©Copyright 1999-2014 Telekinesys Research Ltd. (t/a Havok). All rights reserved. [1]

[1] Havok.com and the Havok buzzsaw logo are trademarks of Havok. All other trademarks contained herein are the properties of their respective owners.

This document is protected under copyright law. The contents of this document may not be reproduced or transmitted in any form, in whole or in part, or by any means, mechanical or electronic, without the express written consent of Havok. This document is supplied as a manual for the Havok game dynamics software development kit. Reasonable care has been taken in preparing the information it contains. However, this document may contain omissions, technical inaccuracies, or typographical errors. Havok does not accept any responsibility of any kind for losses due to the use of this document. The information in this document is subject to change without notice.

# Contents


**1 Common Havok Components** **6**

1.1 Base Library . . . 7
- 1.1.1 Introduction . . . 7
- 1.1.2 The Base System . . . 7
- 1.1.3 Container Classes . . . 16
- 1.1.4 Error Reporting . . . 19
- 1.1.5 Monitoring and Timing . . . 23
- 1.1.6 Working with Streams . . . 26
- 1.1.7 Microsoft Visual Studio Debugger Expansions . . . 31

1.2 Memory Management . . . 31
- 1.2.1 Overview . . . 32
- 1.2.2 Default Implementations . . . 36
- 1.2.3 Customization . . . 43
- 1.2.4 Reporting . . . 46

1.3 Havok Math Library . . . 62
- 1.3.1 Introduction . . . 62
- 1.3.2 Math Types . . . 62
- 1.3.3 Using the Math Library . . . 63
- 1.3.4 Common Conventions . . . 66
- 1.3.5 SIMD and Non SIMD support . . . 67
- 1.3.6 Optimization . . . 68

1.4 Serialization . . . 69
- 1.4.1 Introduction to Serialization . . . 69
- 1.4.2 Basic Loading and Saving . . . 70
- 1.4.3 File Formats . . . 73
- 1.4.4 Details of Loading and Saving . . . 75
- 1.4.5 Reflection . . . 82
- 1.4.6 Versioning . . . 84
- 1.4.7 Serialization and Physics 2012 . . . 88
- 1.4.8 Using the Infrastructure . . . 93

1.5 Multithreading . . . 103
- 1.5.1 Introduction . . . 103
- 1.5.2 Stepping several multithreaded workloads . . . 107
- 1.5.3 Multithreading Physics 2012 . . . 107
- 1.5.4 Multithreading Animation . . . 111
- 1.5.5 Multithreading Collision Queries . . . 111
- 1.5.6 Multithreading Behavior . . . 111
- 1.5.7 Timers . . . 113
- 1.5.8 The Master thread, and hkThreadNumber . . . 115

1.6 Geometry Processing . . . 115
- 1.6.1 Introduction . . . 115
- 1.6.2 Convex Hull . . . 115

1.6.3 Collision Geometry Optimizer . . . 125
1.7 Visualization . . . 133
1.7.1 Introduction . . . 133
1.7.2 Local visualization in your application . . . 134
1.7.3 Remote visualization with the Visual Debugger . . . 136

**2 Havok Physics 2012 164**
2.1 Introduction . . . 165
2.1.1 What is a Physics Engine? . . . 166
2.1.2 What is Havok Physics 2012? . . . 183
2.1.3 Demo Resources . . . 187
2.2 Creating a Simulation . . . 189
2.2.1 Creating Physics 2012 Objects . . . 189
2.2.2 Simulating a Physics 2012 World . . . 271
2.3 Interacting with a Simulation . . . 304
2.3.1 Collision Filtering . . . 304
2.3.2 General Listeners . . . 314
2.3.3 Collision Listeners . . . 318
2.3.4 Response Modifiers . . . 327
2.3.5 Collision Queries . . . 332
2.4 Optimizing a Simulation . . . 346
2.4.1 Profiling . . . 346
2.4.2 Tweaking Physics 2012 Behavior . . . 347
2.4.3 Optimizing Performance . . . 356
2.4.4 Optimizing Memory . . . 367
2.5 Extensions . . . 374
2.5.1 Character Control . . . 375
2.5.2 Vehicle Physics . . . 405
2.5.3 Saving Contact Points . . . 433
2.6 Articles . . . 435
2.6.1 Rotations, Handedness, and all that. . . . 435
2.6.2 Continuous Physics . . . 459
2.6.3 Integrating Particles with Havok Physics . . . 482
2.6.4 Ragdoll Best Practices . . . 484

**3 Havok Animation 503**
3.1 Introduction . . . 504
3.2 Architecture Overview . . . 504
3.2.1 Object Overview / Glossary . . . 504
3.2.2 Export Path . . . 507
3.3 Animation Runtime . . . 509
3.3.1 Animation Overview . . . 509
3.3.2 Pose Representation . . . 512
3.3.3 Animation and Rig Representation . . . 533
3.3.4 Motion Extraction/Locomotion . . . 536
3.3.5 Animation Compression . . . 548
3.3.6 Playback . . . 555
3.3.7 Multithreading . . . 566
3.3.8 Skeleton Mappers . . . 568
3.3.9 Retargeting Animations . . . 578
3.3.10 Skeleton Partitioning . . . 584
3.3.11 Inverse Kinematics . . . 585
3.3.12 Deformation (Skinning And Morphing) . . . 603
3.4 Integrating Havok Animation with Havok Ragdolls . . . 605
3.4.1 Introduction . . . 605

3.4.2 A Typical Scenario . . . 605
3.4.3 Ragdoll Instances . . . 609
3.4.4 Mappers . . . 612
3.4.5 Ragdoll Controllers . . . 612
3.4.6 The Detect Ragdoll Penetration Utility . . . 614
3.5 Animation Examples and Use Cases . . . 619
3.5.1 Animation Showcase . . . 619

**4 Havok Content Tools** **626**
4.1 Guide To The Havok Content Tools Documentation . . . 627
4.1.1 Introduction . . . 627
4.1.2 Documentation for Content Creators . . . 627
4.1.3 Documentation for Tools Programmers . . . 630
4.2 Introduction and Architecture . . . 631
4.2.1 Introduction . . . 631
4.2.2 Architecture . . . 632
4.3 The Havok Filter Pipeline . . . 635
4.3.1 The Filter Manager . . . 635
4.3.2 Core Filters . . . 642
4.3.3 Physics Filters . . . 653
4.3.4 Physics 2012 Filters . . . 657
4.3.5 Animation Filters . . . 667
4.3.6 Graphics Filters . . . 710
4.4 3ds Max Tools . . . 715
4.4.1 Introduction . . . 715
4.4.2 3ds Max: Scene Exporter . . . 715
4.4.3 3ds Max: Physics Tools . . . 719
4.4.4 3ds Max: Animation Tools . . . 741
4.4.5 Tutorial: Export and Animation Basics . . . 741
4.4.6 Tutorial: Physics 2012 Basics . . . 755
4.4.7 Tutorial: More on Rigid Bodies . . . 772
4.4.8 Tutorial: Rag Doll Toolbox . . . 785
4.5 Maya Tools . . . 800
4.5.1 Introduction . . . 800
4.5.2 Maya: Scene Exporter . . . 802
4.5.3 Maya: Physics Tools . . . 806
4.5.4 Maya: Animation Tools . . . 827
4.5.5 Tutorial: Export and Animation Basics . . . 830
4.5.6 Tutorial: Physics Basics . . . 843
4.5.7 Tutorial: More on Rigid Bodies . . . 859
4.5.8 Tutorial: Rag Doll Setup . . . 869
4.6 XSI Tools . . . 883
4.6.1 Introduction . . . 883
4.6.2 XSI: Scene Exporter . . . 884
4.6.3 XSI: Physics Tools . . . 892
4.6.4 XSI: Animation Tools . . . 912
4.6.5 Tutorial: Export and Animation Basics . . . 915
4.6.6 Tutorial: Physics Basics . . . 928
4.6.7 Tutorial: More on Rigid Bodies . . . 942
4.6.8 Tutorial: Rag Doll Toolbox . . . 952
4.7 Common Concepts . . . 972
4.7.1 Introduction . . . 972
4.7.2 Common: Controlling File Size . . . 973
4.7.3 Physics : Rigid Body Concepts . . . 974

4.7.4 Physics : Constraint Concepts . . . 980
4.7.5 Animation: Repositioning Animations and Motion Extraction . . . 999
4.7.6 Animation: Annotations . . . 1001
4.7.7 Animation: Controlling Compression . . . 1002
4.7.8 Animation: Previewing Compression . . . 1007
4.7.9 Animation: Controlling File Size . . . 1009
4.7.10 Physics and Animation: Local Frames . . . 1010
4.8 The Preview Tool . . . 1011
4.8.1 Introduction . . . 1011
4.8.2 Assets . . . 1012
4.8.3 Main Window Controls . . . 1013
4.8.4 Simulation Controls . . . 1017
4.8.5 Tweaker . . . 1019
4.8.6 Debugging . . . 1019
4.8.7 Restoring defaults . . . 1020
4.9 Integrating with the Havok Content Tools . . . 1020
4.9.1 Introduction . . . 1020
4.9.2 Offline Processing . . . 1020
4.9.3 The Havok Scene Data Format . . . 1023
4.9.4 Extending the 3ds Max Tools . . . 1034
4.9.5 Extending the Maya Tools . . . 1042
4.9.6 Extending the XSI Tools . . . 1046
4.9.7 Writing your own Filters . . . 1054
4.9.8 Customizing your Installation Method . . . 1062
4.9.9 Tutorial: Extending the Toolchain . . . 1069
4.10 Artist Best Practices . . . 1112
4.10.1 Introduction . . . 1112
4.10.2 [Physics] Collision Detection Optimizations . . . 1112
4.11 Troubleshooting . . . 1121
4.11.1 Troubleshooting: Processing Assets . . . 1121
4.11.2 Troubleshooting: 3ds Max Tools . . . 1126
4.11.3 Troubleshooting: Maya Tools . . . 1130
4.11.4 Troubleshooting: XSI Tools . . . 1132
4.11.5 Troubleshooting: Toolchain Integration . . . 1133

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

# Chapter 1

# Common Havok Components

## 1.1 Base Library

### 1.1.1 Introduction

The Havok base library, `hkBase`, provides functionality common to all Havok libraries. This includes

- A platform-independent interface to system services, such memory management and timing
- Basic data types and container classes

Havok has been specifically designed to allow you to customize its default behavior so that it matches how your existing environment functions. Here, we take a detailed and practical look at how to hook in to all of the integration points in the Havok Base Library. We will also show how default behavior can be extended or replaced for many of these system functions.

### 1.1.2 The Base System

#### 1.1.2.1 Configuring and Customizing hkBase

All platforms use common `hkBase` header files. The platform-specific code is out of line so that you can change the implementation without affecting the rest of Havok. `hkBase` is the only Havok library that includes non-Havok headers.

The default `hkBase` implementations are fully usable without modification. Alternatively, you can make your own changes or additions to the provided source code. Havok provides a simple mechanism for updating and replacing many of the `hkBase` classes, which you can find out more about in the Singletons section of this chapter.

**Base Root Directory**

This contains `hkBase.h`, which includes all the other hkBase headers.

**Non-replaceable code**

Some parts of `hkBase` are compiled into the Havok libraries, and are not replaceable without a complete rebuild of all libraries. These include

- the container classes, which have inline methods

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

- the build configuration options in the `config` directory

In general, none of the `hkBase` header files should be modified.

**Compiler Warnings**

An important function of `hkBase.h` is to enable and disable compiler warnings. Havok has determined that some level 4 compiler warnings in the Visual Studio compilers (.NET and Visual C++ 6) can be disabled without affecting performance. Disabling these warnings results in better code readability and class design, without compromising performance. The warnings are disabled in `Common/Base/hkBase.h`. Please note that as almost all Havok headers include `Common/Base/hkBase.h`, if any Havok header is included in a source file you will be disabling certain warnings for that source file. If you require these warnings in non Havok code it is recommended that you `#pragma` reset or re-enable the warnings as necessary.

**Visual Studio Warnings**

The following level 4 compiler warnings are disabled in `hkBase.h` under Visual Studio. Before any of the following pragma warnings are disabled, a pragma push is issued. The push stores the current warning state so that it can be reverted to easily using a pragma pop anytime after these warnings have been enabled.

- C4100 'identifier': unreferenced formal parameter.
  *A function need not use all its arguments.*
- C4127 conditional expression is constant.
  *Constant conditionals are often used inside asserts.*
- C4505 'function': unreferenced local function has been removed.
  *Lots of inline functions are not used in a compilation unit.*
- C4510 'class': default constructor could not be generated.
- C4511 'class': copy constructor could not be generated.
- C4512 'class': assignment operator could not be generated.
  *Many classes are not designed with value semantics.*
- C4514 unreferenced inline/local function has been removed.
  *Lots of inline functions are not used in a compilation unit.*
- C4714 force inlined function not inlined.
  *This warning is only disabled in debug modes.*

`hkBase.h` is implicitly included by all other Havok headers. If you need to fine-tune warning options, it's best to group all Havok includes and issue the pragma pop at the end of the block.

```
#include <Physics/Physics/hknpPhysics.h>
#include <Physics/Physics/Dynamics/Body/hknpBody.h>
// all Havok includes in a single block
#pragma warning(pop)
```

#### Metrowerks Warnings

Havok does not explicitly disable or enable any warnings for Metrowerks CodeWarrior. Instead Havok code is manually tested for certain CodeWarrior warnings prior to each release. This test includes most warnings in the project settings panel with the exception of the warning for unused arguments which is ignored. Due to Havok's class design it is typical for some methods to have parameters but no implementation. Havok recommends that this warning is disabled when including Havok headers. A warning that Havok does monitor closely is the warning for implicit conversion for arithmetic operators. Prior to each release Havok attempts to minimize these warnings in its code base.

#### Compile-time configuration

The `config` directory contains the compile time settings used when building all the Havok libraries–these include global debug output options, and whether to use the memory manager.

Note that any changes you make to this information will require a *full rebuild* of Havok. Changing configuration values with compiled libraries will not work.

### 1.1.2.2 Base Types

The `hkBaseTypes.h` header provides you with some useful macros and basic types.

#### Platform information

`hkBaseTypes` defines a number of macros based on the platform information obtained from the compiler. These are

- `HK_COMPILER_*`: the compiler
- `HK_COMPILER_*_VERSION`: the compiler version
- `HK_ARCH_*`: the CPU
- : the operating system

These definitions, in turn, are used to define some further platform-specific macros in `hkBaseTypes`.

#### Basic types

`hkBaseTypes` provides some Havok-specific basic types, including floating-point types and unsigned and signed integers. The default Havok floating-point type is `hkReal`.

#### Aligned variables

`hkBaseTypes` defines macros that allow you to specify an alignment in memory when declaring a variable, as in the following example:

```
HK_ALIGN16( hkReal myVar );
```

These macros are platform-dependent, and are defined based on the platform information macros.

**hkBaseObject**

This is the base class for all Havok classes that have virtual functions. Its main purpose is to ensure that all objects' vtables are at the very start of the object with all compilers. (For example if `hkReferencedObject` had no parent, some compilers would put the data members before the vtable.)

#### 1.1.2.3 Runtime Initialization

Havok Initialization is a two-stage process. The first stage, memory system initialization, has been separated from the rest of the system for customizability. The second stage is responsible for creating the rest of Havok's subsystems such as error handling, stream handling and debugging utilities.

There are utility classes to perform the default setup. The following block shows a typical initialization. The Memory Management section goes into more details of what an `hkMemoryRouter` is and what the default setup does.

```
hkMemoryRouter* memoryRouter = hkMemoryInitUtil::initDefault( hkMallocAllocator::m_defaultMallocAllocator,
  hkMemorySystem::FrameInfo( 500* 1024 ) );
hkBaseSystem::init( memoryRouter, errorReport );
```

The *errorReportFunction* and *errorReportObject* are used by Havok's error handler. The error handler is responsible for dealing with any asserts, errors, warnings or reports that occur in the engine. The default error handler is `<Common/Base/Error/hkDefaultError.h>`. It uses the supplied *errorReportFunction* to print its messages. For details on how to further customise the error reporting methods and use the *errorReportObject* parameter please see the Error Reporting section.

```
static void HK_CALL errorReport(const char* msg, void* userArgGivenToInit)
{
    if(s_verbose)
        log("%s\n", msg);
}
```

Similarly, to shutdown and delete system objects and memory:

```
hkBaseSystem::quit();
hkMemoryInitUtil::quit();
```

**Note:**
Note that as the memory system is not initialized until after `main()` starts, any statically initialized classes that need to allocate memory will fatally fail. In practice, this means that most non-trivial Havok objects cannot be global or static.

**Per-Thread Initialization**

Each thread used by Havok has several thread local variables which need to be initialized. For example, each thread has a local pointer to a memory manager object and its own timer data. In the previous example, the main thread has already been initialized and only non-main threads need to call the thread initialization functions.

Again initialization has a distinct memory stage, followed by everything else.

```
hkMemoryRouter& memoryRouter = m_memoryRouter;
hkMemorySystem::getInstance().threadInit(memoryRouter, "hkWorkerThreadContext");
hkBaseSystem::initThread(&m_memoryRouter);
```

At the end of the thread's life there is thread shutdown code.

```
hkBaseSystem::quitThread();
hkMemorySystem::getInstance().threadQuit(m_memoryRouter);
```

#### 1.1.2.4 Code Size Customization–Product Features

Normally, the linker is able to strip unused code from your executable. However there are sometimes cases where the linker is not able to guarantee that certain code is unreachable. This is usually because the code path which forces its inclusion depends on a runtime variable. If you know that certain code paths will not be executed, you can explicitly force their removal and reduce the code size.

Different systems have different underlying ways to allow their removal. These differences are hidden by `hkProductFeatures.cxx`. The basic idea is that you `#define` the features to be excluded and then `#include <Common/Base/Config/hkProductFeatures.cxx>` which defines a custom method called `hkProductFeatures::initialize()` which does the appropriate initialization. This method is called during `hkBaseSystem::init()`. Alternatively you can write your own `hkProductFeatures::initialize()` method and not include the `.cxx` file.

This is an opt-out system. Macros corresponding to excludable product features are grouped in varying levels of granularity in `hkProductFeatures.inl`. At the most coarse level, entire products can be excluded. Havok products are normally included according to the presence of a valid keycode, but can be excluded by undefining the product macro. `hkProductFeatures` should usually be used in conjunction with the keycode include, as feature registration depends on what products are present. For example if a keycode for AI is present, but only Havok Physics 2012 is required, AI may be stripped as follows.

```
#include <Common/Base/keycode.cxx>
#undef HK_FEATURE_PRODUCT_AI // was defined by the product keycode
#include <Common/Base/Config/hkProductFeatures.cxx
```

Creating the `hkBuiltinTypeRegistry` will generate a `hkTypeInfo` and `hkClass` for each serializable Havok class present. This will cause code for each library in each product to be included. Individual libraries may be forcibly stripped when not required by defining the corresponding macro. For example the vehicle library can be removed with the following code.

```
#include <Common/Base/keycode.cxx>
#define HK_EXCLUDE_LIBRARY_hkpVehicle // not using hkpVehicle library
#include <Common/Base/Config/hkProductFeatures.cxx>
```

`hkProductFeatures` ties together a number of features in the SDK.

- `hkClass` registration
- Old serialization system
- Versioning patch registration
- Memory tracker registration
- Optional Havok features

The `hkBuiltinTypeRegistry` is automatically generated. It is not necessary to include `hkBuiltinTypeRegistry.cxx` as in pre-2010.1 versions of Havok. By default this will create reflection data for all classes. A custom class list may be employed by defining a `HK_CLASSES_FILE`.

By default the library will link in the information required to patch previous versions of Havok files enabling them to work with the current version. The amount of backward compatibility you require can be configured by setting `HK_SERIALIZE_MIN_COMPATIBLE_VERSION` to one of the values in the file `hkHavokVersions.h`. For example if compatibility is only required from 6.5 beta 1 add the following define

```
#define HK_SERIALIZE_MIN_COMPATIBLE_VERSION 650b1
```

If you know that all of your files have been exported with the current version, you can remove all of the patching information and only allow the loading of the current version of files by using 'Current' as the type.

```
#define HK_SERIALIZE_MIN_COMPATIBLE_VERSION Current
```

`hkSerializeDeprecated` is by default initialised using the versions specified in `hkCompatVersions.h`. A custom list of versions may be provided as required by defining `HK_COMPAT_FILE`.

`hkProductFeatures` will also automatically populate the `hkVersionPatchManager` with product patches according to the product keycodes present. Undefining a specific product macro before the `hkProductFeatures` constructor is called will avoid patches for that product being generated.

Most unused features in the Havok libraries will be automatically dead-stripped. There are however instances where code will be included when it may not be required. Explicitly removing such code can lead to substantially reduced code size. The reduction in code size per feature will vary based on the platform and on the nature of the project. As an example, when the code below is included in the Havok ConsoleExample demo, the resulting reductions in code size as indicated in the comments next to each macro may be achieved. These are ballpark figures representing a Release build using Microsoft Visual Studio on Win32.

```
#include <Common/Base/keycode.cxx>

// Features excluded and resulting reduction in size of executable:
#define HK_EXCLUDE_FEATURE_SerializeDeprecatedPre700 // 4000k
#define HK_EXCLUDE_FEATURE_RegisterVersionPatches // 60k
#define HK_CLASSES_FILE "AFileWithNoClassDefinitions.h" // create an empty hkBuiltinTypeRegistry: 80k
#define HK_EXCLUDE_LIBRARY_hkGeometryUtilities // 50k
#define HK_EXCLUDE_LIBRARY_hkpUtilities // 70k
#define HK_EXCLUDE_LIBRARY_hkpVehicle // 400k
#define HK_EXCLUDE_FEATURE_hkpHeightField // 10k
#define HK_EXCLUDE_FEATURE_hkpContinuousSimulation // 25k
#define HK_EXCLUDE_FEATURE_hkpMultiThreadedSimulation // 50k
#define HK_EXCLUDE_FEATURE_hkpAabbTreeWorldManager // 15k
#define HK_EXCLUDE_FEATURE_hkpKdTreeWorldManager // 40k
#define HK_EXCLUDE_FEATURE_hkpAccurateInertiaTensorComputer // 400k

#include <Common/Base/Config/hkProductFeatures.cxx>

/*
* Can also further reduce code size:
* Only registering those collision agents required // 200k
* Removing references to the VDB // 800k
*/
```

#### 1.1.2.5 Singletons

`hkSingleton` is a utility class used in Havok to define global objects. Singletons are created when `hkBaseSystem::init()` is called, and are deleted automatically when `hkBaseSystem::quit()` is called. You access a singleton using its `getInstance()` method.

Many of the `hkBase` classes described in this chapter, including `hkError` and `hkFileSystem`, are `hkSingleton`s, which facilitates managing and replacing them.

**Replacing singletons**

You can create your own version of any of the `hkBase` singletons–for example, your own error handling class–by subclassing the appropriate class and implementing the virtual functions.

To use your own custom singleton, call `hkSingleton::replaceInstance()`. This allows you to replace any singleton without recompiling Havok.

```
hkError::replaceInstance( new myErrorHandler() );
```

Note that only singletons which are null are created in `hkBaseSystem::init()`. Thus you may set up any replacements before calling `hkBaseSystem::init()` instead of creating the default and immediately replacing it. Similarly, singletons may be destroyed manually by replacing them with null before `hkBaseSystem::quit()`.

**Instantiating singletons**

You can specify that you want an `hkSingleton` to be instantiated in several ways, using macros defined in `hkSingleton.h`. The default is `HK_SINGLETON_IMPLEMENTATION(CLASS)`,

which means that the singleton `CLASS` is automatically created when `hkBaseSystem::init()` is called. For example `HK_SINGLETON_IMPLEMENTATION(hkDebugDisplay)`. When you create a derived singleton which implements an interface, use `HK_SINGLETON_CUSTOM_IMPLEMENTATION`. For example to use `PlatformSpecificSystemClock` as the system clock use `HK_SINGLETON_CUSTOM_IMPLEMENTATION(hkSystemClock, PlatformSpecificSystemClock)`.

If you don't want a singleton to be initialized in `hkBaseSystem::init`, use `HK_SINGLETON_MANUAL_IMPLEMENTATION()` and the singleton instance will remain null until explicitly replaced. In this case the singleton will not be automatically destroyed in `hkBaseSystem::quit()`: you can destroy it whenever you want calling `replaceInstance()` with an argument of `HK_NULL`.

#### 1.1.2.6 Reference Counting

**hkReferencedObject**

`hkReferencedObject` is a subclass of `hkBaseObject`. It's used as the base class for Havok objects that can have multiple owners, and facilitates Havok's object management mechanism. Rigid bodies, constraints, and actions are all `hkReferencedObject`s.

All `hkReferencedObject`s store size information about themselves for use by the memory manager. You can find out more about how the memory manager works in the Memory section.

A `hkReferencedObject` has a reference counter that is incremented each time a reference is added to the object, and decremented when the reference is removed. When a `hkReferencedObject` is created, it automatically has a reference count of 1. The object is deleted from memory when its reference count reaches zero. You can increment and decrement the reference counter using the `addReference()` and `removeReference()` functions.

To see how this works, let's say you create a Havok rigid body in your game code and then add it to the physics engine using the `hkpWorld`'s `addEntity()` method. When the object is created, it has a reference count of 1. This is incremented to 2 by the `hkpWorld` when the object is added to it. When Havok is finished with the object, the reference count is decremented back to 1. As the reference count is not zero, the object is not deleted from memory and can be reused in your code, thus saving the overhead of creating a new object. So, for example, if you create a piece of debris that falls through and out of the physical scene, you can keep removing and then re-adding the same object to the scene to create more debris.

Alternatively, if you don't want to reuse a `hkReferencedObject`, you can relinquish ownership of it by calling `removeReference()`, as in the example below.

```
m_world->addEntity(rigidBody);
rigidBody->removeReference();
```

As you know, this decrements the reference count. Because of this, the rigid body's reference count will be reduced to zero when the physics engine is finished with it, and the object will be deleted automatically. You don't have to worry about deleting the object yourself.

You can quickly find out which Havok classes are `hkReferencedObject`s by checking the `hkReferencedObject` class hierarchy diagram in the Reference Manual.

### hkRefPtr

`hkRefPtr` is a smart pointer which in many cases removes the need to explicitly call `addReference` and `removeReference`. By its nature it is more resistant to resource leaks by omitted `removeReference` calls.

```
const hkpShape* shape1 = rigidBody->getCollidable()->getShape();
// shape1 may become invalid if the body is deleted or its shape changed
hkRefPtr<hkpShape> shape2 = rigidBody->getCollidable()->getShape(); // adds a referernce
// We're guaranteed that shape2 is valid for the lifetime of the refptr
```

There is one complication to be aware of with `hkRefPtr` and newly created objects. Since objects are created with a reference count of 1, the initial assignment adds an extra reference which must be removed.

```
extern hkReferencedObject* createAnObject()
{
  return new hkReferencedObject(); // refcount = 1
}
{
  hkRefPtr<hkReferencedObject> obj = createAnObject(); // refcount = 2
  // ERROR on block exit refcount = 1, resource leak!
}
{
  hkRefPtr<hkReferencedObject> obj = createAnObject(); // refcount = 2
  obj->removeReference(); // OK, refcount = 1, now owned by the refptr
}
```

### hkRefNew and hkRefLoan

`hkRefNew` was introduced to provide some compile time safety and to be explicit about the user's responsibility for returned values. Functions returning `hkRefNew` indicate that the caller must eventually cause a reference to be removed. `hkRefNew` can transfer its ownership to a `hkRefPtr` which makes the code more natural than the previous example.

```
extern hkRefNew<hkReferencedObject> createAnObject()
{
  return new hkReferencedObject(); // refcount = 1
}
{
  hkRefPtr<hkReferencedObject> obj = createAnObject(); // OK, no extra refcount
  // hkReferencedObject* obj = createAnObject(); // ERROR, does not compile
  hkReferencedObject* obj = createAnObject().stealOwnership(); // stealOwnership is a promise to call
    removeReference later
}
```

Classes using `hkRefNew` will normally use a named constructor or factory method pattern.

`hkRefLoan` is a simple pass-through wrapper for a raw pointer. It exists only to be explicit that the pointer is not owned by the caller and that the pointer is valid only for some small scope and may become invalid through a mutation of the callee.

### Conventions

Code which does not use `hkRefNew` and `hkRefLoan` adheres to some simple rules.

- `new` always returns objects with a reference count of 1
- `createX` methods return a new reference
- `getX` methods return a borrowed reference (analogous to `hkRefLoan`)

### 1.1.3 Container Classes

#### 1.1.3.1 Arrays

`hkArray` is the default Havok array class. Arrays of Havok objects must be allocated using a `hkArray` in order to be tracked correctly by the memory system. It is very similar to the standard STL array, with a number of important differences:

**Naming conventions**

The `hkArray` functions use a slightly different naming convention to the STL functions. For example, the corresponding `hkArray` function to the STL `push_back()` function is called `pushBack()`.

**Preserving order**

The `hkArray` `removeAt()` function does not preserve the order of the array elements; instead, the last array element is used to replace the removed element, to save the overhead of copying all the subsequent elements back one place in the array. To get the STL behavior when removing an element, you use the `removeAtAndCopy()` function.

**Unchecked functions**

`hkArray` has special unchecked versions of `pushBack()`, `setSize()` and `expandBy()`. These are called `pushBackUnchecked()`, `setSizeUnchecked()`, and `expandByUnchecked()`, respectively. With these functions, the `hkArray` does not check if the array capacity needs to be increased before performing the task. These functions are useful if you have already reserved an appropriate amount of memory using `reserve()`.

**Type traits**

`hkArray` uses Havok type traits for optimized handling POD and non-POD items.

**Note:**
A POD type in C++ is an aggregate class that contains only POD types as members, has no user-defined destructor, no user-defined copy assignment operator, and no nonstatic members of pointer-to-member type. This implies that a C++ plain old data structure inherits only nonvirtually from other POD types, if at all. These restrictions ensure that the compiler or program can initialize, copy, and assign to POD types using only primitive memory copying functions and machine instructions.

Some code paths are different depending on if the element type is POD or not. All of the builtin fundamental types (int, float, etc.) are treated as POD. All aggregate (struct & class) objects default to being treated as non-POD. In some cases, you may have a type which you wish to treat as POD, usually because the compiler has more optimization opportunities. To do this, use one of the Havok type traits macros, noting that the behavior is undefined if the class contains any non-POD members:

- `HK_DECLARE_POD_TYPE();`

  Use this macro if your class should be treated as POD. This macro must be specified in public area of the class body.

- `HK_DECLARE_POD_TYPE_IF_POD( T );`

  Use this macro to declare this structure as POD if and only if type $T$ is POD. This macro must be specified in the public area of the class template body, where the $T$ is the template argument.

- `HK_DEFINE_AS_POD_TYPE( classname );`

  This is an alternate syntax for when the above declarations cannot be used, for instance for fundamental types or for unmodifiable source. This must be specified outside the class body.

When an non-POD object is used in an `hkArray`, the following must be publicly accessible:

- default constructor
- copy constructor
- assignment operator
- destructor

### Example

For example, to allocate an array of 20 `hkVector4` objects using the default allocator use

```
hkArray<hkVector4> vecArray(20); // synonym for hkArray<hkVector4>::Heap vecArray(20);
```

To allocate an empty array of vectors from the `Temp` allocator, use

```
hkArray<hkVector4>::Temp tempVecArray;
```

### Array allocators

`hkArray` is templated on the allocator type. The default allocator goes to the heap. There are typedefs for using different allocators such as `hkArray<T>::Temp` which uses the frame local allocator. Also of note is `hkArrayBase<T>` which has no builtin allocator. Any operations which may involve reallocation require an allocator to be passed as a parameter. All `hkArray` types are implicitly convertible to `hkArrayBase` and so is the preferred type for const method arguments.

### Using existing arrays

If you have an existing array that you want to use in Havok, you can use your array to create a new `hkArray` with the following constructor. The *ptr* parameter is a pointer to your array data.

```
inline hkArray(T* ptr, int size, int capacity);
```

Note that the original array is not copied, just used in place. You should ensure that the data that *ptr* points to is valid for the scope of the `hkArray`. An existing `hkArray` can also be used to wrap an external array. The functions

```
inline void hkArray::setDataAutoFree(T* ptr, int size, int capacity);
inline void hkArray::setDataUserFree(T* ptr, int size, int capacity);
```

cause the `hkArray` to use the specified storage. Any data already in the `hkArray` is discarded. The `AutoFree` version passes management of the storage to the `hkArray`, which will free it on destruction. The `UserFree` version will not free or reallocate the storage, even on destruction–if the array grows beyond the available size, an entirely new buffer is allocated.

**Inplace arrays**

`hkInplaceArray` is a subclass of `hkArray`. An `hkInplaceArray` has internal storage capacity for its elements within the object itself. The creation of an inplace array is much faster than the creation of an ordinary `hkArray` since no dynamic allocations occur. They can be very useful in cases where the size of an array is likely to be less than a small fixed size.

When you create an inplace array, you specify an initial size for it. If you subsequently make the array too large for all the elements to be stored within the object, Havok converts your inplace array into an ordinary `hkArray`.

There is also a `hkInplaceArrayAligned16` class. Arrays of this class have a small internal storage capacity, aligned to 16 bytes within the object.

**Deallocate mask**

Each `hkArray` has an `m_capacityAndDeallocateFlag` member that stores the array's capacity in the lower 31 bits. The highest bit is used to indicate whether the memory storing the original array elements should be deallocated if the array is resized or destroyed. If the highest bit is set, the memory will not be freed. This flag is set, for instance, if you create a `hkArray` from an existing C-style array or `hkInplaceArray`.

**Sorting**

`hkBase` provides you with two different classes that you can use to sort the elements of an array: `hkHeapSort` and `hkQuickSort`. Each element of the array to be sorted must have the $<$ operator implemented.

The macro `hkSort` is defined per platform to point to either `hkHeapSort` or `hkQuickSort` depending on performance characteristics.

#### 1.1.3.2 Maps

**hkMap and hkMapBase**

`hkMap` and `hkMapBase` are hash tables with open addressing (also known as closed hashing). The key operations (hash, equal, etc.) are specified in an operations structure. The default operations are in

hkMapOperations but may be overridden if desired. The lower level hkMapBase requires an explicit allocator for every potentially mutating operation while the higher level hkMap has an allocator built in to its type.

These classes are slightly unusual in that they require explicit template instantiation. See hkMap.cpp for an example.

**hkPointerMap and hkStringMap**

hkPointerMap is a hashing map for the common case where the keys and values are 4 or 8 byte integers. To minimize template code bloat, all methods are thin wrappers around a single hkMap type.

hkStringMap is similar, except that the hashing function is suitable for strings and that the string hashes are cached for quick string equality tests.

Note that in both cases, the maps hold only the pure key and value pairs - the pointed to objects and strings are not copied.

### 1.1.4 Error Reporting

The Havok libraries have embedded error checking which can be used to help debug problems. There are three types of error supported by Havok: warnings, asserts and errors. Macros are provided for convenience to raise these errors. This also means that they can be easily compiled out so that they use no CPU cycles.

**Note:**
Remember, asserts and warnings are not raised in release builds, so make sure you are linking against the debug versions of the libraries.

#### 1.1.4.1 hkError

The hkError class is an hkSingleton that provides a number of error reporting functions to direct error, warning, and assert information from Havok to an appropriate location.

The default implementation of hkError is hkDefaultError, which prints out a message for each error, warning, or assert to the standard error output, e.g. STDERR. During the initialization of hkBaseSystem a hkErrorReportFunction must be specified. Typically this just forwards to printf but it is also possible to invoke a method on a custom error handler object.

The four levels of error have the following behaviour:

| Error Level | Execution Impact | Disabling | Release/Debug |
|---|---|---|---|
| HK_ERROR | Break Execution | No | Release, Debug |
| HK_ASSERT | Break Execution | Yes | Debug |
| HK_WARN | Continue Execution | No | Debug |
| HK_REPORT | Continue Execution | No | Release, Debug |

| | |
|---|---|
| HK_ERROR | Header files and libraries have different build numbers |
| HK_ASSERT | Length of ray cast must be non-zero |
| HK_WARN | You are setting the collision filter after adding entities. Collisions between these entities will not have been filtered correctly. |
| HK_REPORT | Visual debugger server initialized, my IP address is 10.55.1.10 |

Table 1.1: Example errors raised from within Havok

The file and line number are output in addition to the type of message (warning, assert, error) and a descriptive string that indicates the reason for the error.

#### 1.1.4.2 Raising an Error

The following are examples of how you can raise an error, assert or warning with or without the convenience macros:

```
#include <Common/Base/System/Error/hkerror.h>

HK_WARN( "You haven't initialised the memory manager!" );
HK_ERROR( "Null object passed as parameter." );

// Asserts evaluate an expression and print out the expression and description
HK_ASSERT2(0x473a8c3a, (pRigidBody != HK_NULL), "Null body passed as parameter" );

// You can use the error handler without using the MACROS
hkError::getInstance().message( hkError::MESSAGE_ERROR, 0x10101010, "Directly accessing the error handler.
  ", __FILE__, __LINE__);
```

#### 1.1.4.3 Creating a Custom Error Handler

The error handling system provides a means to redirect the default error output to fit into your existing framework.

There are two main ways of redirecting error output, the first approach is to use Havok's `hkDefaultError` handler in conjunction with user defined print functions or print objects. The simplest approach is to pass a `hkErrorReportFunction` to `hkBaseSystem::init` when the Havok subsystem's (including `hkDefaultError`) are being constructed. You can do this as follows:

```
#include <Common/Base/System/hkBaseSystem.h> // include for hkBaseSystem
extern "C" int printf(const char* fmt, ...); // For printf, used by the error handler

// Stub function to print any error report functionality to stdout
// std::puts(...) could be used here alternatively
static void errorReportFunction(const char* str, void* errorOutputObject)
{
    printf("%s", str);
}

{
...
    hkMemoryRouter mainMemRouter;
    // init memory system
    hkBaseSystem::init( &mainMemRouter, errorReportFunction );
...
}
```

**Note:**

`printf` is not useful for error reporting on consoles, since there is no console window. Instead, the error should be sent to some platform-specific or custom debug handler.

It is also possible to invoke a method on a custom error handler object. Use the *errorReportObject* parameter interface to specify the custom error handler object. A custom *errorReportFunction* can be used to invoke the method on the custom error handler object. For example consider the following pseudo code:

```
class MyErrorHandler
{
    public:
        virtual void printError(const char* error);
};

MyErrorHandler g_gameError; // The instance of your error handler

void errorReport(const char* s, void* errorReportObject)
{
    static_cast<MyErrorHandler>(errorReportObject)->printError(s);
}

{
...
    hkMemoryRouter mainMemRouter;
    // init memory system
    hkBaseSystem::init( &mainMemRouter, errorReport, &g_gameError );
...
}
```

The second approach is to create your own error handler. In this case simply create your own `hkError` subclass, implementing the virtual functions. You can then replace the default error handler using the method described in the hkSingletons section. In this example, we want to redirect the output from the standard error channel to a custom dialog box.

Here's the derived custom error class:

```
// Our custom handler needs to show all warnings, asserts and errors using a dialog box.
class hkCustomError: public hkDefaultError
{
  public:

    HK_DECLARE_CLASS_ALLOCATOR(HK_MEMORY_CLASS_BASE);
    hkCustomError( hkErrorReportFunction errorReportFunction, void* errorReportObject = HK_NULL )
      : hkDefaultError( errorReportFunction, errorReportObject )
    {}

    virtual int message(Message m, int id, const char* description, const char* file, int line)
    {
      // Output everything except reports to my own error output mechanism,
      // in this case a custom dialog class.
      if (m != hkError::MESSAGE_REPORT)
      {
          gDialog->showError( description );
      }

      // We return positive for errors and asserts, but not warnings or reports.
      // A positive return value is used by HK_ASSERT and HK_ERROR to force a breakpoint.
      return m == hkError::MESSAGE_ASSERT || m == hkError::MESSAGE_ERROR;
    }
};
```

It derives from the default error class so that it can redirect errors to the dialog class. We now have an error handler, but it needs to be registered as the handler that Havok will use. When the base system is initialised the default memory object is created. We must replace this with an instance of our own handler using the *hkError::replaceInstance()* method.

```
{
  ...
  // Initialise memory, constructing a default error handler
  hkBaseSystem::init( memoryRouter, errorReportFunction, errorReportObject );

  // Replace default error handler
  hkError::replaceInstance( new hkCustomError( errorReportFunction, errorReportObject ) );
  ...
}
```

#### 1.1.4.4 Disabling errors, warnings and asserts

Be careful! If Havok raises an assert then it usually means that something is fundamentally wrong. Disabling an assert may well let you continue development but should only be done as a short term measure. You should ensure that you address the underlying cause of the assert.

In Havok all warnings and asserts have unique ids which are persistent across releases. Occasionally you may not want to be notified by Havok with an assert or warning every time a certain event happens. To disable warnings, errors or asserts for a particular line of code, you can use the `setEnabled( int id, hkBool enabled )` method, passing it the appropriate assert ID. You can find the ID by examining the output after a particular assert etc. is triggered.

```
#include <Common/Base/System/Error/hkError.h>

// Disable the assert by its id (a hexadecimal number).
hkError::getInstance().setEnabled( 0x44444444, false );
```

#### 1.1.4.5 Stack Trace

`hkStackTracer` provides stack tracing facilities for several platforms. See the implementation folder `Common/Base/System/StackTracer/Impl`. Depending on platform support, there may be support for translating program addresses to symbolic values. For instance, some platforms can resolve addresses to file:line pairs and function names, others can only resolve addresses to function names. For some platforms external programs may be needed (addr2line for platforms with a GNU toolchain for example).

`hkStackTracer` is used primarily by the debug memory manager and by the default assert handler.

### 1.1.5 Monitoring and Timing

In certain situations, you may need to extract and use statistics from Havok. The Havok libs have embedded profiling code that is easy to output and examine. The monitor classes and macros can be found in the monitor directory. All the timer macros are described in hkMonitorStream.h and hkMonitorStream.inl. A utility class is available for manipulating and viewing the monitor data, called hkMonitorStreamAnalyzer.

The Havok monitor macros are very simple and are compiled into the Havok release build. They simply check whether there is any space available in a thread local buffer, which we call a monitor stream, and if there is they write a string pointer and a timer value together with a command identifier into the stream. If the stream of size 0 then the compiled in monitors do not degrade performance. You can use the macros in your own code too, and if you do your own timers and monitors will show up when you output a monitor analysis.

#### 1.1.5.1 Creating a Timer

You can use a the HK_TIMER macros to time sections of code. You create and use a timer as in the following example:

```
HK_TIMER_BEGIN("Sweep", HK_NULL);

//your code to be timed here

HK_TIMER_END();
```

The first parameter is a string identifying the monitor. The second parameter will allow you to specify a particular object to be timed - however, this functionality has not yet been implemented, so for now you should use `HK_NULL`. You finish timing by calling `HK_TIMER_END`.

#### 1.1.5.2 Creating a Monitor

You can also insert the value of any arbitrary variable into the monitor stream using the HK_MONITOR_ADD_VALUE macro. An example of this is shown below:

```
hkArray<Agent> myArray;
//...

HK_MONITOR_ADD_VALUE("num agents", hkReal(myArray.getSize()),
                     hkMonitor::HK_MONITOR_TYPE_INT);
```

As with the HK_TIMER macros the first value is the string identifier. The send value is the variable you wan to capture. The final parameter is currently unused, and should be set to HK_MONITOR_TYPE_INT. Variables captured in this way will show up in the timer dump files.

#### 1.1.5.3 Setting Up and Printing Monitor Results

In order to view and use monitor results, you can print them as they are updated to the screen or a file. To do this you use the hkMonitorStreamAnalyser class. The code below is from the demo CommonApi/DetailedTimers/DetailedTimersDemo, and shows how to use the hkMonitorStream and the hkMonitorStreamAnalyser.

The hkMonitorStream is a singleton that wraps a thread local buffer into which timers are written during simulation. By default this stream is zero size, so no timers will be captured. To enable timers you have to resize this stream. The following code shows how to do this, and should go in your world setup code:

```
/*
 * At startup / construction time e.g. when you create the hkpWorld
 */
{
        // Allocate space for monitor results
        hkMonitorStream::getInstance().resize( 2 * 1024 * 1024  );// 2 meg for timer info per frame
}
```

This prepares a buffer for the profiling results. Here we allocate 2 Mb, which is certainly enough for capturing the timers from one frame. If you do not allocate enough the timers simply stop being written to the stream when it gets full, so you will have a stream with inconsistent timer results (i.e. missing HK_TIMER_END tokens). Next you need to create a hkMonitorStreamAnalyzer class. You use this class to store multiple frames of data, and to print the data out to file when your timing session is finished.

```
// Create the monitor stream analyzer
  hkMonitorStreamAnalyzer analyzer( 10000000 ); // 10 Mb buffer for multiple frames capture
```

This is a very large buffer that will capture a lot of data. If you do not have this space in memory you can create a smaller buffer.

Before each step you reset the hkMonitorStream buffer. After each step you copy the data from the

hkMonitorStream into the analyzer buffer. The code for this is:

```
/*
 * In your main loop
 */
{
  // Reset monitor stream
  hkMonitorStream::getInstance().reset();

  // Step the world
  m_world->stepDeltaTime( deltaTime );

    hkMonitorStreamFrameInfo frameInfo;
        frameInfo.m_indexOfTimer0 = 0;
        frameInfo.m_indexOfTimer1 = -1;
        frameInfo.m_heading = "usecs";
        frameInfo.m_timerFactor0 = 1e6f / hkReal(hkStopwatch::getTicksPerSecond());
  analyzer.captureFrameDetails(hkMonitorStream::getStart(), hkMonitorStream::getEnd(), frameInfo );
```

When you complete your timer trace, you can output the timers in the hkMonitorAnalyzer to a text file, as follows:

```
// record stats to file
  hkOstream ostr (fileName);
  ostr << TEST_DEMO_NAME "   Timers: \n";
  streamUtility.writeStatistics( ostr );
}
```

If you are running multithreaded, you need to do all this on a per thread basis. Each thread must have its own hkMonitorStream and hkMonitorStreamAnalyzer buffers initialized. Each step each thread must reset the hkMonitorStream, and after each step each thread must call captureFrameDetails to copy the info into its hkMonitorStreamAnalyzer. The threads created by the hkpMultithreadingUtil class in the hctUtilities library create their own buffers. When you wish to output the data to a text file, you need to call hkMonitorStreamAnalyzer::writeMultiThreadedStatistics and pass it the list of streams. See the CommonApi/DetailedTimers/DetailedTimersDemo for the code to do this.

#### 1.1.5.4 Stopwatches

The hkStopwatch class implements a timer that stores time internally as a 64 bit integer.

hkStopwatches are used internally by hkMonitors, and the most straightforward approach to timing sections of your code is to use the hkMonitorBank macros described above. However, you can also use hkStopwatches directly if you like. hkStopwatch has functions for starting, stopping, and resetting a stopwatch, and allows you to get the total running time or the "split" time (the time since the last split time was requested), in either ticks or seconds.

The following snippet illustrates getting both the split time and the total running time from a hkStopwatch:

```
hkStopwatch sw;
sw.start();
function();
sw.stop();
hkcout << "Iter:" << sw.getSplitTicks() << "\tTotal:" << sw.getElapsedTicks();
```

#### 1.1.5.5 System clock

hkSystemClock is an hkSingleton used by hkStopwatch to get its tick counter and ticks per second values. As it is an hkSingleton, it can be easily replaced by your own clock implementation.

### 1.1.6 Working with Streams

#### 1.1.6.1 Low level I/O

The low level input and output of raw bytes is done through the `hkStreamReader` and `hkStreamWriter` interfaces. Havok provides implementations of these classes for

- Memory blocks
- Expandable buffer (hkArray)
- Buffering on top of an unbuffered stream
- Platform specific I/O devices (hard disk, DVD emulator etc)

These low level interfaces are discussed later in this document.

#### 1.1.6.2 hkOstream and hkIstream

The `hkOstream` class is a simple ASCII formatted output stream modelled after `std::ostream`. It contains the usual << operators for basic types. A `hkOstream` simply formats its arguments and forwards them to an `hkStreamWriter` object for writing as a stream of characters.

The following snippet shows two examples of using a `hkOstream`.

```
hkcout << "Hello world\n";
hkcout.printf("Hello world\n"); // equivalent to above.
```

By default, the hkcout `hkOstream` writes to a console window.

Similarly, `hkIstream` is a simple input stream that provides comparable functionality to `std::istream` and requires an `hkStreamReader` to do the actual raw reading of characters.

#### 1.1.6.3 hkOArchive and hkIArchive

Whereas the "Stream" classes operate on text, the "Archive" classes provide I/O for binary data in an endian safe manner. The "Archive" classes perform endian conversion as necessary and again forward to an `hkStreamReader` or an `hkStreamWriter` for the raw byte I/O.

The following snippet shows an example of using a `hkOArchive`.

```
hkOArchive oa("file.dat");
struct {
  int id;
  float scale;
  char data[10];
} foo;
oa.write32( foo.id );
oa.writeFloat32( foo.scale );
oa.writeArray8( foo.data, 10 );
```

#### 1.1.6.4 hkStreamWriter

As you know, Havok streams use an `hkStreamWriter` to write their data.

The `hkStreamWriter` class has two virtual functions that must be implemented. These are:

- `int write(const void* buf, int nbytes) // write the buffer buf`
- `bool isOk() const // has the stream encountered an error?`

For example an implementation which used stdio could be as follows

```
class hkStdioStreamWriter : public hkStreamWriter
{
  public:

    hkStdioStreamWriter(const char* fname)
      :  m_owned(true)
    {
      m_fhandle = fopen(fname, "rb");
    }

    virtual ~hkStdioStreamWriter()
    {
      if(m_fhandle) { fclose(m_fhandle); }
    }

    virtual int write( const void* buf, int nbytes)
    {
      HK_ASSERT(m_fhandle != HK_NULL);
      return fwrite(buf, 1, nbytes, m_fhandle );
    }

    virtual bool isOk() const
    {
      return m_fhandle != HK_NULL && !feof(m_fhandle);
    }

    //
    // This simple example does not implement seek/tell
    //
  protected:

    FILE* m_fhandle;
};
```

Additionally if the stream supports random access, it should override `seekTellSupported()` to return true and must then implement `seek()` and `tell()`.

Prior to version 2.3, buffering was built into every stream interface. From 2.3 onwards buffering is done in a proxy class which can buffer any underlying writer. This class (`hkBufferedStreamWriter`) can handle alignment and block size issues allowing the underlying classes to be considerably simplified.

In the above example, stdio handles buffering. We can easily add buffering if required by wrapping a writer in a `hkBufferedStreamWriter`. For example:

```
hkStreamWriter* openWriter(const char* filename)
{
    hkStreamWriter* custom = new MyCustomWriter(fileName);
    hkStreamWriter* buffered = new hkBufferedStreamWriter(custom);
    custom->removeReference(); // buffered now owns the underlying custom writer
    return buffered;
}
```

#### 1.1.6.5 hkStreamReader

The `hkStreamReader` interface parallels the writer interface with the mandatory methods

- `int read(void* buf, int nbytes) // read into buffer buf`
- `bool isOk() const // has the stream encountered an error?`

As before, random access may be implemented if desired.

The `hkStreamReader` interface has one additional aspect worth mentioning, namely the mark/rewind interface. When parsing, it is common to read a few characters and then put some back. This can be done very efficiently through the mark/rewind interface. Normally users will not need to implement this interface - it is much simpler to wrap the reader in an `hkBufferedStreamReader` instead. This class also handles block alignment and size issues.

For example a minimal reader:

```
class MyCustomReader : public hkStreamReader
{
    public:

        MyCustomReader(const char* fname)
        {
            m_fd = myOpen(fname, MY_RDONLY);
        }

        virtual ~MyCustomReader()
        {
            if (m_fd >= 0) { myClose(m_fd); }
        }

        virtual int read( void* buf, int nbytes )
        {
            if( m_fd != -1 && nbytes > 0 )
            {
                int nread = myRead(m_fd, buf, nbytes);
                if( nread > 0 )
                {
                    return nread;
                }
                myClose(m_fd);
                m_fd = -1;
            }
            return 0;
        }

        virtual hkBool isOk() const
        {
            return m_fd >= 0;
        }

        int m_fd;
};
```

This simple reader does not handle buffering nor the mark/rewind interface which is used by xml parsing for instance. We can support the mark/rewind interface and likely improve read performance by wrapping it in a buffered stream thus:

```
hkStreamReader* openReader(const char* filename)
{
    hkStreamReader* custom = new MyCustomReader(fileName);
    hkStreamReader* buffered = new hkBufferedStreamReader(custom);
    custom->removeReference(); // buffered now owns the underlying custom reader
    return buffered; // buffered now handles all buffering and mark/rewind calls
                     // and calls custom->read as necessary
}
```

#### 1.1.6.6 Creating Stream Readers and Writers

Often you won't directly create stream readers and writers as `hkIstream`, `hkOstream` have constructors for most common cases (filename, memory buffer and explicitly created readers/writers). However, some low level interface require raw streams. You can use the high level formatted stream and archive classes as a shortcut for creating the most common types of raw stream readers and writers. For example:

```
hkOstream ostreamToFile("filename.txt");
hkStreamWriter* writer = ostreamToFile.getStreamWriter();
// writer is owned by ostreamToFile
// or we can take ownership by adding a reference.

char myMemBuffer[1000];
hkIstream istreamFromMemory(myMemBuffer, sizeof(myMemBuffer));
hkStreamReader* reader = istreamFromMemory.getStreamReader();
// again reader is owned by the hkIstream and we can
// take ownership by adding a reference.
```

Havok uses the `hkFileSystem` singleton to create streams which are connected to files. You can create your own `hkFileSystem` class and replace the default using the method described in the singletons section. This is useful if you want to specify that your own custom stream should be used instead of the default for your platform. This is described in the following section.

**Note:**
The default factory wraps a buffer around streams to improve performance and also to ensure that the mark/rewind interface is always available. See the examples in `hkStreamReader` and `hkStreamWriter` sections.

#### 1.1.6.7 Integrating Custom Streams

If you do not want to use one of the provided implementations, you can create your own custom stream readers and/or writers. To ensure that your custom class is instantiated and used by the Havok libraries you must do the following:

- create a factory to instantiate this type of stream objects instead of the default and
- register this factory with the base system, overwriting the default factory

The factory will look like this:

```
class MyCustomFileSystem : public hkFileSystem
{
  public:

    virtual hkStreamWriter* openWriter(const char* name)
    {
      return new MyCustomStreamWriter(name);
    }

    virtual hkStreamReader* openReader(const char* name)
    {
      return new MyCustomStreamReader(name);
    }
};
```

Once we register this factory, all calls to open a named stream will result in a custom stream being used. Here's how we register the factory just after the base system init() call:

```
// create the custom streambuf factory
hkFileSystem::getInstance().replaceInstance( new MyCustomFileSystem() );
```

Now all of Havok's filesystem operations are passed through the custom implementation.

### 1.1.7 Microsoft Visual Studio Debugger Expansions

If you are using Microsoft Visual Studio with Havok, you can edit its `autoexp.dat` file to make Havok objects display in a more useful way in the Visual Studio debug window. For example, you can specify that you want a `hkVector4`'s x, y, z, and w components to display on the top line, avoiding the need to click through several expansions.

Adding the following to the file will affect the display of `hkVector4s`, `hkQuaternions`, `hkBaseObjects`, `hkArrays`, and `hkStrings`.

```
hkVector3=<m_quad.m128_f32[0],g>, <m_quad.m128_f32[1],g>, <m_quad.m128_f32[2],g>,
        <m_quad.m128_f32[3],g>
hkQuaternion=<m_vec.m_quad.m128_f32[0],g>, <m_vec.m_quad.m128_f32[1],g>,
           <m_vec.m_quad.m128_f32[2],g>, (<m_vec.m_quad.m128_f32[3],g>)
hkBaseObject=<,t>
hkArray<*>=<,t> <m_size> / <m_capacityAndDeallocateFlag&0x7fffffff>
hkString=<,t> <m_string,s>
```

## 1.2 Memory Management

Havok's memory management system is designed to maximize performance based on how Havok uses memory, as well to provide flexibility in order to fit into the memory requirements of your application. This chapter describes the components of the memory system, how to customize the system to your needs, and how to get detailed reports on memory use at any time.

### 1.2.1 Overview

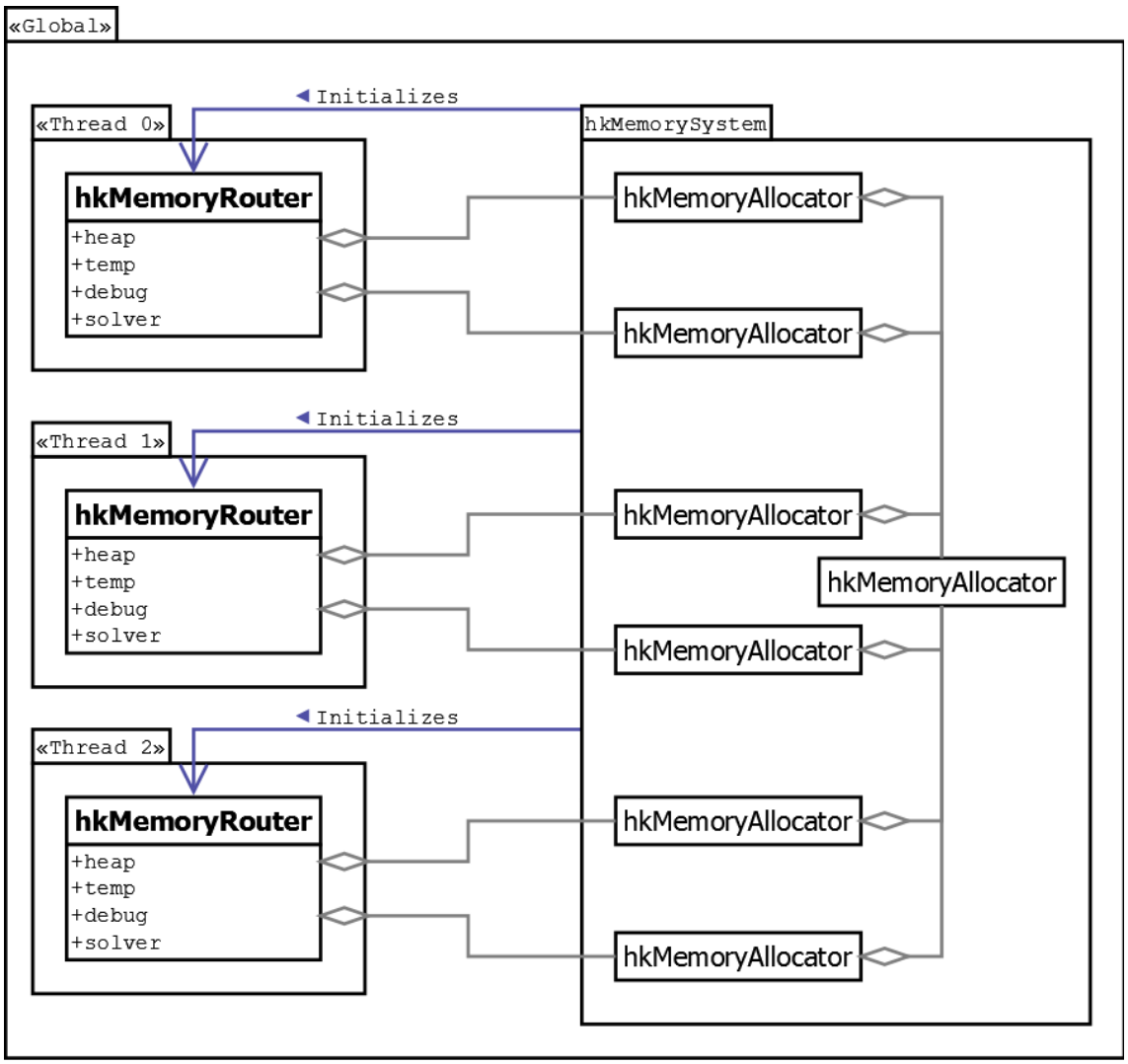
There are three major components in the Havok memory system:

**hkMemoryAllocator:** The lowest level interface through which blocks can be allocated and freed

**hkMemoryRouter:** A per-thread collection of `hkMemoryAllocator`s, e.g. it contains allocators for regular heap, temporary allocations and debug allocations.

**hkMemorySystem:** High level interface to the system as a whole, e.g. statistics, setting up a new threads memory router. Generally contains all the allocators for all threads.

The following diagram shows their relationships: the memory system owns and arranges the plumbing of the allocators; the memory routers point to allocators and only see a small part of the system; and the allocators themselves have an even more restricted overall view.




Each of these components is discussed in more detail in the following sections.

#### 1.2.1.1 hkMemoryRouter Class

`hkMemoryRouter` is a facade object through which all allocations and deallocations are routed. It is little more than a structure containing pointers to allocators for each allocation pattern. The class is nonvirtual–its behaviour is customized by replacing the allocator pointers:

**heap:** Allocations which will typically last longer than a single frame, for example rigid bodies, constraints, the world and their contained arrays. May be freed by any thread and have no particular lifetime.

**temp:** Temporary allocation. Lifetime is restricted to the current scope, thus the same thread allocates and frees. Very often but not always in a LIFO pattern.

**debug:** Allocator used by profiling and checking code so as not to change the main memory profile. For example monitor streams, memory checking utilities and debug visualizers all use this allocator.

**solver:** Special allocator used exclusively by the Physics 2012 constraint solver. Requests are only made from the constraint solver and their lifetime is temporary. Thus solver memory may be freed outside of the physics step. Requests are often for large contiguous blocks so this has been separated from the temp allocator and will likely be shared between all threads unlike the temp allocator.

**stack:** Allocator for pure last-in-first-out allocations, essentially a restricted form of the temp allocator. On most platforms with the default setup, this *is* the temp allocator. (On the PlayStation®3 SPU, this is not actually a virtual allocator class but is a simple pointer bump and all calls are inlined.)

Each thread has its own router instance, though usually many of the objects behind the facade are shared across multiple threads.

#### 1.2.1.2 hkMemoryAllocator Interface

`hkMemoryAllocator` is the basic interface for allocations, providing the pure virtual methods `blockAlloc` and `blockFree` to allocate and deallocate blocks of memory. Note that the size must be explicitly given to `blockFree` which reduces per-allocation overhead.

**Note:**
By default, Havok expects allocated memory to be 16-byte aligned. You will need to take this into account if you override the memory allocation methods.

**Buffer allocations**

The `bufAlloc`, `bufFree` and `bufRealloc` methods may be overridden to optimize handling of resizable blocks. They are used where the caller cares that bigger than requested block size may be returned by the allocator and also to give the allocator opportunity to resize an allocation without copying. `hkArray` is the major user of this interface. An implementation may take advantage of this fact by trying to leave space after buffer allocations so that a future reallocation will succeed without moving the buffer. The default implementation of the buffer allocator implemented using the block allocator.

**Optimizing for space**

Memory managers often store "magic" information, such as size, just in front of the allocated memory. On modern machines the overhead for this may be up to 64 bytes because of alignment issues. Havok's memory manager tries to save this space using two techniques:

- Many simple classes (e.g. `hkAabb`) know their size, and so the memory allocator doesn't need to explicitly store it; the class knows how much space to free when it is deleted. These classes have their sizes compiled directly into their overridden operator new and delete operators. See `HK_DECLARE_NONVIRTUAL_CLASS_ALLOCATOR`.
- For classes derived from `hkReferencedObject`, the allocated size is stored directly in the instance. See *m_memSizeAndFlags* in `HK_DECLARE_CLASS_ALLOCATOR`.

**Allocator utilities**

The default allocators all supply 16 byte aligned allocations and require the size to be given for deallocating. `hkMemoryRouter` has the utility methods `easyAlloc` and `alignedAlloc` to allocate blocks with greater alignment and to stash the allocation size so it need not be supplied. They work by allocating a little more than requested and storing some bookkeeping information before the returned pointer, so the corresponding utility free must be used.

**Marking unused memory**

Some memory allocator implementations overwrite freed/uninitialized memory with known values to aid in debugging. `hkFreeListAllocator` overwrites freed memory and newly allocated memory with `0x7ffa110c` when `HK_DEBUG` is defined. The allocator used in `hkCheckingMemorySystem` always overwrites its memory.

#### 1.2.1.3 hkMemorySystem Interface

The `hkMemorySystem` is for operations on the memory system as a whole. For example it has methods for getting memory statistics and for setting up and tearing down the memory system.

**Initialization**

The memory system used by Havok must be initialized before the `hkBaseSystem` is initialized.

To initialize a memory system, two tasks need to be performed:

- Set a global memory system object for global methods on the memory system
- Initialize each of the allocators in a `hkMemoryRouter` instance and return it

Since all knowledge of the memory setup is encapsulated in the memory system class, each thread must ask the system to set up its memory router appropriately. One thread is must be designated the "main" thread–not necessarily the main system thread, just the first Havok aware thread to be initialized and the last to be deinitialized. The relevant methods on `hkMemorySystem` are `mainInit` and `mainQuit`. (Note that `hkMemoryInitUtil` calls these methods for you.)

After the main thread has been set up, each worker thread should call `threadInit` and `threadQuit`. These methods are implicitly called from the `mainInit` and `mainQuit` so should not be called in the main thread.

**Garbage Collection**

Many allocators cache their freed memory to satisfy future requests and do not release it to their parents unless explicitly requested. The memory system has several methods to release such cached memory. Because of the lockless nature of the thread local allocators, normally each thread needs to release its own cache back to the shared pool and a later call releases the shared cache back to the higher level allocator. This must be done manually by calling the memory system's `garbageCollect` method (see also `garbageCollectThread` and `garbageCollectShared`). This method will cause the allocator to optimize memory in space, potentially making more, and larger chunks of memory available. Doing a garbage collection takes some work, so it is not recommended that it is called every frame. It is better to call it in between levels, or at a point where you know your memory usage is going to change significantly.

Note that doing a garbage collection cannot make available all 'free' memory available; it only frees a block of memory if all of the contained blocks are free. Therefore if you allocated all of the memory with 128 byte chunks and then freed every other chunk, performing a 256 byte allocation will still fail, as no pool will have been freed, and moreover memory is now fragmented such that there is no contiguous block of 256 bytes.

When `hkFreeListAllocator` cannot allocate it will automatically perform a garbage collection and attempt to allocate again. It is still recommended that you perform a garbage collect independently of this mechanism, as otherwise the automatic garbage collection could happen at a time critical section of your application.

Note that calling `threadQuit` implicitly releases cached memory. Also `hkFreeListMemorySystem` will release cached memory when the temporary parts are released.

### Reusing Temporary Memory

Some memory is only needed during specific intervals of execution. For instance the temp allocators, solver allocators and all thread stacks are completely empty outside of Havok SDK calls and could be reused while the Havok SDK is not being used. Most calls will require both a stack and a temp allocator, but the solver allocator is only ever accessed from the Physics 2012 constraint solver, so can safely be reused when the world is not being stepped.

To support this, memory systems may support partially tearing down a threads resources. The following example shows a thread freeing its temporary parts while sleeping between workloads. The main thread would have a similar `mainInit` and `mainQuit` pair which would free any shared resources.

```
hkMemorySystem& memSystem = hkMemorySystem::getInstance();
hkMemoryRouter memRouter;
memSystem.threadInit( memRouter, "worker", hkMemorySystem::FLAG_PERSISTENT );
hkBaseSystem::initThread(&memRouter);

while( waitForStartSignal() == HK_SUCCESS )
{
    memSystem.threadInit( memRouter, "worker", hkMemorySystem::FLAG_TEMPORARY );
    doWork();
    memSystem.threadQuit( memRouter, hkMemorySystem::FLAG_TEMPORARY );
    sendWorkDoneSignal();
}

hkBaseSystem::quitThread();
memSystem.threadQuit( memRouter, hkMemorySystem::FLAG_PERSISTENT );
```

### Memory Statistics

All memory systems support getting a summary of all memory allocated and in use by the memory system, via the `getHeapStatistics` method. This contains information for each allocator in the router. Values which are not applicable for a given memory system are set to -1.

Some memory systems also support getting snapshots to gather detailed data about the memory that is currently allocated by the memory system, via the `getMemorySnapshot` method. These snapshots contains the following information about each allocation, and can be used to give detailed reports on memory use:

- A pointer to the allocation.
- The size (in bytes) of the allocation.
- A trace of the call stack when the allocation was made.

Please see the Memory Reporting section for more information on working with memory statistics.

#### 1.2.1.4 Overriding Class New and Delete

Havok uses several macros to declare class local memory management. These ensure that all Havok objects are allocated by the appropriate allocators in the Havok memory system.

The `HK_DECLARE_CLASS_ALLOCATOR` macro overrides class new and delete so that classes derived from `hkReferencedObject` are automatically handled correctly by the memory system. Note that the memory manager expects to be able to find a `hkReferencedObject` at the object address, so when using multiple inheritance the `hkReferencedObject` should come first in the inheritance list.

Unfortunately, non virtual classes cannot inherit from a memory management class, since some compilers do not implement the empty base object optimization. For each of these classes we require them to use the `HK_DECLARE_NONVIRTUAL_CLASS_ALLOCATOR` macro. The memory class parameter is the same as in the virtual case. The name of the class must also be passed to get its size information and is compiled into their operator new and delete.

Some classes can only usefully provide placement operators new and delete (most notably the memory classes themselves). These classes use `HK_DECLARE_PLACEMENT_ALLOCATOR()` and must be allocated statically or via another memory system and constructed in-place.

### 1.2.2 Default Implementations

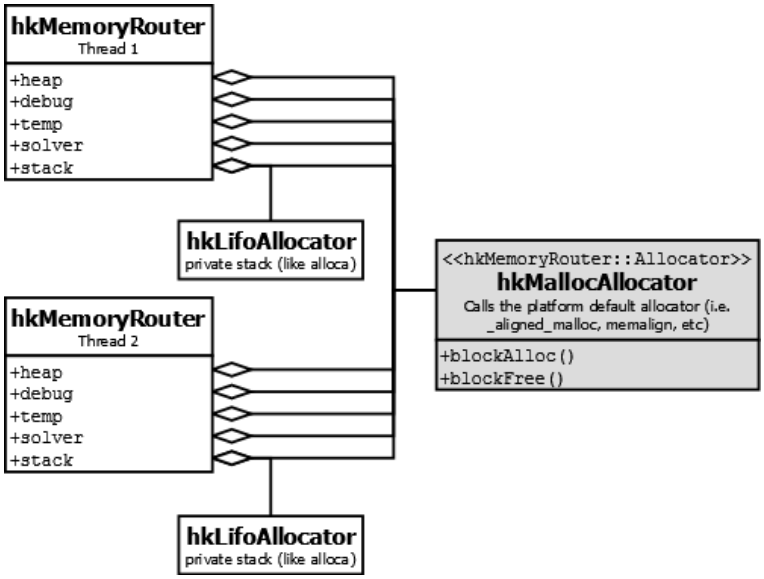
#### 1.2.2.1 Allocators

There are several allocators available in the Havok SDK (`Common\Base\Memory\Allocator`). The more important of these are described in the following sections.

**hkThreadMemory**

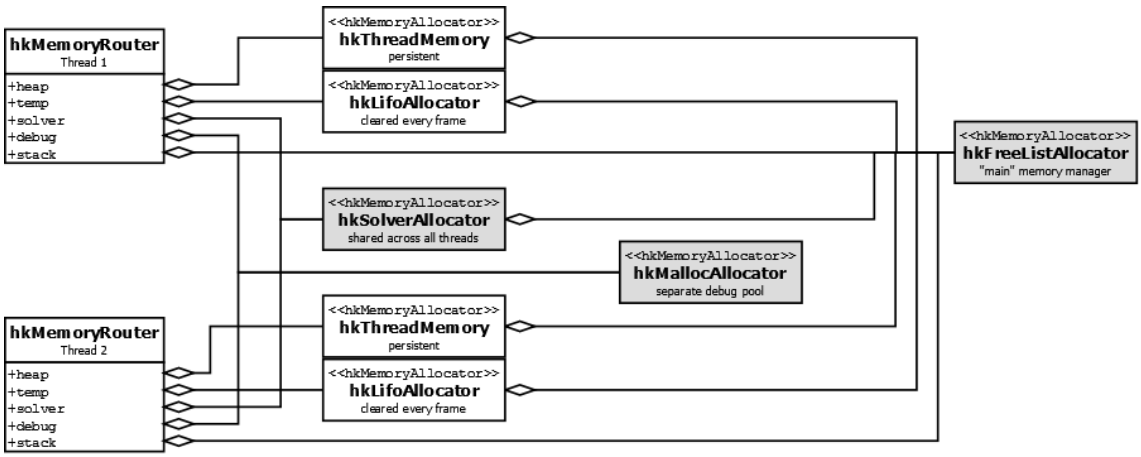
Havok's memory managers are thread-safe. For this reason they might pose a serious performance bottleneck when used intensively by multiple threads. For example, the following diagram shows the

simplest possible setup: all threads using the platform's default allocator and with their own small private stack. This setup is very inefficient, as every allocation will require a system call and must be locked for thread safety.




To overcome this, the default memory system (`hkFreeListMemorySystem`) gives each thread used by Havok its own instance of `hkThreadMemory`. `hkThreadMemory` has the ability to cache memory blocks freed by a thread and use them for subsequent memory allocations without the need to reference a global memory manager. This reduces lock contention in the main memory allocator. It is faster too as most requests won't need any locking whatsoever.

The following diagram demonstrates this, where each thread has a `hkThreadMemory` instance that points to a shared higher level allocator. There is also a second per-thread `hkLifoAllocator` for frame local allocations which reduces fragmentations and which may be completely purged between SDK calls. The buffer underlying the solver allocator may also be freed between solver calls. Classes which must implement locking are shown with a shaded background.




Note that:

- All heap allocations per thread are handled by the `hkThreadMemory` class.
- All allocations are shared between the threads. This means a block which is allocated by one `hkThreadMemory` can be freed by another.

In addition:

- `hkThreadMemory` is highly optimized to handle memory for a single thread. It does so by caching a limited number of blocks in a thread-local free list. There are free lists for each different block size.
- Each free list (for each size) can hold only a maximum number of items (`m_maxNumElemsOnFreeList`).
- If `maxNumElemsOnFreeList == 0` then the free list is not checked at all, and the `hkThreadMemory` simply forwards all requests.
- If, for a given size, there is no free block on the free list, a block is fetched by calling `blockAllocBatch()` on the higher level allocator.
- If the number of elements on a particular free list exceeds `m_maxNumElemsOnFreeList`, some blocks are forwarded to the higher level allocator using `blockFreeBatch()`.

**hkLifoAllocator**

`hkLifoAllocator` is designed for fast temporary allocations. Memory allocated with `hkLifoAllocator` should be preferably be deallocated in reverse order, although the allocator can handle a few out of order deallocations. When used in a LIFO fashion, `hkLifoAllocator` performs comparably to a stack based allocator, provided the inlined versions for its allocation functions are used. Note that the performance degrades linearly with the number of out-of -order deallocations performed.

`hkLifoAllocator` needs to be initialized by providing pointers to three additional allocators: a *"slab" allocator*, a *"large" allocator*, and an allocator to be used for its internal bookkeeping. The slab allocator is used to perform on-demand allocations of large fixed-sized blocks (16 KB by default) where the actual allocations will occur. The large allocator is used when `hkLifoAllocator` is asked to allocate a block larger than the slab size. By default, the solver allocator is used for both slab and large allocations. Since allocations handled by `hkLifoAllocator` are thread-local and typically short-lived, this is unlikely to interfere with the ordinary solver allocations. Note that the first slab is handled specially, and not allocated using the solver memory.

**hkFreeListAllocator**

The `hkFreeListAllocator` is designed to provide fast and efficient allocation of small allocations. For allocations less than or equal to 640 bytes, the `hkFreeListAllocator` uses a 'freelist' allocation strategy from which it is named. For each block size, there is a freelist containing blocks "owned" by Havok but available for use. If an allocation is requested, and a suitable free element is available on the appropriate freelist, the free block is returned to the caller. If there isn't an available block, a new allocation is made to the underlying memory system for a large block of memory which can hold many smaller blocks; these blocks are then added to the freelist and available for use. When an allocation is freed, the block is added back to the freelist, allowing again for a quick allocation (of similar size) at a later time.

For large allocations (greater than 640 bytes) the `hkFreeListAllocator` uses the underlying allocator passed to the `hkFreeListAllocator` constructor.

It is sometimes useful to be able to control how these freelists are internally constructed. The last parameter of the `hkFreeListAllocator` constructor provides this control.

There are two handy methods that can be used to set up the `hkFreeListAllocator::Cinfo` simply. `hkMemoryAllocator::setDefaultCinfo` sets up the freelists in the same style previous versions of Havok have used and is the default style used if the info parameter is passed as `HK_NULL`. `hkMemoryAllocator::setFixedSizeCinfo` sets up the freelists so they all use the same underlying block size. This can be advantageous if the underlying allocator is designed to work with a limited block sizes.

Below is an example of setting up a `hkFreeListAllocator` to use blocks of the same size to the underlying allocator. Note that allocations larger than 640 bytes will still have to be serviced by the underlying allocator. Also note that if the `hkMemoryAllocator::ExtendedInterface` parameter to the `hkFreeListAllocator` is passed as `HK_NULL` (as is here), the allocator will still work, but memory limiting and walking will not work. Not all implementations of hkMemoryAllocator also include an implementation of `hkMemoryAllocator::ExtendedInterface`.

```
// Forwards to the platform malloc
hkMallocAllocator systemMalloc;

// Lets go with 8k underlying blocks
hkFreeListAllocator::Cinfo info;
hkFreeListAllocator::setFixedSizeCinfo( 1024 * 8, info);

// Set up the allocator
// NOTE! That since the hkMemoryAllocator::ExtendedInterface is not set - memoryLimiting and walking will
  not work
hkFreeListAllocator allocator(&systemMalloc, HK_NULL, &info);
```

### hkLargeBlockAllocator

The `hkLargeBlockAllocator` is designed to handle larger blocks than the `hkFreeListAllocator`, and blocks of a wide variety of sizes with minimal overhead. The overhead of an allocation from the large block allocator is 16 bytes per allocation; however, with this informational overhead the large block allocator can fit allocations tightly together and fuse neighboring free blocks, limiting wasted space otherwise caused by fragmentation.

The `hkLargeBlockAllocator` can receive its memory supply from either an underlying `hkMemoryAllocator`, or from a single fixed block of memory.

Using `hkLargeBlockAllocator` with a pre-allocated block of memory:

```
// Memory block where all memory allocations will be located.
char memoryBlock[1024 * 1024];

// Large block allocator using a single block of memory for all allocations
hkLargeBlockAllocator allocator(memoryBlock, sizeof(memoryBlock));
```

Using `hkLargeBlockAllocator` with malloc:

```
// Forwards to the platform malloc
hkMallocAllocator systemMalloc;

// Large block allocator using the systemMalloc for all underlying allocations
hkLargeBlockAllocator allocator(&systemMalloc);
```

Since the `hkLargeBlockAllocator` is designed for larger blocks it is not very efficient with smaller blocks, and is unable to allocate blocks smaller than 256 bytes. This means that it is not ideally suited as the 'main' allocator Havok uses, but it is ideally suited as the allocator used to service allocations from the `hkFreeListAllocator`. Doing so allows small allocations to be handled rapidly and efficiently by the freelist mechanism, and larger blocks of any size to be handled without undue overhead by the `hkLargeBlockAllocator`. This arrangement also allows for a configuration where an `hkFreeListAllocator` allocates all memory from a single contiguous block.

Setting up a `hkFreeListAllocator` so that it uses a `hkLargeBlockAllocator` for its underlying allocations:

```
// Forwards to the platform malloc
hkMallocAllocator systemMalloc;

// Large block allocator using the systemMalloc for all underlying allocations
hkLargeBlockAllocator largeBlockAllocator(&systemMalloc);

// Lets go with 4k underlying blocks
hkFreeListAllocator::Cinfo info;
hkFreeListAllocator::setFixedSizeCinfo( 1024 * 4, info);

// Set up the freelist allocator
// Note that hkLargeBlockAllocator implementes hkMemoryAllocator::ExtendedInterface, so it is passed as
  the second
// parameter - meaning memory limiting and walking will work in this arrangement.
hkFreeListAllocator freeListAllocator(&largeBlockAllocator, &largeBlockAllocator, &info);
```

### hkPooledAllocator

`hkPooledAllocator` is a special allocator for small, temporary allocations. It is chained with another allocator, and allocates large blocks (16 kB by default) from this allocator. It then satisfies allocations very efficiently from within these blocks. Freed memory is not available until the `hkPooledAllocator` is destroyed, and all of the memory blocks are released. This allows a large number of small allocations to be satisfied efficiently without the overhead of a more complex allocator, however there will be a memory penalty if the allocations are long-lived.

### hkSpuStack

`hkSpuStack` is not an actual allocator, but a simple wrapper around a fixed-size buffer for *stackframe scope* allocations. That is, deallocations must always occur in reverse order to allocations.

`hkSpuStack` is only used in the PlayStation®3 SPU, and all its allocations are 128-byte aligned.

#### 1.2.2.2 Systems

There are several memory systems available in the Havok SDK (`Common\Base\Memory\System`). The more important of these are described in the following sections.

**The default "Free List" Memory System**

`hkFreeListMemorySystem` is designed to avoid memory fragmentation and is fairly complex, composed of multiple layers. The first layer is a `hkFreeListAllocator`. Allocation requests from `hkFreeListAllocator` check an internal structure of free lists, and potentially make an allocation from the next layer, a `hkLargeBlockAllocator`. Finally, `hkLargeBlockAllocator` gets its memory from a user defined allocator or from a fixed memory block. Please see the previous sections for more detail on each of these allocators.

This is the recommended memory system for most use cases.

**The "Checking" Memory System**

`hkCheckingMemorySystem` is a special memory system implementation that provides much more strict error checking than the default `hkFreeListMemorySystem`. In addition, on shutdown it prints the context of any memory leaks. Memory leak tracking is dependent on stack tracing facilities being available. See the stack tracing section. To use this:

- Ensure you are compiling in *debug* configuration. Please note that *dev* configuration is an optimized build with line numbers enabled. Stack tracing will not work in *dev* configuration as defined by Havok.
- Pass an instance of `hkCheckingMemorySystem` into `hkBaseSystem::init(...)`.
- Ensure that frame pointer optimisations are turned off, as they can mangle the call stack

If there are *no* memory leaks, you should see the following printed out when the allocator is shut down:

```
*************************************************************
--------------- NO HAVOK MEMORY LEAKS FOUND ------------------
*************************************************************
```

However if there are any memory leaks, the report will look like this:

```
*************************************************************
                BEGIN MEMORY LEAK REPORT
*************************************************************
80 bytes leaked. (chunk==1)
../source2\hkcollide/shape/hkpShape.h:24:hkpShape::operator new
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi.cpp:54:TriangleShapeApi::TriangleShapeApi
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi.cpp:109:TriangleShapeApi::TriangleShapeApi
G:\demos\common\menu\MenuGame?.cpp:438:MenuGame::startCurrentGame
G:\demos\common\menu\MenuGame?.cpp:599:MenuGame::stepMenuGame
G:\demos\common\menu\MenuGame?.cpp:142:MenuGame::stepGame
G:\framework\hkdemoframework\hkDemoFramework.cpp:475:hkFrameworkMain
G:\demos\main.cpp:46:main
crt0.c:206:mainCRTStartup
(null):0:ProcessIdToSessionId
-------------------------------------------------------------------
132 bytes leaked. (chunk==1)
../source2\hkdynamics/entity/hkpEntity.h:43:hkpEntity::operator new
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi.cpp:93:TriangleShapeApi::TriangleShapeApi
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi.cpp:109:TriangleShapeApi::TriangleShapeApi
G:\demos\common\menu\MenuGame?.cpp:438:MenuGame::startCurrentGame
G:\demos\common\menu\MenuGame?.cpp:599:MenuGame::stepMenuGame
G:\demos\common\menu\MenuGame?.cpp:142:MenuGame::stepGame
G:\framework\hkdemoframework\hkDemoFramework.cpp:475:hkFrameworkMain
G:\demos\main.cpp:46:main
crt0.c:206:mainCRTStartup
(null):0:ProcessIdToSessionId
-------------------------------------------------------------------
240 bytes leaked. (chunk==1)
../source\hkbase/baseobject/hkBaseObject.h:17:hkBaseObject::operator new
G:\source2\hkdynamics\entity\hkpRigidBody.cpp:48:hkpRigidBody::hkpRigidBody
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi.cpp:93:TriangleShapeApi::TriangleShapeApi
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi.cpp:109:TriangleShapeApi::TriangleShapeApi
G:\demos\common\menu\MenuGame?.cpp:438:MenuGame::startCurrentGame
G:\demos\common\menu\MenuGame?.cpp:599:MenuGame::stepMenuGame
G:\demos\common\menu\MenuGame?.cpp:142:MenuGame::stepGame
G:\framework\hkdemoframework\hkDemoFramework.cpp:475:hkFrameworkMain
G:\demos\main.cpp:46:main
crt0.c:206:mainCRTStartup
(null):0:ProcessIdToSessionId
-------------------------------------------------------------------
96 bytes leaked. (chunk==1)
..\hkdynamics/entity/hkpEntityDeactivator.h:22:hkpEntityDeactivator::operator new
G:\source2\hkdynamics\entity\hkpRigidBody.cpp:101:hkpRigidBody::hkpRigidBody
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi.cpp:93:TriangleShapeApi::TriangleShapeApi
G:\demos\api\shapesapi\triangleshapeapi\TriangleShapeApi.cpp:109:TriangleShapeApi::TriangleShapeApi
G:\demos\common\menu\MenuGame?.cpp:438:MenuGame::startCurrentGame
G:\demos\common\menu\MenuGame?.cpp:599:MenuGame::stepMenuGame
G:\demos\common\menu\MenuGame?.cpp:142:MenuGame::stepGame
G:\framework\hkdemoframework\hkDemoFramework.cpp:475:hkFrameworkMain
G:\demos\main.cpp:46:main
crt0.c:206:mainCRTStartup
(null):0:ProcessIdToSessionId
-------------------------------------------------------------------
*************************************************************
                  END MEMORY LEAK REPORT
*************************************************************
```

The report here contains 4 leaks, with a call stack of each, detailing all calls resulting in the allocation causing the leak. In this demo, a reference was not removed from a rigid body, so it was never deleted. In addition, the shape and entity deactivator owned by the rigid body were also not deleted.

You can see the type of object which leaked by looking at the first line of each leak block. The interesting bit is the following line which gives the file and line number of the call to new the object, which should

give you an indication of what class was supposed to "own" the object, and hence forgot to either delete it later, or remove a reference to it.

Note that some leaks obscure others, or are really caused by the same leak. It's up to you to find out the real cause, which is generally the "largest" object not deleted. In this case, one line (`rb->removeReference()`) fixes 5 leaks, but this may not be obvious from the debug output. You'll need to fix, re-run, fix, re-run etc. until you get the "`NO HAVOK MEMORY LEAKS FOUND`" all-clear.

The Checking Memory System also allows you to get information about its current allocations. See the section on Memory Snapshots for more information.

**Note:**
You can use the checking memory system in the Havok demos with command line option '*-c*'. Please see the quick start guide for details.

#### 1.2.2.3 hkMemoryInitUtil

This utility provides methods to easily create some standard memory systems. It exists mainly for brevity and its use is optional. When getting started with Havok it is common to use one of the sample memory systems from `hkMemoryInitUtil`.

Each memory system initialization method takes two parameters: the base allocator to use and a `hkMemorySystem::FrameInfo` specifying the size of the Physics 2012 solver buffer. If you are not using Physics 2012, this should be set to zero. For simple systems, the base allocator should usually be set to `hkMallocAllocator::m_defaultMallocAllocator`, a statically allocated allocator that passes all allocations to the system malloc implementation. Systems with more complex memory requirements may require a custom allocator.

If you need more flexibility than the interface allows, your choice of `hkMemorySystem` may of course be instantiated directly. Note that in any case the currently active memory system is accessible via `hkMemorySystem::getInstance()`.

### 1.2.3 Customization

#### 1.2.3.1 Redirecting all base allocations

All the supplied memory systems eventually allocate memory from one base allocator. The simplest customization is to replace this base allocator. All memory systems have methods to set the base allocator or if you're using `hkMemoryInitUtil`, you may pass in `hkMallocAllocator::m_defaultMallocAllocator`. This is a statically allocated `hkMallocAllocator` that forwards allocations to the system allocator. Here we show a minimal suitable allocator.

```
static void* custom_alloc( int numBytes ) { return HK_NULL; }
static void custom_free( void*, int numBytes ) { }

class HK_EXPORT_COMMON CustomAllocator : public hkMemoryAllocator
{
    public:

        virtual void* blockAlloc( int numBytes ) HK_OVERRIDE { return custom_alloc(numBytes); }
        virtual void blockFree( void* p, int numBytes ) HK_OVERRIDE { return custom_free(p, numBytes); }
        virtual void getMemoryStatistics( MemoryStatistics& u ) const HK_OVERRIDE { }
        virtual int getAllocatedSize(const void* obj, int numBytes) const HK_OVERRIDE { return numBytes; }
};
```

#### 1.2.3.2 Replacing the Memory System

The following listing shows the bare bones of what is required to create a memory system which uses a single thread safe allocator, as done by the `hkSimpleMemorySystem`.

Each of the init and quit methods takes an optional flag which indicates which part of the system to operate on. The frame local allocators (stack, temp and solver) may be completely purged each frame and reallocated next frame without disturbing the persistent allocators (heap and debug).

```
hkMemoryRouter* hkSimpleMemorySystem::mainInit(const FrameInfo& info, Flags flags)
{
    if( flags.get(FLAG_PERSISTENT) )
    {
        // nothing extra to do
    }
    if( flags.get(FLAG_TEMPORARY) )
    {
        // set up solver buffer
    }
    // do thread local setup too
    threadInit(m_mainRouter, "main", flags);
    return &m_mainRouter;
}

hkResult hkSimpleMemorySystem::mainQuit(Flags flags)
{
    threadQuit(m_mainRouter, flags);
    if( flags.get(FLAG_TEMPORARY) )
    {
        // free solver buffer
    }
    if( flags.get(FLAG_PERSISTENT) )
    {
        // nothing
    }
    return HK_SUCCESS;
}

void hkSimpleMemorySystem::threadInit(hkMemoryRouter& router, const char* name, Flags flags)
{
    if( flags.get(FLAG_PERSISTENT) )
    {
        // setup heap, debug
        // all others set to null
    }
    if( flags.get(FLAG_TEMPORARY) )
    {
        // set up stack, temp and solver
    }
}

void hkSimpleMemorySystem::threadQuit(hkMemoryRouter& router, Flags flags)
{
    if( flags.get(FLAG_TEMPORARY) )
    {
        // quit stack, temp and solver
    }
    if( flags.get(FLAG_PERSISTENT) )
    {
        // quit heap, debug
    }
}
```

`hkMemoryInitUtil` has more example setups. You'll notice that many of the memory classes are written with almost empty constructors/destructors and have explicit init/quit methods. This is to facilitate using them as static variables.

### Replacing only the heap allocator

If the goal is just to replace the heap allocator rather than replace the whole memory system, there is a simpler way of doing this by just implementing a `hkMemoryAllocator` for the allocator you wish to use. The `hkMemoryInitUtil` has a method `initHeapAllocator()` which will create a create memory router

and system from such an `hkMemoryAllocator`.

The method takes a *memoryAllocator* which is used for allocations other than heap allocations, such as for the solver memory buffer. The `heapInterface` parameter provides an extended interface to the *heapAllocator* parameter. The `hkMemoryAllocator::ExtendedInterface` is needed because the `hkMemorySystem` has methods such as `garbageCollect()` and `memoryWalk()` that the `hkMemoryAllocator` doesn't have. So the *heapInterface* provides a mechanism to provide implementations of such features to your underlying *heapAllocator*. It is not essential to provide an implementation, and the parameter can be passed as `HK_NULL` but in doing so methods such as `garbageCollect()` on the `hkMemorySystem` will do nothing.

Note that you can pass the same allocator to both *memoryAllocator* and *heapAllocator* if it is not important to separate heap allocations from other types of allocations.

#### 1.2.3.3 Tweaking the Physics 2012 Solver Buffer size

The solver buffer is used for allocations during the Physics 2012 solver step. The Havok demos aim to comprehensively test the runtime system, and explicitly set a very large solver buffer size. It is likely that simpler systems will require a much smaller buffer. The buffer size must be set by passing a `hkMemorySystem::FrameInfo` to `hkMemorySystem::mainInit` (either directly or through the `hkMemoryInitUtil`). The `hkMemorySystem::FrameInfo` constructor takes one parameter, the size of the buffer to allocate. For example:

```
hkMemorySystem::FrameInfo finfo(500 * 1024);    // Allocate 500KB of Physics solver buffer
hkMemoryRouter* memoryRouter = hkMemoryInitUtil::initDefault(hkMallocAllocator::m_defaultMallocAllocator,
  finfo);
hkBaseSystem::init( memoryRouter, errorReport );
```

See the solver memory section in the Physics 2012 user guide for the consequences of making the solver buffer too small.

If you are not using Physics 2012, the solver buffer size should be set to zero.

### 1.2.4 Reporting

Havok's memory system provides two approaches to reporting memory usage:

- A quick, high-level heap blocks summary via `hkMemorySystem::getHeapStatistics()`.
- A detailed, per-allocation breakdown via `hkMemorySystem::getMemorySnapshot()`.

The following sections discuss each of these approaches, with emphasis on the latter.

#### 1.2.4.1 Heap Summary

The following snippet will get a summary of the current heap usage:

```
void getHeapStatistics(hkMemoryAllocator::MemoryStatistics& heapStats)
{
    hkMemorySystem::MemoryStatistics stats;
    hkMemorySystem::getInstance().getMemoryStatistics(stats);
    for (int i = 0 ; i < stats.m_entries.getSize(); ++i)
    {
        if ( hkString::strCmp(stats.m_entries[i].m_allocatorName, "Heap") == 0 )
        {
            heapStats = stats.m_entries[i].m_allocatorStats;
            return;
        }
    }
}
```

`hkMemoryAllocator::MemoryStatistics` values which are not applicable for a given memory system are set to -1. Note that the amount of detail given in the output varies considerably between memory systems. Currently `hkFreeListMemorySystem` and `hkCheckingMemorySystem` provide the most detailed statistics.

Calling the following method will print out the same data:

```
hkOstream ostr(lStr);
hkMemorySystem::getInstance().printStatistics(ostr);
```

The following is an example of output of this function call. This data was taken from the multi-pendulum example in the Havok demo framework. There are 500 rigid bodies and 500 ball and socket constraints in this demo:

```
hkFreeListMemorySystem memory overview:
=======================================

    Allocation totals:

              1827888 allocated by heap
             29274772 allocated by debug
              6291456 allocated by solver
              -------
             37394116 total allocated

    Heap usage:

               132128 ( 7%) unused in thread local freelists
               170080 ( 9%) unused in main heap
              1525680 (83%) used in main heap
              -------
              1827888 allocated by heap

    Peak usage:

              1657808 (92%) peak heap used (versus current heap used)
               154128 ( 2%) peak solver used (versus available)
```

First of all, we are currently using the free list allocation system. Looking at each section in turn we see that the heap allocator has requested 1,827,888 bytes from its parent allocator. We also show the totals for debug and solver. The bulk of these are the monitor (profiling) streams which are quite large

to accommodate some of the larger demos.

Next is a breakdown of the heap usage. We see that 7% is spread across the lockless thread local caches and that 9% is in the shared cache. The remaining 83% is in use. (Percentages are rounded)

Finally we show peak sizes. The current heap usage is at 92% of the peak heap usage.

Because this demo default demo setup is extremely conservative, this demo has a solver buffer which is far too large–only 2% was ever used. Memory can be reclaimed by shrinking the buffer. Note that this buffer is only required during a physics simulation step, and need not be permanently allocated. Be aware that making the buffer too small will have a performance penalty (see the solver memory section in the physics guide).

#### 1.2.4.2 Detailed Statistics: the Memory Tracker system

Havok's Memory Tracker system aims to provide detailed information about memory usage largely automatically. The tracker allows for creating a "snapshot" of memory usage which can then be processed by a variety of reports to provide information about memory usage. The snapshot captures memory allocations, as well as a description of how memory is used. This section describes how the tracker works and how to use it.

**Overview**

The memory tracker uses a variety of techniques to produce information about how memory is used, and how memory blocks relate to one another.

Key to the trackers functionality is that whenever an object is '`new`ed' or '`delete`d', the `hkMemoryTracker` instance will be informed. Specifically when an object is '`new`ed' the tracker is informed of the objects location, size and type. This is generally implemented behind the scenes with the use of the `HK_DECLARE_CLASS_ALLOCATOR` and `HK_DECLARE_NONVIRTUAL_CLASS_ALLOCATOR` macros.

Reflection information for most types is produced automatically by Havok build scripts. This reflection information allows the tracker code to work out how one type relates to another, by knowing the field types.

There needs to be a fall back position if there isn't type or reflection information. This could be because the type is a template and therefore there isn't type reflection, or because the block of memory being examined has no type (for example a chunk that is allocated via `hkAllocateChunk`). Later on we will describe how these cases can be handled using handlers when necessary. If no specific handler is provided, the block will still be considered in snapshots, but no attempt will be performed to discover pointed blocks.

The memory tracker knows about every object allocated on the heap using `new` and `delete` thanks to the Havok allocators; this information is combined in a snapshot with the allocations actually reported by the Havok memory system through `hkMemorySystem::getMemorySnapshot()`. The memory system and the memory tracker are therefore considered two independent sources and their information is combined in the snapshot depending on the report.

Features of memory tracker system:

- Generally no code needs to be written to track memory.

- All memory allocated by a `hkMemorySystem` is tracked.

- Produces a database of memory relationships, which can be used to produce many different types of report.

- Has support extending functionality, for example adding extra code to handle templates, or types that have specialized memory layouts.

Limitations of memory tracker system:

- Ownership of memory can be hard to determine.

- It requires extra reflection information, just for memory tracking.

- There may be limitations on tracking accuracy on heterogeneous processors (such as Cell).

**Getting Started**

Prerequisites for using memory tracking

- You must be using a `hkMemorySystem` that implements `getMemorySnapshot()`.
  - `hkCheckingMemorySystem` and `hkFreeListMemorySystem` do.
  - In Havok demos you can enable the checking memory system by passing the "*–c*" argument on the command line.
- You must have a build compiled with `HK_MEMORY_TRACKER_ENABLE` (currently enabled if `HK_DEBUG` is enabled).
- You must explicitly enable memory tracking by calling `hkMemoryInitUtil::initMemoryTracker`.

Using a `hkMemorySystem` that implements tracking of stack traces of allocations can also be very useful, as it will allow the memory tracker to dump out stack traces of blocks whose ownership cannot be determined.

To try out memory tracking in the Havok demos use the options "–c –me" on the command line. "*–me*" will cause the demo system to output a collection of memory tracker reports once a demo is exited. "*-c*" makes the Havok demos use the `hkCheckingMemorySystem`, which provides stack traces for all allocations as well as implementing `getMemorySnapshot()`, a feature required for memory tracking operation. If the demo uses a lot of memory it can take a fair amount of time (perhaps minutes) to dump out all of the reports.

To incorporate memory tracking into your own project, a good starting point would be to check out `hkMemoryInitUtil` which has two methods `initMemoryTracker` and `quitMemoryTracker`. The tracker should be initialized just after the memory system and before `hkBaseSystem::init`, otherwise some allocations will be missed. Similarly the tracker should be quit after `hkBaseSystem::quit()` and before the memory system shutdown.

With a HK_MEMORY_TRACKER_ENABLE build all new and delete calls to types that use the standard HK_DECLARE_CLASS_ALLOCATOR or HK_DECLARE_NONVIRTUAL_CLASS_ALLOCATOR macros will also inform the singleton hkMemoryTracker of when an object gets 'newed' or 'deleted'.

To examine memory using the memory tracker system, a snapshot of the memory state has to be taken. The easiest way to take a hkTrackerSnapshot is to call hkTrackerSnapshotUtil::createSnapshot(). The hkTrackerSnapshot can be thought of as a kind of database describing how memory is being used at the time of capture. A hkTrackerSnapshot can then be processed to produce reports.

NOTE: Care needs to be taken with threads when a snapshot is taken.

Threads cannot

- Modify memory that the main hkMemorySystem has allocated
- Allocate of deallocate memory on the main hkMemorySystem.
- Make calls to the hkMemoryTracker.
  - Generally through new/delete.

Not adhering to these rules will produce undefined results, including potentially a crash.

There are a range of reports in the SDK in the directory 'Common/Base/Memory/Tracker/Report'.

Read the documentation on each to find out what kinds of information they produce. Many reports take a hkScanReportUtil::FollowFilter* filter parameter. For now just set this parameter to HK_NULL to disable filtering. Most reports write out text to a stream passed as the last parameter. Some reports give the option of writing output to '.csv' files, which are just text files which can be loaded into Microsoft Excel and other spreadsheets.

**Allocations and Blocks**

The memory tracker system separates the idea of memory 'Allocations' and tracker memory 'Blocks'. An allocation is a chunk of memory that has been requested from a memory system. A 'Block' describes how a piece of memory is being used. For example a Block could describe an area of memory inside of an Allocation that was being used to store an array.

A Block can belong to at most one Allocation. Generally Blocks are held in a single allocation, although that is not necessarily the case. For example a Block could describe a piece of memory on the stack. Alternatively an Allocation could hold multiple Blocks. A Block is not allowed to straddle Allocations

Determining Allocations is relatively easy, in that a hkMemorySystem can provide getMemorySnapshot() functionality, which will describe all of the allocations made on that memory system. This is how the hkTrackerSnapshot determines all of the Allocations, and for it to work correctly Havok must have been run with a memory system that implements the function, such as hkCheckingMemorySystem.

Determining Blocks is more complicated and is really a question of determining how memory has been used. The memory tracker system works on objects that are created and destroyed with 'new' and 'delete' as providing critical information in determining memory usage. On a build that has HK_MEMORY_TRACKER_ENABLE defined, the allocator macros insert code such that after an object is 'newed'

or ‘deleted’ the tracker is informed. Key here is that the tracker is informed of the memory location, size and the type of the allocation.

The first set of Blocks then are these allocations and their types. Using reflection information it is now possible to determine other Blocks of memory. For example if you have an object hkFoo and it contains an hkArray<hkBar>. If hkFoo is ‘newed’ the tracker knows this. Through reflection the tracker can also determine that hkFoo contains an hkArray, and from this it can determine there is another Block, which is the block of memory holding the array of hkBar elements.

Note that the tracker system does not directly specify the difference between used and unused memory. If there is a Block then that memory is used, and the Block type will specify how it is used. For every Allocation collected from the memory system we know if it is considered used, unused or overhead, and every Block allocated should always be part of used memory Allocations.

**Uninitialized and ‘Dangling’ Pointers**

Generally the tracker has no implicit knowledge about a type. If reflection says a type contains a pointer, and it is set to something that is an allocation, it will assume they are associated. It may be the case that the pointer is uninitialized, and this is just coincidence. With typical debug memory allocator all allocations are set to some predefined value which will avoid this problem. Another scenario is that the pointer is set to point to an allocation, but now effectively it doesn't any longer. To avoid such problems its recommended (in a memory tracker build at least) to set such pointers to HK_NULL so they are not followed.

**Container Types**

For types that are really ‘containers’, or more specifically do not have a layout that is the same across instantiations, specific handlers should generally be written so that the tracker can find out all references going out of the Blocks.

**Obfuscated Pointers**

For the memory tracker to function correctly generally it needs pointers to be un-obfuscated. Obfuscation can be in the form of storing a pointer in a type which isn't a pointer (such as hkUlong). If it is reflected, it will not work correctly because the reflection does not know that the hkUlong can hold a pointer, and so the member will not be reflected and therefore tracked.

A more difficult type of obfuscation is when a pointer is modified to hold additional information. An example might be knowing that a pointer can only point to a 32-bit boundary and using the bottom 2 bits for storage. The normal tracker operation has no way of knowing that some bits are being used in this way, and so may not track the ‘pointer’ correctly.

Both scenarios can be solved by having handlers (described later) to handle traversal of those specific types.

**Stack Objects**

Objects instantiated on the stack cannot be directly seen by the memory tracker. This is because if they are instantiated on the stack, ‘new’ is not called, and ‘new’ is the hook into the memory tracking system.

If an object allocated on the stack allocates memory (such as a hkArray) that memory will be seen by the tracker in that the getMemorySnapshot() will inform the tracker of all allocations. But because the hkArray definition is not seen it cannot infer what the memory holds.

**Filters and Ownership**

Memory ownership is an important concept in many of the reports. Memory could be reported just as the blocks and their types (if available), but doing so doesn't provide much information on what structures are at the root of the memory consumption.

In order to get around this problem, and to try and group memory allocations together, a simple concept of 'ownership' is introduced. The idea is to allow a block to potentially be owned by another block. Each block can at most be owned by one other block. To give an example image a class `hkFoo` that contains an `hkArray` member. The memory contained in that array could be thought of as being 'owned' by the instance `hkFoo`. If the `hkArray` was of type `hkArray<hkBar*>` things are a little more difficult–`hkBar` objects could perhaps be shared, and so cannot be 'owned' by `hkFoo`.

Determining ownership is generally performed as a breadth first traversal. A block (generally known as a root block) is determined, and then other blocks that can be reached from the root block become owned by the root block if they are not already owned and so forth. All of the blocks in the tree of ownership that is produced can be thought of as ultimately belonging to the root block, and moreover, the tree shows how that ownership was reached.

There are two keys issues to the algorithm for determining ownership

- What blocks are 'roots'
- What blocks can be reached from a block

In the `hkTrackerScanSnapshot` each Block lists the other blocks that it references. These have been determined by reflection information, scanning and perhaps handlers for specialized types.

**Extending Functionality with Handlers**

Sometimes its best to write some code to describe specifically how to handle a type. This is particularly useful for template containers, and perhaps for a type that has a particularly unusual way of accessing other Blocks.

The mechanism to write special case code revolves around implementations of the `hkTrackerLayoutHandler`. The `hkTrackerLayoutCalculator` is a type that is used to create `hkTrackerTypeLayout` based on types. It can do this in two main ways. The first way is to use the reflection data that has been registered with the `hkMemoryTracker`.

If a type is scanned it can have a `hkTrackerTypeLayout` created with the fields found, and registered with the `hkTrackerLayoutCalculator`.

If the `hkTrackerLayoutCalculator` doesn't know a type, it will see if there is a `hkTrackerLayoutHandler` that is registered for that type. If there is, it will ask the handler to perform two functions:

1. Get the size of the specific instance given the type node.
2. For an instance of that type, to describe Blocks that that instance knows about.

An implementation of `hkTrackerLayoutHandler` has 2 methods that need to be implemented. One returns the size for the type and the other returns descriptions of any memory blocks an instantiation knows about.

For a hkTrackerLayoutHandler to be used it must be registered with the hkTrackerLayoutCalculator. A single handler can be referenced with multiple names, for example a hkLocalArray, hkArray will require similar code, so its probably easiest to implement a single handler and register it under both names.

**Reflection and Types**

In order to be able to traverse memory and find out what blocks reference other blocks accurately there needs to be reflection data describing fields that can reference other blocks. This reflection data is normally produced automatically. For Havok projects the generated reflection is stored in files in the 'Classes' directory in projects. The name of the file(s) is of the form XXXTrackerClasses.cpp. The file will contain definitions of the fields and their types. Only fields which reference other sections of memory are reflected, as that's the only information needed to work out the relationship between memory Blocks.

Each field is specified with a name, an offset, a type and some flags. The type is given as a C++ style type definition. In order to be able to use such a definition, it's useful to be able to turn it into something that can be processed.

To do this there is the hkTrackerTypeTreeParser. The main method on this class is parseType which will parse a C++ style string type definition into a syntax tree constructed out of hkTrackerTypeTreeNode objects. The syntax tree makes it much easier to work out how to process a type. For example if the type was hkArray<int>* the tree would consist of a TYPE_POINTER node containing a TYPE_CLASS_TEMPLATE of type hkArray<int>. In the example given, the code can check if the pointer is not HK_NULL; if so, it can continue processing with hkArray<int> on the memory that is being pointed to.

The tracker reflection system doesn't reflect any information about 'const', and structs are reflected as class. In some situations a type may be specified when its underlying type (struct, class, enum, etc) is not determined. In this case a node will just be specified as TYPE_NAMED.

All of the types reflected are registered with the hkMemoryTracker. To find a type that has been registered there is the method findTypeDefinition.

**Type Determination in 'new'**

C++ has a method for determining a types information: through RTTI. Unfortunately that capability has some serious limitations for cross platform code, in particular that the string returned from typeid(TYPE).name() is entirely compiler-defined. Additionally many developers are wary of building code with the RTTI feature enabled, due to performance and code footprint considerations.

The memory tracker system operates without the need for RTTI, but to do so it does need to use a compiler dependent feature in the form of __PRETTY_FUNCTION__ and its equivalents. This compiler variable, equates to the current function signature. By implementing a method in a class that returns this string it is possible to determine the type of the class. There are some issues mainly in that different compilers return different styles of strings.

To get around these issues there are the facilities found in hkCurrentFunction. It provides a compiler independent way to return something that contains the class type in HK_METHOD_CLASS_NAME. It also contains a compiler specific function that will extract the type name from what HK_METHOD_CLASS_NAME returns.

The advantage of using this technique over say passing in the name of the type to the allocator macros, is that it will produce the right result even if the type is templated.

### Creating Snapshots and Registering Handlers

Snapshots are databases used by the tracking system to store the current state of memory. The two types involved are `hkTrackerSnapshot` and `hkTrackerScanSnapshot`. The `hkTrackerSnapshot` is a simple structure that stores the current memory walk, and the current data held in the `hkMemoryTracker` instance. The `hkTrackerScanSnapshot` is more like a database describing all of the memory Allocations and Blocks, their types and how they are related. All of the reports use the `hkTrackerSnapshot` as the key repository of information to produce the report.

The easiest way to get a `hkTrackerScanSnapshot` is to use `hkTrackerSnapshotUtil::createSnapshot`, which will produce a snapshot of the current memory usage.

For more flexibility the `hkTrackerScanSnapshot` can be produced manually. In particular this is useful if custom handlers are to be registered. First a `hkTrackerSnapshot` needs to be produced. Generally it is constructed on the stack, so no additional allocation is produced.

```
hkTrackerSnapshot snapshot;
snapshot.init();
```

The implementation of `init` removes allocations in the `hkTrackerSnapshot` that are known shouldn't be part of the snapshot. On PlayStation®3 (tm) there has to be special handling for allocations which have been freed and then reallocated on the SPU. The memory tracker does not run on the SPU, and so the `hkMemoryTracker` will not be informed of the morphing of the type. In order to work around this there is a utility `hkVerifySnapshotTypesUtil` used internally in `init` which using types structure will try and determine if any types have been morphed.

If custom handlers are wanted, they have to be registered with the `hkTrackerLayoutCalculator`. The code snippet below shows how this can be done:

```
// All of this code must be after snapshot.init() - so that the hkTrackerSnapshot is not polluted with
  allocations used here

hkTrackerLayoutHandler* customHandler = new MyCustomTypeHandler;

hkTrackerTypeTreeCache* typeCache = new hkTrackerTypeTreeCache;
hkTrackerLayoutCalculator* layoutCalc = new hkTrackerLayoutCalculator(typeCache);
typeCache->removeReference();

// Register custom handlers here!!
layoutCalc->addHandler("MyCustomType", customHandler);
// A handler can be registered under multiple names - if the handler can work for the different types
layoutCalc->addHandler("MyCustomTypeAnother", customHandler);
// The layoutCalc will keep the ref
customHandler->removeReference();

hkTrackerScanSnapshot* scanSnapshot = scanCalc.createSnapshot(&snapshot, layoutCalc);
layoutCalc->removeReference();
```

### Examples from the demo framework

When the "*-memstat*" flag is given to the demos, the demos will write a '`captureMemoryDump.txt`" file on exit. This file will start with the raw block information appended by several additional sections.

### Heap Memory Breakdown

The following section shows how the heap memory is used. In our example about 540k of heap memory is reachable by walking from the demo, 409k stack memory has been allocated from the heap, 84k which are in thread local caches (`hkThreadMemory`) and 308k which is not reachable from the demo. That 308k can be examined with the `hkMatchSnapshotStatisticsCollector` described in the next section. All memory used by Havok Physics 2012 will be part of the memory owned by the demo.

```
Heap Memory (   1343072 Bytes) used by:
===================================
      540032 Bytes owned by demo (hierarchy)
           ? Bytes not owned (with stack trace), use debug memory to get this data
           0 Bytes allocated by hkSolverAllocator
      409600 Bytes allocated by thread local hkStackMemory
       84912 Bytes allocated by thread local hkThreadMemory
      308528 Bytes unknown
```

### Hierarchical View of all Memory

The report contains a hierarchical view of the memory found with the collector. The first column shows the names of the items which were given to the collector and the number of instances of each. The second and third columns show the used and allocated block sizes. The difference is because of allocators padding allocation requests up to a suitable size and partially full buffers (mostly `hkArray`s). Example: Say we have an `hkArray` of bytes which has a size of 1 and a capacity of 9. Our allocator does not support 9 byte allocations but will round up to 16 bytes. In our memory report the left number will be 1 (byte used) and the right number will be 16. For our memory statistics the allocated block size of 16 is more important than the memory used by the engine.

```
Root
Demo                                    432900.000:   540032.000
  self                                       240.000:      256.000
  hkpWorld                                432516.000:   539616.000
    self                                      848.000:     1024.000
    activeIslands                              52.000:      128.000
    Active (13)                             379552.000:   445424.000
      self (13)                                1404.000:     1664.000
      Entity (480)                           276900.000:   324944.000
        self (480)                             261120.000:   307200.000
        constraintRuntime (479)                 15328.000:    15328.000
        Shape (2)                                  96.000:       96.000
        CollAgtPtr (100)                          328.000:     1616.000
// ... report truncated
```

The final section is a flattened view of the previous hierarchical view, ordered by size. For each item in the report, the entry with no indent (in this case Entity) is the one being examined. Below the entry are the allocations reachable from entities.

Allocation ownership is generally a directed acyclic graph rather than a pure tree as we represented it in the hierarchical view. Above "Entity" we see three items Active, Inactive and Fixed. These correspond to items which contain Entity allocations, in this case simulation islands. All entity allocations which appeared in different subtrees of the hierarchical view are merged for the flattened view. In most cases, there is only a single containing item.

```
Active (13)                                379552.000:   445424.000:
    Inactive (20)                               17360.000:    20208.000:
    Fixed (1)                                    3104.000:     3632.000:
Entity (505) 63.3%                         291392.000:   341904.000:
    self (505)                                 274720.000:   323200.000:
    constraintRuntime (499)                     15968.000:    15968.000:
    Shape (7)                                     336.000:      336.000:
    CollAgtPtr (103)                              336.000:     1664.000:
    constraintsSlave (46)                          32.000:      736.000:
```

### Flat view of the Remaining Allocations

Previously we saw that 308k was not reachable by walking memory from the demo. We can examine what was missed by using the `hkCheckingMemorySystem` with a `hkMatchSnapshotStatisticsCollector`. To do this simple add the *-c* flag to the command line when running the Havok demos. This allows us to get at least a stack trace for every allocation in the system As a result our heap memory usage now has more information.

```
Heap Memory (    778690 Bytes) used by:
====================================
      475320 Bytes owned by demo (hierarchy)
      303114 Bytes not owned (with stack trace)
           0 Bytes allocated by hkSolverAllocator
           0 Bytes allocated by thread local hkStackMemory
           0 Bytes allocated by thread local hkThreadMemory
         256 Bytes unknown
```

After the flattened statistics section, we see a list of allocations which were not traversed during the memory walk. The format is similar to the memory leak report, i.e. size information and a call stack. Here we see that the job queues in `hkDefaultPhysics2012Demo` were not included.

```
*********************************************************************************
* Dump of allocations which weren't included in the calcStatistics:             *
* Summary :  113 Blocks of total size       303114  (allocated       303114)     *
*********************************************************************************
* 16 blocks of size  16384 =   262144 (allocated 262144) not included in calcStatistics *
*********************************************************************************
Blocks: 0E1BCD90 0E1B8D30 0E1C4E40 0E1C0DE0 0E1CCEF0 0E1C8E90 0E1D4FA0 0E1D0F40 ...
    NOTE! Not all allocs may have exactly this trace...
c:\dev\700release\source\common\base\memory\system\checking\hkcheckingmemorysystem.cpp(267):'
  hkCheckingMemorySystem::checkedAlloc'
c:\dev\700release\source\common\base\memory\system\checking\hkcheckingmemorysystem.cpp(20):'
  hkCheckingMemorySystem::AllocatorForwarder::blockAlloc'
c:\dev\700release\source\common\base\memory\router\hkmemoryrouter.h(271):'hkAllocateChunk<hkJobQueue::
  JobQueueEntry>'
c:\dev\700release\source\common\base\container\queue\hkqueue.inl(91):'hkQueue<hkJobQueue::JobQueueEntry>::
  setCapacity'
c:\dev\700release\source\common\base\thread\jobqueue\hkjobqueue.cpp(126):'hkJobQueue::hkJobQueue'
c:\dev\700release\demo\demos\democommon\demoframework\hkdefaultdemo.cpp(184):'hkDefaultDemo::hkDefaultDemo
  '
c:\dev\700release\demo\demos\democommon\demoframework\hkdefaultphysicsdemo.cpp(117):'
  hkDefaultPhysics2012Demo::hkDefaultPhysics2012Demo'
c:\dev\700release\demo\demos\physics\api\collide\broadphase\multipendulums\multipendulumsdemo.cpp(144):'
  MultiPendulumsDemo::MultiPendulumsDemo'
c:\dev\700release\demo\demos\physics\api\collide\broadphase\multipendulums\multipendulumsdemo.cpp(243):'
  hkCreateMultiPendulumsDemo'
c:\dev\700release\demo\demos\democommon\utilities\menu\menudemo.cpp(1512):'MenuDemo::startCurrentDemo'
c:\dev\700release\demo\demos\democommon\utilities\menu\menudemo.cpp(333):'MenuDemo::MenuDemo'
```

### Using the Memory Snapshot Analyzer

The Havok memory analyzer is an experimental tool introduced to provide a graphical view for analyzing a memory snapshot.

**Warning:**
This is a Beta tool, and is likely to change significantly in future versions

### Requirements for Capturing a Memory Snapshot

First, ensure you are running the Havok Visual Debugger server and that you can connect to your game using the Visual Debugger client.

In order to acquire a memory snapshot containing callstack information, you must use the `hkCheckingMemorySystem`. The simplest method to use the checking memory is to call `hkMemoryInitUtil::initChecking` before initializing the Havok base system. See the Getting Started guide to the Memory Tracker, above for more information.

### Acquiring a Memory Snapshot

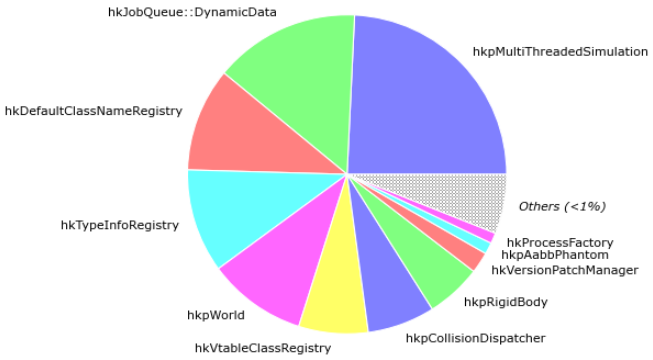
Finally, connect the Visual Debugger to the game, and in the "Viewers" menu, select "* Grab Memory Snapshot". This will open a dialogue as below prompting you for a filename to save the snapshot to.

### Using the Snapshot to Analyze your Memory Use

The Havok Memory Snapshot Analyzer is distributed separately from the rest of the SDK. After opening the tool and selecting a snapshot file to analyze, you are presented with the 'types' screen.

### The Block Types Window

| Name | Total Size (Bytes) | Count |
|---|---|---|
| hkpMultiThreadedSimulation | 28,400 | 1 |
| hkJobQueue::DynamicData | 17,312 | 3 |
| hkDefaultClassNameRegistry | 12,320 | 1 |
| hkTypeInfoRegistry | 12,320 | 1 |
| hkpWorld | 11,748 | 1 |
| hkVtableClassRegistry | 8,232 | 2 |
| hkpCollisionDispatcher | 8,016 | 1 |
| hkpRigidBody | 6,576 | 12 |
| hkVersionPatchManager | 2,512 | 1 |
| hkpAabbPhantom | 1,368 | 6 |
| hkProcessFactory | 1,266 | 1 |
| UNKNOWN | 1,100 | 2 |
| MenuDemo | 748 | 1 |
| hkJobQueue | 644 | 1 |
| hkpViolatedConstraintArray | 516 | 1 |
| hkpSimulationIsland | 448 | 2 |
| hkCpuJobThreadPool | 424 | 1 |
| AddRemoveBodiesDemo | 288 | 1 |
| hkpMtThreadStructure | 288 | 1 |
| hkVisualDebugger | 246 | 1 |
| hkpWorldCinfo | 240 | 1 |
| hkVersionRegistry | 228 | 1 |
| hkpPhysicsContext | 152 | 1 |
| hkpShapeDisplayViewer | 140 | 1 |
| hkpProcessCollisionInput | 128 | 1 |
| hkpEntity* | 120 | 2 |
| hkPseudoPad | 120 | 2 |
| hkDefaultError | 108 | 1 |



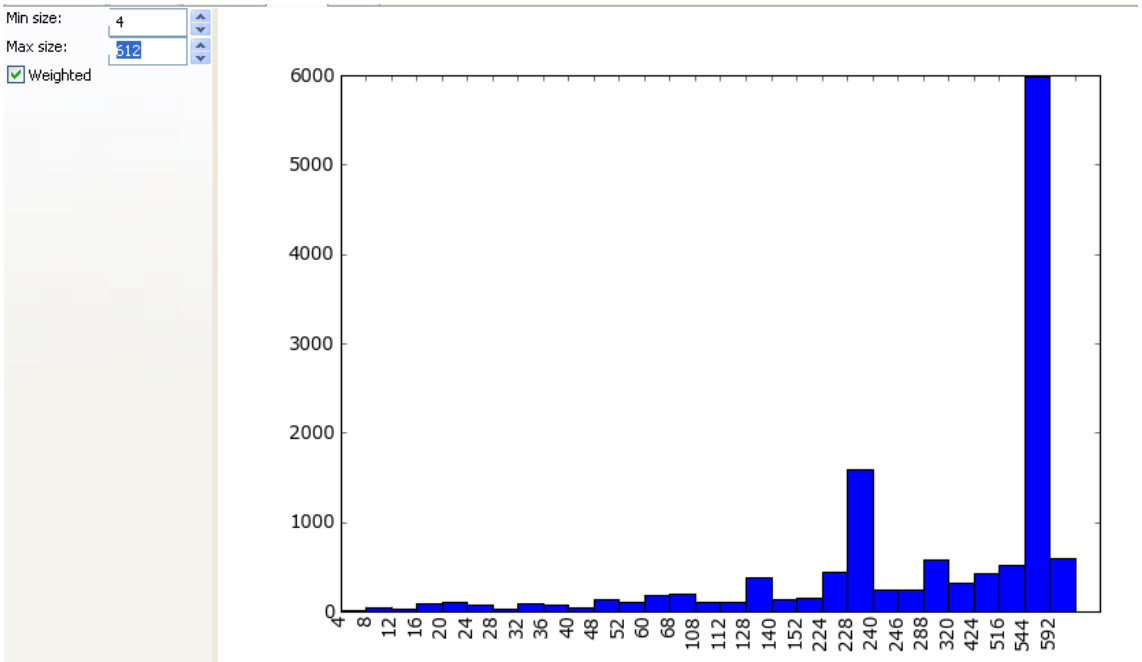

This view summarizes the snapshot based on the type and size which was allocated.

Note, however that some of these numbers may be misleading; We attempt to consolidate contained data into its owner. So for example, you may see the number of `hkpRigidBody`s increase as their count incorporates types wholly owned by the rigid body.

Double-clicking a type will show the unique callstacks which allocated this type, allowing you to inspect where these allocations occurred. Double-clicking on a frame in this dialog will bring you to the equivalent location in the callstack window.

### The Block Sizes Window

The sizes window acts in a similar manner to the types window. You are presented with a list of Blocks presented by the size of the requested amount of memory. Selecting one of these sizes allows you to break this down by type and callstack.




### The Call Tree Window

The call tree window provides an interface to navigate through the call graph to determine which functions allocated the most memory through the Havok memory system.

| Function | Inclusive (Bytes) | % Inclusive |
|---|---|---|
| hkAllocate<char> | 968 | 0.83 |
| strDup | 968 | 0.83 |
| assign | 968 | 0.83 |
| hkMemoryRouter::easyAlloc | 968 | 0.83 |
| hkStringPtr::operator= | 936 | 0.80 |
| hkCreateMenuDemo | 906 | 0.77 |
| hkDemoDatabase::createDemo | 906 | 0.77 |
| frameworkStartDemo | 906 | 0.77 |
| hkpWorld::operator new | 864 | 0.74 |

| Child functions | Inclusive (Bytes) | % Inclusive |
|---|---|---|
| AddRemoveBodiesDemo::addBody | 396 | 50.77 |
| hkDefaultPhysicsDemo::stepDemo | 384 | 49.23 |

| Parent functions | Inclusive (Bytes) | Local (Bytes) | % Inclusive |
|---|---|---|---|
| MenuDemo::stepCurrentDemo | 780 | 780 | 100.00 |

This window is split into three panels; The largest lists all functions which have been seen in the callstacks in the memory snapshot and the amount of memory allocated by this function. This is inclusive of the function's children, as a function does not allocate memory by itself–each allocation will be recorded by the Havok memory manager.

The next window lists the functions which were called by the currently selected function, and gives a proportional breakdown of which child functions, after being called by the currently selected function allocated memory.

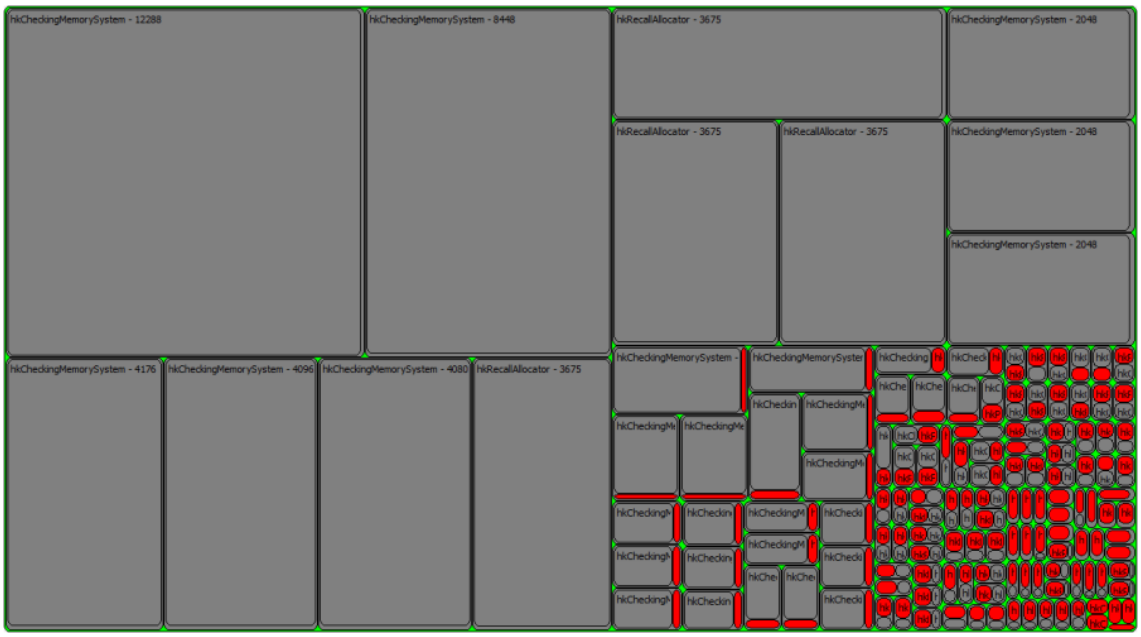
The final window shows the parents of the currently selected function (i.e. the functions which called the current function) and works similarly to the child view.

### The Fragmentation Window

This window uses the Allocations saved in the memory snapshot to draw a memory layout of them,

placing them one inside the other depending on the level of the provider that reported the allocation. A different color will be used depending on the Allocation type (used, unused, overhead).

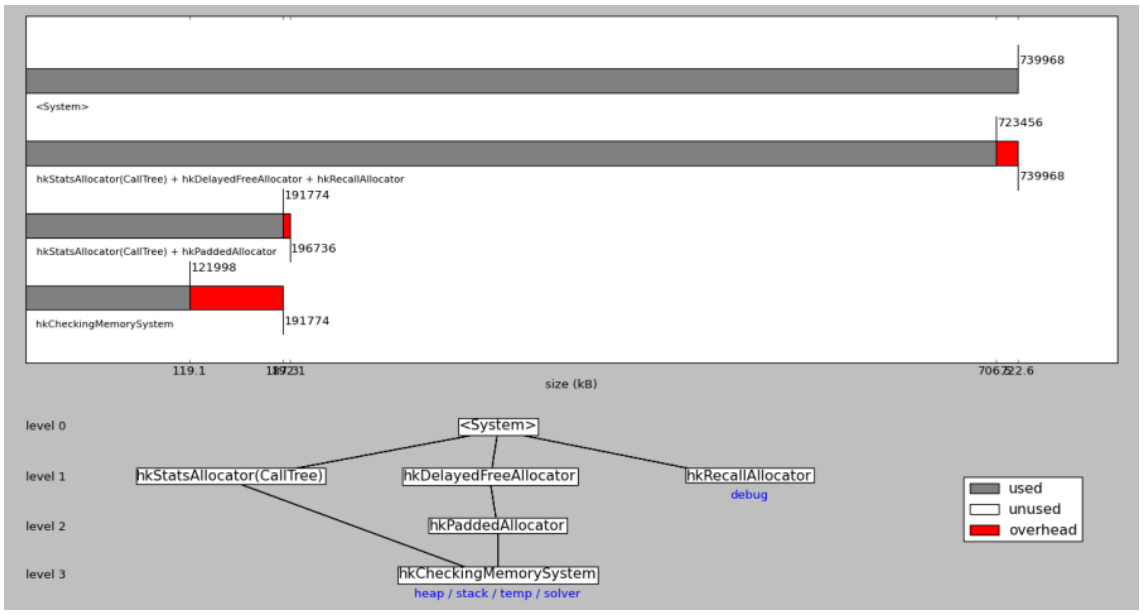
A provider is a generic entity in the memory system which provides memory to some other entity and obtains it from a lower-level provider. A provider is roughly the same as an allocator, but in some memory systems the memory system itself acts as a provider for specific allocations; for this reason we are not using the word 'allocator'.




### The Provider Stack Window

This window provides very high-level information about the different memory providers in the memory system and how the memory is used within them.

A provider is a generic entity in the memory system which provides memory to some other entity and obtains it from a lower level provider. A provider is roughly the same as allocator, but in some memory systems we have that the memory system itself acts as provider for specific allocations, for this reason we are not using the word 'allocator'.




This window is trying to give a layered view of the memory system, highlighting overhead and unused blocks, and also providing a graph that shows the wiring of providers in the used memory system.

### The Raw Report Window

This window is a simple report that includes the statistics collected from the memory system using `hkMemorySystem::printStatistics()`.

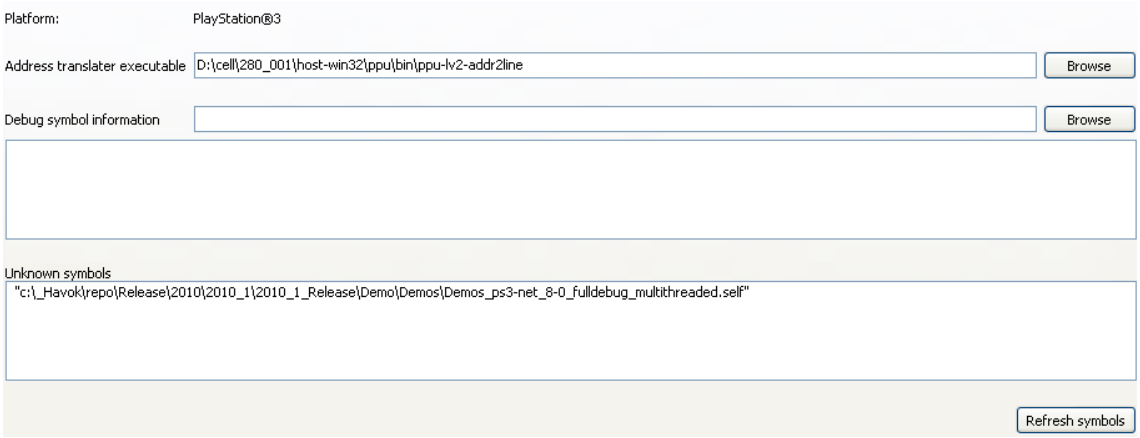
```
hkCheckingMemorySystem memory overview:
=======================================

    Allocation totals:

              89198 allocated by heap
             526848 allocated by debug
              32800 allocated by stack
                  0 allocated by temp
                  0 allocated by solver
              16384 in delayed frees
               4962 in allocation padding
              40960 in callstacks
              28672 in active pointer map
                144 in allocator forwarders
            -------
             739968 computed total
           50000000 reported total

    Heap small block (<=640) summary:

                 16 *      34 =     0k : >>>>>>>
                 32 *      38 =     1k : >>>>>>>>>>>>>>>>>>>>>>>>>>
                 48 *      18 =     0k : >>>>>>>>>>>>>>>>>>
                 64 *       3 =     0k : >>>
                 80 *       2 =     0k : >>>
                 96 *       0 =     0k :
                112 *       0 =     0k :
                128 *       1 =     0k : >>>
                144 *       3 =     0k : >>>>>>>>>>>
                160 *       0 =     0k :
                176 *       1 =     0k : >>>>
                192 *       2 =     0k : >>>>>>>>>>
                208 *       6 =     1k : >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
                224 *       1 =     0k : >>>>>
                240 *       1 =     0k : >>>>>
                256 *       1 =     0k : >>>>>>
                272 *       2 =     0k : >>>>>>>>>>>>>>>
                288 *       0 =     0k :
                304 *       0 =     0k :
                320 *       1 =     0k : >>>>>>>
                336 *       0 =     0k :
                352 *       0 =     0k :
                368 *       0 =     0k :
                384 *       1 =     0k : >>>>>>>>>>
```

### The Modules Window

Some platforms do not have the introspection required to get symbol information at runtime, and so these

platforms have limited or missing callstack information. However, it is possible to process the reported addresses offline to get the missing callstack information.

If the memory analyzer detects you are using a platform which is missing callstack information, you are presented with the following screen on loading a memory snapshot




This window lists the detected platform and list of files providing symbol information. You must provide a path to a tool which can translate the addresses to line numbers. For PlayStation®3 this should be **ppu-lv2-addr2line** ; for Xbox 360, this should be **x360-addr2line** (included with the memory analyzer). You should also provide the path to the unsigned executable that generated the snapshot. Pressing "Refresh symbols" will attempt to refresh the addresses in the callstack information with the correct symbols.

## 1.3 Havok Math Library

### 1.3.1 Introduction

The `hkMath` library provides a platform independent interface to simple linear algebra primitives such as vectors, matrices and quaternions. The interface is designed to be able to take advantage of platform specific optimizations, especially SIMD (vector) instructions, even if compiler support is lacking.

This chapter introduces you to the `hkMath` classes, highlights some common usage problems and some of the conventions used in `hkMath`, as well as some optimization tips for working with the library. For more detailed API information for each `hkMath` class, see the Reference Manual.

### 1.3.2 Math Types

#### 1.3.2.1 Basic Types

- **hkReal**

  The default floating point type, defined as '`float`'.

- **hkQuadReal**

  A type representing a SIMD register defined as four `hkReals`. This will generally be a typedef to a native type if the compiler has such support. e.g. `vector float` for alitvec platforms.

- **hkSimdReal**

  When SIMD is enabled, a single `hkReal` in a SIMD register, i.e in the 'x' component of an `hkQuadReal`. When SIMD is disabled, an `hkSimdReal` is a simple `hkReal`. Using this type instead of `hkReal` can allow the compiler more optimization opportunities.

- **Other hkMath functions**

  Functions such as `hkMath::sqrt()`, `hkMath::sin()`, and so on.

#### 1.3.2.2 Compound Types

- **hkVector4**

  `hkVector4` is Havok's general purpose vector class. Most of the platform specific optimization is done here, and the other math classes are mostly compositions of `hkVector4s`.

  Each vector has four `hkReal` elements, for ease of use with vector processors. However, it can be used to represent either a point or vector (x,y,z) in 3-space, i.e. a 3-vector. The final, or w, value of the `hkVector4` defaults to zero.

- **hkQuaternion**

  Class for quaternions, which you can use to represent rotations. *Note that hkQuaternion transformation methods assume a normalized (unit) quaternion, which is the most common case.*

- **hkMatrix3**

  Class for a 3X3 matrix of `hkReals`.

- **hkRotation**

  Class to store an orthonormal rotation matrix. In Havok there are two ways to store rotations: `hkQuaternion` and `hkRotation`. If you are operating on `hkVector4`s e.g. rotating `hkVector4`s then using `hkRotation` will be faster than an `hkQuaternion`. If you need to store many rotations, `hkQuaternion` is more space efficient. Also, quaternions are faster to renormalize.

- **hkTransform**

  This class represents a rotation and a translation. It is a composition of an `hkRotation` and an `hkVector4`.

- **hkVector4Util**

  Special purpose routines for working with vectors that do not fit into `hkVector4`, but deserve platform specific optimizations. For example `hkVector4Util::rotatePoints(...)`.

- **hkQsTransform**

  A decomposed (translation vector + quaternion + scale vector) transform. Used by the Havok Animation system; check the Havok Animation chapter in this manual for more details.

### 1.3.3 Using the Math Library

This section aims to highlight some of the common issues you may run into when using the `hkMath` library.

#### 1.3.3.1 Stack Allocations and 16-byte alignment issues

Havok expects math objects memory to be 16-byte aligned. This is because unaligned loads and stores from the SIMD unit are typically very expensive. On platforms which pass SIMD parameters on the stack (e.g. IA32), this means that you cannot reliably pass `hkMath` objects by value, but should pass them by reference instead. The most common cause of misaligning a `hkVector4` is through passing a stack allocated `hkVector4` by value.

The following examples shows common cases where a `hkVector4` is stack allocated. Note that these cases will not always cause a crash as occasionally they will be correctly aligned. It is this fact that make it difficult to link a runtime crash to an incorrectly allocated `hkVector4`, hence they should be caught as early as possible.

*Example 1:*

Here a `hkVector4` will be stack allocated before it is passed to the user method `animateToPosition(...)`.

```
// incorrect - compiler bugs mean the temporary may not be correctly aligned
animateToPosition( hkVector4( 10.0f, 0.0f, 0.0f) );

// correct - works on all compilers
hkVector4 pos( 10.0f, 10.0f, 10.0f );
animateToPosition( pos );
```

*Example 2:*

Here a stack allocated `hkVector4` is returned by value from a conversion method. This is a common case as many users have an in-house math library and need to convert `hkVector4`s to and from their own proprietary format.

```
// incorrect: creates and returns a stack allocated hkVector4
hkVector4 convertMyVec2HKVec( MyVec in )
{
    return hkVector4( (float)(in[0]), (float)(in[1]), (float)(in[2]) );
}

or

// incorrect: creates and returns a stack allocated hkVector4
hkVector4 convertMyVec2HKVec( MyVec in )
{
    hkVector4 temp;
    temp.set( (float)(in[0]), (float)(in[1]), (float)(in[2]) );
    return temp;
}

// correct: passes the hkVector4 to be set by reference
void convertMyVec2HKVec( const MyVec& in, hkVector4& out )
{
    out.set( in[0], in[1], in[2] );
}
```

#### 1.3.3.2 Correctly constructing hkMath objects

All `hkMath` default constructors are lightweight and do not initialize their data members. This means that code such as:

*Example 1:*

```
hkRotation rotation;
hkQuaternion quat(rotation)
```

Will cause an assert the following assert in the `hkQuaternion` constructor:

```
HK_ASSERT2(isOk(), "hkRotation used for hkQuaternion construction is invalid. hkQuaternion is not
  normalized/invalid!");
```

To prevent this class of assert ensure you construct `hkMath` objects with valid data.

#### 1.3.3.3 Aliasing of arguments

`hkVector4` methods are explicitly safe against argument aliasing. All other math classes are not. For instance:

```
a.setAdd4(b,c) -> a = b+c
a.add4(b) -> a += b
```

This is done in order to avoid creating temporaries for the common case of no aliasing.

#### 1.3.3.4 Denormal numbers

Occasionally, Havok code may produce denormal or subnormal values. Operations on these denormals take significantly longer to execute, up to one or two orders of magnitude, depending on the architecture. Havok code assumes that the processor is set to flush these values to zero and may produce incorrect results if it is not. Any worker threads spawned by Havok will automatically enable this mode, but any threads that call Havok functions must also set it first. This can be done with the following code:

```
#if defined(HK_COMPILER_HAS_INTRINSICS_IA32) && HK_CONFIG_SIMD == HK_CONFIG_SIMD_ENABLED
// Flush all denormal/subnormal numbers (2^-1074 to 2^-1022) to zero.
// Typically operations on denormals are very slow, up to 100 times slower than normal numbers.
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);
#endif
```

#### 1.3.3.5 Viewing asserts in hkMath

The assert mechanism in the Havok library is designed to, among other things, catch bad or out-of-range data in the engine. A common place that bad data is trapped is in the `hkMath` library. It is essential that you are able to view asserts that the `hkMath` library generates. For details on how to view asserts or override the assert mechanism please consult the Base System Integration guide. Note that asserts will only fire in the `Debug` Havok libraries, in the release libraries bad data would generally cause a crash or machine exception.

#### 1.3.3.6 Nonstandard operator=

Normally, the member method `operator=` returns a reference to itself so that assignments can be chained.

```
object a,b,c; // fictitious type 'object'
a = b = c;
```

Even when the return value is unused, some compilers are unable to optimise the return value away and generate extra redundant code. Since the Havok math classes are used so pervasively and are so performance critical, objects do not return a reference to themselves.

#### 1.3.3.7 No overloaded operators + , - , *

The definitions of operators such as + , - and * are required to return their result by value. Unfortunately, because of compiler issues, the generated code is often either extremely inefficient or causes a crash as a result of alignment issues. (See the previous section)

### 1.3.4 Common Conventions

#### 1.3.4.1 Location of methods

Methods are located in the class to which the result is assigned. For example, the operations to transform a `hkVector4` by rotations and transformations are part of `hkVector4` because the result is a `hkVector4`.

#### 1.3.4.2 Naming of methods

Methods that begin with `set`, such as `hkVector4::setAdd4()`, set the value as a function of its input arguments only.

Methods that do not begin with `set` mutate or use the existing value of the object. For instance `hkVector4::add4()` adds to the existing value and `hkVector4::dot3()` uses the existing value.

```
a.setAdd4(b,c) -> a = b+c
a.add4(b) -> a += b
```

#### 1.3.4.3 Method suffixes

Some methods in `hkVector4` have the suffix "3" or "4". As you know, in Havok, all vectors actually store four elements. We use the suffixes to make explicit which elements are being operated on: the first three, or all four. For example, there is both a `normalize3()` and a `normalize4()` method in this class.

However, many `hkVector4` methods have a "4" variant but not a "3". This is because there is no generic hardware support for the "3" version of the operation. These include `add4()` and `setAdd4()`.

Other `hkMath` classes do not have these suffixes because there is no similar ambiguity.

### 1.3.5 SIMD and Non SIMD support

#### 1.3.5.1 hkSimdReal and hkReal

The Havok math library is carefully constructed so that there is no penalty for platforms that do not have SIMD units. This is done by never passing `hkQuadReals` by value which results in a register move on SIMD machines but possibly a stack copy on those without.

Havok currently provides SIMD support for version SSE1. For compilers that have SIMD support, we define `hkQuadReal` to be the compiler's native four-element SIMD type (for example `__m128` or `__v4sf`). `hkSimdReal` is an `hkQuadReal` for which only the 'x' element contains a valid value. Implicit conversions are provided between `hkSimdReal` and `hkReal` for ease of use.

For compilers that do not have SIMD support, `hkQuadReal` is a structure with four `hkReals` and 16-byte alignment, and `hkSimdReal` is a typedef for `hkReal`.

#### 1.3.5.2 Ambiguous overloads

Overloaded operators are provided for `hkSimdReal` so that it behaves much like an `hkReal`. Thus:

```
hkSimdReal s0, s1;
hkReal r0, r1;

s0 * s1; // Ok, result is an hkSimdReal.
r0 * r1; // Ok, result is an hkReal.

s0 * r0; // Error ambiguous overload
r0 * s0; // Error ambiguous overload
```

Mixing types is an error because the operation could be performed on either the FPU or SIMD unit. Use an explicit cast in this case.

```
hkSimdReal s0, s1;
hkReal r0, r1;

hkReal(s0) * r0; // Ok, calculation on FPU, result is an hkReal
s0 * hkSimdReal(r0); // Ok, calculation on SIMD unit, result is an hkSimdReal
```

### 1.3.6 Optimization

#### 1.3.6.1 FPU/SIMD conversions

Most compilers find it difficult to optimise conversion between SIMD types and FPU types. Where possible use the SIMD types and operators when working with `hkMath`.

The following examples illustrate this:

```
hkVector4 v0, v1;
hkReal r = v0.dot3(v1);
hkVector4 v2;
v2.addMul4(r, v0);
```

The above gives the compiler more optimization opportunities if written as:

```
hkVector4 v0, v1;
hkSimdReal r = v0.dot3(v1);
hkVector4 v2;
v2.addMul4(r, v0);
```

Similarly:

```
// ok
hkVector4 v0;
hkReal r = v0(0)
hkVector4 v2;
v2.addMul4(r, v0);

// better
hkVector4 v0;
hkSimdReal r = v0.getSimdAt(0);
hkVector4 v2;
v2.addMul4(r, v0);
```

#### 1.3.6.2 Load/Store Hazards

Some platforms incur a penalty for overlapping loads and stores of different sizes.

```
hkVector4 v0, v1;
v0(3) = 0;   // store 4 byte float
v1.add4(v0); // read 16 byte simd value
```

Try to avoid mixing element wise write access with whole vector access.

You can safely and quickly get read only access to a vector element using the `getSimdAt()` method of `hkVector4`. Similarly the `setXYZ0` and `setXYZW` can be used for write access.

#### 1.3.6.3 Multiple dot products with the same vector

Dot products are generally quite slow because of the need to perform a horizontal add–an operation not directly supported by most SIMD units. For example, this is the dot product for Pentium III and IV.

```
hkQuadReal x2 = _mm_mul_ps(m_quad,a.m_quad);
hkQuadReal broadcastY = _mm_shuffle_ps(x2,x2,_MM_SHUFFLE(1,1,1,1))
hkQuadReal xySum = _mm_add_ss(broadcastY, x2);
hkQuadReal broadcastZ = _mm_shuffle_ps(x2,x2,_MM_SHUFFLE(2,2,2,2))
hkQuadReal xyzSum = _mm_add_ss(broadcastZ, xySum);
```

If you have, for instance, three dot products with the same vector, you can put the three vectors into the rows of a matrix3 and get three results for only one horizontal add.

## 1.4 Serialization

### 1.4.1 Introduction to Serialization

Serialization is the process of converting data to a writable format. Once serialized, the original data can be accurately reconstructed later on. This means that serialized data can safely be written to disk or transmitted across a network. In a game development context, serialization is used to transfer information to and from the game world. It is generally used when saving and loading data, or in networked multiplayer games. This chapter explains how serialization works in Havok, and how best to use it in your games.

#### 1.4.1.1 Features

- File formats
  - Cross-platform binary tagfile format.
    - Cross platform load and save.
    - Compact file size.
    - Can losslessly convert to/from binary and XML formats.
  - High performance binary format.

    - Binary file format is a memory image.
    - Loading consists of single raw read and very fast pointer patching.
    - Zero allocation loading into a user supplied buffer.
  - Flexible XML format.
    - Human readable XML format.
    - Cross platform load and save.
    - Can process with standard XML tools.
    - Can losslessly convert between XML and binary formats.

- Asset pipeline support.
  - Cross platform export. Generate console assets directly from a PC.
  - Files are introspectable. Files optionally contain the metadata for its contents.
  - Metadata is reusable by tools outside the SDK.
- Version compatibility
  - Don't need to re-export your assets when your objects change.
  - Objects can be optionally converted at load time.
  - Can add custom versioning code when automatic versioning is not enough.
  - Versioning code can be stripped for final assets.
- Small runtime footprint.
  - Configurable size/features trade off.
  - Binary loading code is very small.
- Extensible
  - Tools supplied to serialize user classes.
  - Building blocks supplied so custom file format/streams are possible.
  - Supports generic tweakers/editors through object reflection.

### 1.4.2 Basic Loading and Saving

#### 1.4.2.1 Loading with hkSerializeUtil

If you have a Havok asset, `hkSerializeUtil` takes care of the details of loading. It will detect whether the file is tagfile, XML or binary packfile. Also if the asset was produced for a different version of the SDK, `hkSerializeUtil` can update the contents to the latest version on load.

There are two options when loading. One is to load the file into a single flattened block where reference counting is disabled and the block must be destroyed as a whole (see memory considerations). The other is to load the file into individual reference counted objects, just as if created by code. We will refer to these as "flattened" and "object" styles respectively. This choice is independent of the file format: any format may be loaded with either loading style.

The following uses the flattened style. The `hkResource` both stores the memory block and the list of destructors to call when the resource is freed.

```
hkResource* loadedData = hkSerializeUtil::load("filename.hkx");
SomeClass* contents = loadedData->getContents<SomeClass>();
// hkResource owns the loaded data. You must ensure hkResource is
// not destroyed while you access contents.
```

In the object style, `hkObjectResource` is a temporary handle to the contents object. After taking ownership of the data, the object resource may be destroyed and the contents persist. If ownership is not taken, the object resource will clean up its contents on destruction. Reference counted objects will have their count decremented and other objects will be deleted.

```
hkObjectResource* loadedData = hkSerializeUtil::loadOnHeap("filename.hkx");
SomeClass* contents = loadedData->stealContents<SomeClass>(); // steal ownership
delete loadedData; // 'loadedData' no longer owns the data and may now be destroyed
```

There is also a shortcut if access to the `hkObjectResource` is not necessary.

```
SomeClass* contents = hkSerializeUtil::loadObject<SomeClass>("filename.hkx");
```

### Error Checking

Most loading methods take an optional `ErrorDetails` parameter. On load failure and if the parameter is not null, extra information about the failure cause is written into this object.

### hkSerializeUtil Variants

The `hkSerializeUtil` loading methods all come in three variants

- `const char* filename`: The file is opened using the `hkFileSystem` singleton
- `hkStreamReader* reader`: The file is read directly from the stream
- `const void* buf`, `int bufSize`: The file is read directly from memory. The buffer may be freed after loading.

```
// We as the hkFileSystem to open a stream from a filename
hkResource* res0 = hkSerializeUtil::load("Resources/Common/Api/Serialize/SimpleLoad/simple.xml");
res0->removeReference();

    // Fill buffer using external read
    // This has a fast path for up to date binary packfiles
    // The buffer is not modified
char* buf = HK_NULL; int bufSize = 0;
hkResource* res1 = hkSerializeUtil::load(buf, bufSize);
res1->removeReference();

    // There is also a non-copying version for up to date binary packfiles.
    // The buffer must remain valid for the lifetime of the contained objects.
    // Don't forget to call hkNativePackfileUtils::unloadInPlace to destruct the contents.
/* void* contents2 = */ hkNativePackfileUtils::loadInPlace(buf, bufSize);
// contents2 = HK_NULL;
hkNativePackfileUtils::unloadInPlace(buf, bufSize);

    // The data may be streamed instead of loaded all at once.
hkStreamReader* sr = HK_NULL;
hkResource* res3 = hkSerializeUtil::load(sr);
res3->removeReference();
```

**hkLoader Utility Class**

`hkLoader` is a thin wrapper on top of `hkSerializeUtil`. It always uses the flattened loading style and owns all the loaded `hkResource`s. The simple interface and automatic cleanup make it particularly suitable for demo code.

#### 1.4.2.2 Serialize Feature Registration

Some file features require extra code and data to be linked into your executable. In particular, loading and versioning functions may be stripped. See

- `HK_EXCLUDE_FEATURE_SerializeDeprecatedPre700`
- `HK_EXCLUDE_FEATURE_RegisterVersionPatches`

Most people will want to register patches, which enables versioning of assets from the 7.0.0 release.

#### 1.4.2.3 Saving with hkSerializeUtil

Use `hkSerializeUtil::save()` to save your data in the default format (currently tagfile). To save an object, it must have reflection data (an associated `hkClass`).

```
extern const hkClass hkObjectClass;
Object* object = ...;
hkSerializeUtil::save( object, hkObjectClass, hkOstream(FILENAME).getStreamWriter() );
```

There are also related methods which specify allow specifying the output format. When saving, some

objects require special handling. For instance the fixed rigid body in `hkpWorld` belongs to the world and should not be serialized. Since the serialization library is product independent, we get this fine grained control by means of callbacks. In this case `hkpHavokSnapshot::ConvertListener` prevents the fixed body from being saved and also generates storage versions of mesh and heightfield shapes which allows us to reload the asset outside of the game. (Non-storage versions contain pointers to arbitrary external data.)

```
hkpHavokSnapshot::ConvertListener defaultListener;
Object* object = ...;
hkSerializeUtil::saveTagfile( object, hkObjectClass, hkOstream(FILENAME).getStreamWriter(), true, &
  defaultListener );
```

```
hkpHavokSnapshot::ConvertListener defaultListener;
hkPackfileWriter::Options options; // can set more options here
Object* object = ...;
hkSerializeUtil::savePackfile( object, hkObjectClass, hkOstream(FILENAME).getStreamWriter(), true, options
  , &defaultListener );
```

**Redirecting Output**

All `hkSerializeUtil` save methods take a `hkStreamWriter` for output. Rather than create streams directly, it is convenient to use `hkOstream` as a factory. Note that we recommend using an anonymous temporary or a nested scope to ensure the stream is destroyed and all output is flushed into the buffer.

```
// This is the object you want to save
const void* object = HK_NULL;
const hkClass* objectClass = HK_NULL;

// This will expand the buffer to fit
hkArray<char> expandingBuf;
hkResult result0 = hkSerializeUtil::save(object, *objectClass, hkOstream(expandingBuf).getStreamWriter() )
  ;

// This will truncate the file and return HK_FAILURE if the buffer is too small.
char fixedBuf[10000]; int fixedBufSize = 10000;
hkResult result1 = hkSerializeUtil::save(object, *objectClass, hkOstream(fixedBuf, fixedBufSize)
  .getStreamWriter() );
```

### 1.4.3 File Formats

#### 1.4.3.1 Tagfiles

Tagfiles are a platform independent binary file format and are the default and recommended format to store serialized objects and data.

Tagfiles use a variety of data encoding techniques. This makes them smaller than binary packfiles, but requires somewhat more CPU and memory resources to read them back. The difference in CPU time between tagfiles and binary packfiles averaged under a factor of 4 in our in-memory tests. The difference is generally dwarfed by the time taken to read from the storage medium. Conceptually the files may

be thought of as strongly typed nested property lists. Tagfiles are closely related to `hkDataObject`, an abstract and simplified (vs. `hkClass`) reflection interface.

Tagfiles are written using `hkSerializeUtil` (recommended) or directly via the `hkBinaryTagfileWriter`, `hkXmlTagfileWriter` or `hkTextTagfileWriter` classes. Note that text tagfiles are for data debugging, there is no API provided to read text tagfiles back into game.

Tagfiles are loaded using `hkSerializeUtil` (recommended) or `hkXmlTagfileReader` or `hkBinaryTagfileReader` if low level access is needed. The low level load method returns `hkDataObject` which must be converted to native C++ objects using `hkDataObjectUtil::toResource()` or `hkDataObjectUtil::toObject()` as appropriate. Access to this layer is generally only needed if the contents need to be edited in some way. E.g. the `hkSerializeUtil` load methods version the file contents before converting to usable C++ objects.

#### 1.4.3.2 Packfiles

Serialized data may also be stored in packfiles. Packfiles, the default file format in previous versions, have largely been superseded by tagfiles.

They are conceptually similar to object files–raw memory images with pointer patch tables. Because of the memory image, they are naturally compiler and platform specific. Versioning is naturally more complex with memory images and so for most cases, tagfiles are a better choice. The one exception is when the file contents is guaranteed to be up-to-date (e.g. perhaps a cache file). In this case, packfiles can be loaded with a single allocation and pointer patches. `hkSerializeUtil::load` uses this fast path if available or `hkNativePackfileUtils` may be used directly.

Packfiles are loaded into your games using `hkSerializeUtil` (recommended) or `hkXmlPackfileReader` and `hkBinaryPackfileReader`. XML packfiles are platform independent; therefore XML packfiles can be created on any platform and loaded onto any platform. However, binary packfiles are platform specific, so they can only be loaded by a reader running on the specified target.

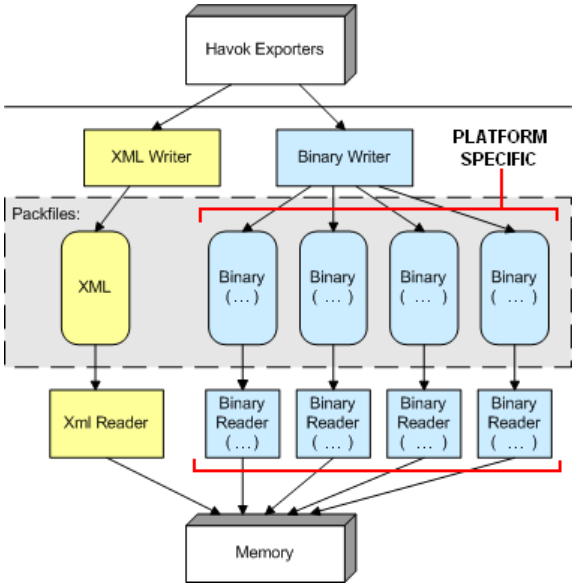



Figure 1.1: Loading packfiles.

#### 1.4.3.3 Deprecation of Packfile Versioning

Prior to Havok 7.0, we used to version the C++ objects directly. However this requires a copy of the metadata for each intermediate version as well as the file and executable versions. As the number of versions increased, this snapshot based approach became less viable. Havok 7.0 introduced a new delta based scheme which is considerably smaller. The old versioning system is enabled by default by `hkProductFeatures` as it calls `hkFeature_serializeDeprecatedPre700()`, and it can be used by linking with the hkCompat library. It may be disabled by defining `HK_EXCLUDE_FEATURE_SerializeDeprecatedPre700` before including `hkProductFeatures.cxx` in your project.

#### 1.4.3.4 Choice of Format

Regardless of the choice of format to be used in the final product, it is recommended that tagfiles be used as the default output format from tools. They are fundamentally more malleable than packfiles and can be converted losslessly to a platform specific format at a later date. We ship some command line utilities to convert between formats, see `Tools/PackfileConvert/AssetCc2`.

### 1.4.4 Details of Loading and Saving

#### 1.4.4.1 Tagfile Loading Steps

As you'll see tagfiles are strongly related to the `hkData*` classes. Briefly, `hkDataObject`s are property bags, `hkDataClass`es are the description of allowed properties and types and `hkDataWorld` contains a collection of both. These are the high level steps used by `hkSerializeUtil` to load a tagfile.

**Read the tagfile into a hkDataWorld.:** Tagfiles are self describing and the data in them is described in terms of the `hkDataObject` types. Thus the first stage of loading can be performed using only the input stream and a tagfile reader. Generally, we'll use a `hkBinaryTagfileReader` or a `hkXmlTagfileReader` and supply a `hkDataWorldDict` to read the data into. This is because the class descriptions in `hkDataWorldDict` are mutable which will be important for the next step, versioning.

**Modify the hkDataWorld. Typically this means versioning.:** The versions of the data on disk won't in general exactly match the version of the same objects compiled in to the executable. It would be possible to apply some ad hoc rules to match data which was slightly out of date, but we need to support arbitrary SDK changes and so have created a system to describe how such changes should be handled. The main method of interest here is `hkVersionPatchManager::applyPatches()`.

**Copy the hkDataWorld to native objects. This uses their hkClass descriptions.:** Now that the data closely resembles the compiled in version, it may be copied into its final location. `hkDataObjectUtil::toResource()` or `hkDataObjectUtil::toObject()` will be used here depending on if the "flattened" or "object" style allocation is preferred. This step uses the compiled in `hkClass` reflection data to decide how to populate the native C++ objects. We need to be able to find a `hkClass` instance given its name; this is the purpose of the `hkClassNameRegistry` which appears in some methods.

**Call "finish" constructors to make loaded objects usable.:** At this stage all the data is in place, but there are some parts of the object which cannot be directly initialized. For instance, virtual function tables need to be set up for polymorphic objects. Also, cached data or non-serialized data

may need to be initialized. To achieve this, any classes which require post-processing or 'finishing' declare a special purpose constructor (the one taking a `hkFinishLoadedObjectFlag`) which does not initialize its serialized already-initialized members. We call these constructors for each objects that has them.

#### 1.4.4.2 Tagfile Saving Steps

These are the steps `hkSerializeUtil` takes to save a tagfile:

**Create a hkDataWorld which contains the data to be saved:** Rather than copy the objects to be saved into a `hkDataWorldDict`, we use instead a `hkDataWorldNative`. This provides the same interface but has the advantage that it can reference the native data in place without copying. `hkDataWorldNative` is built on top of `hkClass` reflection.

**Traverse the world finding all data to be saved.:** The tagfile metadata appears before any data in the file so we first traverse the data, looking for any classes used. Whenever we encounter an object, a callback is raised to allow the object to be ignored or even replaced. See the object added listener section.

**Actually write the data now that all objects and types are known.:** The file header is written, followed by the metadata, followed by the objects in depth first order. The text tagfile writer shows the order of the objects and members in the output stream.

#### 1.4.4.3 Native Packfile Loading Steps

We also provide a special low level loader which loads from an in-memory packfile image into a single user-supplied buffer, discarding temporary data such as pointer patch tables. The loaded memory footprint is smaller than using the packfile reader directly. Also, using only this API for loading means that all the versioning code and generic loading code can be dead stripped. The following steps are taken:

**Memcpy the persistent parts of the packfile into a user supplied buffer.:** We don't need pointer patch tables and file container data to persist. We also store a list of destructors to call in the buffer.

**Apply pointer patches and call finish constructors.:** Iterate over the tables stored in the source buffer.

`hkSerializeUtil` uses this interface when possible. If the file is already in memory, using the `(const void*, int size)` loading interface gives the minimal amount of copying. Otherwise the data must be copied from the stream into a temporary source buffer.

Using the low level `hkNativePackfileUtils` allows the persistent buffer to be allocated by the user. See `hkNativeResource` for an example of wrapping the native interface in the common `hkResource` container.

`hkNativePackfileUtils::load(const void* packfileData, int dataSize)` allocates a new buffer to hold the loaded objects, and wraps it in a `hkResource`. This `hkResource` holds all of the contained objects and they will be destroyed when it is destroyed.

`hkNativePackfileUtils::load(const void* packfileData, int dataSize, void* outBuffer, int outBufferSize)` creates the objects in a user-supplied buffer, returning the top-level

object in the file. The buffer must be large enough to hold the generated objects (given by `hkNativePackfileUtils::getRequiredBufferSize(const void* packfileData, int dataSize);`), and the buffer must remain allocated for the lifetime of the objects. Call `hkNativePackfileUtils::unload(void* loadedBuffer, int bufferSize);` to destroy the objects before freeing the buffer.

`hkNativePackfileUtils::loadInPlace(const void* packfileData, int dataSize)` creates the objects in the same buffer that the packfile is stored in, returning the top-level object in the file. The generated objects will be smaller than the original buffer, so no additional permanent allocation is performed, and this may be more efficient if the buffer will remain allocated anyway. The packfile data is overwritten by the generated objects. The buffer must remain allocated for the lifetime of the objects. Call `hkNativePackfileUtils::unloadInPlace(void* loadedBuffer, int bufferSize);` to destroy the objects before freeing the buffer. A buffer loaded in place can not be reloaded after being unloaded.

#### 1.4.4.4 Flattened Memory

**Contiguous Block**

Objects loaded with the flattened style method are stored contiguously in memory. A `hkResource` (or derived `hkPackfileData`) instance contains a handle to the memory allocations made when the file was loaded. The memory containing the loaded data can be deallocated at any time by removing deleting the resource.

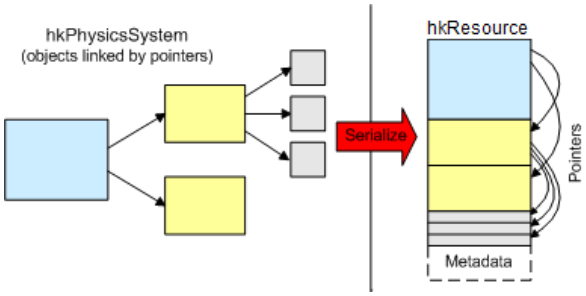



Figure 1.2: The internal structure of a flattened resource.

**hkArray resizing**

`hkArray`s are represented as a pointer and a size, with the array elements stored further down in the memory block. Some objects can contain dynamic `hkArray`s, which can be increased in size after being loaded. If a `hkArray` is increased in size its elements are copied elsewhere in memory to allow it to expand, as can be seen in the diagram.

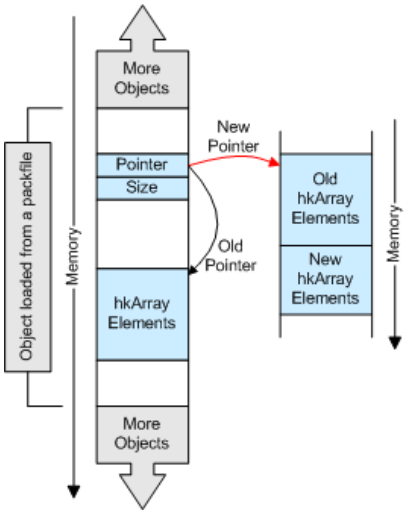



Figure 1.3: Memory re-allocation for arrays.

Therefore, objects that contain `hkArray`s and were loaded from a packfile must be parsed before they are deleted, and destructors called for any `hkArray`s found in case they have moved elsewhere in memory. The destructor checks the `hkArray`'s `DONT_DEALLOCATE_FLAG`, and if not set then the `hkArray` has moved and gets explicitly deallocated.

Note that once an array has moved its original memory is not referenced by anything and cannot be freed up until the section of memory that the packfile was loaded into is deallocated.

### Reference counted objects

Reference counted objects are usually freed when their reference count falls to zero. However, because multiple objects are packed into a single allocation when serialized, it is not possible to free individual objects. Therefore, calls to increment and decrement the reference count are ignored. More precisely, `addReference` and `removeReference` calls are ignored for objects which have `hkReferencedObject::m_memSizeAndFlags==0`.

### Destructors of packfile objects

When objects are loaded, we call the finish constructors (see Finishing). The `hkPackfileData` remembers these constructors and will automatically call the corresponding destructors when the data is destroyed. Note that we almost always call `removeReference` in destructors instead of an explicit `delete`, because the former correctly handles flattened and heap objects.

Usually memory owned by reference counted serialized objects should not be explicitly deallocated in the object destructor. Explicit frees generally would break the choice of flattened or object loading. Havok container classes (`hkArray`, `hkStringPtr` etc) have an internal flag which specifies whether the memory should be freed or not. As discussed above, `removeReference` is safe.

Note that if you have several files which reference each other, you will need to do cleanup in two passes - first call all the destructors (`hkResource::callDestructors()`), then free the memory (`hkResource::~hkResource()`). This avoid problems such as calling `removeReference()` on memory which has already been freed.

#### 1.4.4.5 Object Add Listeners

When generating the list of objects to be saved, we issue a callback for each object. This allows customization of exactly what is saved on a per instance basis. For instance when we encounter rigid bodies, we generally want to include them in the contents. However the one exception is if the body is the world's internal fixed rigid body, we don't want the internal body to appear in the file so set the pointer to null when saving.

Another case is when an object's members refers to non-serializable data. These members will usually have a reflection tag indicating they should be skipped for serialization. However this means that to load a file containing such an object requires the non-serializable parts to be recreated and reconnected. For debugging we often want the files to have minimal external dependencies, so we may replace these objects with self-contained ones, e.g. `hkpMeshShape`s are converted to `hkpStorageMeshShape`s.

When saving, we assume that objects won't be freed and will remain valid until the save completes. Thus if the listener provides a replacement, it will preserve the replacement until the listener is destroyed. See `hkpHavokSnapshot::ConvertListener` for a packfile listener example.

#### 1.4.4.6 Finish Constructors

Finish constructors are the very last step to make the objects usable. At this stage all the data is in place, but there are some parts of the object which cannot be directly initialized. For instance, virtual function tables need to be set up for polymorphic objects. Also, cached data or non-serialized data may need to be initialized. To achieve this, any classes which require post-processing or 'finishing' declare a special purpose constructor (the one taking a `hkFinishLoadedObjectFlag`) which does not initialize its serialized already-initialized members.

The special constructor is also used to extract the vtables for a form of RTTI. This is why the code is only executed when `flag.m_finishing` is nonzero. Virtual classes must have a finish constructor to initialize the vtable. Nonvirtual classes may implement it as needed.

```
hkpListShape::hkpListShape( class hkFinishLoadedObjectFlag flag )
:    hkpShapeCollection(flag)
,    m_childInfo(flag)
{
    if( flag.m_finishing )
    {
        setType(HKCD_SHAPE_TYPE_FROM_CLASS(hkpListShape));
        m_collectionType = COLLECTION_LIST;
    }
}
```

Since C++ doesn't allow taking the address of constructors, we take the address of tiny stub functions which call the constructors. Similarly we have a destructor stub. These are wrapped together in a `hkTypeInfo` and the association with class names is in a `hkTypeInfoRegistry`.

The following are required for a class finish constructor:

- Must chain to the parent's finish constructor (to initialize inherited class members correspondingly).
- Must chain to members' finish constructor (e.g. to prevent the default `hkArray` constructor being

called, virtual embedded class member).

- Must be executed only when `hkFinishLoadedObjectFlag::m_finishing` is not 0.

There are several hints that your finish constructor has errors. Arrays which are unexpectedly empty or `unexpectedly null hkRefPtr`s are the most common symptoms which result from not chaining the finish flag to all members. Luckily detection of many common errors has been automated in `hkSerializationCheckingUtils::verifyReflection()`. The `ReflectionVerify` demo shows usage of this utility.

#### 1.4.4.7 Stream Requirements

If you are using custom streams, note that the readers may require some optional interfaces to be implemented. See the `hkBase` stream section for a shortcut to implementing the mark interface.

| | | |
|---|---|---|
| `hkBinaryTagfileReader` | load | none |
| `hkBinaryTagfileWriter` | save | none |
| `hkXmlTagfileReader` | load | none |
| `hkXmlTagfileWriter` | save | none |
| `hkXmlPackfileReader` | loadEntireFile | **mark** |
| `hkBinaryPackfileReader` | loadEntireFile | none |
| `hkBinaryPackfileReader` | loadEntireFileInplace | n/a |
| `hkBinaryPackfileReader` | loadSection | **seek** |

#### 1.4.4.8 In-memory Streams

All `hkSerializeUtil` load methods have a variant which takes a buffer input. The following shows how to write to a resizable buffer.

```
// Poor-man's clone - saving and reloading from an in-memory buffer.
Object* object = ...;
hkArray<char> buf;
hkSerializeUtil::save( object, ObjectClass, hkOstream(buf).getStreamWriter() );
Object* clone = hkSerializeUtil::loadObject<Object>( buf.begin(), buf.getSize() );
```

#### 1.4.4.9 Imported/Exported Objects

A packfile may contain pointers to data which is not contained in the packfile, e.g. created in code or located in another packfile. We call each pointer an *Import*. Imports are identified with plain C strings. Similarly a packfile may expose some of its contained objects using an *Export*. Use `hkPackfileData::getImportsExports()` or `hkPackfileData::getImportsExportsInPlace()` to examine the Imports and Exports.

We do not impose a policy of how these references should be resolved, but a sample linker which places all identifiers into a single global namespace is included in `demos/Utilities/Asset/ResourceLinker.h`. The SymbolImportExport demo shows its usage.

There are several approaches to setting up imports/exports. If you have relatively few objects (e.g. mesh

shapes), a simple option is to call `addExport` and `addImport` before `setContents`. For an indeterminate number of objects, you can use the `addObjectCallback` to select by adddress or type.

#### 1.4.4.10 Exporting Platform Specific Packfiles

You can use `hkSerializeUtil` or `hkBinaryPackfileWriter` to export packfiles to platforms with different binary layouts. Do this by setting `hkPackfileWriter::Options::m_layout` to the desired platform. `hkStructureLayout` contains several static layouts corresponding to supported compiler/platform combinations.

```
hkOstream ostream(FILENAME);
hkBinaryPackfileWriter writer;
writer.setContents(&simpleobject, SimpleObjectClass);
hkPackfileWriter::Options options;


// export the data in PlayStation(R)2 gcc3.2 format
options.m_layout = hkStructureLayout::Gcc32Ps2LayoutRules;
writer.save(ostream.getStreamWriter(), options);
```

If your platform does not have a static entry in `hkStructureLayout`, you can find the correct values by examining the `hkStructureLayout::HostLayoutRules` on the target platform. This always corresponds to the running programs rules.

#### 1.4.4.11 Embedded Metadata

Packfile format assets can optionally include metadata for all contained objects. This is the default was used in the past for versioning. Currently, tagfiles are a better option for files which may need versioning. For packfiles which don't need versioning, metadata is redundant and may be stripped by setting `hkPackfileWriter::Options::m_writeMetaInfo` to false.

Tagfiles always contain compressed metadata.

#### 1.4.4.12 Type of Contents Object

For consistency all Havok exporters export the same top level data structure. This data structure can be found in `Common/Serialize/Util/hkRootLevelContainer.h`; it simply contains an array of arbitrary named objects (`hkVariants`). Each Variant is a structure containing a pointer to some data together with its `hkClass`. Utility routines (findObjectByType and findObjectByName) are provided to make searching this root level container easier.

Note however that the top level object is not fixed and any reflected object may be used as the contents. E.g. the snapshot utility uses a `hkpPhysicsData` object.

### 1.4.5 Reflection

#### 1.4.5.1 Type Registration

**Uses of the Registry**

Much of the serialization infrastructure relies on having registries of `hkTypeInfo` and `hkClass` objects. The definitions for the `hkTypeInfo` and `hkClass` objects associated with a particular Havok or external class using reflection are generated using `Tools/Serialize/generateReflections.py` (see Generating the Reflection Data).

**Saving: vtable to hkClass:** When saving polymorphic types, we need to know their exact type in order to save them correctly. This is done by registering each type's virtual table in a `hkVtableClassRegistry` along with its `hkClass` instance. Note that a single virtual table cannot be registered with multiple classes, and this situation can occur when certain compilers generate polymorphic types that share the same virtual table. When inheriting from `hkReferencedObject`, one workaround is to override the `getClassType` virtual function to force a unique virtual table.

**Loading: name to finish function:** When loading polymorphic types or types which need post-load "finishing", we need to be able to call an initializing function on the object, based on the type name. `hkTypeInfoRegistry` stores this information.

To ease the burden of creating and initializing these registries, there is the `hkBuiltinTypeRegistry` singleton which contains the above registries and is created once using a compiled-in list of types. Most serialization functions have variants which take explicit registries and ones which implicitly use the built-in registry, e.g. `hkPackfileWriter::setContentsWithRegistry()` and `hkPackfileWriter::setContents()`. Other methods implicitly use the builtin registries, i.e. `hkSerializeUtil::save()`.

**Register All Types**

To start with, the easiest thing to do is to register all available classes. To do this use the following code snippet. Note that the `hkProductFeatures.cxx` file will automatically perform this step if the feature is enabled. This can be disabled by defining `HK_EXCLUDE_FEATURE_RegisterReflectedClasses` before including `hkProductFeatures.cxx`.

**Note:**
Usage of `HK_EXCLUDE_FEATURE_RegisterReflectedClasses` relies on the linker being smart enough to discard the `hkBuiltinTypeRegistry` singleton object completely. Most modern linkers will be capable of doing this, but to obtain the desired behavior the Havok Base library must always be linked before the Havok Serialize library. If a linker is not capable of discarding the `hkBuiltinTypeRegistry` singleton object, it won't be possible to use `HK_EXCLUDE_FEATURE_RegisterReflectedClasses`.

```
#include <Common/Base/KeyCode.h>
#define HK_CLASSES_FILE <Common/Serialize/ClassList/hkKeyCodeClasses.h>
#include <Common/Base/Config/hkProductFeatures.cxx>
```

**Note:**
If `HK_CLASSES_FILE` is not defined, `hkKeyCodeClasses.h` will by default be included by `hkProductFeatures.cxx`. This code defines the serialization type registry static data, i.e. `hkBuiltinTypeRegistry::StaticLinkedTypeInfos` and `hkBuiltinTypeRegistry::StaticLinkedClasses`. Note that `hkProductFeatures.cxx` must be included in your executable to avoid link errors.

### Custom Registration

Registering all types is easy but it can lead to bloat because storing virtual tables can cause linkers to link all the methods referenced from the virtual table. You can strip these methods by linking only the used types.

To set up the class registry, create a file which contains declarations of the types used, e.g. `MyClasses.h`.

```
// these are always used
HK_STRUCT(HK_EXPORT_COMMON,hkClass)
HK_STRUCT(HK_EXPORT_COMMON,hkClassEnum)
HK_STRUCT(HK_EXPORT_COMMON,hkClassEnumItem)
HK_STRUCT(HK_EXPORT_COMMON,hkClassMember)

// declare non polymorphic types with HK_STRUCT
HK_STRUCT(HK_EXPORT_MYGAME,SimpleObject)

// declare polymorphic types with HK_CLASS
// e.g. if BaseObject is an abstract class and
// PlayerObject, EnemyObject are concrete derived types.
HK_CLASS(HK_EXPORT_MYGAME,PlayerObject)
HK_CLASS(HK_EXPORT_MYGAME,EnemyObject)

// abstract base types should be registered too
// HK_ABSTRACT_CLASS(HK_EXPORT_MYGAME,BaseObject)

// We can use the usual preprocessor directives
#if defined(EXTRA_TYPES)
#  include "MyExtraTypes.h"
#endif
```

Next, use the super macro (also known as x-macro) file `hkProductFeatures.cxx` to generate the correct definitions. Alternatively, you may include `hkBuiltinTypeRegistry.cxx` separately from `hkProductFeatures.cxx`. Again, note that `hkProductFeatures.cxx` must be included in your executable to avoid link errors.

```
// place this somewhere in your code (e.g. MyClasses.cpp)
#define HK_CLASSES_FILE "MyClasses.h"
#include <Common/Base/Config/hkProductFeatures.cxx>
```

**Note:**
The lists in `Common/Serialize/ClassList/*.h` provide a good starting point for your custom class lists. You can remove an entire library's classes by commenting out a single line.

**Note:**
Do not use `#pragma once` in your custom classes file as this file is included multiple times by `hkBuiltinTypeRegistry.cxx`.

Alternatively, you may also register your custom types at runtime using `hkBuiltinTypeRegistry::addType(const hkTypeInfo*, const hkClass*)`. You must provide pointers to the `hkTypeInfo` and `hkClass` static data describing your custom type. The static data is usually generated by the serialization scripts. Make sure that Havok is initialized before you access the `hkBuiltinTypeRegistry` singleton.

```
#include <Common/Serialize/Util/hkBuiltinTypeRegistry.h>

// Forward declaration of hkClass and hkTypeInfo for
// the PlayerObject and EnemyObject custom classes.
extern const hkClass BaseObjectClass;
extern const hkClass PlayerObjectClass;
extern const hkTypeInfo PlayerObjectTypeInfo;
extern const hkClass EnemyObjectClass;
extern const hkTypeInfo EnemyObjectTypeInfo;

// Register the custom classes now.
// Make sure the Havok environment is initialized before you register the classes.

// Abstract class has no hkTypeInfo, so register hkClass only.
hkDefaultClassNameRegistry::getInstance().registerClass(&BaseObjectClass);

// Register normal classes.
hkBuiltinTypeRegistry::getInstance().addType(&PlayerObjectTypeInfo, &PlayerObjectClass);
hkBuiltinTypeRegistry::getInstance().addType(&EnemyObjectTypeInfo, &EnemyObjectClass);
```

### 1.4.6 Versioning

The versioning system is solely responsible for the backward compatibility of asset loading. One of the design goals for the system is that objects need not carry any baggage from previous versions, that there is a clean separation between the current and previous versions. Another goal is correctness. That is, that it is as bad to silently and implicitly modify the data on loading as it is to fail to load a valid file.

The system is based on the idea of patches. These are small data bundles which describe the metadata changes from version to version, i.e. member added/removed/renamed. Additionally for changes which are not easily described by data, function calls may be included in the patch to perform the necessary updating. Each component of the patch is applied sequentially in the order it appears. The following shows an example of the patch needed when a `Square` class is refactored to be a `Rectangle` with width and height.

```
// old class name, old version, new class name, new version
HK_PATCH_BEGIN("Square", 4, "Rectangle", 1)
  // old and new member name
  HK_PATCH_MEMBER_RENAMED("sideLength", "width")
  // new member name and type info
  HK_PATCH_MEMBER_ADDED("height", TYPE_REAL, HK_NULL, 0)
  // an external C function to call to initialize the new member
  HK_PATCH_FUNCTION(Square_4_to_Rectangle_1)
HK_PATCH_END()
```

The snippet shows the patch mini-language. Preprocessor macros expand this into blocks of static const C++ data. The patch may be described with raw C++ code, but it's considerably more error prone.

In the SDK all patches are available. This means that ideally it is possible to load an asset that was originally created with the first Havok version. In most cases all assets used will be produced with a recent Havok version; in this case all the additional patching code registered will merely increase the executable size without any advantage. It is possible to use the `HK_SERIALIZE_MIN_COMPATIBLE_VERSION` preprocessor definition to select a specific minimum compatible version (oldest compatible version). See the Code Size Customization section for more information.

#### 1.4.6.1 Patch Types

**Simple Patch Types**

The patch itself is a `hkVersionPatchManager::PatchInfo`. Each component of the patch corresponds with an enum value and struct in `hkVersionPatchManager`.

**HK_PATCH_MEMBER_ADDED(const char* name, hkDataObject::Type type, const char* className, in**
Add the member with the given name and type. *className* is only used if the member does not have an associated class. *count* is the number of array items in the member, zero if the member is not an array, e.g. 5 for `int m_vals[5]`.

**HK_PATCH_MEMBER_ADDED_INT(const char* name, hkInt64 value):** Add an integer member with the given default value. This value will be applied to the class member if it does not already have a value on serialization.

**HK_PATCH_MEMBER_ADDED_REAL(const char* name, hkReal value), HK_PATCH_MEMBER_A**
Similiarly, real and bool values with a default.

**HK_PATCH_MEMBER_ADDED_VEC_4(NAME, ...), HK_PATCH_MEMBER_ADDED_VEC_12(N**
Add `hkVector4`, `hkQuaternion`, `hkTransform` or `hkQsTransform` with a default value. The value is a list of floats, in the order in which they are initialized.

**HK_PATCH_MEMBER_ADDED_CSTRING(NAME, ...), HK_PATCH_MEMBER_ADDED_POINTE**
Add strings or pointers, with a default. Pointer defaults are not checked.

**HK_PATCH_MEMBER_ADDED_VEC_4_IDENTITY(NAME), HK_PATCH_MEMBER_ADDED_V**
Add `hkVector4`, `hkTransform`, `hkQsTransform` and `hkQuaternion` with a default value of an identity transform.

**HK_PATCH_MEMBER_REMOVED(const char* name, hkDataObject::Type type, const char* classNam**
Remove a member; identical parameters as for member added.

**HK_PATCH_MEMBER_RENAMED(const char* oldName, const char* newName):**
Rename a member.

**HK_PATCH_FUNCTION(hkVersionPatchManager::Function func):** Call the named function which will modify the `hkDataObject`.

**HK_PATCH_PARENT_SET(const char* oldParent, const char* newParent):** Change the inheritance hierarchy. Null means no parent.

**HK_PATCH_DEPENDS(const char* className, int version):** The named class must be at the named version when this patch is applied.

**HK_PATCH_MEMBER_DEFAULT_SET_XXX(const char* name, ...):** Follows the same format as the various `HK_PATCH_MEMBER_ADDED` macros except to change the default for an existing member.

See `Common/Compat/Patches` for many examples of patches.

**Composite Patches**

Occasionally, some changes are not described by the built-in simple patch types. In these cases we can describe a changes with several of the basic types. For instance, there is no patch to describe a single floating point value changing into a vector value. However here is a composite one for this case:

```
HK_PATCH_BEGIN("SomeObject", 0, "SomeObject", 1)
  HK_PATCH_MEMBER_ADDED("strength3", TYPE_VEC_4, HK_NULL, 0)
  HK_PATCH_FUNCTION(SomeObject_0_to_1) // set x,y,z to the original float value, w to 0
  HK_PATCH_MEMBER_REMOVED("strength", TYPE_REAL, HK_NULL, 0)
  HK_PATCH_MEMBER_RENAMED("strength3", "strength")
HK_PATCH_END()
```

### Version Numbers

When objects are serialized, we save the version number with them. This number is taken from the objects `hkClass` and is an arbitrary 32-bit integer. By convention we use small monotonically increasing values. The values themselves are declared in the class declaration (see metadata annotations). E.g.

```
class hkpEntity : public hkpWorldObject
{
  public:
    //+version(2)
    ...
```

The version number should be manually updated at the time a patch is committed. It is not necessary to update the version number for every minor change, but the system will warn on load when there seem to be changes with no corresponding patch.

### Function Patches

Functions called by patching need to accept a `hkDataObject` by reference and return void. They modify the object in-place. Syntax is similar to associative arrays or dictionaries in dynamic languages. Operator= is overloaded for many types, however the right hand side of assignments need explicit type hints (`.asXXX`) to satisfy C++'s static typing.

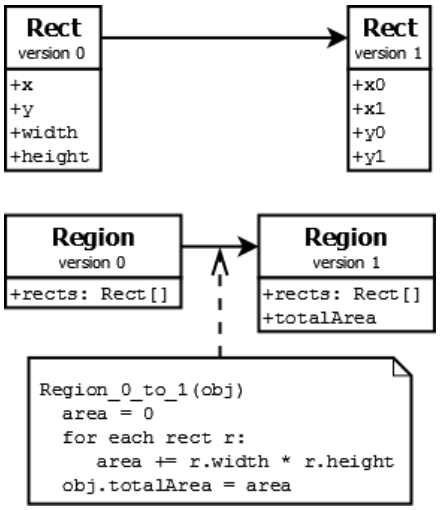
For instance the patch function for the previous "composite" patch would be:

```
static void SomeObject_0_to_1(hkDataObject& obj)
{
  hkDataArray strength3 = obj["strength3"].asArray();
  hkReal s = obj["strength"].asReal();
  strength3[0] = s;
  strength3[1] = s;
  strength3[2] = s;
  strength3[3] = 0.0f;
}
```

### Patch Dependencies

By convention and for human ease, patches are roughly in chronological order, though the order of patches in a file and the order in which they are registered is unimportant. This is necessary to support multi product patches and custom non-Havok classes. Let's ignore for a moment the obvious order of a chain of patches needed to bring a single object to the current version. The order of application of patches for different types of object, does not matter *except* when there is a function component in the patch.

When a function is called, it must explicitly declare which object versions it expects to find, if any. In this example, the patch for Region must declare that it expects Rect version 0. If a dependency is omitted, the patches may be applied in arbitrary order and fail at runtime because Rect has been updated before the function call and r.width and r.height no longer exist.




```
HK_PATCH_BEGIN("Region", 0, "Region", 1)
  HK_PATCH_MEMBER_ADDED("totalArea", TYPE_REAL, HK_NULL, 0)
  HK_PATCH_FUNCTION(Region_0_to_1)
  HK_PATCH_DEPENDS("Rect", 0)
HK_PATCH_END()
```

Dependency patches must be included whenever a function accesses any variable which is not declared in the class itself, not including inherited members. If members from a parent class are accessed, the dependency must be specified like any other.

#### 1.4.6.2 Creating Patches

**What's New?**

To reduce the burden of keeping patches up to date, the system has a utility to introspect the patches and currently registered classes and tell you what is missing, if anything. See the `PatchesVersionVerify` demo for an example of using this utility. Note that it may take several iterations of examining the output and following recommendations before succeeding. The utility tries to pinpoint only the next immediate issue since a single issue can have cascading effects.

**Registering Patches**

Assuming you use the macros and your patches are in `MyPatches.cxx` (Havok uses the `.cxx` extension for files which are included by `.cpp` files), define `HK_PATCHES_FILE` and include the super macro (x-macro) patch manager file. The macros assume the existence of a local `hkVersionPatchManager` variable named `man`. The patches will be added to this manager.

Order of registration of patches in unimportant since dependencies are computed when patches are

applied.

```
// any functions used by patches

void HK_CALL registerMyPatches(hkVersionPatchManager& man)
{
#define HK_PATCHES_FILE <MyPatches.cxx>
#include <Common/Serialize/Version/hkVersionPatchManager.cxx>
#undef HK_PATCHES_FILE
}
```

By convention, we create a file containing version functions, if any, followed by a patch registration function. This allows the functions to be static and they need not be forward declared.

### 1.4.7 Serialization and Physics 2012

When serializing Havok Physics 2012 objects, only the properties needed to recreate the object later on are saved, such as position, orientation, dimensions and impulses, but not contact points. There are a number of classes in the Havok Physics 2012 SDK worth discussing with regard to serialization.

#### 1.4.7.1 The hkpRigidBody and hkpPhantom Classes

`hkpRigidBody` and `hkpPhantom` are two serializable Havok Physics 2012 classes containing `hkArray`s which are resized during simulation. Particular care needs to be taken when deallocating a section of memory loaded from a packfile if it contains either of these objects. As described in the Memory Considerations section, the `hkArray` destructor needs to be called for their arrays in case they have increased in size and are now located elsewhere in memory. `hkResource` will handle this for you unless you have explicitly disabled it.

#### 1.4.7.2 Userdata pointers

Some objects contain userdata pointers. They are typeless and may contain integer, float or pointer values. By default Havok saves them as zeros. If in fact they are pointers to serializable objects, you can have the system treat them as such.

To do this, edit the header file and add an `//+overridetype()` comment after the `member` element. For instance if the pointer is really a shape pointer, you could use the following. After making this change, you will need to run the `Tools/Serialize/generateReflections.py` script and recompile the generated `ReflectedClasses.cpp`.

```
hkUlong m_userData;
      //+overridetype(hkpShape*)
```

Alternatively, you can edit the reflection generated data in `Classes/Reflections/*Reflection.cpp` directly, e.g. `hkpPhysicsSystemReflection.cpp`. Note that running `generateReflections.py` overwrites

the `*Reflection.cpp` files.

```
// This is the default, as shipped.
{ "userData", HK_NULL, HK_NULL, hkClassMember::TYPE_ULONG, hkClassMember::TYPE_VOID, 0, 0, HK_OFFSET_OF(
  hkpWorldObject,m_userData), HK_NULL }
// This sets the userdata to be a shape pointer as in the previous example.
{ "userData", &hkpShapeClass, HK_NULL, hkClassMember::TYPE_POINTER, hkClassMember::TYPE_STRUCT, 0, 0,
  HK_OFFSET_OF(hkpWorldObject,m_userData), HK_NULL }
```

**Note:**
When overriding a pointer as an integer, you'll need to be careful with 32/64 bit issues. Overriding a type as a type of different size can cause the binary writer to fail. In the case of `hkpWorldObject`, the alignment requirements of adjacent variables ensures that the object layout is still computed correctly for 64 bit machines.

#### 1.4.7.3 The hkpPhysicsSystem Class

A `hkpPhysicsSystem` is a container class which stores collections of objects and their associated actions and constraints for serialization together. Conceptually, a physics system will be a single in-game entity that is always considered as a whole, such as a ragdoll or a vehicle. Even if two ragdolls are lying on top of each other, and hence in the same simulation island, they are separate game entities and can be considered as two separate physics systems for the purposes of serialization.

`hkpPhysicsSystems` are made up of just four component types: `hkpRigidBody`, `hkConstraint`, `hkpAction` and `hkpPhantom`. For example, ragdolls consist of a number of rigid bodies (usually eleven: two for each limb and the torso, and one for the head), and the constraints that hold them together, whereas vehicles are generally defined by a rigid body, a phantom, and an action. Collections of objects such as piles of crates or barrels can also be stored in a single `hkpPhysicsSystem`.

#### 1.4.7.4 The hkpPhysicsData Class

The `hkpPhysicsData` class is the top level container class for Havok Physics 2012 data. It contains a `hkpWorldCinfo` and a `hkArray` of `hkpPhysicsSystems`, allowing anything from groups of ragdolls to entire game levels to be serialized together. A `hkpPhysicsData` scene can be captured and serialized at any time via `hkpHavokSnapshot`.

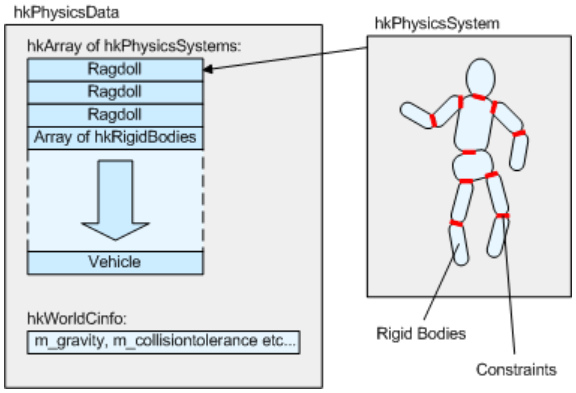



Figure 1.4: hkpPhysicsData and hkPhysicsSystems

The SimpleLoadApi demo (in Common/Api/Serialize/SimpleLoad) demonstrates how to load Havok Physics 2012 data.

```
// Load the file
hkSerializeUtil::ErrorDetails loadError;
m_loadedData = hkSerializeUtil::load(HK_GET_DEMOS_ASSET_FILENAME(path.cString()), &loadError);
if ( !m_loadedData )
{
#ifdef HK_PLATFORM_RVL
    if ( loadError.id == hkSerializeUtil::ErrorDetails::ERRORID_DEPRECATED_NOT_INITIALIZED )
    {
        HK_WARN_ALWAYS(0xa6451543, "Could not load file(" << path << "). The error is:\n" <<
          loadError.defaultMessage.cString() );
        return;
    }
    else
#endif
    {
        HK_ASSERT3(0xa6451543, m_loadedData != HK_NULL, "Could not load file (" << path << "). The error
          is:\n" << loadError.defaultMessage.cString() );
    }
}

// Get the top level object in the file, which we know is a hkRootLevelContainer
hkRootLevelContainer* container = m_loadedData->getContents<hkRootLevelContainer>();
HK_ASSERT2(0xa6451543, container != HK_NULL, "Could not load root level object" );

// Get the physics data
hkpPhysicsData* physicsData = static_cast<hkpPhysicsData*>( container->findObjectByType(
  hkpPhysicsDataClass.getName() ) );
HK_ASSERT2(0xa6451544, physicsData != HK_NULL, "Could not find physics data in root level object" );
HK_ASSERT2(0xa6451535, physicsData->getWorldCinfo() != HK_NULL, "No physics cinfo in loaded file - cannot
  create a hkpWorld" );

// Create a world and add the physics systems to it
{
    m_world = new hkpWorld( *physicsData->getWorldCinfo() );
    m_world->lock();

    // Register all collision agents
    hkpAgentRegisterUtil::registerAllAgents( m_world->getCollisionDispatcher() );

    // Add all the physics systems to the world
    for ( int i = 0; i < physicsData->getPhysicsSystems().getSize(); ++i )
    {
        m_world->addPhysicsSystem( physicsData->getPhysicsSystems()[i] );
    }
    m_world->unlock();
}
```

#### 1.4.7.5 Havok Tools Output

The Havok tools for Maya and 3ds Max group Havok Physics 2012 objects into an instance of hkpPhysicsData. This hkpPhysicsData object contains a hkpWorldCinfo and a hkArray of hkpPhysicsSystems as usual, with the physics objects serialized from the exporters grouped into hkpPhysicsSystems in the following order:

1. Fixed Objects: All fixed objects are saved into the first hkpPhysicsSystem. Some fixed objects (e.g. the base of a chandelier) may be attached to the world and also conceptually belong

to a `hkpPhysicsSystem` (the chandelier). To avoid such systems being grouped into one large `hkpPhysicsSystem` all fixed objects are isolated from any systems they would otherwise be a member of.

2. Dynamic Objects: All dynamic objects without constraints (e.g. crates, barrels etc.) are stored in the second `hkpPhysicsSystem`.

3. Keyframed Objects: All keyframed objects (such as moving platforms etc.) make up the third `hkpPhysicsSystem`.

4. Other Physics Systems: A `hkpPhysicsSystem` is created for each set of objects in the scene linked by constraints, such as ragdolls, vehicles etc. Any groups of objects constrained to a fixed object are saved as a `hkpPhysicsSystem` without the fixed object (e.g. the aforementioned chandelier is saved as a `hkpPhysicsSystem` without its base).

5. Phantoms: Any remaining standalone phantoms are grouped together in the final `hkpPhysicsSystem`.

#### 1.4.7.6 Snapshot Utility

Havok includes a snapshot utility which can easily load and save your world. The utility takes care of most of the underlying details.

**Saving a snapshot of the world**

Note that when the file is being saved, the snapshot may print warning messages to the console if it encounters objects which it does not understand. See troubleshooting.

```
// Save the world into the file at "path"
static hkResult saveWorld( hkpWorld* world, const char* path, bool binary )
{
    hkOstream outfile( path );
    return hkpHavokSnapshot::save(world, outfile.getStreamWriter(), binary);
}
```

**Loading a world snapshot**

Note the `allocsOut` parameter for tracking allocations. See Memory Considerations

```
// Loads snapshot at "path".
static hkpWorld* loadWorld( const char* path, hkPackfileReader::AllocatedData** allocsOut )
{
    hkIstream infile( path );
    hkpPhysicsData* physicsData = hkpHavokSnapshot::load(infile.getStreamReader(), allocsOut);
    return physicsData->createWorld();
}
```

**Troubleshooting snapshots**

You can verify that your snapshot worked by changing the path in the `api/serialize/SimpleLoad` demo to point to your snapshot file. If the snapshot does not load, it is most likely that when saving,

the snapshot encountered an object which it did not know how to deal with. In this case, the snapshot warns and replaces the object pointer with a null pointer.

The snapshot output is configurable using the `hkpHavokSnapshot::Options` structure. This should be initialized using a combination of the following options OR-ed together:

- `SNAPSHOT_BINARY_TAGFILE`: The default output format.
- `SNAPSHOT_TEXT`: Output an XML file. This outputs an XML tagfile by default, or a packfile if selected. The XML format is human-readable and editable.
- `SNAPSHOT_PACKFILE`: Output a packfile.

Havok serialization uses objects vtables as type identifiers. When a polymorphic pointer is encountered (i.e. `hkpShape*`), we check a registry of vtable to `hkClass` instances to determine its exact type and thus how to save it.

If some of your object types are not in the registry (`hkVtableClassRegistry`) you have several choices.

- Register it as its base type.

  This is the quickest method to implement and is a good choice if you are snapshotting for debugging and are not particularly interested in saving the extra information in derived classes.

  E.g. you have a class `MyRigidBody` which inherits from `hkpRigidBody`. You can cause the snapshot to treat them as if they were rigid bodies using the following code.

```
hkVtableClassRegistry* reg = hkBuiltinTypeRegistry::getInstance().getVtableClassRegistry();
reg->registerVtable( *(void**)anInstanceOfMyRigidBody, &hkpRigidBodyClass);
```

- Convert it to another type.

  If there is no suitable base type (e.g. you derive directly from `hkpShape`) or your shape has data which you would like to save, you can save a proxy object instead.

  When objects are added to a packfile, there is a callback fired which allows one to modify or replace objects. For instance some classes (`hkpExtendedMeshShape`) contain external data not owned by Havok. When saving, we can convert such types to "self contained" equivalents. (`hkpStorageMeshShape`).

  We can even change a saved objects type entirely. For instance to replace `MyShape` instances with `hkpCapsuleShape` instances.

```
hkVtableClassRegistry* reg = hkBuiltinTypeRegistry::getInstance().getVtableClassRegistry();
const hkClass MyShapeClass(0,0,0,0,0,0,0,0,0,0); // dummy hkClass
reg.registerVtable( *(void**)anInstanceOfMyShape, &MyShapeClass);
// hkpHavokSnapshot::save( ... );
```

```
// in hkpHavokSnapshot.cpp
void StorageListener::addObjectCallback( ObjectPointer& ptrInOut, ClassPointer& klassInOut )
{
    extern const hkClass MyShapeClass;
    if( hkpExtendedMeshShapeClass.isSuperClass(*klassInOut) && klassInOut != &
      hkpStorageExtendedMeshShapeClass )
    {
        // convert meshes to storage meshes
    }
    else if( MyShapeClass.isSuperClass(*klassInOut) )
    {
        ptrInOut = new hkpCapsuleShape( myShape->getVertex(0), myShape->getVertex(1), 1);
        klassInOut = &hkpCapsuleShapeClass;
        m_storage.pushBack(ptrInOut); // we own the new proxy capsule shape
    }
}
```

- Add serialization information.

  This is the highest fidelity method as your object is saved as it exists in memory. This is the best method if you need to reload the scene from a saved snapshot.

  This topic has its own section.

### 1.4.8 Using the Infrastructure

This section describes how to make your custom class work with Havok's serialization infrastructure.

Havok serialization is an *object reflection*-based system. Each object type has a data block which describes the members of such objects. Object loading, saving and manipulation is done in a generic fashion using this reflection data. An overview of the reflection system is shown below. In this example, the reflected data for the `hkBase` project is being regenerated. The first step is to locate all of the header files in the project. These are checked by `generateReflections.py`, and any that are out of date are re-parsed using the Havok parser and added to the per-project reflection database (`reflections.db`). The various output plugins are then run on the database, to generate the output files, in the `Classes/` directory. The parser is automatically called by the reflection scripts and is not designed to be called directly.

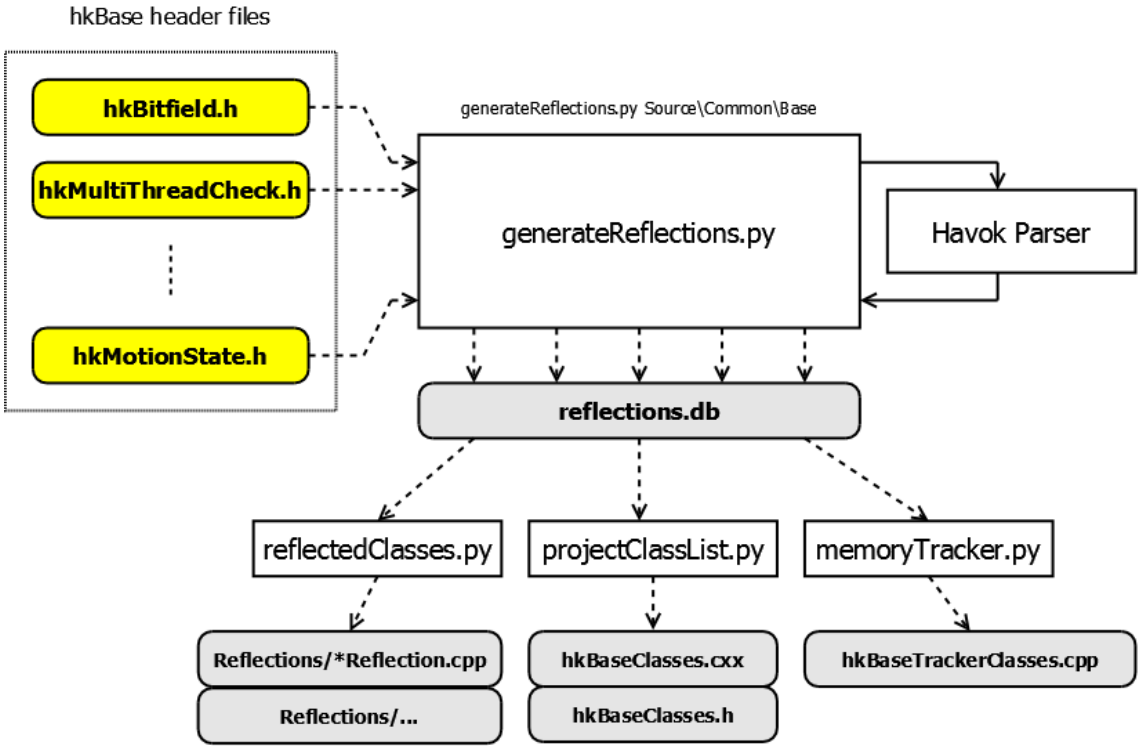



Figure 1.5: Infrastructure Overview

Internally, `generateReflections.py` uses a special compiler based on `LLVM` and `clang` to produce a database of reflected classes for each project (`reflections.db`) which is then used by `generateReflections.py` to produce the final files.

#### 1.4.8.1 Python

The scripts used in the serialization system are written in Python, and require at Python version 2.5, 2.6 or 2.7. Python 3 is not supported. If `Cygwin` is installed for Microsoft Windows, ensure that the native windows Python appears in the system path before any `Cygwin` version.

#### 1.4.8.2 Compiler independent types

The size and alignment characteristics of some types (most notably bool, enum and bitfields) vary between compilers or with compiler options. We define several types to combat this.

- `hkBool`: a bool which is always 1 byte in size.

  Note that while `hkBool` is typically used for object member variables, `hkBool32` (a typedef for int) is sometimes used in functions and local variables as it is more optimization friendly.

- `hkEnum<enum,storage>`: an enumeration with explicitly defined size.

  `hkEnum` stores exactly one value of the enumeration. The storage type can be 8,16 or 32 bits.

- `hkFlags<enum,storage>`: a fixed-sized bitfield of named values.

`hkFlags` stores bitwise combinations of the values defined in the enumeration. The storage type can be 8,16 or 32 bits.

#### 1.4.8.3 The Havok Parser

Havok makes extensive use of reflection in serialization and other systems such as memory reporting. Rather than requiring the user to generate the reflection data, we extract it directly from the header files, using a custom-written header parser. This is based on the LLVM C++ compiler, and therefore requires each header file to contain valid C++.

**Installation and Configuration**

The parser executable should be unzipped and copied into any directory. The environment variable `HAVOK_PARSER_ROOT` should be set to this directory (the base directory, not the `/bin` directory inside the distribution).

The parser executables are 32-bit applications, compiled for Windows, Mac and Linux. For 64-bit Linux distributions, it may be necessary to install 32-bit versions of the system libraries and headers, please consult your distribution documentation for more information. On non-Windows platforms it may be necessary to set the file permissions to allow execution of `clang-interpreter` by the current user.

The Havok parser is essentially a C++ compiler (LLVM) frontend and may require some configuration. In order to pass extra configuration, create a file named `havokParserConfig.txt` in the same directory as the parser executable, for system-wide configuration, or in the root directory of a project, for project-specific configuration (both locations are searched and both files will be read if both are present). Each line of this file is read as a command-line argument, and is inserted directly into the parser command line. See http://clang.llvm.org/ for more information, however in general, this should only involve setting paths or defining macros:

- `-DMACRO=value`: Define a macro.
- `-I/path/to/include`: Add a path to the include search path.

Note that only one item should appear on each line.

Please note that using our LLVM-Clang based parser imposes some limitations on the source code that it will examine:

- Standard C++ syntax must be used for `#include` preprocessor directives, (i.e. use `#include <Common/Base/hkBase.h>`, not `#include <Common\Base\hkBase.h>`).
- Files should be included in the same way in all header files. If you use the Havok `Source/` directory as base include dir, then all includes should be like `#include <Common/Base/hkBase.h>` and not a mixture of `#include <Common/Base/hkBase.h>` and other paths `#include "../../../Base/hkBase.h"`.

**Custom Project Configuration: reflectionSettings.cache**

More advanced configuration may be carried out by creating a file named `reflectionSettings.cache` in the root directory of the project. The parser compiles each file and must have enough information to

access all required header files. Havok projects already contain this file, user projects may need to add one. This file contains a list of settings, in the form of a Python dictionary. There are four configuration values that can be set from this file:

- `addIncludePath`: This is a list of extra directories to add to the search path while searching for header files.
- `setPchFile`: Set a precompiled header file. While precompiled headers are not used by the parser, this file is automatically included for each file in the project.
- `setPrefix`: Set the prefix to prepend to any generated files. This may be used to distinguish between files from different projects.
- `addParserExcludeDir`: Specify a list of directories relative to the `reflectionsSettings.cache` file that should not be parsed.

A sample `reflectionSettings.cache` file is shown below:

```
{'addIncludePath': ['C:/Havok/hk2010_1_0/Source', 'C:/Code/Src'], 'setPchFile':
['stdafx.h'], 'setPrefix': ['hkb'], 'addParserExcludeDir': ['Spu']}
```

The file must take this format. The brackets and quotes are all required syntax. Extra includes are added inside the square brackets after `addIncludePath`, as a comma separated list.

Once one `reflectionSettings.cache` file is encountered while searching a directory, the build system assumes that all projects contain this file and it will ignore directories that do not.

**Troubleshooting**

LLVM is a highly standards-conforming C++ compiler and may produce warnings for code that is permitted by other compilers. In order to exclude files from parsing, either add the directory to the `setParserExcludeDir` list, or add the string `//HK_REFLECTION_PARSER_EXCLUDE` to the file. In either case, the file will not be parsed. Note that reflected classes in these files will not be reflected correctly. If a portion of a file is causing problems, it can be excluded using the macro `__HAVOK_PARSER__`. This is defined during header processing, but not during normal compilation.

Running `generateReflections.py` with the `-v` (or `--verbose`) option will print out the command line of each parser run, which may be useful in debugging configuration problems.

The most common problem is that each header is parsed independently. Any classes or header files that are needed for that header should be included from the header itself, or the project pre-compiled header file, even if this is not required in normal compilation (due to the location from where it is included).

In order to generate a completely clean rebuild of generated files, delete the `reflections.db` file in the project root directory, and any files in the `Classes/` directory, and re-run `generateReflections.py`. The file `reflections.db` is a cache and is always safe to remove.

In many cases, if the parsing fails, a script is automatically run that will test for some common configuration errors. This will be visible if `generateReflections.py` is run from the command line.

**Manually excluding certain files**

When generating the list of files to pass the parser, any files containing the string `//HK_REFLECTION_PARSER_EXCLUDE_FILE` will be excluded from the list.

#### 1.4.8.4 Generating the Reflection Data

To ease the burden on the programmer, the reflection data does not have to be manually created, but can be extracted directly from C++ class definitions. The script (`Tools/Serialize/generateReflections.py`) parses the C++ definitions for the entire project at once and writes the reflection data (C++ code) to the `Classes/` directory in the project. Only classes containing the `HK_DECLARE_REFLECTION()` declaration will be considered for reflection.

E.g. If a project named `Project` contains a single file `Foo.h` containing `class Bar`, the generated file will be named `Project/Classes/Reflections/FooReflection.cpp` and will contain one const `hkClass` instance `BarClass`. All reflected classes for that project will be created in the same directory and will be grouped together in the same `*Reflection.cpp` file if the original classes were in the same header. You may wish to add an extern declaration to `Foo.h` (`extern const hkClass BarClass`) for users of `Foo`.

The `generateReflections.py` script should be passed a list of directories to scan. For each scanned directory, a directory named `Classes` is generated, containing the reflection files for that project. Some common arguments for the script are:

- *–force-rebuild*: Force the script to try to rebuild cached file databases, even if it thinks they are up to date.
- *–force-output*: Force the script to try to regenerate output files, even if it thinks they are up to date.
- *–verbose*: Produce more verbose output than normal.
- *–output–dir=path/to/dir*: Generated files are created in the specified directory and not the default location.

When a directory name is passed to `generateReflections.py`, the following rules are used to determine what is parsed:

- If the specified directory contains a `reflectionSettings.cache` file, it and its subdirectories are parsed as a single project.
- If `reflectionSettings.cache` files are found in subdirectories of the specified directory, each directory containing a `reflectionSettings.cache` file and its subdirectories are parsed as a single project.
- If no `reflectionSettings.cache` files are found, the specified directory and its subdirectories are parsed as a single project. The default include paths are used.

**Migration from previous versions**

The format of the generated reflected classes has not changed from the parsers used up until release 2011.2, however the class information is not stored in the same place. Prior to version 2010.1, each reflected header file generated a corresponding `Class.cpp`. From 2010.1 to version 2011.2, the parser generated one file per project, containing all of the reflection information from that project. The current parser generates a single reflection file for each reflected header file (similarly to what happened prior to version 2010.1). This to allow fine-grained control of which reflected classes and functions are linked into the final executable (even when using file-level linkers). The reflected files are now stored in the

`Classes/Reflections/` directory in the project. The hints and disambiguations required by the pre-2010.1 parser are no longer necessary. Class files generated by the old parsers remain valid and may be used, however any new files generated will be in the new layout. Note that the new reflection files will conflict with the older class reflected files containing classes of the same name. For this reason, any old reflection generated file encountered is automatically renamed. These should be removed from the project.

Header files are parsed as standalone files, no matter how they are included in the source. This means that any includes and definitions required by the header file must be explicitly included, and any required include paths must be set by a `reflectionSettings.cache` file.

### C++ parser notes

The serialization tool uses the Havok C++ parser. This must be installed on the system used to generate reflected class files.

While the parser can usually accurately parse any valid C++, there are some cases that require special consideration. To avoid the overhead of invoking the parser once for each header, all headers for a project are concatenated and the resulting file is passed to the parser. Thus, each header file must parse independently, explicitly including any other header files and defining any types that it requires (apart from an implicit include of the projects precompiled header).

The same generated files are used on all platforms. So while preprocessor defines are handled correctly, care is needed with conditional compilation. Also, because gcc may essentially emulate other compilers, predefined values may not be as expected. Only platform-independent Havok types may be used in reflected classes (e.g. use `hkBool` and not `bool`).

Note that the `m_` prefix is expected on all member variables. If this is not the case, a class may override the prefix, using the `memberprefix` annotation as shown:

```
class MyClass {
        //+memberprefix(_)
        //Note there are no quotes around the string supplied to the memberprefix attribute.
        //Note that the HK_DECLARE_REFLECTION() macro comes after the attribute.
        HK_DECLARE_REFLECTION();
        hkVector4 _pos;
        int _flags[4];
}
```

Note that the `HK_DECLARE_REFLECTION()` macro must come after the `memberprefix` attribute. Note also that the empty string may also be a valid prefix, specified as:

```
//+memberprefix()
```

### Metadata overrides and annotations

Metadata annotations are specially formed comments which are extracted by the metadata generation script. They have the form `+identifier(value)`. Each annotation is applied to the *previous* item, and the parser continues to scan forward until either another declaration is encountered or a comment that does not declare a valid attribute. Care should be taken when commenting out members–parsing stops

once a comment that is not an annotation is encountered. Several annotations may be made per item, e.g.:

```
struct MyStruct
{
  //...

    // Treat tableIndex as an int. For simple classes this can save us having
    // to generate reflection data for the "TableIndex" type.
    // Alternatively we could override globally in typeMapping.txt (see above)
  TableIndex m_index; //+overridetype(int)

    // This is a pointer to a reflected object, but we don't want to save it.
    // It will be null for serialized objects.
  Reflected* m_pointer; //+serialize(false)

    // We don't save the cache - it is set to zero on load.
    // Cache is not a reflected object so "+serialize(false)"
    // won't work here since the notion of "serialized" is separate from
    // the notion of "reflected".
  Cache* m_cache; //+nosave

    // Aggregate type default.
  hkVector4 m_startPos; //+default(100,100,0,0)
};
```

The most important member annotations (with default values if unspecified) are:

**+serialized(boolean=true):** Indicates that the member should not be saved when taking a snapshot. By default all members are serialized.

**+nosave:** Indicate that a member should not be serialized. When serialized, the space will be zero filled. This is similar to `+serialized(false)` except that additionally the member is only partially reflected, e.g. `member.getType()` will be correct but `member.getClass()` will return null.

**+overridetype(newtype):** Force the metadata for member to be "newtype".

**+default(value):** Default values are especially useful for versioning–defaults are applied to new members when versioning.

**+reflected(boolean=true):** If false, the member is completely absent from metadata. Classes which have nonreflected members cannot be serialized.

**+owned(boolean=true):** For a pointer, indicates that the pointed to memory should be included in memory statistics for this object.

For classes:

**+serializable(boolean=true):** The class cannot be serialized, probably because it has some non reflected members.

**+defineattribute(boolean=false):** This class is used as a custom attribute. Currently used by the .NET language bindings which require special treatment for attributes.

**+memberprefix(string=m_):** This class has the given prefix on all member variables.

**+hk.MemoryTracker(ignore=True):** Tell the memory tracker to ignore this class.

For enums:

**+defineflags(boolean=false):** The enumeration values are intended to be combined bitwise instead of being used as discrete values. Currently used by the .NET language bindings which require special treatment for flags.

### Custom metadata annotations

You can attach arbitrary metadata to `hkClass` and `hkClassMember`s. For example:

```
class MyGuiGridPosition // Stores the layout of attributes in the attribute editor grid
{
    //+defineattribute(true)
    HK_DECLARE_REFLECTION();
    int m_row;
    int m_col;
    int m_colSpan;
};
class MyHelpString
{
    //+defineattribute(true)
    HK_DECLARE_REFLECTION();
    const char* m_help;
};
class DebrisGameObject
{
    //+My.HelpString("Attributes can apply to classes and members.")

    hkVector4 m_position;  //+My.GuiGridPosition(1,0)
    hkQuaternion m_rotation; //+My.GuiGridPosition(1,1,1)
    const char* m_name;
        //+My.GuiGridPosition(colSpan=2)
        //+My.HelpString(help="The name is used to resolve the ...")
        //+My.Tooltip("Unique identifier")
};
///...
void GenerateGuiLayout( void* data, const hkClassMember& member )
{
    if( hkVariant* helpStringAttr = member.getAttribute("My.HelpString") )
    {
        MyHelpString* helpString = static_cast<MyHelpString*>(helpStringAttr.m_object);
        //... use the helpString object.
    }
}
```

The custom attribute class should specify `//+defineattribute(true)` when you need to expose it to .NET.

Custom attributes can be arbitrary reflected objects. If you have many attributes, you may spread them over several lines. The mapping from the attribute definition to the C++ data blob is implemented by means of a Python function call. You can see that the attribute syntax matches the very flexible Python function call syntax, so we can easily support positional, variable length and keyword arguments. See the Python docs for how function arguments are mapped.

Note that custom attributes all contain a dot. When the C++ parser finds an attribute which is not built-in (i.e. contains a dot), it will try to import an attribute plugin from `Serialize/attributes`, e.g. in this case `Serialize/attributes/My.py`, and will call methods from that module to create the metadata. The

method call should return a string containing C++ code which is pasted into the `ReflectedClass.cpp`. See `Serialize/attributes/Example.py` and `Demos/Common/Api/CustomAttributes` for several examples.

The attribute name does not need to match the reflected class name. Instead it can be thought of as simply a tag which is used in two places:

- The string argument to `getAttribute()`.
- To find the callable item in the Python module.

In fact the C++ type corresponds to the fully scoped name of the Python class with the periods removed. Thus same reflected class may be reused for several different attribute names. In the above example, we don't need to define separate classes for `HelpString` and `Tooltip`. We reuse the C++ `HelpString` class for both, using the code below.

```
## Serialize/attributes/My.py

import attributes
#...
class HelpString(attributes.Attribute):
    def __call__(self, value):
        return 'static const MyHelpString %s = { "%s" };' % (self.symbol, value)

# Tooltip is an alias for HelpString
# This line is similar to the C++ "typedef HelpString Tooltip;"
Tooltip = HelpString
```

### Virtual Tables

The situation very commonly occurs where we have a pointer to a polymorphic type and need to access its associated `hkClass`. We do this by assuming that at the start of every polymorphic type is a *virtual table* (vtable), the address of which uniquely identifies each type. In practice, all compilers can be made to adhere to this assumption by having an empty base-most class (`hkBaseObject`).

Most "save" methods require such an association (`hkVtableClassRegistry`) as a parameter.

The other common situation is that we have loaded an object and need to initialize its vtable (it may change each time the executable is compiled) and optionally perform some other "load-time" initialization. We associate a string identifier (class name) with initialization function. Often initializing the vtable is as easy as placing the table address at the start of the object. However, it quickly gets more complicated with multiple inheritance when there are multiple vtables.

Most "load" methods require such an association (`hkTypeInfoRegistry`).

Because of the complications of initializing vtables, we let the compiler do the low level work. Each serialializable class should define a special constructor which does not initialize its members. Then calling placement new with the special constructor on the object initializes the vtable without disturbing the loaded values. Note that the flag is chained to the parent to avoid implicitly chaining the default constructor.

```
struct MyClass : public MyBase
{
  HK_DECLARE_REFLECTION();
  MyClass() {} // default constructor zeros the m_indices array
   \textbf{MyClass(hkFinishLoadedObjectFlag f) : MyBase(f), m_indices(f) {} // non-initializing
     constructor}
  hkArray<int> m_indices;
};
```

### Multiple Inheritance

For simplicity, we support only Java-style inheritance. That is, only the first parent may have data members and all others must be pure interface (no member data) classes. Additionally, virtual inheritance is not supported at all. Because most Havok types (except Listeners) derive from `hkReferencedObject`, this means that a Havok base class must be the first parent in the hierarchy.

### Templates

When a reflected class depends on a template type (e.g. the reflected class inherits from a specific template instance or it has a member object of a specific template instance type) the template class itself must be reflected in order to rebuild the objects correctly. That is, information about the specific template instance class is saved in specific `hkClass` and `hkTypeInfo` objects (with companion helper functions). Because we only feed the reflection data generation system with header files, we can only detect the needed template instance classes if they are explicitly referred in the header file. All of the following cases are correctly supported:

- Reflected classes with reflected template instance parent type
- Reflected classes with members of reflected template instance types
- Reflected classes with member pointers to reflected template instance types
- Reflected classes with member references to reflected template instance types
- Reflected classes with member arrays of reflected template instance types

In any of the above cases, the generated `hkClass`, `hkTypeInfo` and reflection functions will be saved in reflection files that are part of the current project. If a project uses an external template in a reflected class in one of its headers, the reflection data associated with the specific instance will be saved in that project (even if the template itself belongs to another project). This is done to limit dependencies between Havok libraries. If the same template instance is used in different libraries that are then linked together in the final executable, multiple objects files containing the exact same reflection data will be present. This does not cause any problem on Havok supported compilers, as only one of the identical objects files will be linked in the final executable.

When using member pointers and/or references to template instance types in non-template reflected classes, care must be taken to be sure that reflection data is correctly generated for the template instance. The simple definition of a pointer or reference to a template instance is not enough for clang to actually define the template instance; the type will remain incomplete until any access to the template instance class is done (e.g. calling the constructor). Because we only feed clang with header files, only references available in the preprocessed header files will be considered. If the type remains incomplete, it is assumed that the specific template instantiation is available somewhere else and if this does not hold a linker error will be generated.

**hkTypeInfo**

The virtual table details are stored in an `hkTypeInfo` instance. This class stores the type's name, vtable and functions to finish (e.g. initialize its vtable) and cleanup object. The type info is generated automatically along with a `hkClass` for each reflected class. Note that `hkTypeInfo` is not generated for abstract classes as you cannot instantiate an object of abstract class, thus you do not need `hkTypeInfo`.

**Summary**

Here are the steps to reflect a `Foo` in a file named `Foo.h`.

- Mark the class for reflection with `HK_DECLARE_REFLECTION();`.
- Run `tools/serialize/generateReflections.py` to generate the needed reflection data
- (optional) Add override types and defaults and rerun script
- Add vtable initializing constructor (`hkFinishLoadedObjectFlag`).

At this point all the infrastructure is in place. All that remains is to add the reflection data to the relevant registries as in the custom registration section.

## 1.5 Multithreading

### 1.5.1 Introduction

All Havok products are designed to run in a multithreaded environment. This chapter aims to give you a basic understanding of the way multithreading is approached by Havok products. It covers the mechanics of how multithreading works, how products interoperate, and how to setup and work with multithreaded Havok applications. For more platform specific information please refer to the quickstart guide. Also some specific products have additional multithreading information in their user guide sections.

Running Havok multithreaded involves the concept of a "master" thread and "worker" threads. On the PlayStation®3 platform, the worker threads are implemented by default as SPURS tasks, and on the PC as CPU threads. There is no interface difference between a worker thread running on an SPU and a worker thread running on a CPU. There are two other main concepts which we will talk about in the following sections: the *queue*, which depending on the type of work load will be a `hkJobQueue` or a `hkTaskQueue`, and the *thread pool* (`hkThreadPool`).

If you are only interested in stepping physics multithreaded, you can skip the more detailed explanations below, and refer directly to the `consoleExampleMt` demo.

#### 1.5.1.1 hkJobQueue

Products using *jobs* as their unit of multithreaded work use the *job queue* as the central container for them. A job (`hkJob`) is a unit of work that can be processed by any thread (although there are some specific jobs that can only be processed on CPU or SPU). Any thread can add jobs to the queue and any thread can ask the queue for jobs. Although it is generally for internal Havok use, the job queue may be

used for your own custom jobs. The following pseudo code shows the job queue being used by a single thread.

```
{
     create hkJobQueue
     register job handlers with hkJobQueue

     while (1)
     {
          add jobs to hkJobQueue
          hkJobQueue::processAllJobs
     }

     delete hkJobQueue
}
```

Note that the above code is purely singlethreaded. The thread creates a job queue and then registers handlers with it. These are functions that are linked with particular job types, and are called when a job of a particular type is retrieved from the queue. Each Havok product using jobs has a function to register all its functions with the job queue. Once the job queue has been initialized in this way, you can add jobs to it. To process the jobs there is a function `processAllJobs()` which will continue taking and processing jobs from the queue until there are no more remaining. Note that some jobs (in particular those used during physics processing) will add new jobs to the queue. These new jobs must also be processed before `processAllJobs()` will return.

`hkJobQueue::processAllJobs()` can be called from multiple threads (or SPUs) simultaneously. When this happens each thread will take and process jobs until there are no more left in the queue. Also, as jobs may add more jobs, all threads wait until the execution of the last job is complete before returning, as it may add more jobs for the other threads to process. Therefore when any thread returns from `processAllJobs()`, all the work specified by the jobs that were on the job queue is guaranteed to be completed.

#### 1.5.1.2 hkTaskQueue

*Tasks* are used by some products instead of jobs as the unit of multithreaded work. A task is a class implementing the `hkTask` interface, which declares a virtual `process()` method for worker threads to call to process the task. This eliminates the need for keeping a register of task processing functions and allows for better encapsulation of the task processing logic and data.

Products usually return groups of tasks in the form of *task graphs* (`hkTaskGraph`) to be processed by worker threads. A task graph is simply a collection of tasks together with their *dependency* relations and a *multiplicity* per task indicating the number of times that that task has to be processed.

The *task queue* (`hkTaskQueue`) helps with the scheduling of these tasks graphs. Tasks graphs are queued in the task queue usually by the master thread while worker threads obtain tasks from the queue, process them and mark them as finished in the queue to allow its dependent tasks to be made available. Multiple task graphs may be queued simultaneously with their tasks being interleaved according to their priority. This is an example of a normal task queue work flow:

```
// Task queue instance shared between all threads. Usually alive for the lifetime of the application
hkTaskQueue queue

// Master thread
{
    // Graph of tasks to be executed produced by one of Havok's products
    hkTaskGraph* graph = ...

    // The moment a graph is added to the queue its tasks without unresolved dependencies are made
      available to worker threads
    graphId = queue.addGraph(graph)

    // Join processing but only those tasks that belong to our graph. Will return when all tasks in the
      graph have been finished
    queue.processGraph(graphId, hkTaskQueue::WAIT_UNTIL_ALL_TASKS_FINISHED)
    queue.removeGraph(graphId)
}

// Worker thread
{
    // Process tasks as they are made available until the queue is closed
    queue.processTasks(hkTaskQueue::WAIT_UNTIL_QUEUE_CLOSED)
}
```

#### 1.5.1.3 hkThreadPool

The `hkThreadPool` is an abstract class that allows multiple threads to process jobs from a job queue and tasks from a task queue. It has two implementations, one for CPU and one for SPU, although it is possible to provide your own (if for example you have your own scheduler for the PlayStation®3). Typically a single thread pool is created for the duration of an application. This keeps a set of threads running, and synchronises calls to set the threads operating on the job or task queue depending on whether `processJobQueue()` or `processTaskQueue()` is called. If you are working with Havok products that use both multithreading systems, job-based and task-based, you will have to alternate between the two methods. The following pseudo code shows the general work flow:

```
{
     create hkThreadPool
     create hkJobQueue
     create hkTaskQueue

     while (1)
     {
          // Jobs
          place jobs on hkJobQueue
          hkThreadPool::processJobQueue( hkJobQueue )
          <optional> hkJobQueue::processAllJobs
          hkThreadPool::waitForCompletion

          // Tasks
          place graph on hkTaskQueue
          hkThreadPool::processTaskQueue( hkTaskQueue )
          <optional> hkTaskQueue::processGraph
          hkThreadPool::waitForCompletion
     }

     delete hkTaskQueue
     delete hkJobQueue
     delete hkThreadPool
}
```

The call `hkThreadPool::processJobQueue()` will simply call `hkJobQueue::processAllJobs()` on all its worker threads. The next call, `hkJobQueue::processAllJobs()`, causes the master thread to join in the processing with the worker threads. If you call this function it will return when all work is complete. However it is still necessary to call `hkThreadPool::waitForCompletion()`. This call will make sure that all relevant cleanup takes place, such as marshalling timer data etc, from the worker threads, and must be called before you call `hkThreadPool::processJobQueue()` or `hkThreadPool::processTaskQueue()` again.

`hkThreadPool::processTaskQueue()` will basically call `hkTaskQueue::processTasks()` on all its worker threads. The SPU implementation of the thread pool has a different internal logic but the end result is similar to having all SPUs calling `processTasks()`; check out `hkSpuThreadPool` for the details.

Note that the calls from the master thread to `processAllJobs()` and `processGraph()` have been marked as optional in the code above. This means that the master thread can work on something while the worker threads are processing jobs/tasks. However this call is not optional for physics; the thread that creates the job queue must participate in processing the jobs. For other products please refer to their sections below.

It is possible to use several job queues with one thread pool, however you may not process jobs on more than one job queue at one time. You may, for example, add jobs to a second job queue while the thread pool is processing, and after waiting for that processing to complete, kick off the processing of jobs of the second job queue. The same goes for task queues, although you are less likely to want to do that as they have better support for processing and waiting for independent workloads.

It is also possible to have multiple `hkThreadPool`s. However, currently it is not possible to do so on the PlayStation®3.

### 1.5.2 Stepping several multithreaded workloads

In this context a "workload" is a conceptual term; Havok has no workload structure as such. Workloads generally refer to stepping a Havok module forward in time. There are several workloads depending on what Havok products you are using, they are:

- Physics 2012
- Animation
- Collision Queries (part of physics product)
- Cloth
- Behavior

It is theoretically possible to step these workloads in parallel, by adding jobs from multiple workloads to a `hkJobQueue` and then calling `processAllJobs()` as described above. However this is not currently fully supported, and there are several issues with this. The first is that workloads are not typically independent. For example, collision queries depend on physics transforms, so performing them while physics is being simulated is not safe. Cloth and Behavior have the same issue. However, each module is designed to utilise multiple threads well. So each module should be stepped independently in a "staged" fashion. In other words, add and process the jobs for one module, and wait for completion. Then add and process the jobs for the next module and so on.

### 1.5.3 Multithreading Physics 2012

Please refer to the demo "`ConsoleExampleMt`" to complement this section of the documentation. It shows a basic physics setup, with no graphics, running multithreaded. It does however connect to the visual debugger, so you can run the demo, open the visual debugger and connect to localhost to see the physics simulation running.

Physics 2012 multithreading is the most complex of all Havok products. It is not necessary to understand how it works, however this documentation will go some way to explaining it so you can customise the execution if you need to. The basic interface to multithreading is relatively simple and issues such as cross platform coding, monitor stream maintenance, and SPU parameter passing (for PlayStation®3) are automated for the user.

To enable multithreading physics you must create your world using `hkpWorldCinfo::.m_simulationType = hkpWorldCinfo::SIMULATION_TYPE_MULTITHREADED`.

The following code example shows some of the important calls. Please see the `ConsoleExampleMt` demo for the full code listing.

```
{
     ...

     // Create thread pool and job queue
     hkThreadPool* threadPool = new hkCpuThreadPool( threadPoolCinfo );
     hkJobQueue* jobQueue = new hkJobQueue();

     // Create a physics world as usual. Make sure the simulation type in worldCinfo is
       SIMULATION_TYPE_MULTITHREADED
     hkpWorld* physicsWorld = new hkpWorld(worldCinfo);

     // Register physics job handling functions with job queue
     hkpWorld::registerWithJobQueue( &jobQueue );

     while( simulating )
     {
          // Step the world using this thread, and all the threads or SPUs in the thread pool
          physicsWorld->stepMultiThreaded( &jobQueue, threadPool, timestep );
     }
}
```

This code snippet does not have any calls to `hkJobQueue::processAllJobs`, etc., that were discussed in the previous section. Unless you need to segment the timestep, perform asynchronous stepping, or use the main thread for other processing, you do not need to go into the details. `stepMultiThreaded()` represents a full "stage" of processing, in that it adds all the necessary jobs to the queue, kicks off the worker threads, and calls `waitForCompletion()`. You should not call this function with any other jobs added to the queue, as they will also then be processed and may not be compatible with processing physics simultaneously (see notes in the previous section). Also, on the PlayStation®3 platform, `hkpWorld::stepMultiThreaded()` must be called from the same thread that initializes the `hkThreadPool`. This is generally the main thread.

The implementation of `stepMultiThreaded()` is shown in the following code snippet. Note that all functions are public and designed to be called independently if desired.

```
{
     // Place the initial physics jobs on the job queue
     world->initMtStep( jobQueue, physicsDeltaTime );

     // Tell the worker threads to start processing all jobs
     threadPool->processJobQueue( jobQueue )

     // Also use this thread to process all jobs
     jobQueue->processAllJobs( );

     // Wait for the worker threads to complete
     threadPool->waitForCompletion();

     // Finish the step, including performing continuous simulation
     world->finishMtStep( jobQueue, threadPool );
}
```

Note that the thread that calls `stepMultiThreaded()`, or these functions above *must* be the same thread that created the `hkJobQueue`.

When running multithreaded, Havok requires that at least two threads are processing jobs, with at least one on a CPU (a PPU in the case of the PlayStation®3). Therefore, while the call to `jobQueue->processAllJobs()` is not always necessary, there are several situations where it is **manda-**

**tory** :

- It is always mandatory on the PlayStation®3.
- It is mandatory on non-PlayStation®3 platforms if you are using a single worker thread in the thread pool. As stated above, when running multithreaded, Havok will hang if it doesn't have at least two threads processing jobs.
- It is mandatory on non-PlayStation®3 platforms if you are using no worker threads in the thread pool. This essentially reverts to running Havok in a singlethreaded manner, and no simulation work will be performed unless the main thread processes the jobs. The simulation will not hang, but nothing will be updated.

On the PlayStation®3 the majority of the time will be spent with the PPU thread waiting on a system semaphore, as it is only used for internal marshaling jobs, the majority of calculations are performed by SPU. If you have other lower priority work to be done on the PPU it can be scheduled in another PPU thread to fill the gaps. If you'd prefer to process other tasks on the main thread, as apposed to a separate thread, you will need to create an additional `hkCpuThreadPool`. This is required on the PlayStation®3 platform because some physics tasks can only be executed on the PPU. Therefore, if a PPU thread does not process these jobs, the `hkJobQueue` will block until `hkJobQueue::processAllJobs` is called. The following code snippet shows how to step using both a `hkCpuThreadPool` and a `hkSpuThreadPool`. This allows PPU threads in the `hkCpuThreadPool` to process PPU jobs while the main thread does other tasks.

```
{
     // Place the initial physics jobs on the job queue
     world->initMtStep( jobQueue, physicsDeltaTime );

     // Tell the worker threads to start processing all jobs (SPU & PPU)
     spuThreadPool->processJobQueue( jobQueue );
     cpuThreadPool->processJobQueue( jobQueue );

     // Do other work

     // Wait for the worker threads to complete
     spuThreadPool->waitForCompletion();
     cpuThreadPool->waitForCompletion();

     // Finish the step, including performing continuous simulation
     world->finishMtStep( jobQueue, threadPool );
}
```

The final statement, `finishMtStep()`, requires some further explanation. This function performs continuous simulation, and performs any singlethreaded cleanup necessary. The parameters allow you to control whether collision detection is performed in parallel during continuous simulation. If you pass in a *jobQueue*, and a *threadPool*, they will be used for this. When it is called from `stepMultithreaded()` it will take the *jobQueue* and the *threadPool* and will perform parallel collision detection in this step. However if called separately (as above) the default values of `HK_NULL` will mean all continuous simulation is performed single threaded. Only the master thread must call this function. It must be called when all threads have returned from calling `stepProcessMt()`, i.e. the master thread has called `waitForCompletion()` on the thread pool.

The last part of this step, `finishMtStep()` does not typically multithread very well, as it performs continuous physics which is a largely sequential task. In a continuous simulation, Time Of Impact (TOI) events have to be solved sequentially and 2 TOIs cannot be solved by 2 independent threads. As a result

the possibilities of multithreading TOI solving are limited. However the collision detection part of each TOI can be parallelized and solved by different worker threads or SPUs. As a result you should see a TOI solving speedup if a fast complicated (e.g. list shape) object hits a pile of other complicated objects. However you will not get any speedup (but rather a slowdown) if a simple object hits another single object, like a ragdoll hitting a landscape.

It may be advantageous to choose to process other tasks that are independent from physics (such as animation) on the thread pool, and process `finishMtStep()` single-threaded. To do this, use the code in the above code section, but pass in `HK_NULL` for the parameters of `finishMtStep()`. Prior to calling `finishMtStep()` you need to kick off the thread pool with any jobs you wish to process on the job queue.

If you need to step multiple `hkpWorld`s you can do this with a single (or multiple) `hkThreadPool`s. The following code snippet shows how to do this with a single `hkThreadPool`.

```
// Blocking approach
{
     for (int i = 0; i < numberOfWorlds; ++i)
     {
         world[i]->stepMultithreaded(jobQueue, threadPool, timestep);
     }
}

// Non-blocking approach
{
     // Place the initial physics jobs on the job queue
     for (int i = 0; i < numberOfWorlds; ++i)
     {
          world[i]->initMtStep( jobQueue, timeStep );
     }

     // Tell the worker threads to start processing all jobs
     cpuThreadPool->processJobQueue( jobQueue );

     // Do other work

     // Wait for the worker threads to complete
     cpuThreadPool->waitForCompletion();
     jobQueue->processAllJobs( );

     // Finish the step, including performing continuous simulation
     for (int i = 0; i < numberOfWorlds; ++i)
     {
          world[i]->finishMtStep( jobQueue, threadPool );
     }
}
```

There are some additional issues to be aware of when running physics multithreaded. For an overview of these, please refer to the section "Multithreading Issues" in the physics documentation.

#### 1.5.3.1 Asynchronous stepping

Asynchronous stepping refers to stepping the engine out of sync with your game stepping. This is typically done when you take variable timesteps. It is generally not recommended because it can be more difficult to control the physics and get exactly the behavior you want, but it is possible. A discussion of asynchronous stepping is provided in the dynamics chapter of the physics documentation, entitled "Stepping The Simulation Forward". Multithreading asynchronous stepping is done with the following

code:

```
{
     world->setFrameTimeMarker( frameDeltaTime );

     world->advanceTime();
     while ( !world->isSimulationAtMarker() )
     {
          physicsWorld->stepMultiThreaded( &jobQueue, threadPool, timestep );
     }
}
```

Again, prior to calling `stepMultiThreaded()` the *jobQueue* should be empty, and the *threadPool* should not be processing, i.e. `waitForCompletion()` should have been called.

### 1.5.4 Multithreading Animation

This is discussed in more detail in the Animation Runtime > Multithreading chapter of the Animation documentation. The process of multithreading animation sampling is simple from a thread control point of view–you place your sampling jobs onto the queue, then call `processJobQueue()` using the thread pool. You then can call `waitForCompletion()` to ensure that all jobs have finished. Additionally each job takes a pointer to a semaphore, or a flag. This allows you to wait for a particular job or group of jobs (if you point multiple jobs to that semaphore/flag), without having to wait until all the added jobs are finished.

### 1.5.5 Multithreading Collision Queries

This is discussed in more detail in the Collision Detection chapter of the physics documentation. The process of multithreading collision queries is simple from a thread control point of view–you place your collision query jobs onto the queue, then call `processJobQueue()` using the thread pool. You then can call `waitForCompletion()` to ensure that all jobs have finished. Additionally each job takes a pointer to a semaphore. This allows you to wait for a particular job or group of jobs (if you point multiple jobs to that semaphore), without having to wait until all the added jobs are finished.

If you register the physics world with the job queue, the collision query functions will be also registered.

Special note for PlayStation®3: It is possible to add collision query jobs to the job queue that cannot be run on the SPU. These are jobs that reference shapes that are not supported by the SPU. Such shapes are contained by collidables that will have the `m_forceCollideOntoPpu` set to a non-zero value. A warning is raised if you add these jobs to the queue. If these jobs are on the queue, then you must call `hkJobQueue::processAllJobs` in the PPU thread, or these jobs will not be processed and the program will hang in `waitForCompletion()`.

### 1.5.6 Multithreading Behavior

The optimization of Havok Behavior for multithreading was guided by the use case in which there are more characters in the scene than available threads. The algorithm does not process a single behavior

graph on multiple threads. The algorithm is primarily intended for running multiple behavior graphs on one or more threads.

Before stepping Behavior using multiple threads, you must first register Behavior with the job queue:

```
{
     create hkJobQueue

     // register the Behavior jobs with your job queue
     hkbBehaviorJobQueueUtils::registerWithJobQueue( jobQueue );
}
```

You can now use the thread pool and job queue to step your behaviors at each frame of simulation.

#### 1.5.6.1 When using hkbWorld

If you are using an `hkbWorld`, you simply call the step method like this:

```
hkbWorld* world = ...;

world->step( timestep, jobQueue, threadPool );
```

It will use the thread pool and job queue that you pass in to multithread the step.

Since by default the `hkbWorld` steps the physics world, you may want the physics step to also be multithreaded. To make the `hkbWorld` step the physics multithreaded you must initialize it with an `hkbMultiThreadedPhysics2012Stepper` like this:

```
hkbPhysics2012Stepper* physicsStepper = new hkbMultiThreadedPhysics2012Stepper( jobQueue, threadPool );

hkbWorldCinfo worldInfo;
{
    worldInfo.m_up = myWorldUp;
    worldInfo.m_physicsWorld = physicsWorld;
    worldInfo.m_useDefaultSceneModifiers = true;
    worldInfo.m_physicsStepper = m_physicsStepper;
}

hkbWorld* world = new hkbWorld( worldInfo );
```

#### 1.5.6.2 When not using hkbWorld

If you are not using an `hkbWorld` you will need to independently multithread the `update()` of your behavior graphs (including the handling of events) and the `generate()`. Please refer to the implementation of `hkbWorld::stepMultithreaded()`.

To perform the `update()` you must create an `hkbUpdateCharacterJob` for each character and then

process them.

To perform the `generate()` you must populate an `hkbSceneCharacters` object with information about your characters and then call this utility function:

```
static void HK_CALL hkbGenerateUtils::generateMultiThreaded(
    hkbSceneCharacters& sceneCharacters,
    hkJobQueue& jobQueue,
    hkThreadPool& threadPool );
```

The `hkbSceneCharacters` contains parallel arrays of `hkbCharacter`s, `hkbGeneratorOutput`s, and `hkbContext`s. The `hkbGeneratorOutput` and `hkbContext` that you must provide for each character/behavior do not need to persist from one frame to the next. You can choose whether to keep them around or create them anew each frame.

**Note:**

If you want to process other kinds of jobs at the same time as Behavior jobs you will need to replace the implementation of `hkbGenerateUtils::generateMultiThreaded()` with your own code.

### 1.5.7 Timers

There are a couple of additional things to do to collect timers. If you want to collect timers and display them in the VDB, each thread or SPU must have a buffer allocated to it to collect these timers.

```
{
     ...
     // Allocate timer buffer for this thread (200K)
     hkMonitorStream::getInstance().resize(200000);

     // Create thread pool and job queue
     // Specify that the thread pool should allocate buffers per thread for timer collection
     hkCpuThreadPoolCinfo threadPoolCinfo;
     threadPoolCinfo.m_timerBufferPerThreadAllocation = 200000;
     hkThreadPool* threadPool = new hkCpuThreadPool( threadPoolCinfo );

     // Create a physics world and register with queue.
     hkJobQueue jobQueue;
     hkpWorld* physicsWorld = new hkpWorld(worldCinfo);
     physicsWorld->registerWithJobQueue( &jobQueue );

     // Setup the visual debugger. Just showing the first line of this setup code,
     // see ConsoleExampleMt demo for the remaining code to setup the VDB
     hkpPhysicsContext* context = new hkpPhysicsContext();


     while( simulating )
     {
          // Step the world using this thread, and all the threads or SPUs in the thread pool
          physicsWorld->stepMultithreaded( &jobQueue, threadPool, timestep );

          // Copy timer information from the thread pool to the VDB context, so it can be viewed by the
            VDB
          // This function also copies the timers from this thread.
          context->syncTimers( threadPool );
          vdb->step();

          // After the VDB has been stepped, reset the timer streams in this thread and in the threads of
            the thread pool
          hkMonitorStream::getInstance().reset();
          threadPool->clearTimerData();
     }
}
```

To collect timers from all the threads you need to use code from the above snippet. You must make sure that space is allocated for the timer buffers, and that each frame they are synced with the VDB context and then after the VDB is stepped, they are reset. If you do this you will be able to view the timer information in various forms in the VDB, on a per thread basis. Note that this code is identical for both CPU and SPU thread pools. If the SPU thread pool is used, the thread pool class takes care of managing the buffers and syncing them from SPU to main memory.

#### 1.5.7.1 Viewing timers in the Visual Debugger

To enable Havok performance monitors, first turn on the Statistics Viewer (via the Viewers > Statistics menu item) and then enable the Stat Graph Overlay (via the View > Render State > Stat Graph Overlay menu item). To see text versions of the performance monitors, enable the Perf Stats Summary from the Window menu or from the "T" icon on the Windows tool bar. In general, Green stands for integration, Blue stands for collision detection and Red stands for wait in the Stats Graph Overlay. For specific details hover the mouse over the Stat Graph Overlay Window.

### 1.5.8 The Master thread, and hkThreadNumber

Each thread that runs with the job queue must have a unique thread number. This is stored in the thread local variable `hkThreadNumber`. This variable is set automatically in the following way. In the constructor of the job queue the `hkThreadNumber` is set to 0. In the `hkThreadPool`, each thread created gets its `hkThreadNumber` set from 1-N. If you do not use the `hkThreadPool` you must set this number in an identical manner for all worker threads.

The job queue has the concept of a "Master" thread. This is the thread whose `hkThreadNumber` variable is set to 0. By default it will be the thread that created the `hkJobQueue`. The master thread *must* call process next job for physics simulation. If you create the job queue in a different thread to the thread you wish to use as a master thread, you must manually set the `hkThreadNumber` of that thread to 0.

## 1.6 Geometry Processing

### 1.6.1 Introduction

Havok contains geometric processing technology which can be used to modify shapes and meshes in different ways, customizing the source geometries for various needs.

### 1.6.2 Convex Hull

Havok includes robust, fast and exhaustive support for convex hull generation, querying and processing. It is accessible through the following header file: `/Common/Internal/GeometryProcessing/ConvexHull/hkgpConvexHull.h`. This document is intended to be a quick overview of the functionality provided by the convex hull API.

As a reference, most of the functionality related to convex hulls are demonstrated in the `Demos/Common/Test/Feature/ConvexHull` demo.

#### 1.6.2.1 Construction

There are various ways to build a convex hull, depending on the input and/or dimensional requirements.

Construction can be configured by passing a `hkgpConvexHull::BuildConfig` structure as the last parameter of the build methods. Here is a list of the main members of this structure and their default values:

- *m_allowLowerDimensions* (`hkBool`, default: false)
  If true, the algorithm will try lower dimensional convex hulls if a 3D convex hull cannot be built.
- *m_buildIndices* (`hkBool`, default: true)
  If true, the algorithm will automatically compute indices.
- *m_buildMassProperties* (`hkBool`, default: false)
  If true, the algorithm will automatically compute mass properties.

### 3D Convex Hull from Points

A 3D convex hull can be built from a set of points using the method `hkgpConvexHull::build(const hkVector4* points, int numPoints, const BuildConfig& config=BuildConfig())`.

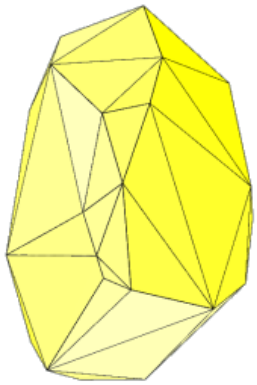


Figure 1.6: 3D convex hull built from a random points cloud.

### 3D Convex Hull from Planes

A 3D convex hull can be built from a set of planes using the method `hkgpConvexHull::buildFromPlanes(const hkVector4* planes, int numPlanes, const BuildConfig& config=BuildConfig())`.

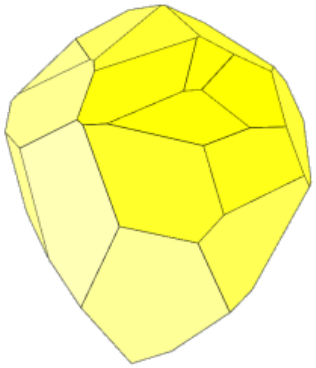


Figure 1.7: 3D convex hull built from a random planes set.

### 2D Convex Hull from Points

A 2D convex hull can be built from a set of points and a projection plane using the method `hkgpConvexHull::buildPlanar(const hkStridedVertices& points, const hkVector4& plane, const BuildConfig& config=BuildConfig())`.

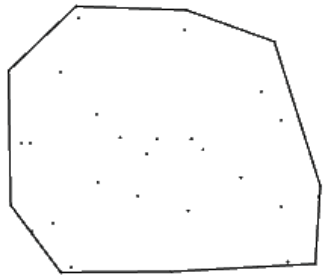


Figure 1.8: 2D convex hull built from a random points cloud

#### 2D Convex Hull from Planes

A 2D convex hull can be built from a set of bounding planes and a projection plane using the method `hkgpConvexHull::buildFromPlanes(const hkVector4* planes, int numPlanes, const hkVector4& projectionPlane, const BuildConfig& config=BuildConfig())`.

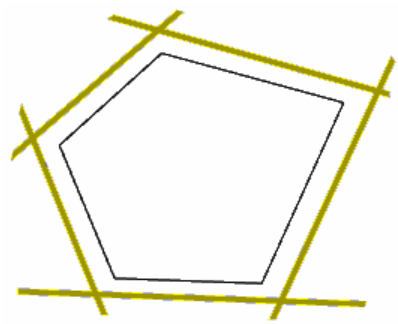


Figure 1.9: 2D convex hull built from a random planes set

### 1.6.2.2 Querying

Several methods can used to retrieve information about a built convex hull.

**Note:**

Because internally the convex hull might use slightly different positions than the ones used to build the hull, some functions require you to specify the type of coordinates that you wish to retrieve:

- `hkgpConvexHull::SOURCE_VERTICES`
  The original vertices used as input to build the convex hull.
- `hkgpConvexHull::INTERNAL_VERTICES`
  The internal vertices used to robustly compute the convex hull.

In most situations, it is recommended that you use the internal representation as it guarantees convexity.

#### General Information

Once a convex hull is built, the following methods can be used to query general information about its current status:

- `int getDimensions()`

  Retrieve the number of dimensions of the convex hull or -1 if the convex hull is empty.

- `const BuildConfig& getConfiguration()`

  Retrieve the configuration used to built this convex hull.

- `const hkVector4& getProjectionPlane()`

  Retrieve the projection plane used to build the convex hull if `getDimension()` is 2.

### Vertex Information

Once a convex hull is built, the following methods can be used to access vertex information:

- `int getNumVertices()`

  Retrieve the number of vertices.

- `Vertex* getFirstVertex()`

  Retrieve the first vertex.

- `Vertex* getNext(const Vertex* vertex)`

  Retrieve the next vertex.

- `hkVector4 getPosition(const Vertex* vertex)`

  Retrieve the position of the given vertex.

- `void fetchPositions(Inputs inputs, hkArray<hkVector4>& positionsOut)`

  Retrieve all vertex positions and store them in the output array.

**Note:**
This method is automatically called by default (when building the convex hull) unless specified otherwise using hkgpConvexHull::Config::m_buildIndices.

### Triangle Information

Once a convex hull is built, the following methods can be used to access triangle information:

- `int getNumTriangles()`

  Retrieve the number of triangles.

- `Triangle* getFirstTriangle()`

  Retrieve the first triangle.

- `Triangle* getNext(const Triangle* triangle)`

  Retrieve the next triangle.

- `Vertex* getVertex(const Triangle* triangle, int index)`

  Retrieve a vertex of a triangle.

### Plane Information

Once a convex hull is built, the following methods can be used to access plane information:

- `int getNumPlanes()`

  Retrieve the number of planes.

- `const hkVector4& getPlane(int index)`

  Retrieve the normalized plane equation.

- `void fetchPlanes(hkArray<hkVector4>& planesOut)`

  Retrieve all planes.

### Geometry Information

Once a convex hull is built, the following methods can be used to access its geometrical properties:

- `void generateGeometry(Inputs inputs, struct hkGeometry& geometryOut)`

  Generate the geometry of the convex hull.

- `hkAabb getBoundingBox(Inputs inputs, hkReal scale=1, hkReal radialScale=0)`

  Retrieve the world AABB of the convex hull.

- `void getOrientedBoundingBox(hkVector4& halfExtents, hkTransform& worldTransform)`

  Retrieve the oriented bounding box and world transform of the convex hull.

- `void getOrientedRectangle(const hkVector4& projectionPlane, hkVector4& halfExtents, hkTransform& worldTransform)`

  Retrieve the oriented rectangle and world transform of the convex hull given a projection plane.

- `const hkVector4& getCentroid()`

  Retrieve the geometric center of the convex hull.

### Index Information

The internal representation of a convex hull is a closed triangular mesh (or a closed polygon if planar). Thus, to retrieve higher-level faces, one must call the `hkgpConvexHull::buildIndices` method to build that information. This method takes the following parameters:

- *minCosAngle* (`hkReal`)

  Defines the minimum (cosine) of an angle between two triangles for them to be classified as part of the same face.

- *forceRebuild* (`hkBool`)

  Force the algorithm to rebuild indices even if previous indices exist.

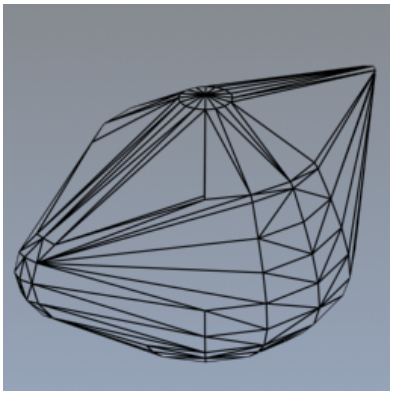


Figure 1.10: Triangles of the convex hull of a teapot

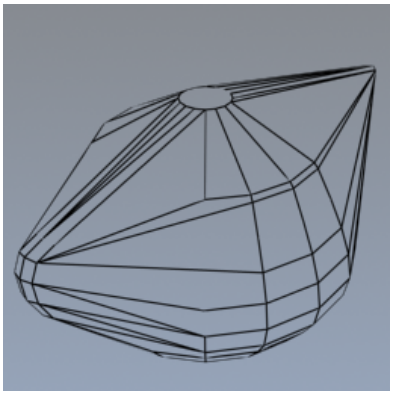


Figure 1.11: Faces of the convex hull of a teapot after having called buildIndices

**Note:**
This method is automatically called by default unless specified otherwise using *hkgpConvexHull::Config::m_buildIndices*.

Once built, faces topology can be retrieved by calling `generateIndexedFaces(Inputs inputs, hkArray<int>& verticesPerFace, hkArray<int>& indices, hkBool maximizeArea)`.

### Mass Property Information

Once a convex hull is built, one can compute the mass properties for it. Mass properties can be computed for 2D and 3D convex hulls alike.

Before querying mass properties, one must call the `hkgpConvexHull::buildMassProperties()` method. Once completed, the following methods can be used to retrieve mass properties related information:

**Note:**
Mass properties are *not* automatically built, you must explicitly call `buildMassProperties()` or set *hkgpConvexHull::Config::m_buildMassProperties* to true.

- `const hkVector4& getCenterOfMass()`

  Retrieve the center of mass of the convex hull.

- `void getLocalInertia(hkTransform& inertiaSpace, hkVector4& inertiaDiagonal)`

Retrieve the inertia tensor in local space.

- `hkReal getVolume()`

  Retrieve the volume of the convex hull.

- `hkReal getSurfaceArea()`

  Retrieve the surface area of the convex hull.

#### Custom Data

It can be handy to attach custom data to a convex hull instance. Two ways are provided to accomplish this:

- Unmanaged user pointer

  Using the function `userData()`, one can retrieve a *reference* to a void pointer.

  This is a light-weight method to store and retrieve custom data,

- Managed user object

  Using the function `userObject()`, one can set or get a *reference* to an instance of an object implementing the hkgpConvexHull::IUserObject interface.

  This object will be destroyed automatically when the convex hull is destroyed.

  **Note:**

  Some convex hull functions need to create either compounds and/or copies of managed user data; for this purpose, the `IUserObject` interface allows the implementation to overload the methods `IUserObject* append(const IUserObject*)` and `IUserObject* clone()` respectively.
  The default implementation of these methods raise an error using the `HK_ERROR` Havok macro.

### 1.6.2.3 Processing

Various methods are available to process convex hulls.

#### Simplification

Two main method of simplification are provided by the convex hull API:

- Vertex decimation

  Using the method `hkBool decimateVertices(int numVerticesToKeep, hkBool ensureContainment)` one can enforce the maximum number of vertices.

- Volume simplification

  One can also decimate vertices based on volume error: call the method `hkgpConvexHull* volumeSimplify(const hkgpConvexHull* hull, hkReal threshold, hkBool ensureContainment)`. This method internally uses the `decimateVertices()` method.

Note that in both cases, if *ensureContainement* is set to true, the simplified convex hull will be constrained to enclose the original vertices.

**Warning:**
Containment constraint is an expensive operation, use only during offline processing or on a simple convex hull.

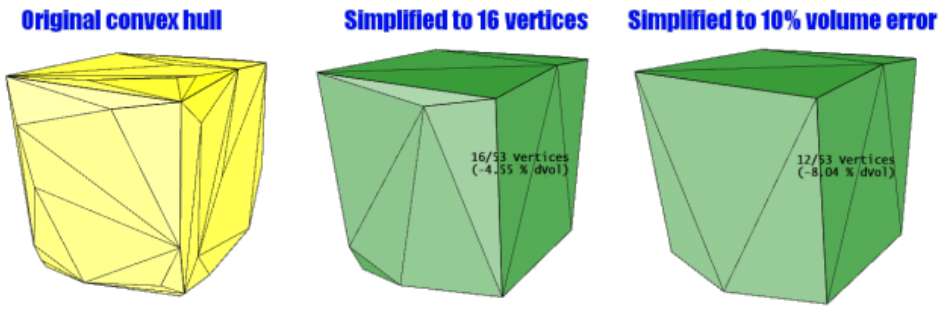



Figure 1.12: Convex hull simplification

### Boolean Operations

To compute the intersection of two convex hulls, use the `hkgpConvexHull::booleanIntersection()` method.

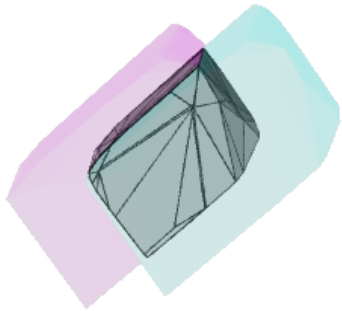


Figure 1.13: 3D intersection of convex hulls

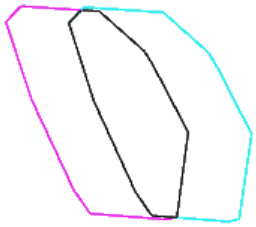


Figure 1.14: 2D intersection of convex hulls

To subtract one convex hull from another, use the `hkgpConvexHull::booleanSubtract()` method.

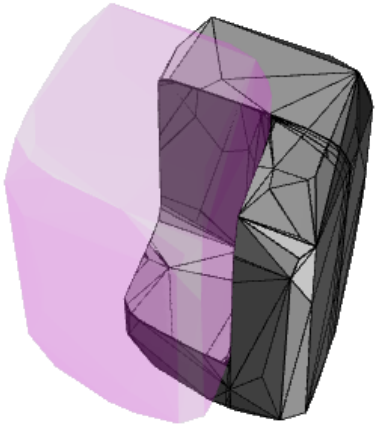


Figure 1.15: 3D subtraction of convex hulls

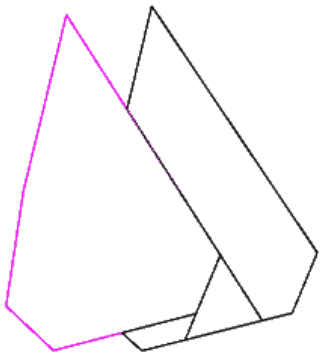


Figure 1.16: 2D subtraction of convex hulls

**Voronoi Cells and Edges**

To compute the Voronoi cells of a convex hull, use the `hkgpConvexHull::createVoronoiCells()` method.

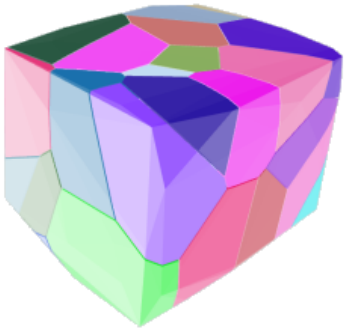


Figure 1.17: 3D Voronoi cells

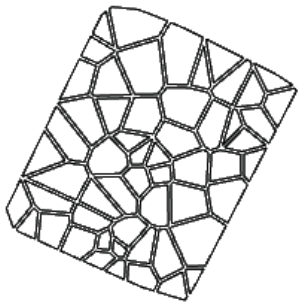


Figure 1.18: 2D Voronoi cells

To compute the Voronoi edges of a convex hull, use the `hkgpConvexHull::createVoronoiEdges()` method.

Figure 1.19: 3D Voronoi edges

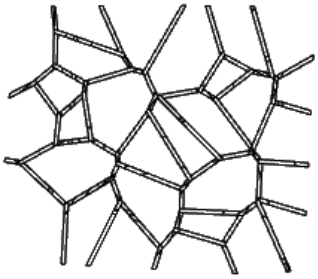


Figure 1.20: 2D Voronoi edges

### 1.6.3 Collision Geometry Optimizer

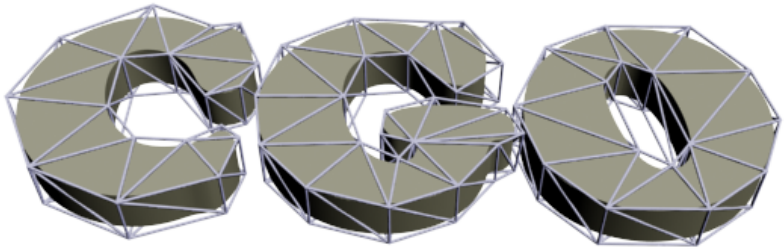


Figure 1.21: Densely tessellated mesh with its CGO-simplified cage (edges)

#### 1.6.3.1 Introduction

**About the Collision Geometry Optimizer**

Please note that this functionality is currently in beta. Please send us feedback on results or improvements for this technology. Havok collision geometry optimization technology (referred to as CGO from this point) is specifically designed to enable the automatic and conservative simplification of geometries used for collisions. More specifically, it provides a preprocessing step for geometry destined to be fed into convex decomposition or stored in a compressed mesh (`hkpBvCompressedMesh`) for fast and accurate collisions.

**Note:**
CGO is conservative in the sense that it will never shrink the mesh in order to optimize it unless it is specifically asked to do so by the user.

**CGO API**

CGO functions can be made accessible by including <`Common/Internal/GeometryProcessing/CollisionGeometryOptimize`

```
// Include CGO header.
#include <Common/Internal/GeometryProcessing/CollisionGeometryOptimizer/hkgpCgo.h>
```

CGO requires you to pass in a simplification configuration structure in order to control the process.

```
// Declare a CGO configuration.
hkgpCgo::Config myConfig;
// Set the maximum allowed error (in unit of distance).
. = 0.01f; // One centimeter.
```

Then simply call the `optimize` method with your geometry stored in a `hkGeometry` format.

```
// Optimize geometry.
hkgpCgo::optimize( myConfig, myGeometry);
```

This is it. The next sections will describe the most relevant configuration settings in detail.

### CGO Configuration

Here is a list of the most important configuration members and their meaning.

- `hkgpCgo::Config::m_maxDistance`, unit of distance.

  Control the maximum allowed error during optimization. This is a hard constraint–the optimization process will stop as soon as no more optimization below this value is possible. This is the recommended setting to control the accuracy of the optimization process as it is independent of the tessellation of the model.

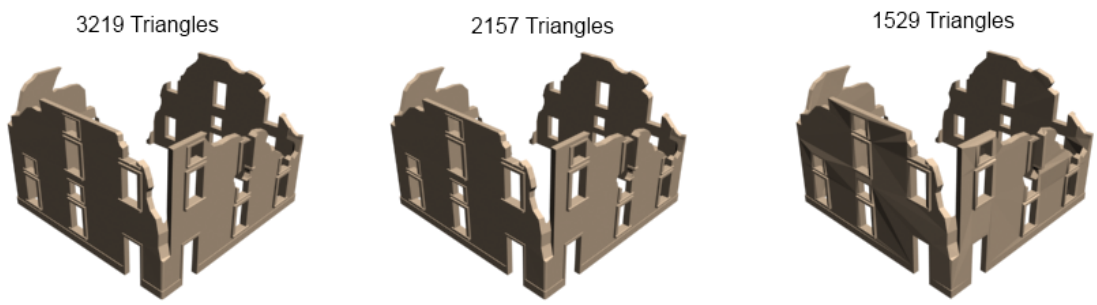



Figure 1.22: Series of optimizations applied to a model with increasing values for *m_maxDistance*

- `hkgpCgo::Config::m_maxShrink`, fraction between 0 (no shrinking allowed) to 1 (force shrink to model).

  By default, CGO will never shrink the model, thus guaranteeing that collisions are conservative. But in some situations where this constraint can be lifted, this setting can allow a better fit and produce more highly optimized models.

Figure 1.23: Optimized chair with and without shrinking.

- `hkgpCgo::Config::m_protectNakedBoundaries` and `hkgpCgo::Config::m_protectMaterialBoundaries` (booleans)

  By default, CGO will simplify naked edges (edges not shared by two triangles) and will optimize edges shared by different materials. This behavior can be changed with these two settings.

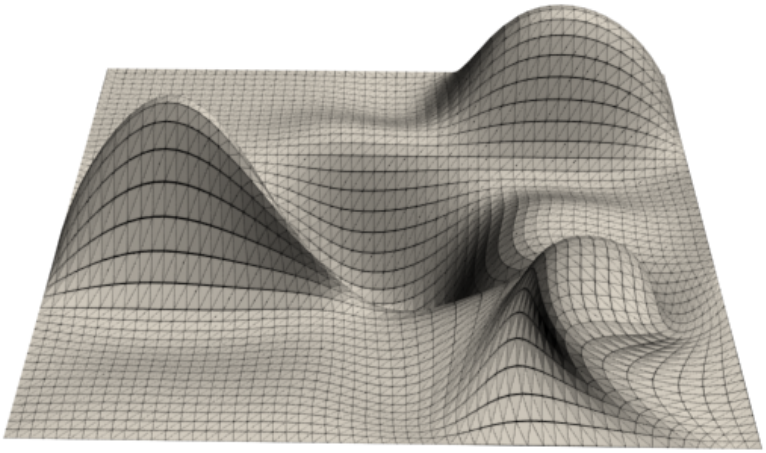


Figure 1.24: Base surface model with edges visible

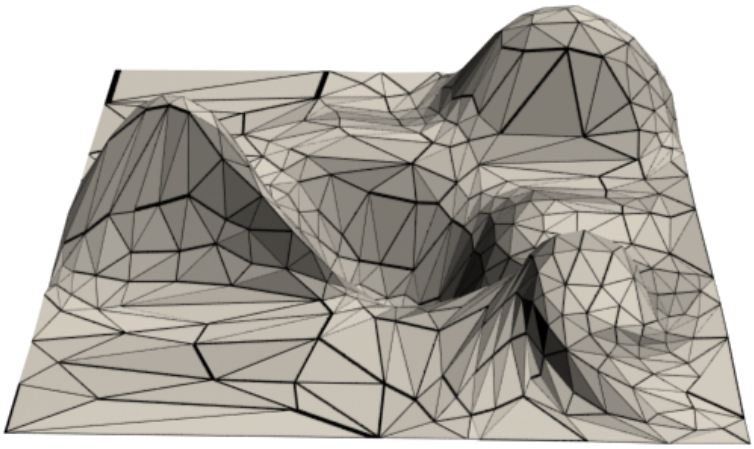


Figure 1.25: Optimized surface model without naked edge protection

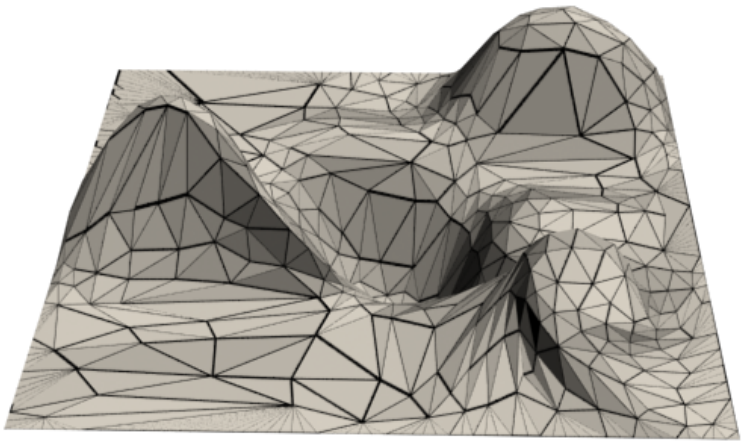


Figure 1.26: Optimized surface model with naked edge protection

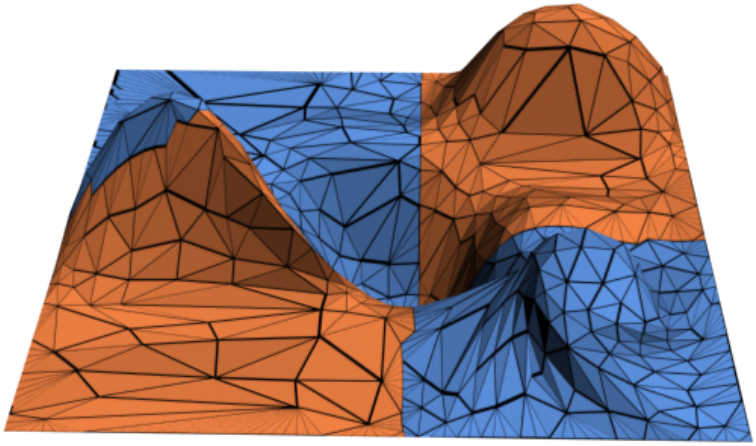


Figure 1.27: Optimized surface model with material boundaries protection

- `hkgpCgo::Config::m_semantic`

  By default, CGO will treat all vertices in the same manner (unless protected). This can be changed with this setting and the vertices W coordinate appropriately.

  This is very important feature that allow you to control optimization precisely and allows you

to highly optimize models and keep them usable for collision purposes (i.e. protect windows and doors).

| Value | Effect | Vertex W component. |
|---|---|---|
| `hkgpCgo::Config::VS_NONE` ( **Default** ) | None. | Unused. |
| `hkgpCgo::Config::VS_WEIGHT` | Control the relative maximum allowed error on a per-vertex basis. | Fraction of `m_maxDistance` between 0 and 1. |
| `hkgpCgo::Config::VS_DISTANCE` | Control the absolute maximum allowed error on a per-vertex basis. | Override the `m_maxDistance` setting. |

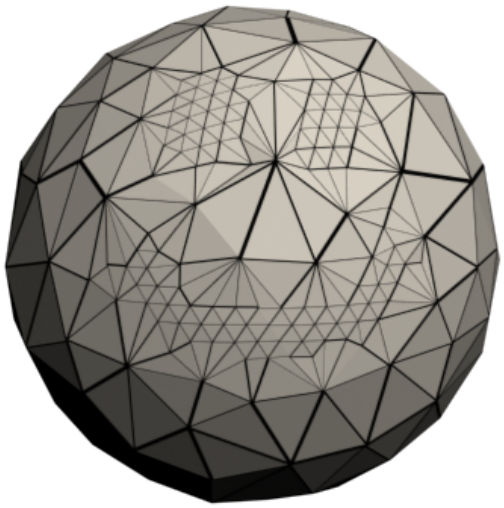


Figure 1.28: Smiley face painted in the W component of the vertices of a sphere

- `hkgpCgo::Config::m_multiPass`

  Once the value of `hkgpCgo::Config::m_maxDistance` as been chosen, enabling `hkgpCgo::Config::m_multiPass` can dramatically improve the results of the optimization. However this can be computationally expensive, and disabled by default.

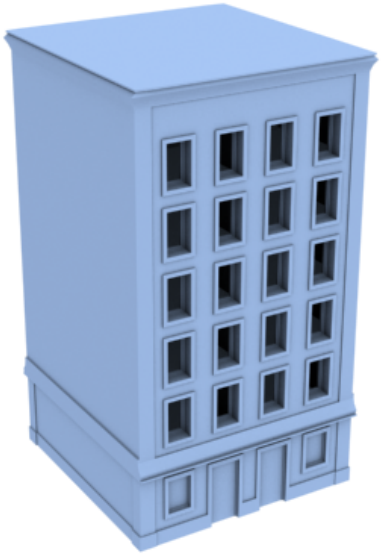


Figure 1.29: Base model, 7,518 Triangles

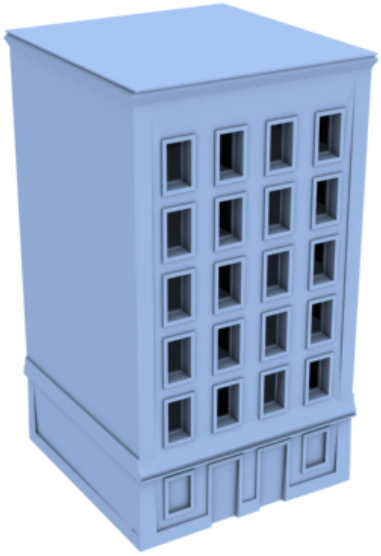


Figure 1.30: Optimized model, 5 cm error, single pass, 4,350 Triangles

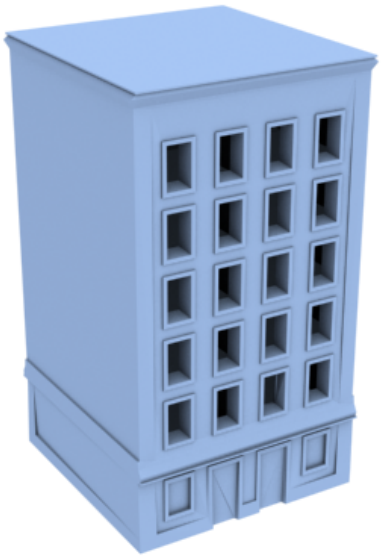


Figure 1.31: Optimized model, 5 cm error, multi pass, 3,484 Triangles

**Comparison with built-in modeler simplification methods**

In order to visually compare results, here is a representative sample of models simplified with CGO as compared to standard techniques used by common modeling packages.

For each picture, from left to right: Original model (red), CGO result (green), Other result (blue) with the same number of triangles.

In each picture, pay careful attention the loss of volume and shrinking introduced by other techniques which make them unsuitable for collision purposes.

Figure 1.32: Optimized chair model, from 788 to 208 triangles

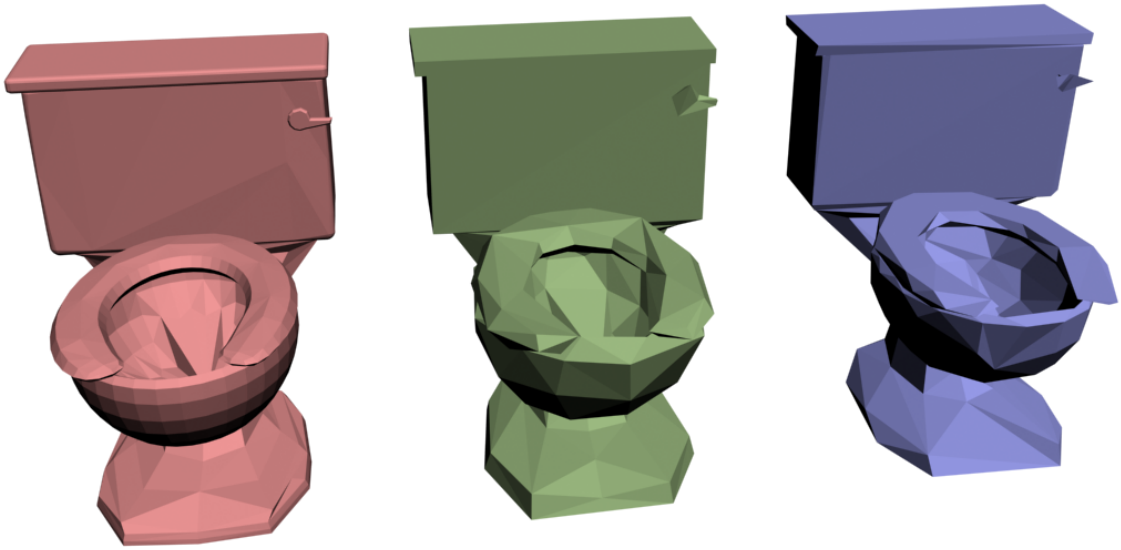


Figure 1.33: Optimized toilet model, from 2,398 to 456 triangles

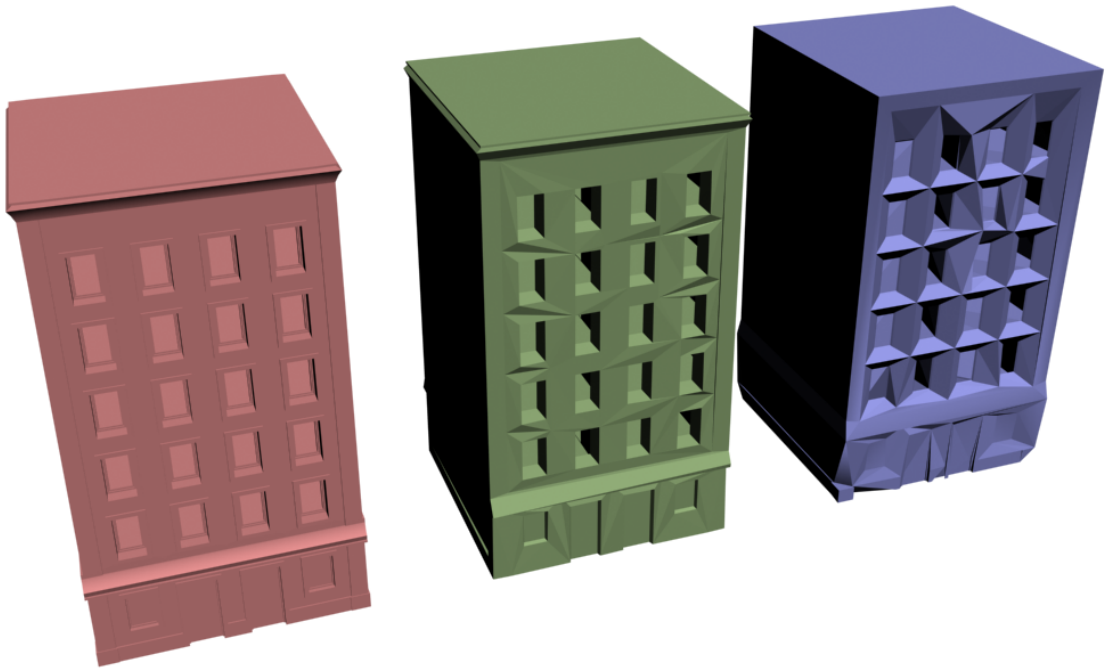


Figure 1.34: Optimized building model, from 7,518 to 1,922 triangles

## 1.7 Visualization

### 1.7.1 Introduction

Havok provides a framework to display debug information in a remote client, i.e. the Visual Debugger (VDB), or in your local application. The interface to this system is the singleton `hkDebugDisplay` which allows users to execute debug display functions like adding, updating and removing display geometry, updating the camera or displaying lines and text, all through a set of macros. `hkDebugDisplay` does not perform any actual displaying but forwards these calls to all its registered **debug display handler** objects which must implement the `hkDebugDisplayHandler` interface. There are currently two implementations of the debug display handler interface: `hkgDisplayHandler` is used by the demo framework to perform all displaying in the local application using Havok's render engine and `hkServerDebugDisplayHandler` is used to serialize and send all display commands to the VDB.

On top of this infrastructure to execute primitive debug display operations locally and remotely the visualization framework adds the **viewer processes** functionality to help visualizing interesting aspects of the system and the different products running on it, such as statistics captured through monitor streams, collision detection geometry in a physics simulation or navigation meshes in an AI powered application. **Local** viewer processes receive information about particular aspects of the system through **contexts** and produce visual outputs issuing calls to a debug display handler. **Remote** viewer processes send debug information to the VDB remote client and receive process control commands from it. All processes derive from the abstract class `hkProcess` that declares functions to initialize and step the process and holds references to the relevant contexts provided on construction and to a display handler. Viewer processes must be registered in the **process factory** , singleton `hkProcessFactory`, providing a

name and a creation function, and should be created through it as well. The process factory keeps a list of all available viewer processes to inform the VBD and help remote creation.

Note that this document refers to Havok Physics 2012 integration with the VDB, as it has not yet been updated to the latest version of Havok Physics.

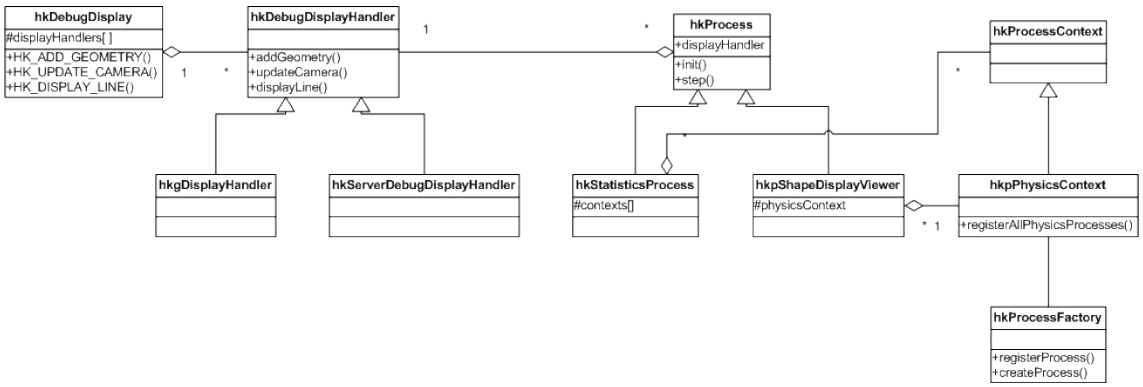



Figure 1.35: Simplified class diagram of the main classes of the visualization framework and some of its relevant members

### 1.7.2 Local visualization in your application

To use Havok's visualization framework to display visual debug information inside your application, both debug visualization macros and viewer processes output, you will have to implement the `hkDebugDisplayHandler` interface in your render engine.

These are the steps to follow to use your own implementation of `hkDebugDisplayHandler` (see Visualize demo for the full example):

1. Obtain a reference to your implementation of `hkDebugDisplayHandler`:

```
hkDebugDisplayHandler* displayHandler = yourRenderSystem->getDisplayHandler();
```

2. Create contexts for the aspects of the application you want to visualize:

```
hkpPhysicsContext* physicsContext = new hkpPhysicsContext();
physicsContext->addWorld(DemoWorldData.m_world);
contexts.pushBack(physicsContext);
```

3. Register viewer processes in the process factory and create the ones you are interested in:

```
hkpPhysicsContext::registerAllPhysicsProcesses();
processNames.pushBack(hkpShapeDisplayViewer::getName());
processNames.pushBack(hkpBroadphaseViewer::getName());
processNames.pushBack(hkpActiveContactPointViewer::getName());
```

```
hkArray<hkProcess*> processes;
for (int i = 0; i < processNames.getSize(); ++i)
{
    hkProcess* process = hkProcessFactory::getInstance().createProcess(processNames[i], contexts);
    HK_ASSERT2(0x30675a6b, process, "Failed to create - did you register all the processes for the product
      (s) you are using?");
    process->m_inStream = HK_NULL;
    process->m_outStream = HK_NULL;
    process->m_processHandler = HK_NULL;
    process->m_displayHandler = displayHandler;    // Your displayhandler used here
    process->init();
    processes.pushBack(process);
}
```

4. Inside the frame loop:

   Use `hkDebugDisplay` macros to display debug information:

```
HK_DISPLAY_FRAME( demoWorldData.m_transform, 5);
```

   Step processes at the end of the frame:

```
for (int pi = 0; pi < processes.getSize(); ++pi)
{
    processes[pi]->step(frameTime * 1000);
}
```

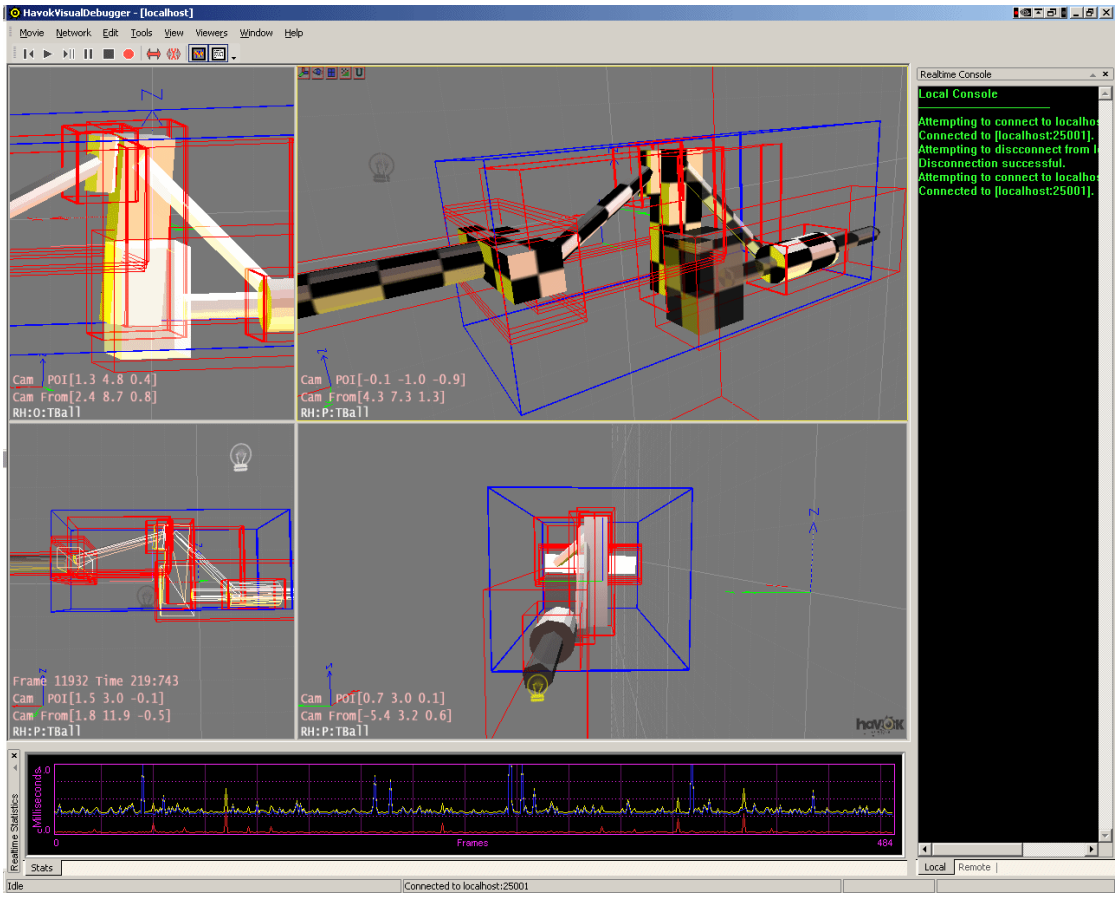
   Many viewer processes will not do anything in their step function because they are callback driven.

5. Release resources at the end of the application:

```
for (int i = 0; i < processes.getSize(); ++i)
{
    delete processes[i];
}

// Delete contexts. Some of these may be referenced objects, in which case it
// would be better (if you know their type) to use removeReference()
for(int i = 0; i < contexts.getSize(); i++)
{
    delete contexts[i];
}
```

### 1.7.3 Remote visualization with the Visual Debugger

#### 1.7.3.1 Introduction




The Visual Debugger connects from a PC (client) directly to your game running on a host platform (the game console, referred to as the Server) and displays a real-time graphical view of an active Havok simulation. These visualizations can be viewed in real-time or captured to a file for future analysis. File capture can also be used for submitting support queries. The ability to view the physics independently of the graphics integration can make it easier to find and correct errors in both your simulation code and the graphics integration. Detailed performance statistics (both performance timings and memory usage) can also be displayed and captured.

The local area network is used to stream the state of the physics world between the simulation - your game - and the Visual Debugger application in real-time. The Visual Debugger is implemented based on the client/server model:

- A stand alone Windows application that connects to your game and displays the state of the physics and user request information in real-time.
- A game side that links with your game along with the rest of the Havok runtime libraries. The game side also allows you to stream the state of the physics to a file directly without using the client application.

#### 1.7.3.2 Quick Start Guide

To get the Visual Debugger up and running follow the quick start guide for your specific platform.

**Connecting to the Demos**

Once you have completed all the platform specific preparation for the Visual Debugger you should be able to get it up and running with the demos that ship with your Havok package. For help compiling the demos please refer to the main Havok Quick Start Guide in the docs directory of your Havok installation.

- Run the Havok demos on a supported console or PC with -d as one of the command line arguments. You can use the `hkdemo.cfg` file in the demos directory to specify command-line arguments as well, especially if running on a console.
- Find out the IP address or network name of the game side (which is actually the server). Consult the platform specific quick start guides for information on how to obtain a suitable IP address. Please note that the Visual Debugger game side will only be created once a game has been run and the -d command line argument has been specified. Specifically, no Visual Debugger exists on the game side when browsing the available demos using the demos menus system, it is created when you run a demo and a Havok Physics 2012 world is created.
- Run the Visual Debugger application located under the tools directory in your Havok installation. The Visual Debugger application has been tested on the latest Microsoft Windows and requires the DirectX 8.1 runtime or later to function.
- Select **Network - Connect** and enter the IP address of the Visual Debugger game side where your game is running.
- The demo should be displayed in the Visual Debugger application main window. Try turning on and off some viewers by toggling the entries in the **Viewers** menu. Note that the **Viewers** menu will be empty until a valid connection has been established.
- You can also try to record a movie by selecting **Movie - Record...** and specifying a file name. Be warned: these files can be quite large as they are an exact dump of the network traffic from the server, but they compress well (using zip for instance) if you need to send them to Havok support.

#### 1.7.3.3 The Visual Debugger Game Side

In order to use the Visual Debugger (VDB) with a Havok Physics 2012 game, you must perform the following steps:

- Include the Visual Debugger and the physics related container:

```
#include <Common/Visualize/hkVisualDebugger.h>
#include <Physics2012/Utilities/VisualDebugger/hkpPhysicsContext.h>
```

- Instantiate a context so that the VDB will know the physics worlds it can visualize:

```
hkpPhysicsContext* physicsContext = new hkpPhysicsContext;
physicsContext->addWorld( myWorld ); // add all worlds as you have
```

We define a process as anything that gets run by the VDB, such as a viewer. A process context is a way of getting information to those processes in a non physics-specific way. When a process (viewer) is created it is given this array of contexts. As such the context can hold whatever you like. In this case we want one that holds physics related information, so we use a `hkpPhysicsContext` which is just a container that holds a `hkpWorld`s and listens to their deletion so that the list is always valid. Note that the `hkProcessContext` itself is not Havok memory managed, it is just an interface so that you can use your own container classes for whatever data you want to expose to processes you create and use with the VDB. The `hkpPhysicsContext` sub-class is however Havok memory managed.

- Register all processes (viewers) you want to expose to the VDB:

  There is a singleton called the `hkProcessFactory` that supplies any instance of the Visual Debugger with a list of what viewers are available (by name) and how to create them. When you create a Visual Debugger, it will know about the common processes that do not specifically depend on Physics 2012 or Animation. These common processes are the DebugDisplay and Statistics viewers. As we also want to be able to view physics specific data, such as broadphases and shapes, we need to register which ones we want. Normally you want to register all of them. `hkpPhysicsContext::registerAllPhysicsProcesses()` registers all the ones in hkpUtilities (it is a static method). If you have custom processes (viewers), register them with the `hkProcessFactory` too, but normally all you need to do is call:

```
hkpPhysicsContext::registerAllPhysicsProcesses();
```

- Create the Visual Debugger server:

  We can then create a Visual Debugger, given a list of the contexts that you want the processes to see.

```
hkArray<hkProcessContext*> contexts;
contexts.pushBack( physicsContext );
hkVisualDebugger* vdb = new hkVisualDebugger( contexts );
```

- Create a network game side:

```
vdb->serve(/* optional port number */);
```

  or open up a capture file to stream directly to a file:

```
vdb->capture(captureFilename);
```

Note: You do not have to call `hkVisualDebugger::serve(portNumber)` method to use the `hkVisualDebugger::capture(captureFilename)` or vice versa, they function independently.

- Finally you must step the display by calling:

```
vdb->step( frameTimeInMs );
```

once per frame. If several physics steps pass before the debugger is stepped, then the debugger will only display the positions of bodies at the *last* step. If you don't give the VDB a 'current time' to report, by leaving it at 0, then the VDB will use its own internal timer to work out the time since the last step call. The time is used to govern movie playback so that it can optionally happen in 'real time' and to provide information to the client during connection.

If you want to use the statistics viewers (highly recommended), then just before you call `step()` above, you should let the VDB know where all the timer data is. There is potentially one timer stream per thread used for Havok (see `hkMonitorStream::resize()` in the Base documentation). The code to pass each timer block would look like:

```
struct TimerStreamInfo
{
    const char* m_streamBegin;
    const char* m_streamEnd;
};
hkArray<TimerStreamInfo> threadStreams;

//
// In each of your worker threads in turn:
//
{PerThread} {
    TimerStreamInfo info;
    info.m_streamBegin = hkMonitorStream::getInstance().getStart();
    info.m_streamEnd = hkMonitorStream::getInstance().getEnd();
    timerStreams.pushBack(info);
}

//.. don't make any more Havok calls (or else the end pointer above will not take those times into account
  )

//
// In your main thread before vdb->step():
//
physicsContext->m_monitorStreamBegins.setSize(0);
physicsContext->m_monitorStreamEnds.setSize(0);

for ( int i = 0; i < threadStreams.getSize(); ++i )
{
    physicsContext->m_monitorStreamBegins.pushBack(threadStreams[i].m_streamBegin);
    physicsContext->m_monitorStreamEnds.pushBack(threadStreams[i].m_streamEnd);
}

vdb->step( frameTimeInMs );

//
// In each of your worker threads in turn you can now reset the mointors for the next sim step etc:
//
{PerThread} {
    hkMonitorStream::reset();
}
```

- Once you no longer require the VDB, delete it (it is reference counted) and whatever contexts you gave it. Any open network connections will be dropped and any open files will be closed.

```
vdb->removeReference();
physicsContext->removeReference();
```

An example of how to setup the Visual Debugger can be found in the hkDefaultDemo.cpp file that ships with your Havok package.

**Note:**
For the Visual Debugger to compile, the application must link with the appropriate libraries and in some cases specific hardware must be installed. Please consult the Quick Start Guide more information for the platform specific libraries required by the visual debugger game side.

### Required Havok Libraries

The following Havok libraries are required on all platforms in order to compile and link with the Visual Debugger game side:

- hkBase (where `hkSocket`, etc. are defined).
- hkMath (where `hkVector4` and `hkTransform` are defined).
- hkVisualize (where the class `hkVisualDebugger,` etc. are defined, and the non-physics related viewers).
- hkpUtilities (for the default physics viewers, but you can make your own viewers without it).

### Viewers

Viewers are the link between you game and the Visual Debugger game side. It is possible to write your own viewers. The current list of viewers that ship with the Havok SDK and their corresponding string IDs is as follows (common ones are in hkVisualize, the physics ones in hkpUtilities):

| Common Viewers | ID (Unique Name) | Description |
|---|---|---|
| `hkDebugDisplayProcess` | "DebugDisplay" | Shows the global debug points and lines output that result from using HK_DEBUG_LINE etc. |
| `hkStatisticsProcess` | "Statistics" | Transmits detailed performance statistics. See documentation on the hkMonitorStreams and HK_TIMER calls for more information. |

| Physics 2012 Viewer | ID (Unique Name) | Description |
|---|---|---|
| `hkpShapeDisplayViewer` | "Shapes" | Displays the simulation geometries. This viewer is used by default. |
| `hkpBroadphaseViewer` | "Broadphase" | Displays the broadphase AABBs (Axis Aligned Bounding Box) in red. |
| `hkpSimulationIslandViewer` | "Islands" | Displays the simulation islands as AABBs. Active islands are colored blue and inactive islands are colored green. |
| `hkpRigidBodyInertiaViewer` | "Tensor" | Displays a magenta AABB representation of the Inertia Tensor for each rigid body in the world. |
| `hkpRigidBodyCentreOfMassViewer` | "CentreOfMass" | Shows the center of mass of each rigid body in the world. |
| `hkpContactPointViewer` | "Contact Points (Active/Inactive)" | Displays all contact points in the simulation at the end of each frame. The contact points in active simulation islands are drawn in blue, the contact points in inactive simulation islands are drawn in green. This colour scheme matches the simulation island viewer's colour scheme. |
| `hkpToiContactPointViewer` | "Contact Points (TOI)" | Displays TOI contact points as a red arrow. This viewer has some limitations: The TOI contact point is drawn in the contactPointProcess callback. Users may modify the contact point in this callback, and the changes may or may not appear in the VDB. The SDK is free to call user callbacks in any order, so the VDB may draw the contact point before or after the user modifies it. |
| `hkpToiCountViewer` | "Toi counter" | Counts the number of TOIs a body has been involved in since the viewer was added and displays the number near the body. Also displays the number of TOIs for just the last frame. |
| `hkpConstraintViewer` | "Constraint" | Displays a graphical representation of all the constraints in the simulation. |
| `hkPhantomViewer` | "Phantom" | Displays the Phantoms in the world as yellow AABBs. |
| `hkpSweptTransformDisplayViewer` | "SweptTransforms" | Displays the Continous Simulation start and end frame for objects. |
| `hkpWorldMemoryViewer` | "WorldMemory" | Transmits per-step memory usage for all `hkpWorld`s (this asks each object how much it is using, so is relatively slow). |
| `hkpVehicleViewer` | "VehicleViewer" | Displays debug information, including the wheels and suspension, for any Havok vehicles in the simulation. |
| `hkpWeldingViewer` | "Welding" | Displays the snapped normals at every welded edge at the angle with the triangle normal. |
| `hkpInconsistentWindingViewer` | "InconsistentWindingViewer" | Shows pairs of inconsistent winding triangles which are important for welded meshes. The triangles are coloured in green and the common edge in red. Note that this viewer will only work if you select the option "Mark Inconsistent Winding Edges" in the Create Rigid Bodies filter. |
| `hkpWorldSnapshotViewer` | "* Grab World Snapshot (Xml/Binary Tagfile)" | On selection will send a snapshot of the world to the Visual Debugger. A dialog box appears where the destination filename can be specified. The snapshot can be saved as either an XML or a binary tagfile. |

### Using Viewers

Once a successful connection has been made to the visual debugger application a list of all the available viewers is presented under the **Viewers** menu. Using this menu viewers can be created and destroyed at runtime.

You specify that you want to use a particular process (viewer) by default using the VDB's `addDefaultProcess(const char*)` method and passing it the appropriate string ID (listed above). This registers interest in that process on the game side. This means that when a Visual Debugger application connects, an instance of that viewer will be instantiated by default.

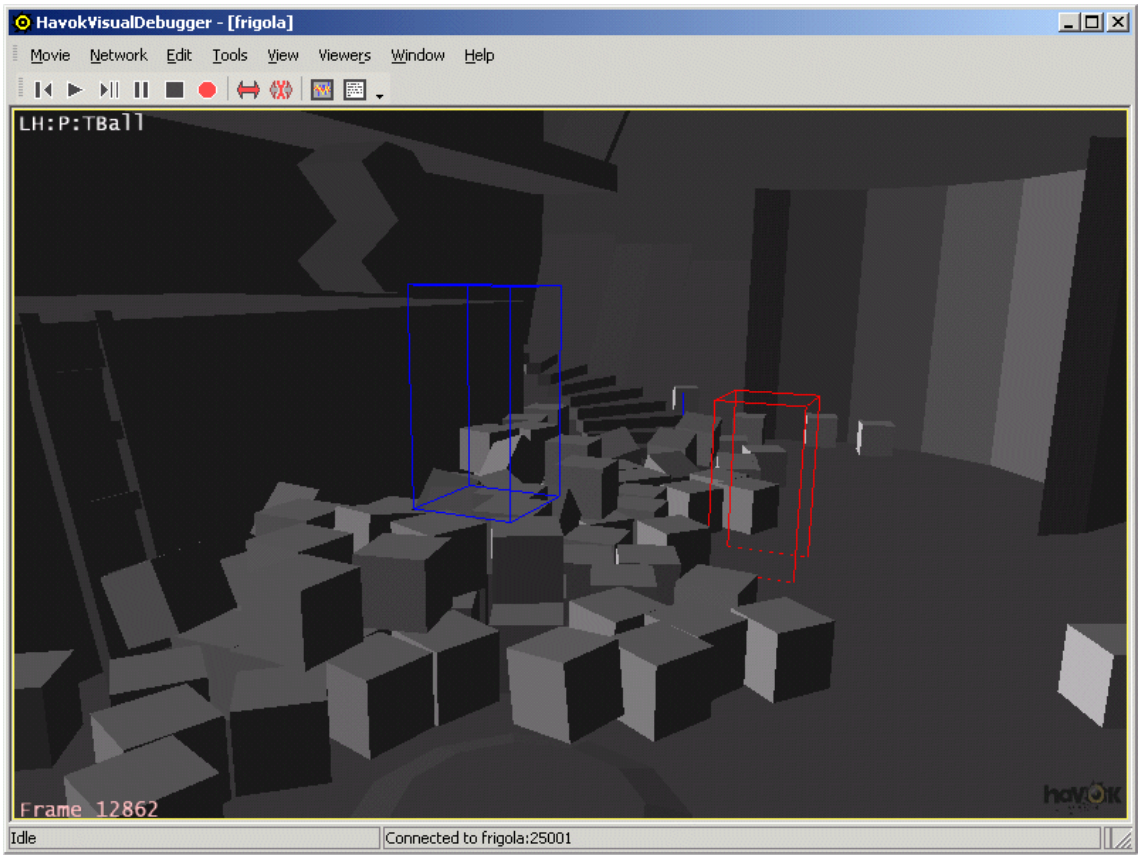
Similarly, you can remove interest in a viewer (such as the default `hkpShapeDisplayViewer`) using `removeDefaultProcess("Shapes")`. However, this will not remove the viewer from any existing in-

stances of the visual debugger application. You can always enable and disable viewers from the Visual Debugger application. The default list only specifies which viewers will be enabled by default. This is especially useful when capturing directly to a file using `hkVisualdebugger::capture(...)`. You can see examples of how to add viewers in the `hkDefaultDemo.cpp` included in your Havok package.

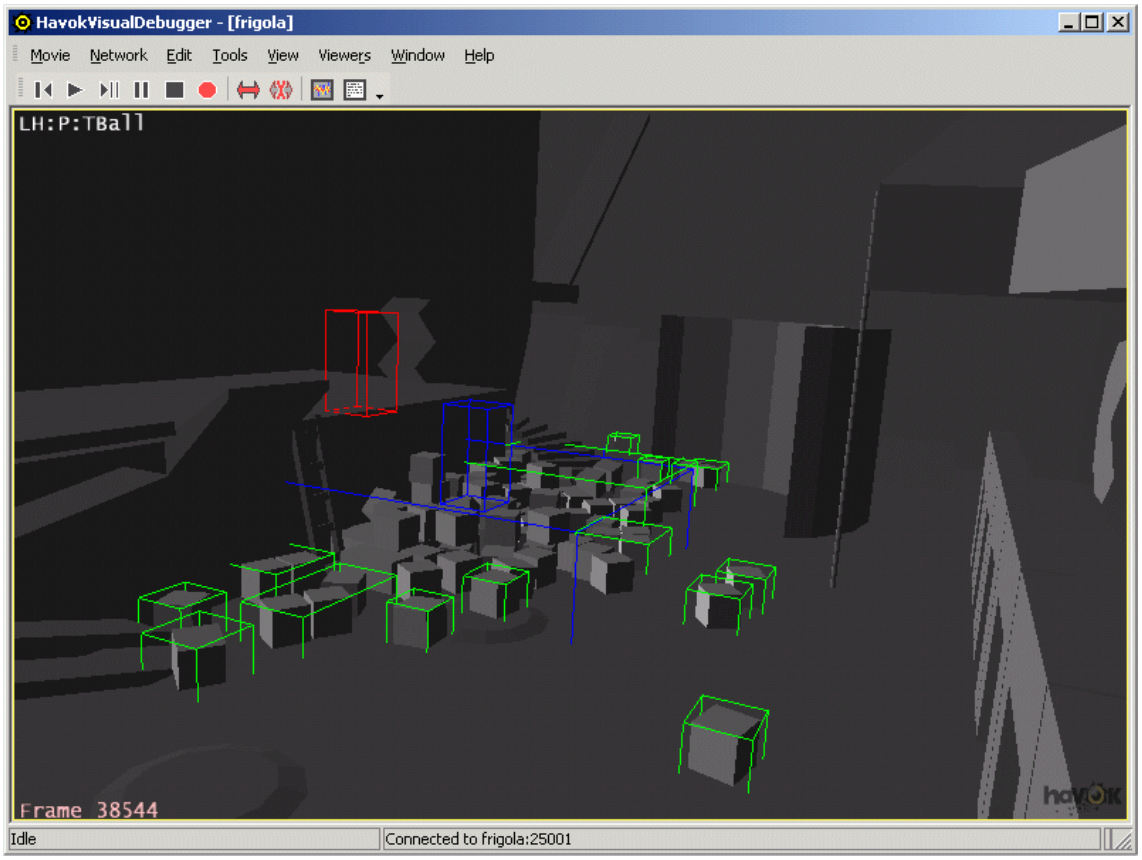
From a performance point of view, bear in mind that as more viewers are instantiated more network bandwidth will be consumed, which can slow down the simulation. The Statistics based viewers (performance and world memory) are the largest bandwidth users.

### Viewers Examples

By default, the Visual Debugger enables just the `hkpShapeDisplayViewer` and the `hkDebugDisplayProcess`. This displays the geometries that exist in the simulation and all the global debug points and lines. In the following example image you can see all of the rigid bodies, the landscape and the AABBs of the characters. The debug display was used to display the character AABBs. This is a convenient way to display information that is not automatically displayed by the Visual Debuggers' built in viewers. The debug display is described in detail below.



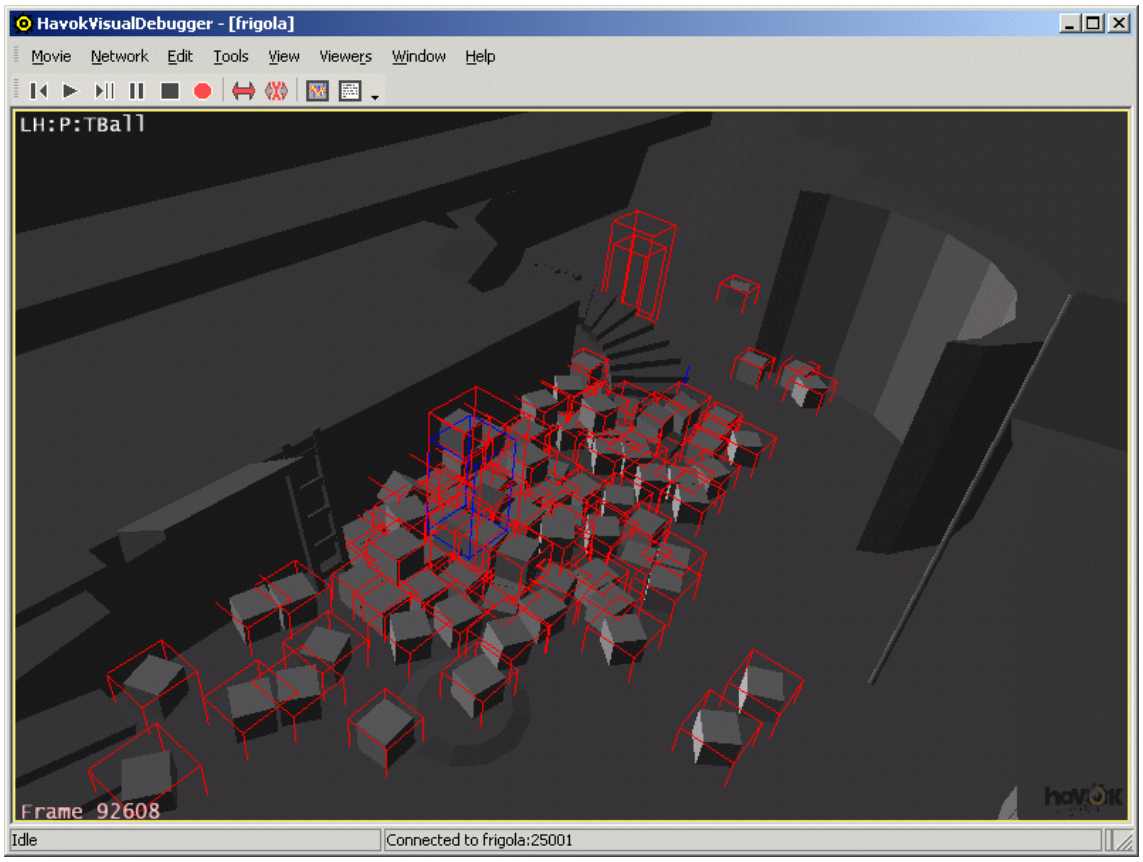

The `hkpSimulationIslandViewer` lets you see information about the simulation islands, such as which objects are being simulated together and which islands have been deactivated:



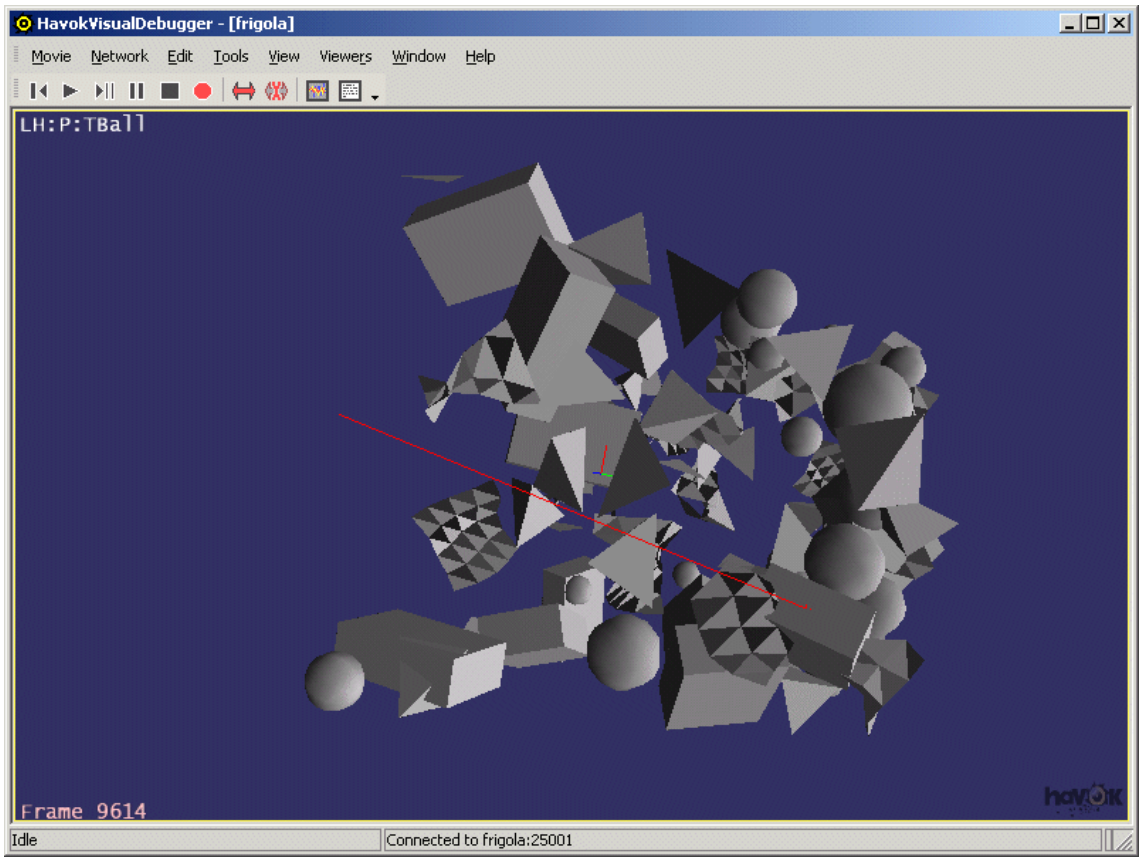

As you can see, inactive islands are shown in green and active islands are shown in blue. Fixed bodies have no island information displayed, as they do not take part in the simulation, they do however take part in the collision detection process. To have the simulation island information displayed by default (so that you don't have to manually select it in the UI) simply call the `addDefaultProcess("Islands")` on the visual debugger game side just after it has been created:

```
myVdb->addDefaultProcess("Islands");
```

In the next example, the `hkpBroadphaseViewer` has been turned on in the debugger by selecting the Broadphase viewer from the Viewer menu. Note that the Viewer menu will not be present (or will be empty) until it has connected to a game. The Broadphase viewer displays the AABB used for each object in the collision detection broadphase.




Debug points and lines are a useful tool for debugging your game. They can be used to debug or visualize anything from forces, velocities and directions to game logic and AI. All debug display output can be viewed by turning on the Debug Display viewer (`hkDebugDisplayViewer`), and only debug information issued after the viewer is turned on will be viewed. Again the simplest way to do this is to select it from the Viewers menu in the Visual Debugger application. In the example below the raycast is visualized using red user-defined debug lines.




To display debug points and lines you can call the following macros in your code:

```
HK_DISPLAY_POINT(position, color);
HK_DISPLAY_LINE(start, end, color);
```

The example below outlines how to display a debug point and a debug line. Note the header file required to use the macros:

```
// Comment in the line below to compile out the debug points and lines...
//#define HK_DISABLE_DEBUG_DISPLAY
#include <Common/Visualize/hkDebugDisplay.h>

...

void drawMyLines()
{
   ...

   hkVector4 start(0.0f, 0.0f, 0.0f);
   hkVector4 end(0.0f, 100.0f, 0.0f);
   HK_DISPLAY_LINE(start, end, hkColor::BLUE);

   hkVector4 position(1.0f, 1.0f, 1.0f);
   HK_DISPLAY_POINT(position, hkColor::WHITE);
}
```

The section entitled "The Debug Display" details all the functionality provided by the debug display.

**The Statistics Viewer**

This viewer is unique in that it does not display information in the graphical viewports. Instead it outputs information to special real-time graph windows. There are 6 windows, one for each virtual channel in the timer stream.

By default, channel 0 is the performance timer on most platforms. This graph can be enabled by selecting the **Window - Perf Ch0** and so on for each of the 6 channels, or by selecting the appropriately numbered Stat icon in the default toolbar. In a multithreaded environment such as the PC or the Xbox360, the 6 channels represent timer 0 from each of the 6 potential threads. For more information on the graph display see the section entitled The Real-time Statistics View.

For Havok statistics to function properly the follow extra requirements must be met:

- Call `hkMonitorStream::resize(bufferSize)` at least once. Specify a buffer large enough to capture all the stats for a single frame. The size of a frame is typically at least 100 kilobytes (normally 1MB if you have a complicated simulation). If you want to check that there is enough memory call `hkMonitorStream::memoryAvailable()` at the end of a frame before calling `hkMonitorStream::reset()`. If it returns false then you may have run out of memory.
- Call `hkMonitorStream::reset()` at the beginning of every frame OR at the end of every frame but AFTER `hkVisualDebugger::step()` is called. (`hkVisualDebugger::step()` must be called for the Visual Debugger to function properly.)

If you are not obtaining statistics as you expect then double-check the following:

- The Statistics viewer is enabled and you have a valid connection to your game. Note that, in the case of a movie, the viewer will have to have been turned on when the movie was created. You can check if a movie has statistics by looking at the **Viewers** menu during playback. If there is a Statistics entry and it is checked (but grayed out) then statistics are enabled.
- Check that enough memory is being allocated for the `hkMonitorStream`.
- Check that the range of the real-time graph is suitable for your statistics.
- Check that the statistics you are interested in are selected in the Select Stats or Legend dialogue boxes (right-click on the Real Time Statistics View).
- Ensure that your game is calling `hkMonitorStream::resize(bufferSize)` and `hkMonitorStream::reset()` appropriately.

In a multithreaded simulation, you will notice that the performance in each channel is quite variable. This does not represent spikes in the simulation itself but simply that the jobs are being done in different threads each frame. When you step a multithreaded simualtion, the first thread to call `stepWorld()` is the master thread and takes more of some types of jobs than others. The remaining threads take jobs as they arise and add more to the job queue when necessary and so on.

Thus depending on which thread gets in first and how the subsequent threads arrive to `stepWorld()`, they will get different jobs each frame. The order of work is dependent on platform thread scheduling. The overall work is the same, just the statistics for the jobs will jump from one channel to another each frame.

### The Debug Display

The Visual Debugger game side incorporates a debug display (class `hkDebugDisplay` in `hkvisualize/visualdebugger`). The header file defines many global macros which can be used anywhere in your code. Remember though that the calls have immediate effect only, so if the DebugDisplay Process is not enabled the calls will do nothing (apart from also going to the default local debug display if one).

Here is a list of all the macros available:

```
HK_DISPLAY_POINT(position, color)
```

Displays a debug point in your scene. The position should be specified as an hkVector4 and the color is an unsigned int in 0xARGB format. You can also use the `hkColor` class for ease of use.

```
HK_DISPLAY_LINE(start, end, color)
```

As above but takes a start and end point to draw a line.

```
HK_DISPLAY_TEXT(text, color)
```

Sends text to the console in the Visual Debugger application. This text is displayed in the Remote tab of the dockable Real-time Console view. Currently the color has no effect in the Visual Debugger application.

```
HK_DISPLAY_GEOMETRY(geometries, color)
HK_DISPLAY_GEOMETRY_WITH_TRANSFORM(geometries, transform, color)
```

Displays an array of geometries in wireframe. The array is a `hkArray` of `hkDisplayGeometry` pointers (i.e. a `hkArray<hkDisplayGeometry*>`). The `hkArray` class is documented in the Havok Base Library chapter. The `hkDisplayGeometry` class and its derivatives (such as `hkDisplayAabb`) are in the hkVisualize Havok library. In HK_DISPLAY_GEOMETRY_WITH_TRANSFORM, a transform can be specified to position the array of geometries in the world; transform is a reference to a `hkTransform` (see the Havok Math Library documentation for more information). `hkDisplayAabb` is used by the Broadphase Viewer to efficiently transmit AABBs over the network by sending only the min and max extents (6 floats) as opposed to 12 individual debug lines (70 floats!).

The following macros can also be used but are a little more tricky to work with:

```
HK_ADD_GEOMETRY(geometries, transform, id)
```

Adds a geometry to the display world which can be updated by setting its transform using the HK_UPDATE_GEOMETRY macro and removed from the display by calling the HK_REMOVE_GEOMETRY macro below. The geometries parameter is a `hkArray` of `hkDisplayGeometry` pointers as described above. These geometries form part of the same local coordinate space whose position and orientation is governed by a single transform.

The transform is a reference to an `hkTransform`. The id is an unsigned integer and should be unique with respect to all other geometries added to the display world. The convention used by the Shape Viewer for example is to use the pointer to the Havok `hkpCollidable` instance as an id. You can use whatever convention you like as long as your ids are unique.

```
HK_SET_OBJECT_COLOR(id, color)
```

This function will set the color of an object that has been added to the display world. Please note that this function does have some limitations. For example, if you set the color on an object and then later connect to your game with the Visual Debugger, the color change will have been missed. However for events that happen periodically this is not usually a problem.

```
HK_UPDATE_GEOMETRY(transform, id)
```

Updates the transform of a geometry that has been previously added to the display world with a call to HK_ADD_GEOMETRY. The id must be the same as the id used in the original add geometry call and the transform is a reference to an hkTransform.

```
HK_REMOVE_GEOMETRY(id)
```

Removes a geometry from the display world. The id should be the same as the one used in the corresponding HK_ADD_GEOMETRY call.

```
HK_UPDATE_CAMERA(from, to, up, nearPlane, farPlane, fov, name)
```

The update camera command allows the user to specify their own cameras. Use this to have the Visual Debugger track your vehicles and characters in your game. You can supply as many as you like and update them as often as you like. (Please note that only the first 32 will be listed in the Visual Debugger applications drop down menu!). The from, to and up parameters are specified as Havok `hkVector4`. nearPlane and farPlane are specified as `hkReal` (which is just a float by default) and name is a standard c string identifying the camera.

**Customizing Shape Viewers**

The `hkpShapeDisplayViewer` class can be extended to allow the display of shapes in the VDB based on some criteria. As an example consider the display of Trigger Volumes. A Trigger Volume is a shape with a certain property, so we might want to be able to view them separately from any other shape. The `hkpShapeDisplayViewer` class has already been extended to handle Trigger Volumes, so we will use this

as a case study for implementing our own shape viewers. Other scenarios could include the filtering of shapes based on material (`hkMaterial`), collision filters (`getCollisionFilterInfo`) or position. To that end we define a filter function that decides whether or not a specific shape should be displayed, write a creator function that instantiates our customized viewer, and register the view process with the process factory.

In the example below we first define a filter function:

```
hkBool HK_CALL hkpShapeDisplayViewer::isTriggerVolume( hkpEntity* entity )
{
    return entity->hasProperty(HK_PROPERTY_TRIGGER_VOLUME);
}
```

This function checks whether a given entity is a Trigger Volume by checking its properties. If the function returns true the entity will be displayed by our viewer.

Next we provide a creator function that will instantiate our viewer:

```
int hkpShapeDisplayViewer::s_tagTriggerVolumeViewer = 0;

hkProcess* HK_CALL hkpShapeDisplayViewer::createTriggerVolumeViewer(const hkArray<hkProcessContext*>&
  contexts )
{
    return new hkpShapeDisplayViewer(contexts, s_tagTriggerVolumeViewer, isTriggerVolume);
}
```

This function will create a new Shape Display Viewer which is customized to only display Trigger Volumes. The process tag (`s_tagTriggerVolumeViewer`) is assigned when the process is registered; see below. We also specify the filter function to be used with this viewer (`isTriggerVolume`).

Lastly we extend the `registerProcess()` function to register our viewer with the process factory:

```
void HK_CALL hkpShapeDisplayViewer::registerViewer()
{
    s_tagShapeViewer = hkProcessFactory::getInstance().registerProcess( "Shapes", createShapeViewer );

    // This line is added
    s_tagTriggerVolumeViewer = hkProcessFactory::getInstance().registerProcess( "Trigger Volumes",
      createTriggerVolumeViewer );
}
```

Here we added a line to register the viewer process using a distinct name (i.e. "Trigger Volumes"). The already existing shape viewer ("Shapes") is created and registered in the same way, that is with a creator function and a filter.

#### 1.7.3.4 The Visual Debugger Application

This is a GUI application which runs on Microsoft Windows XP/Vista/7. The user interface is configurable and has several components: a graphical viewport, a menu, a configurable tool bar system and dockable components for displaying console output and real-time statistical information. Each area has a context sensitive menu.

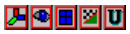
By default the application uses Direct3D, but you can make it use OpenGL if you so desire by using the *-ogl* flag at the application's command line. Some cards do not have good OpenGL support , but if you are experiencing visual difficulties try the OpenGL renderer, and vice versa. The main difference (feature-wise) between them is that the Direct3D code is currently limited to 65K vertices per buffer, so large objects (landscapes etc) may exceed this and some crazy triangles will pop up. This will be fixed in future releases. By default, the VDB will first try DX 9.0c, then DX 8.1, and finally OpenGL.

For multiple monitors, the VDB will initialize by default on the adapter (monitor) that you last used. So if you start on your primary monitor, and want it on your secondary, simply pull the window on to the secondary and then the next time you run the VDB it will initialize on that secondary monitor (so giving full speed in DirectX etc). If you want to force the VDB to a specific montior (and ignore the last window placement), use the command line option *-device0* and *-device1* (etc.) command line options. They will use the given number device to initialize on and will run the VDB maximized in the work area of that monitor.

All menu item shortcut keys are customizable through Tools->Customize and the UI keys for movement are customizable through the Edit->User Keys menu. The speed of movement is also customizable through the Edit->Movement Speeds menu. For further customizations, contact support@havok.com.

**Viewports and Cameras**

The view port area can be configured to contain 1, 2 or 4 viewports and each of these can be used to display a graphical representation of different parts of your scene. The user can manipulate the camera in each viewport in many ways using several navigation modes. These modes can be selected via hot keys (customizable) or through the View menu in the main menu bar. This view menu is also available in the top left hand corner of the current viewport:



The first icon is the Axis icon, and holds the View->Axis submenu. The second icon is an eye and holds the View->Camera submenu. The third icon is the View->Viewports submenu that allows you to configure the number of views. The fourth icon is the View->Render State submenu. The last icon, shaped like a U, only appears if there are User Cameras present. A User Camera is a camera sent by the game server to the Visual Debugger.

**View –> Axis**

Games tend to vary a lot in the convention they use for handedness (left or right) and the direction of axis relative to the viewer. One axis can go any of 6 directions, one of the others can then go one of four directions (relative to viewer) and then the other is fixed for a given handedness. Luckily the VDB can display all combinations. First choose your handness in the Axis menu (use Left or not). Then choose the Up axis as either X,Y or Z. Finally choose if the Up axis is the positive or negative version of that axis.

You can see the Handedness of the current view in the lower left hand side of the window. For example, if

you see *RH:P:TBall*: RH means Right Handed, P means Perspective Mode, and TBall means Trackball camera control.

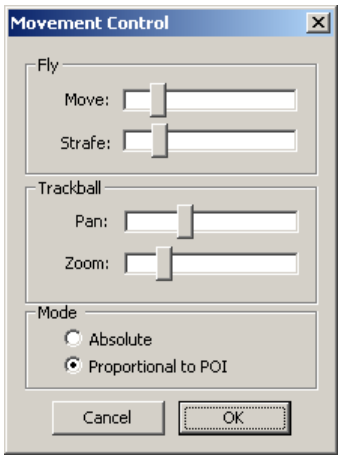
The other options in the Axis menu toggle the main Unit axis at the origin (Unit Axis), the unit axis at the Point of Interest (often referred to as POI in this doc) and the grid on the plane perpendicular to the Up axis. This grid will either be in units of 1 or 10, depending how close the viewer is to it. It is easy to see which resolution you are in if the POI axis is enabled.

**View –> Camera**

Camera control is always an issue with an application that tries to cater for both large and small scales as well as users that prefer lots of different modelling packages. We provide two common modes: a virtual trackball (the SGI style) and a free-fly mode (a Quake style God-mode). The keys for both are mostly customizable, as are the speeds at which they operate. As it is handy to flick back and forth between the two modes, we recommend you choose a handy short-cut key for each.

In trackball mode (indicated by TBall in the viewport information label) the world can be rotated by left clicking with the mouse button, and panned with the right button, SHIFT-left or middle button will zoom (the Shift key is the 'Secondary' key in the User Keys dialog). The mouse wheel can also be used to zoom. All viewports will be in this navigation mode by default.

Free-fly mode (indicated by Fly) allows the user to fly through the scene using the WASDQZ keys (also customizable) and steer using the mouse while holding the left mouse button down. Strafe is the right mouse button and does the same as the ADQZ default fly mode keys (left, right, up, down). Notice that a set of coordinate axes are displayed (of length 1 unit) at the camera's to position. The speed at which the camera flies through the scene is proportional to the distance of the camera to the POI. This can be adjusted by zooming using the wheel on the mouse or through the middle mouse button, or you can change the overall speed of the fly mode in the Edit->Movement Speeds dialog:




The speed sliders range from .1 to 10 times the default speed, and control the fly mode and the trackball mode. The Mode radio button at the bottom governs whether you want the speed to be absolute (same speed no matter what the distance to the POI is) or proportional to the distance to the POI (default, and handy if you can position your POI correctly).

**Camera Object Tracking**

There are two options to allow the camera to track the currently selected object local origin or the point-

of-interest (POI) on the selected object. Select is the Space Bar by default, for the object under the mouse pointer, and the POI is taken as being the exact point of the pick. The camera POI will either be set to the tracked object origin or the selected POI on that object. The camera From position will optionally be kept the same distance away if the relative option is chosen.

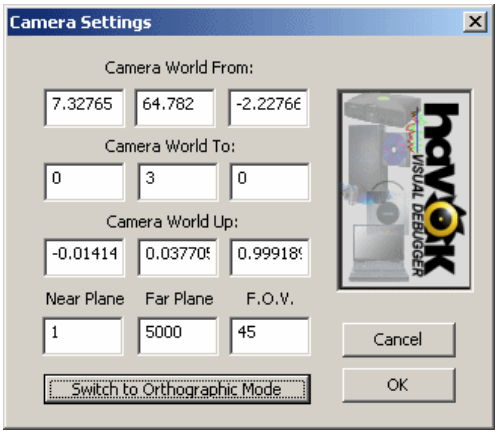
**Camera Fitting and Roll Back**

You will notice the ability to Store the current camera (one stored per viewport) and recall it at a later time (Reset to Stored). The camera you have at the application shutdown is the camera at the start of the next, and also the default stored camera. The Default camera is a trackball mode camera, 10 units along Z, 10 units along Y, with +Y up and Right-Handed.

Fitting the camera to the world or current selection uses a simple computation of what distance the camera has to be along the plane perpendicular to the Up axis to see the whole bounding volume (given a 45 degree field of view). It does not take occluding objects into account so it is easiest to use this fitting in trackball mode and just rotate around, as the POI will be set to the center of the bounding volume and thus you will spin around it. You can choose to reset the POI to any object via the Use Selected POI which will change the POI to the current selection point, but unlike tracking will not update it from that set point every frame. Thus it is handy to map this to a key near the select key on the keyboard so that you can select and re-position easily.

**Exact Camera Settings and Projection Mode**

Through the View->Camera->Camera Settings menu you can change the exact vector basis that makes up the camera as to position the camera exactly where you want:




The Field-of-View (FOV) is measured vertically. Notice that you can switch the mode between Orthographic and Perspective projections. With Orthographic mode, even though you specify the camera as a perspective camera (with F.O.V.), the camera internals will compute an orthographic matrix such that the same amount of the world will be visible at the given Point of Interest (To vector) and the given field of view.

Don't be tempted to set the far clip plane too far from the near plane as the accuracy of the Z buffer has to cover the whole space between the two planes, so a small near plane value and a very large far plane value will result in a reduced accuracy and you will get artifacts.

**View –> User Cameras**

Viewports can also be set to lock to User Cameras, which are sent from the user's game (see the end of the Debug Display section for more information on generating user cameras). When recording a movie all cameras (all Viewports and User cameras) get recorded and are available during playback.

The User Camera menu is only populated when there are user cameras present. For instance, when the Havok Demo Framework is running, it sends its view as a user camera. You can have as many different user cameras as you like. They can be used to follow each AI character or car.

**View –> Viewports**

This allows you to switich the number of views on the world (between 1, 2 and 4). Most options are specific to a view so this can be handy to not only look at the world from a different camera but also from a different rendering style. To resize the proportion of the views, hold down the right mouse button on the splitter bar of the two views, or the cross over of the two splitter bars in the quad view.

**View –> Render State**

The viewports allow for some simple configuration of the render state: Wireframe, Culling, Blending, Textures, ShowLights and Background color. The first five are viewport specific, so you can set them on a preview basis. The last option, the Background color, is across all views (as it is the clear color of the render device).

The Wireframe mode is self explanatory, as is the Backface Culling option. Additive Blending toggles the render state between 'modulate' alpha blending and 'additive' blending. With additive blending the alpha component drives the color component towards white when combined, so gives an X-ray-vision-like effect. The modulate preserves the object color more, but is affected by the rendering order of the objects (and the objects are not sorted in the rendering engine).

The Texture option enables a default checker texture that is mapped such that it repeats every 1 unit in world space (so each full black square in the texture is 0.5 x 0.5 units). This texture in conjunction with the grids and POI unit axis can really help in figuring out sizes in the scene. The ShowLights toggles the rendering (and thus picking) of the lights in the scene.

**Picking and The Current Selection**

The default key for picking is the Space Bar. It will select the closest icon or object to the camera at the point under the mouse pointer. While the selection key is pressed the VDB will attempt to move that object to the new position. If the object is a light, then it is easy to move it exactly to where you want. If it is a geometric object from a movie then obviously you can't change its position.

When connected to a live simulation the scene can be manipulated with the mouse. Objects that represent Havok rigid bodies can be picked up and dragged by pressing and holding the select key while pointing at a geometry. Releasing the space bar will release the object.

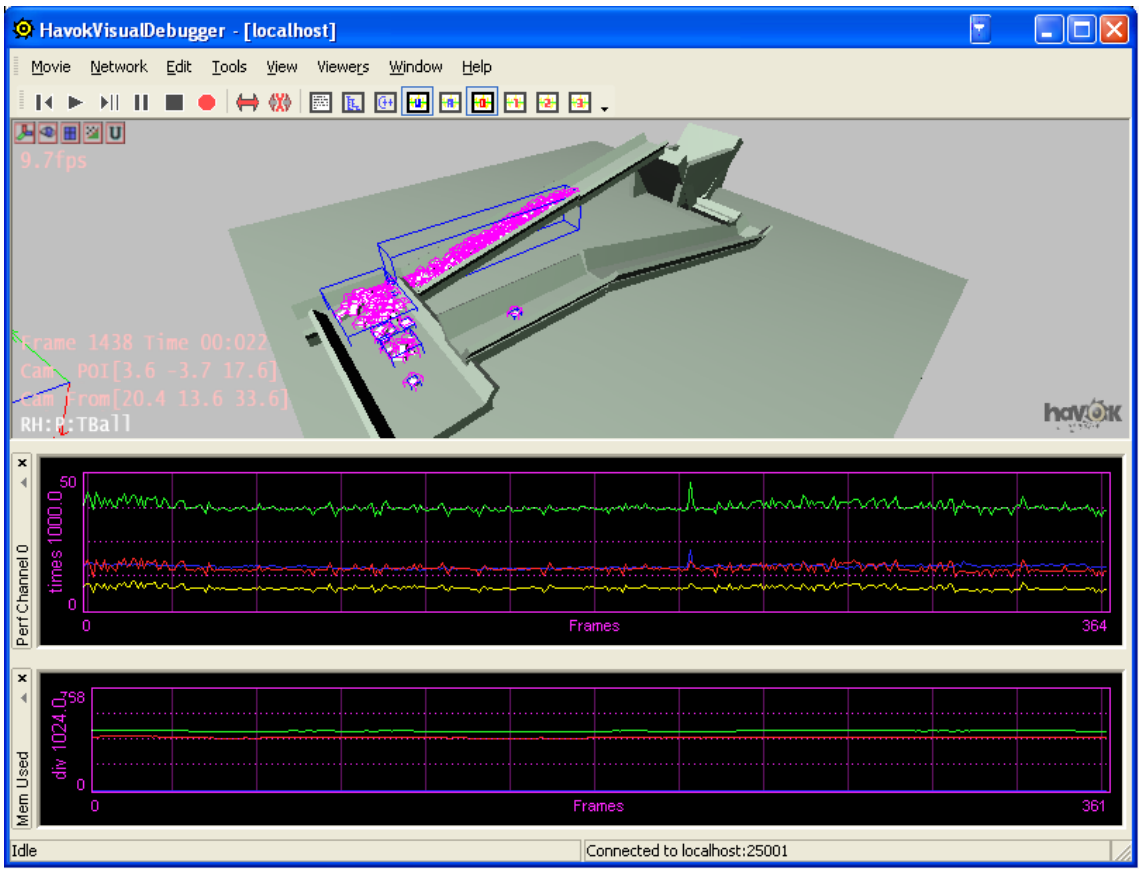
Note that if you double-left-click on a display object you will get a summary of display geometry details. This does NOT directly relate to the Havok collision shape data as it has been converted into a graphical form. It is only there as a temporary feature which people may find useful in the absence of more sophisticated meta-data. The color and alpha value of the display object can also be changed from this dialog box. This color is not persistent in movies or across reconnects.

In that double-click object dialog there is an option to save to file. The format you can save the display geometry to is a .ttk file, which is the normal Havok ASCII .tk file, with a transform appended to the end, as a line of 16 floats, which is column major and follows the same conventions as a `hkTransform`.

To create a `hkTransform` from it just skip the last row (every 4th elem of the 16 floats) to make a 3x4 array. This is so that you can save all the objects from your runtime (or in the case of Havok from your movie file) and re-create an approximation of the actual scene. In future it is planned that the VDB will be able to query for the actual Havok Serialization packets and save them to XML and allow for exact replication of the scene.

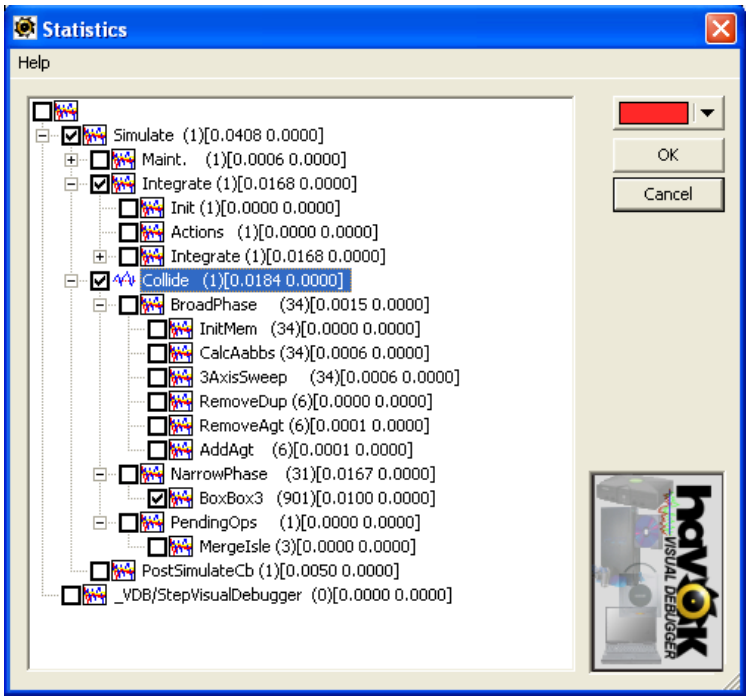
### The Real-Time Statistics View

The image below shows the Visual Debugger displaying real-time performance statistics for a game. This information is transmitted from your game to the Visual Debugger using the statistics viewer (see hkvisualize, class hkStatisticsProcess). For information on how to setup the statistics viewer on the game side see the section above entitled The Statistics Viewer.



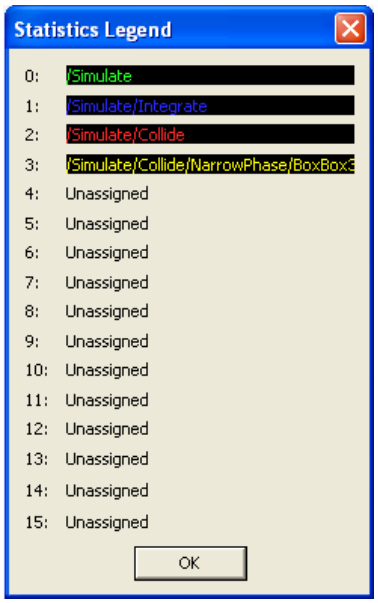

The Real-time Statistics output can be displayed by selecting the **Perf Ch0** option from the **Window** menu. Note that no information will be displayed by this view unless the Statistics viewer has been enabled. To enable this select it from the **Viewers** menu when connected to your game. To display the statistics you are interested in you must use the **Select...** option from the context sensitive window, (as in the image below). Simply right-click on the dockable Real-time Statistics panel. This will bring up a dialog box which displays a hierarchical representation of all the Havok performance timers in your game. Check the box beside the statistic of interest and select an appropriate color for it.

If you are not getting statistics as you expect, there is most likely a problem on the game side. Please refer to the section entitled The Statistics Viewer for help.




Initially the statistics are assumed to be in milliseconds. Depending on the typical values of your statistics you may need to adjust the range of the graph by using the **Options...** option on the context-sensitive menu.

A legend of the currently selected statistics and their corresponding color can be displayed with the **Legend...** option:




Although there are 6 destination stat channels, not all may be updated each frame. The meaning depends on what channels you target with the FrameInfo in `hkStatisticsProcess::step()` and what setup you

have for the perf counter. You can alter this and recompile the viewer to suit what timings you want. On Xbox360 or a multithreaded PC the 6 timers represent timer 0 from each of the potential 6 threads.

**Stats Dumps and the Perf Stat Summary**

The bar graphs are useful to get a general runtime view of what is going on in the sim, but sometimes you may want to dig a bit deeper. One option is to bring up the Perf Stats Summary text window (the T icon in the tool bar). When you have the stats viewer enabled, it will show a navigable statistics tree, as produced by the hkBase lib (and as seen in the Havok demos). Give the window the keyboard focus by clicking on it, and then use the arrow keys to navigate the statistics (Right Arrow to open a node, Left to close etc).

To save all the current statistics for a frame or indeed a series of frames, right click on any of the Perf Stat Bars (graphs) and select Save. You can choose between three types of saves:

- TXT: A text dump, similar to the Perf Stat Summary, but can handle multiple frames and is a full dump.
- TGA: A targa image file dump. A bar graph of important section timings, commonly used with multithreaded apps to find stalls and load imbalance.
- HKS: A binary dump of all the current statistics source information. This file can be read by Havok Dev Rel and they can produce from it all the different stats views available.

When you send a movie (See Movie section below) to Havok Dev Rel, they can extract this timer information too, as long as you enabled the Statistics Viewer while you where recording of course.

**Advanced Profiling Analysis**

The Visual Debugger can also provide real time profiling on a per thread / processor basis. This allows you and us to easily identify multithreading performance bottlenecks. When you save a VDB movie all timer information is also saved. This means that detailed performance analysis can be done either online or offline.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

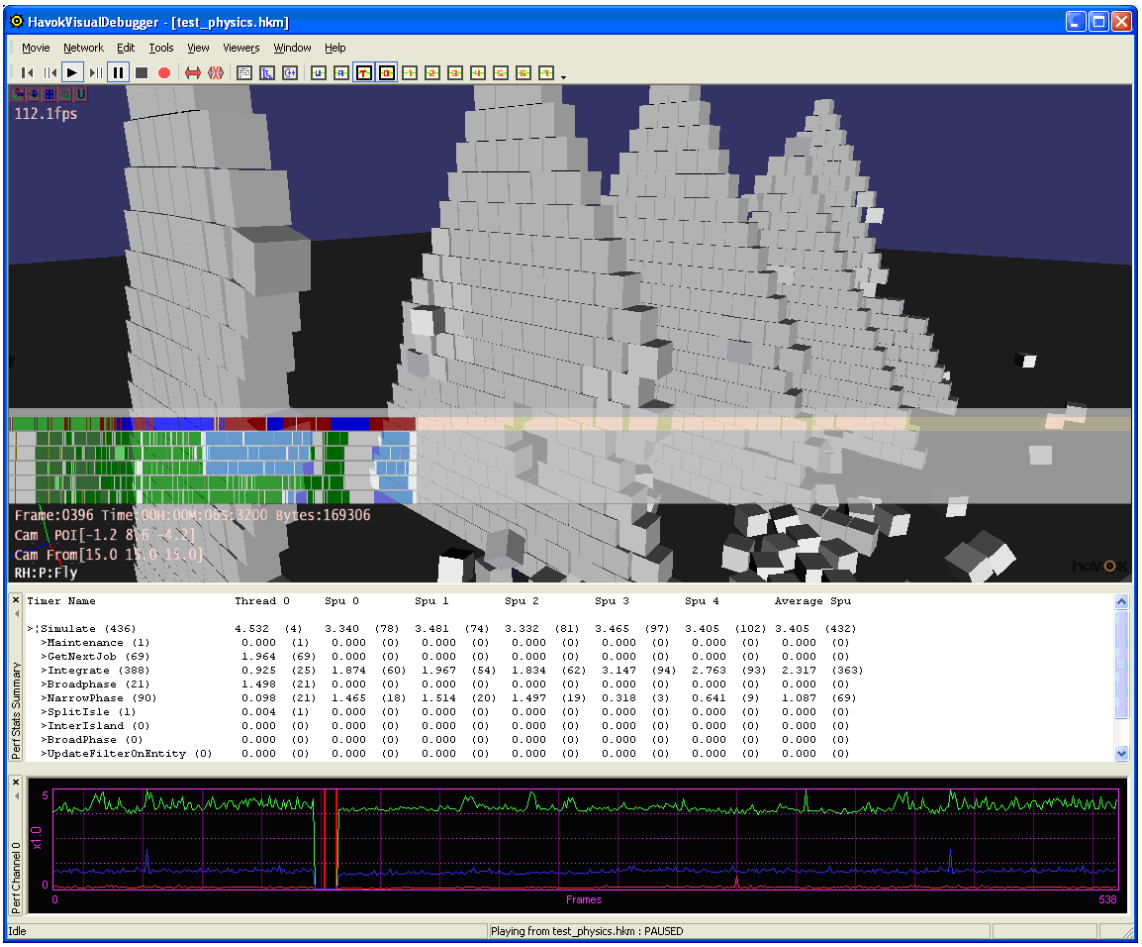



Figure 1.36: Advanced Profiling with the Visual Debugger

The diagram above shows both the per-thread statistics overlay and the detailed performance statistics summary.

### Per-Thread Statistics Overlay

The per-thread statistics overlay give a real-time view of the jobs running on Havok's Job queue. The jobs for each thread are shown horizontally in chronological order, with separate threads stacked on top of each other. To display this view choose View->Render State->Stats Graph Overlay from the menu.

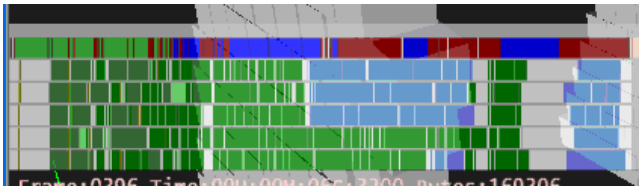


Figure 1.37: Detailed per thread stats overview

In the example above we see six running threads. Individual jobs are shown in different colours. By moving the mouse over any of the colored blocks it will indicate the job type and how long it took. Selecting a region with the left mouse button will allow you to zoom in on that specific region. Double-clicking allows you to zoom back out.

In the example above we can see the top thread (in this case the PPU thread running on Cell) is handling constraint setup jobs (light green) and broadphase jobs (dark blue). These jobs create work for the other threads to do (constraint solving in dark green and narrowphase collision detection in light blue) and then waits for this work to complete (shown in red).

**Detailed Performance Statistics Summary**

The detailed performance statistics summary view can be called up by clicking on the 'Toggle Perf Stats Text' Icon ( a Red T ) in the top toolbar. This launches the following panel:

Perf Stats Summary

| Timer Name | Thread 0 | | Spu 0 | | Spu 1 | | Spu 2 | | Spu 3 | | Spu 4 | | Average Spu | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| >¦Simulate (436) | 4.532 | (4) | 3.340 | (78) | 3.481 | (74) | 3.332 | (81) | 3.465 | (97) | 3.405 | (102) | 3.405 | (432) |
| >Maintenance (1) | 0.000 | (1) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) |
| >GetNextJob (69) | 1.964 | (69) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) |
| >Integrate (388) | 0.925 | (25) | 1.874 | (60) | 1.967 | (54) | 1.834 | (62) | 3.147 | (94) | 2.763 | (93) | 2.317 | (363) |
| >Broadphase (21) | 1.498 | (21) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) |
| >NarrowPhase (90) | 0.098 | (21) | 1.465 | (18) | 1.514 | (20) | 1.497 | (19) | 0.318 | (3) | 0.641 | (9) | 1.087 | (69) |
| >SplitIsle (1) | 0.004 | (1) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) |
| >InterIsland (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) |
| >BroadPhase (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) |
| >UpdateFilterOnEntity (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) | 0.000 | (0) |

Figure 1.38: Performance statistics summary

This is an interactive window showing total times for each thread. The timers are presented hierarchically. Any timer preceded with a '>' symbol can be expanded for further details. Use the up and down arrow keys (when that window has focus) to navigate through the list and use left and right arrow keys to expand and contract timers. For each timer, two values are shown: these are the total time in milliseconds and the number of calls made to the timer. The number of calls is enclosed in parenthesis.

**The Console View**

This displays local and remote textual information. To send text from your game use the HK_DISPLAY_TEXT macro described in the section entitled The Debug Display above. This text will appear in the Remote tab. The current limit on the console text is 1MB, and it will stop taking in text after that.

#### 1.7.3.5 Sending Geometry Data

In 2010.1, several features have been added to reduce the burden on the server when building and sending geometry data for the visual debugger. Both server and client must be from the 2010.1 SDK or later for these features to be used.

Viewers typically build the geometry data corresponding to a shape as one object and send it over the network in one message. For large objects, this creates two problems:

1. The geometry data must exist as a whole on the server. A sufficiently large geometry (e.g. corresponding to a `hkpSampledHeightFieldShape` covering a large area) will cause the simulation to run out of memory.
2. Sending the data over the network is slow and can create a frustrating stall in the simulation at the start of each connection.

To tackle the first problem, it is possible to build the data in parts. The size of the parts is determined by a limit on the number of "simple shapes" it contains. A simple shape typically correspond to a convex

shape, but user shapes and some deprecated shapes count as a single simple shape regardless of how they are constructed.

To tackle the second problem, the visual debugger can cache geometry data, and store it on disk between sessions. The data is identified by a 64-bit hash which is computed on the server (if you want a user shape to be cachable, you must register a callback function capable of creating a hash for the shape to the `hkpShapeHashUtil`. Alternatively, you can provide a hash for a body using the `HK_PROPERTY_OVERRIDE_SHAPE_HASH` property.). You can optionally have the geometry data sent to the visual debugger over several frames (disabling this feature during a connection will cause all outstanding data to be sent in the following frame).

Currently, these features are only supported by three viewers: the `hkpShapeDisplayViewer`, the `hkpConvexRadiusViewer` and the `hkpSweptTransformDisplayViewer`.

Various parameters affect how these features are used. See the section on the Edit Settings Dialog.

Notes for movies:

- The cache is not used when recording or playing movies.
- Movies recorded when shapes are sent in parts will not be compatible with earlier versions of the visual debugger.

#### 1.7.3.6 Visual Debugger Menu Options

This section describes the Visual Debugger menu options:

**Movies**

This menu includes all the Visual Debugger's movie options, along with the **Exit** command.

**Play...**

Plays a movie file. The default file extension for a Havok movie is .hkm. Havok movie files are created using **Movie - Record...** followed by **Movie - Stop** while you are connected to your game. It is also possible to record a movie by calling hkVisualDebugger::capture(const char* movieFilename) on the Visual Debugger game side. (See the hkDefaultDemo.cpp file or the Visual Debugger game side documentation for an example of how to do this.)

If the movie is currently Paused, Play will just unpause (same as hitting Pause again). If the movie is playing and you hit Play again it will assume you want to play a new movie.

**Single Step**

Puts movie playback into single-stepping mode. This option appears on the Standard toolbar by default and repeated clicking causes the movie to advance one frame at a time.

**Back Step**

Puts movie playback into single stepping mode. This option appears on the Standard toolbar by default and repeated clicking causes the movie to rewind one step at a time. This works by restarting playback

from the beginning and advancing until the previous frame - this implies that longer movies may show some delay during backstepping.

**Pause**

Pauses the current movie.

**Rewind**

Goes back to the beginning of the current movie.

**Stop**

Stops playback or recording of the current movie.

**Record...**

Records a new movie to a file. The default file extension is .hkm, and it is an uncompressed format.

**Real Time Playback**

When you step the VDB on the game (server) side, you can pass a time value. This is the 'time' displayed when you show the Frame Info in the viewports. If this time corresponds to the current time when the frame was captured (it may not), then you will probably want the movie to respect that time and playback at that speed. Respecting the frame time is the Real Time Playback option (on by default). If you find your movies stop for no reason at a certain frame, have a look at the Frame Time in the viewport. If it jumps a lot the VDB waits until that amount of time has passed, thus appearing to stop whereas in fact it thinks the movie is just very slow.

Thus make sure that the time you give to the VDB step is insulated from the effects of breakpoints and other stoppages that may occur in your game for lengthy periods of time. Either that or don't use the real time playback option.

**Exit**

Quits the application.

**Network**

This menu allows you to connect and disconnect from the network. The **Network - Connect...** option displays a dialog box in which you can specify the IP address of the game side - the machine the simulation is running on - along with the port number. Use the **Network - Disconnect** to terminate the network connection. The default port number is 25001. **Network - Auto Reconnect** causes the application to automatically reconnect to the game if the connection is closed by the game. Exponential back off is used to avoid redundant network activity.

Using the Visual Debugger it is possible to connect the the HCT Preview Window by connecting to port 25002. However, the HCT Preview Window is single-threaded and so is not representative of in-game performance.

If you shut down the application with Auto Reconnect still on, the VDB will start to Auto Reconnect as soon as it starts up again. This can help speed up your workflow during debugging.

### Edit

Here you can edit the User Keys and the speed of Movement. Both of the settings in these dialogs are stored in the registry across sessions.

### Settings

In the settings dialog, you can edit settings that affect how the client and server interact. Currently, the settings all concern Sending Geometry Data. The settings allow you to specify:

- whether the server creates geometry data in parts and the size of the parts in terms of simple shapes.
- whether the server caches data and the size of the cache.
- Whether the cache is written to a file and the location of that file.
- Whether geometry sending should be staggered over many frames and how many simple shapes should be sent per frame.

### Tools Menu

Tools->Customize allows the user to customize the key bindings and toolbars in the application. All settings are remembered, so please take the time to get all the keys and toolbars the way you like them.

### Viewers

This menu becomes active when there is a valid network connection to a simulation. It displays a list of all viewers available on the game side. Selecting a viewer from this menu toggles the state of the viewer on and off by causing them to be created and destroyed on the game side.

There are several viewers available. In general viewers enabled by default (game side via `hkpPhysicsContext::registerAllPhysicsProccess()`) do not modify the simulation. They are passive and only transmit simulation state information to the VDB. There is however one exception to this rule: the Forced Contact Point viewer **DOES** modify the underlying physics simulation when it is enabled. Internally this viewer modifies the `hkpEntity::m_contactPointCallbackDelay` member of all objects in the simulation. This can hinder performance if you have already specifically set this member to reduce the number of callbacks fired. For more information on the implications of setting `hkpEntity::m_contactPointCallbackDelay` to 0 see the physics documentation section on Callback Frequency.

### Window

Creates and destroys the various dockable windows available.

### Help

### About The Visual Debugger...

Displays a small dialog box outlining what the Visual Debugger does. It also contains protocol version information.

### Help...

Links to the Havok documentation. The link is a relative path so it won't work if you move the application from its original location. It just links to the Havok Physics 2012 chm in the doc directory.

#### 1.7.3.7 Preparing a Visual Debugger Movie for a Support Query

A Havok movie can be created by capturing the simulation directly to a file from your simulation or by using the **Movie - Record...** / **Movie - Stop** menu options in the application. To capture a file directly from a simulation see the section entitled The Visual Debugger Game Side and take a look in hkDefaultDemo.cpp for an example. Be sure that the appropriate listeners are created when capturing your movie. For example, if you want support to examine the performance statistics of your simulation the Statistics viewer must be instantiated/created when capturing your movie otherwise this information will not be streamed to the file. If you will be transmitting the file over a network it is worthwhile zipping it as a compression ratio of 80% is typically achieved.

#### 1.7.3.8 Platform-Specific Issues

**Application Caveats**

The Visual Debugger application requires DirectX 8.1 or later as it uses direct input.

**Server Caveats**

- Raycasts are currently not displayed automatically and must be displayed using the debug display. See the Debug Display section.
- You must call `hkpWorld->stepDeltaTime(...)` and `hkVisualDebugger->step(...)` for the Visual Debugger to work properly. The world step is required for physics viewers that rely on data coming only from physics callbacks. The main VDB step first checks for new clients connecting and then sends its current frame of information, waiting for an acknowledgment back to continue.

# Chapter 2

# Havok Physics 2012

## 2.1 Introduction

**Note:**
This manual describes how to use *Havok Physics 2012*, formerly known as Havok Physics. *Havok Physics* now refers to the new engine available from Havok version 2013.1 onwards, which has a dedicated user guide, separate to this one.

Welcome to the Havok Physics 2012 SDK, a physics middleware technology for fast, real-time rigid body simulation. Havok can be used in applications where objects need to interact realistically within a 3D space. It has been deployed in over 300 game titles across many platforms and across many game genres, including action adventure, role-playing, first and third person shooter, sports, and vehicle games. The full list of games can be found online at our website.

Figure 2.1: a small sample of game genres developed using Havok technology

This chapter begins with an introduction on physics engines, followed by an overview of the Havok Physics 2012 SDK and how to use it in your application. You will find more detailed information about the concepts and components introduced here in the rest of this manual.

### 2.1.1 What is a Physics Engine?

#### 2.1.1.1 Introduction

This section aims to provide a general understanding of the terminology, methodology and behavior associated with a physics engine. With an understanding of how a physics simulation should behave, getting Havok to work within your game will be much easier.

#### 2.1.1.2 Physical Simulation

Physical simulation is not a new concept. Ever since computers started dropping off the assembly line, scientists and programmers have used them to simulate complex situations like rocket trajectories (ballistic motion), liquid flows (fluid dynamics) and other complicated projects. Many of these simulations are very expensive in terms of the CPU resources required. They are concerned with high levels of accuracy and as a result, calculations need to be performed with a very high level of detail. A game physics engine has a different goal: fast, believable simulation executed in a very limited number of CPU cycles.

Rigid body dynamics is the most common form of physical simulation used within video games. For performance reasons, a number of assumptions are usually made about the environment.

- The shape of each object is fixed. Each object is infinitely hard/rigid.

  There are many advantages if the shape of objects can be assumed not to change, irrespective of how large the forces applied to them are. Large speedups are achieved as lots of information about how objects will move can be pre-calculated or simplified.

  In real life, a car panel colliding with a wall will bend (and possibly be torn off) depending on the strength of the impact. In a rigid body simulation, the car panel will remain completely intact.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved.

Figure 2.2: Rigid Objects

- The landscape does not change dynamically.

  Simulating a completely dynamic environment (e.g. blowing holes in the ground with a grenade) can be very computationally expensive. In most games the base level of environment remains fixed and objects within the landscape (such as traffic cones, pedestrians, bins etc.) can be modeled as separate physical objects rather than as part of the landscape.

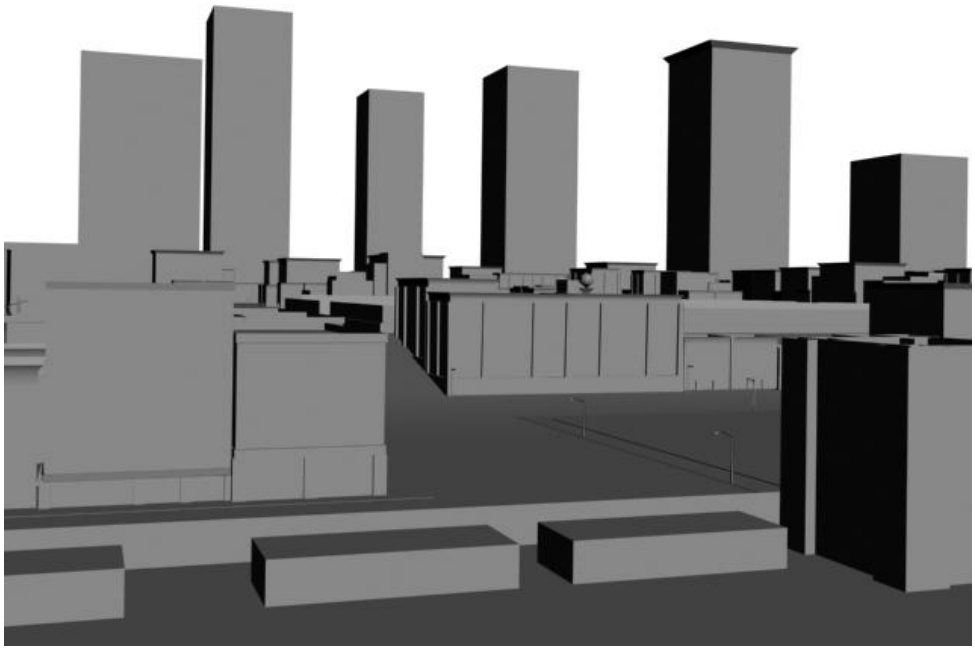


Figure 2.3: Base level Landscapes are generally solid

Sections of the game environment that are required to be dynamic (such as water, collapsible walls, destructible land etc.) can be layered on top of the base static environment.

- The shape of objects can be simplified for physical simulation.

  It is rare for an object to require as much physical detail as its graphical representation. A simplified representation shape is generally used as replacement for a highly detailed in-game polygon mesh without any adverse affect.

Figure 2.4: What the player sees is not always what the physics engine simulates

- Objects in a scene are not all active at the same time.

  Simulating objects at rest is very efficient (zero CPU time). Large worlds with many objects may be created so long as only a reasonable number of them move at any given time. Within a typical game scene, there may be many potentially active environmental objects such as chairs, crates, barrels etc. Most of these remain at rest for most of the time. A physics engine will optimize to only simulate the objects currently necessary.

#### 2.1.1.3 What Does a Physics Engine do?

A physics engine knows and cares little about how the objects it simulates are displayed graphically in a game. It simulates the motion and interaction of objects based on a physical (not graphical) description of the objects, but this information may be used to generate a display that "tracks" the simulation.

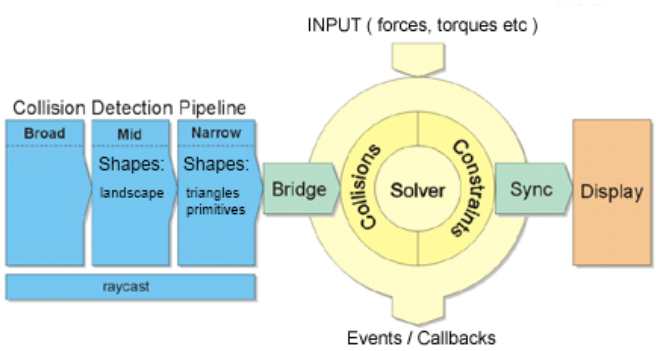



Figure 2.5: The structure of a physics simulation system

A physics engine has three basic tasks to perform in its main simulation loop (once it has setup the initial conditions for a given scene):

1. Determine which objects are overlapping or about to overlap (e.g. a spaceship has collided with a hangar wall, or with another spaceship). This stage is usually divided into three sections: broadphase, midphase and narrowphase.
2. Solve all forces acting on the objects. Some forces occur as a result of collision detection (e.g. two spaceships colliding together), others due to input from outside the simulation (e.g. the gravitational pull of the environment or the driving force of a ship's engines).
3. Having gathered up all the forces, the simulation is advanced by the simulation time step and the new state of the objects (position, orientation, velocity, acceleration etc.) is calculated. This information is then used to update the corresponding display representations of the objects

   Do all other game logic and repeat.

#### 2.1.1.4 Simulating a Cannon Ball

Ignoring collisions for the moment, let's consider the simulation of a cannon ball immediately after it has been fired from the cannon. We know the ball's current position, speed and acceleration. We know its weight and the state of the environment (i.e. air resistance, wind force, gravity). Armed with this knowledge a physics engine can simulate its flight path.

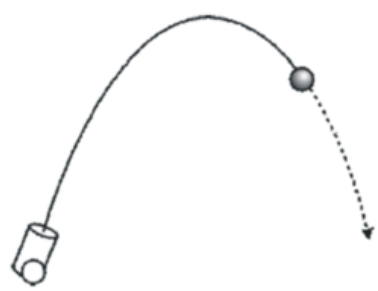


Figure 2.6: The expected ballistic motion of a cannon ball is an arc

This image illustrates what we would like to achieve. Over a period of time the cannon ball's rate of ascent will slow due to gravity, and it will eventually fall to the ground having taken a classic parabolic arc path (assuming no air resistance).

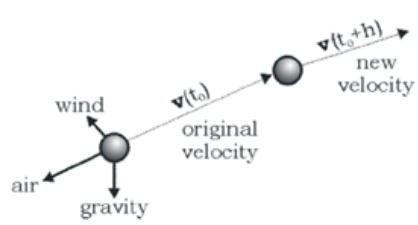



Figure 2.7: Given an initial state of an object, and knowing the forces acting on it, we can estimate the new state of a body in motion over time

At a given point in time we examine the state of the ball (its speed $v$ and acceleration $a$). Knowing the external forces acting on it we make a prediction as to its change in position after a period of time has

elapsed (call this period $h$ seconds), as shown in the figure above. This prediction is based on several assumptions:

1. Newton's laws of motion govern the motion of the ball.
2. During the time period $h$, all external forces acting on the ball are constant (i.e. air resistance and wind and gravity do not change during this time).
3. The math used to calculate the new position is accurate (see below).

In general 1) is usually a good assumption but 2) and 3) cause problems and are closely linked to the time period $h$ over which the calculations are performed.

Imagine planning how to get from your home to your office. The factors affecting your trip are never quite the same and there is no simple equation that can guide you through traffic. You can get more accurate in your plan if you break the trip down into smaller segments and replan after each step.

This is also true of simulation math. As the simulation becomes more complex the math required to calculate the new positions and velocities of objects in a simulation also becomes more complex. As a result, the guesses produced by the math produce less accurate results as time step increases.

The smaller the time step taken, the more accurate the result at the end of the time step. To step forward in time by a large time step $h$ it is better to split this into $n$ consecutive steps of a smaller time interval $h/n$.

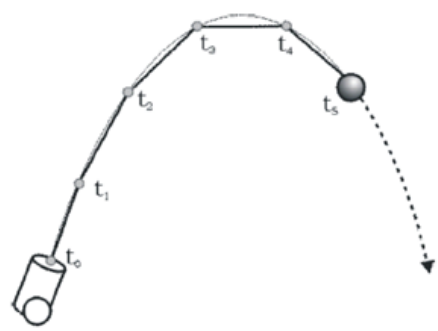



Figure 2.8: Evolution of a simulated cannonball captured in a series of "snap shots"

**Iterations and Accuracy**

With a physical simulation each result depends completely on the previous simulation step. If, in one step, a very inaccurate result is produced, then the next step is likely to be even more inaccurate, resulting in a spiral of decreasing accuracy until eventually the results are garbage (the simulation "becomes unstable" or simply "explodes").

Assume we need to update the geometry in the game so it can be displayed once every $1/60^{th}$ of a second. For simulation accuracy and stability, we might still need to perform the physics simulation step at a higher frequency. We call these sub-time-steps "iterations".

An iteration is an internal physics simulation step that does not interact with the display. The iteration parameter specifies the number of steps the physics engine takes *before updating the display*. This gives control over the granularity of the physics simulation independent of the display update frequency.

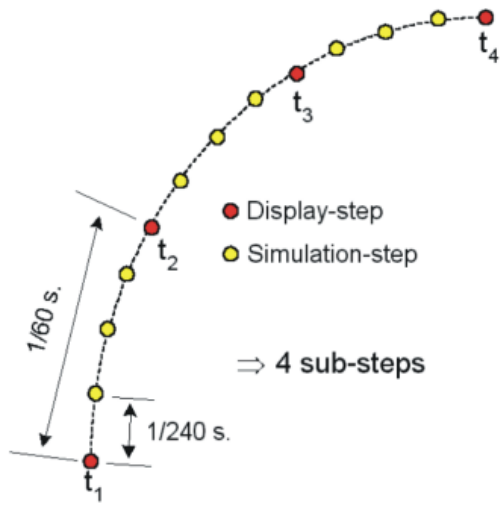



Figure 2.9: Iterations allow simulation frequency to be decoupled from display frequency

In the diagram above, the display is updated at $1/60^{th}$ of a second, but the physics simulation will be stepped at intervals of $1/240^{th}$ of a second. For every 4 iterations the display is updated only once. By setting the number of iterations the accuracy of the physical simulation is controlled independently of the display.

#### 2.1.1.5 Linear Physical Units and Values

When working with a physics engine, it is important to maintain a consistent unit system. Mixing different units indiscriminately will lead to inaccurate, unrealistic results and inconsistent behavior. Working with a consistent set of units is of great value.

There are a large number of different measures in use at any given time. Despite the apparent complexity of dealing with all these units, in most cases you do not need to worry about them. However, it can be useful to know that you can analyze a scene and the units used to determine the approximate values required to achieve a desired goal.

In general, the metric system is used in the following discussions. The metric or SI units are well behaved for our purposes:

- m = Meters.
- kg = Kilograms.
- s = Seconds.
- N = Newtons (measurement of force: 1 N = force required to change the speed of a 1kg object by 1 m/s in 1 s. This can be difficult to grasp but think of it using an example of a 1 kg weight sliding on 'perfect' ice. Imagine this weight slides at a speed of 1m/s (3.6 kph). It would take 1 N of force applied for 1 s in the opposite direction of the movement to bring the weight to a perfect stop).
- rad = Radians. 1 rad = 57.2958 degrees = (180 / PI).

| Name | Units, Metric | Unit, Imperial | Description |
|---|---|---|---|
| Position | m | ft | A better description of this term is displacement or distance. |
| Velocity | m/s | ft/s | Speed–how fast an object is traveling relative to some frame of reference. |
| Mass | kg | Slug | A measure of the amount of matter in a body. How difficult it is to move a body. |
| Momentum | kg m/s | lb/ft$^2$ | Velocity multiplied by mass–this property determines the amount of force required to change the velocity of an object. For example a truck is harder to stop than a scooter even if both are traveling at the same speed, because the truck has greater mass. |
| Acceleration | m/s$^2$ | ft/s$^2$ | The rate of change of velocity over time (i.e. whether an object is speeding up or slowing down) |
| Impulse | kg m/s (or N s) | lb s | A measure of the change in momentum (usually measured in Newton seconds). To instantaneously change the velocity of an object, apply an impulse to it. |
| Force | kg m/s$^2$ (or Newton N) | lb | The basic unit of a physics engine. This quantity measures the effort required to change the velocity of an object (i.e. to give an object an acceleration, either positive or negative). |

Table 2.1: Units and terms

- x/y is read as "x per y" [for example velocity is measured in m/s or meters per second. Note that x/y is sometimes written as x y$^{-1}$].
- x/y$^2$ is read as "x per y squared" or more usually "x per y per y" [for example acceleration is measured in m/s$^2$ i.e. meters per second squared or meters per second per second]

Note: Sometimes, to distinguish from their angular counterparts (discussed later), velocity, momentum and acceleration are defined as *linear velocity*, *linear momentum* and *linear acceleration*.

#### 2.1.1.6 Angular Physical Units and Values

Orientations are often one of the most difficult quantities to become familiar with. Most people (anyone who doesn't spend their entire day working on the space program or writing games!) find it difficult to get a good intuitive grasp of 3D orientation. Please refer to the article on orientation, rotations and handedness for a thorough explanation.

It is important to remember that orientations require a reference frame. Rotations are 'from' somewhere 'to' somewhere else. You always need a fixed reference frame to calculate where you are rotating 'from'.

**Coordinate Systems & Reference Frames**

The usual reference frame for an object is the object's *center of mass* (COM). The COM of an object is not difficult to visualize. It is the point that needs to be supported to keep an object balanced. For example, in a movie where a car teeters over a cliff, the cliff edge is under the COM. Forces applied to an object produce different results depending on the position of the COM.

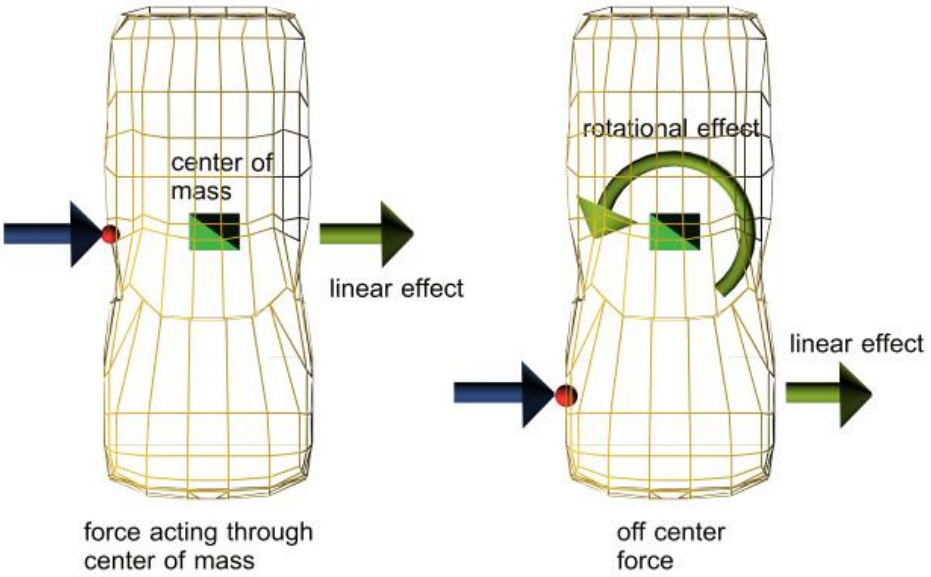



Figure 2.10: Depending on where forces are applied rotation may or may not happen

If a force is applied at or through an object's COM, the object's linear acceleration changes but no rotational acceleration is introduced. In the left car in the diagram above the force in blue is applied at the point of the door panel. The line of force goes through the COM (the green box) and no rotation is created.

On the other hand, when a force is applied so that it does not go through the COM a torque (or twist) is introduced (right car example). This torque alters both the linear and the angular acceleration of the object. In our example, this causes both movement left to right and an anti-clockwise twist. The further the force (or more accurately the line of force) is from the COM the bigger the twist that is introduced. The same applies to impulses.

When applying impulses and forces it is important to specify the point to apply the impulse or force with respect to the COM of the object.

| Name | Units | Description |
|---|---|---|
| Orientation | Axis + Angle (radians) | An object's orientation with respect to the world/game coordinate system. |
| Angular Velocity | Axis + radians/s | The speed at which the object is rotating: i.e. the number of radians per second the object rotates, usually specified with the axis around which it is rotating. |
| Moment of Inertia | 3 x 3 matrix, kg-$m^2$ | Angular equivalent of inertial mass. Represents an object's resistance to a change in its angular velocity about any axis. |
| Angular Momentum | kg-$m^2$/s | Angular equivalent of momentum. This quantifies how much torque is required to halt the object's rotation in a given amount of time. |
| Angular Acceleration | rad/$s^2$ | Angular equivalent of acceleration. This is the rate of change of the angular velocity. |
| Angular Impulse | kg-$m^2$/s or N-m-s | Angular equivalent of impulse. This is a measure of change in angular momentum delivered by a torque over time. Represents the idealization of an instantaneous change to an object's angular velocity. |
| Torque | kg $m^2/s^2$ or N-m | This is the angular equivalent of force. Units are Newton meters. |

Table 2.2: Physical terms and descriptions

#### 2.1.1.7 Physical properties

These properties depend on the object's material(s) and affect its behavior in a physical simulation.

| Name | Units | Description |
|---|---|---|
| Friction | Dimensionless | The friction coefficient (usually a value from 0 to 1) which specifies how "sticky" or "rough" an object is. See below for a more detailed explanation. |
| Restitution | Dimensionless | The restitution coefficient (usually a value from 0 to 1) specifies how much kinetic energy is lost during a collision between two objects e.g. with a value of 0, all energy is lost and objects appear to come to a complete halt when they collide. With a value of 1, no energy is lost and objects bounce off each other with an equal but opposite velocity. For values between 0 and 1, the objects lose some but not all energy with each collision. This is a per object or per object/material property. |
| Mass | kg | A measure of an object's resistance to change in motion, or the amount of matter in an object. Not to be confused with weight, which is the attraction of the earths gravitational pull on a certain mass. |

Table 2.3: Object properties

**Dynamic and Static Friction**

A friction force is one that resists the tendency of two bodies in contact from sliding off each other. Friction is the key factor in allowing stable stacking (i.e. stacks or piles of objects that come to rest, held in place by the friction at the points of contact). During all collisions a certain amount of energy is lost due to friction (and mostly converted to heat in the real world).

Friction manifests itself in two forms, static and dynamic. Static friction operates when objects are at rest; it is the force that prevents objects from moving or sliding. If a force is applied to an object large enough to overcome static friction, the object begins to move/slide. At this point dynamic friction kicks in and as long as the object is in contact with another object or surface, dynamic friction is a force that slows the object down.

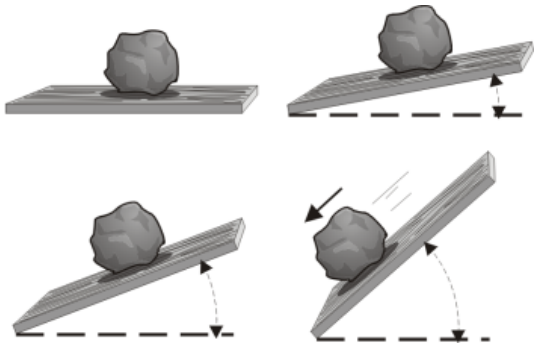


Figure 2.11: Static and dynamic friction in action

The figure above shows that the boulder is held in place by static friction until enough force is applied to break the contact (i.e. the plank has been raised to a sufficient height and gravity forces it to move). When the boulder begins to slide, dynamic friction kicks in which acts against the sliding action, generating heat.

#### 2.1.1.8 Collision Detection

Collision Detection is one of the cores of any physics engine. If naively implemented, it can account for a large portion of the CPU time required for game simulation. We want to know when a player collides with the level, when bullets collide with the player, when bullets collide with the level etc.

A physics engine attempts to minimize the number of possible collisions that must be tested each simulation step. It would be completely inefficient to test every single object against every other object in the simulation frame. Physics engines use locality and frame coherency to reduce the number of tests needed.

There are a number of ways to speed up the collision detection process:

1. Reduce the number of collisions that require detailed collision results to be generated (e.g. use a simpler collision test first)
2. Reduce the complexity of objects tested for collisions.
3. Reduce the number of simultaneously moving objects.

The number of objects active in a given scene (point 3) is the primary source of CPU load. If some objects can be ignored this speeds up the simulation. In the case of point 2, the difficulty in providing detailed collision results is directly associated with the complexity of the objects, and using simplified representations for the physical shape is always encouraged. First, we deal with how the Havok system attempts to achieve 1), a reduction in the number of complex collisions.

**Multiphase Collision Detection**

Physics engines employ a series of collision detection layers, each progressively more complex and returning more detailed results. Each stage filters as many objects as possible from the test process, before moving to the next stage with the resulting data.

**Broadphase Collision Detection**

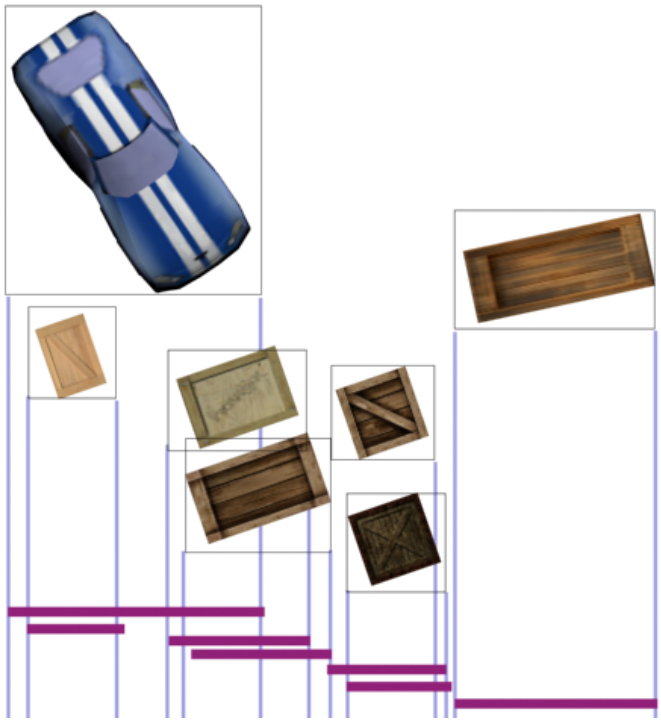


Figure 2.12: First pass in a collision detection system - attempting to eliminate as many collision pairs as early as possible

Above is an example of the first pass of a collision system. The car is driving toward a pile of crates. Each object is approximated by an *axis-aligned bounding box* (AABB) which fully encloses each object.

The purple lines at the bottom show the overlap between the different bounding boxes along a single axis. Only pairs of objects whose projections overlap along all three axes are passed on to the next stage of collision detection. This process is called a 3-axis sweep and prune, and returns all pairs of objects whose AABBs overlap.

Detecting overlaps between AABBs is an extremely fast operation and allows us to filter out a huge number of potentially colliding pairs.

When we consider a single axis then of the potential list of 21 collision pairs in our example ((7x6)/2) we find that only 5 pairs have overlapping boxes (the car and the 3 left most boxes form 3 pairs, boxes 2 and 3 form another pair, and boxes 4 and 5 form the final pair). We can forget about all the other pairs in this case. This is sometimes called *trivial rejection*.

Even though possible colliding pairs of objects have been identified, it does not mean that they are actually colliding, only that they might be colliding. They need to be examined further by a more accurate collision detection layer.

**Midphase Collision Detection**

Now that the set of potentially colliding objects is reduced, more complex tests can be carried out to determine if two objects are potentially colliding before finally passing them to the narrowphase stage. The midphase collision detection is used to reduce the number of colliding pairs between *complex* objects (such as a landscape).

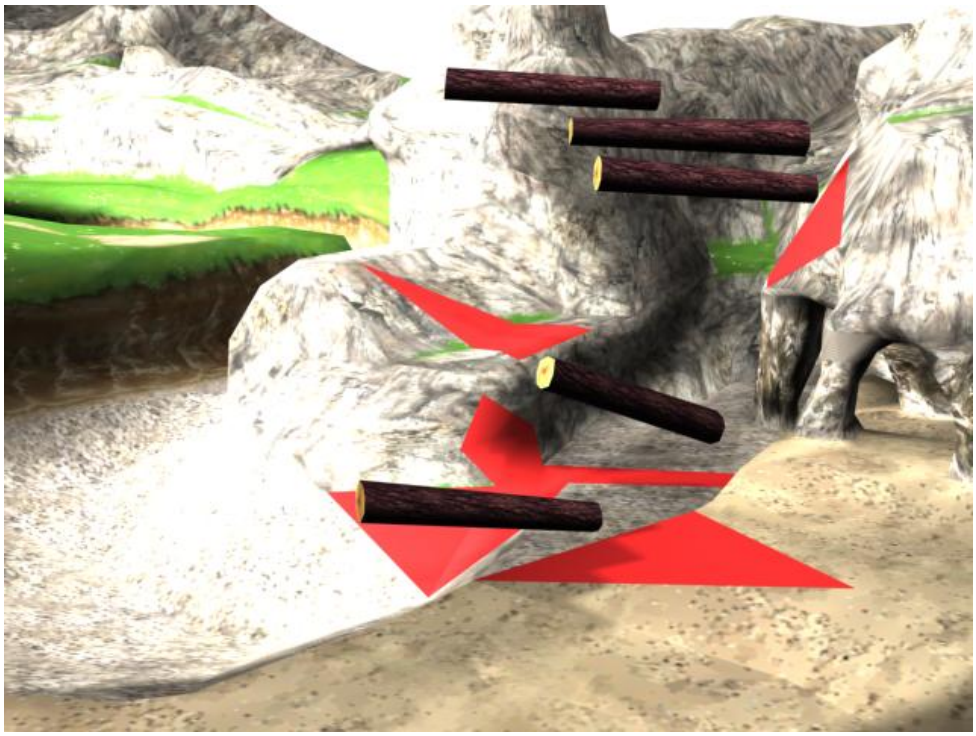


Figure 2.13: Midphase MOPP

In the figure above we see logs about to fall onto a static landscape (the landscape contains 1000's of triangles). The broadphase will have detected that all of the logs are overlapping the AABB of the landscape and all are potentially colliding. The midphase can reduce the problem by identifying the triangles in the landscape which are likely to be colliding with the logs. This much reduced set of possible collisions is now suitable for passing to the final narrowphase.

**Narrowphase Collision Detection**

This final phase performs accurate collisions test to determine if objects actually collide. If they do, relevant collision information is calculated (such as the point of collision, the normal to the objects at the point of collision, etc). These calculations can be quite CPU intensive, which is why the previous two stages are used to prune the set of objects that are processed at this stage.

As all three of these steps are automated by the physics engine, the user is left only with control over the complexity of the collision representation of the shapes. The next section deals with the various types of object shapes that can be used and their complexity for collision detection.

**Rigid Bodies and Collision Geometries**

The shape of objects tested for collisions has a major impact on the speed of collision testing. If assumptions can be made about object shape or the geometry simplified for collision testing, computation costs can be reduced.

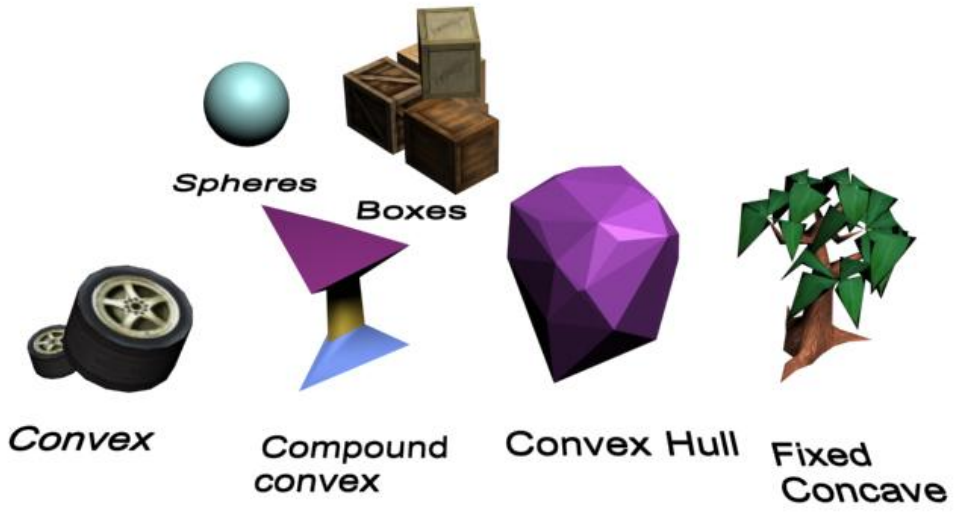



Figure 2.14: Different geometry formats

In a physics engine, objects are classified according to their shapes, and particular shapes have particular properties that make them easier to deal with during collision detection. The following list classifies object shapes in order of increasing complexity:

- spheres
- planes
- boxes
- polygons

For the simple shapes, an implicit mathematical representation of the object can be used for collision test on this. For polygonal representations, the complex representation must be used. In this case objects are classified, in order of increasing complexity, as:

1. Convex: imagine wrapping an object in cellophane. If the cellophane touches every part of the surface (i.e. there are no hollows in the surface that the cellophane does not reach) then the object is convex e.g. a beach ball is convex, a starfish is not. Treating an object as convex implies that it has an *inside* and an *outside*. Many objects can also be approximated with a convex hull (which speeds up collision testing enormously).

   The amount of approximation will represent a trade off between speed of collision and accuracy required. This is typically very context specific; e.g. for the car chassis below we could get by with a simple box representation. If we want to see our cars spinning and wobbling on their roofs then the convex hull gives a pretty good approximation of the graphics mesh.

Figure 2.15: Concave mesh and convex hull representations of a car chassis

2. Polygon Soup: a collection of polygons, not necessarily connected, all grouped and classified as a single object. This is the most expensive format to detect collisions with, but is also the most general. The majority of fixed level data is stored using the polygon soup format.

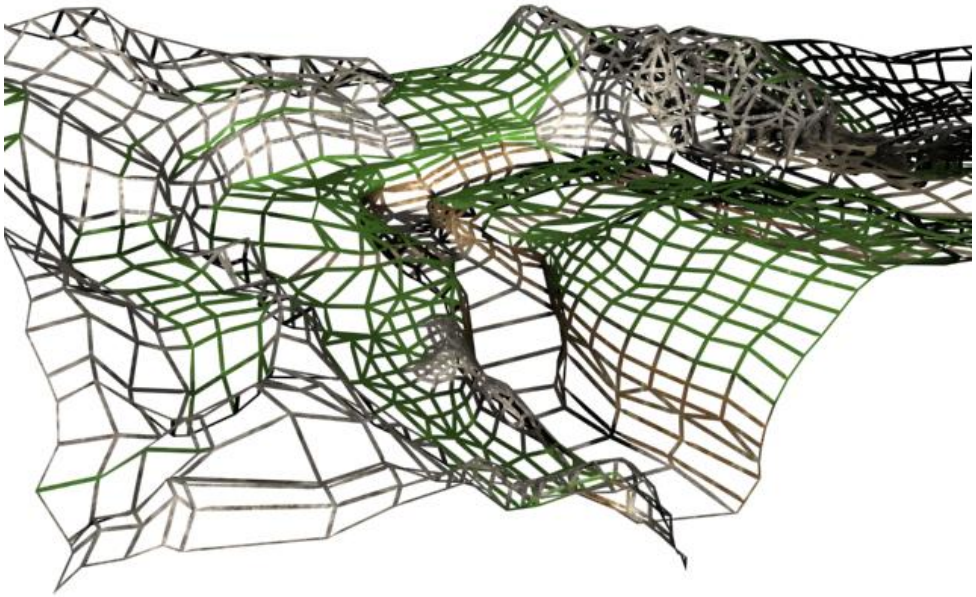


Figure 2.16: Fixed Landscape represented by arbitrary polygons

### Inside and Outside

One major consequence of the assumption of solidity and the problems of interpenetration is that you cannot consider objects to be hollow. A common mistake is to create a box for a room and begin to place objects within the box, thinking these are being placed inside the room. Any objects placed inside are tagged as interpenetrating and will simply be pushed out of the box (possibly at speed!) when simulation starts.

One way to avoid this is to represent the room/box using a polygon soup geometry (unconnected triangles), or a set of boxes. This will have the desired effect of a hollow shell for the room.

### Setting Position and Orientation Directly

Physics engines carefully store the state of all objects in a scene and update this state from one time step to the next. If an external system makes changes to the scene, the physics engine needs to update its state accordingly. This is OK *except* when changes made to the scene violate the simulation stability or consistency that the physics engine attempts to maintain.

It is easy to see how arbitrarily setting the position of an object in the physics scene may cause problems:

- If an object is warped to a position that causes interpenetration, without additional information the physics engine must simply guess how to resolve the inconsistency. A physics engine is able to deal with interpenetrations better when it can plan for it.
- If other objects are resting or stacked above an object that is warped away while the stack is deactivated, the remainder of the stack will remain floating above the space vacated by the warped object.
- If other objects are attached to the moved object by *springs* or *constraints*, it can lead to huge instability problems. By moving the object abruptly, the connections to the other objects are stretched instantaneously and can cause the system to destabilize and the constraints to oscillate wildly.

Rather than moving objects directly it is always better to apply *forces* and *impulses*, either *linear* or *angular*, to the objects in order to "push" them towards the desired goal.

### Setting the Scene

The first stage in any physics simulation is to create the scene. This is slightly different to constructing a 3D scene without physics and you should remember requirements like avoiding interpenetration. If you construct a scene in a purely geometric way you could experience difficulties placing objects so that they rest stably on other objects while avoiding collision problems.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

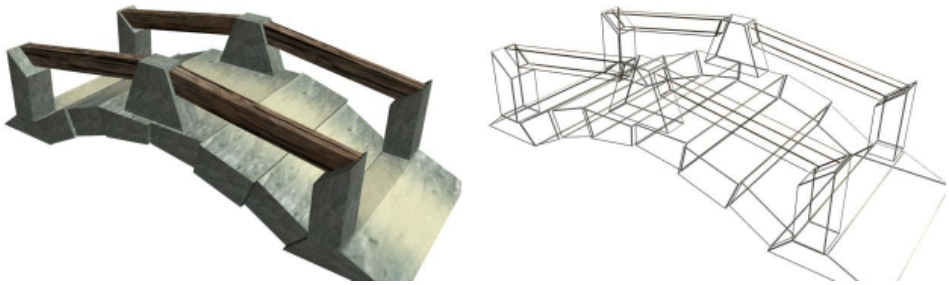


Figure 2.17: Creating a stable stack of objects

As an example take the case of creating a free standing bridge. If stones interpenetrate the stones will all be active and immediately moved away from each other by the engine. Even if everything stabilizes you still get a big performance spike during the first simulation steps.

The best solution is to create the stones initially separated by a small distance. Then simply simulate the scene for a period of time. The stones will fall the small distance and will settle into a stable stack. Eventually they will deactivate as the energy management system kicks in. Now you have a stable stack that is deactivated and ready to be saved. Store the positions of the bridge pieces. Now you can use these positions to create your bridge ready for streaming into your game.

#### 2.1.1.9 Constrained Dynamics

In many cases we want to construct physics systems that have joints or attachments between sub parts of a larger assembly. We need to be able to make machines, robots, traps and puzzles. Physics engines will provide any number of the following (and Havok provides all of these).

- Springs: Forces are applied to objects connected by springs to attempt to keep the objects within some fixed distance of each other (the rest length).
- Dashpots (a more mechanically based spring; in real life dash pots contain oil to smooth out vibrations)
- Mechanical joints: The common joints in the world around us; prismatic joints (e.g. a piston in a machine, sliding door) hinge joints (e.g. a door, the sign swinging above the spooky saloon), axles (e.g. wheels, cogs), human joints (everything from the hinge-like elbow to the more complex body parts like shoulders and hips) and all kinds of vehicle components.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

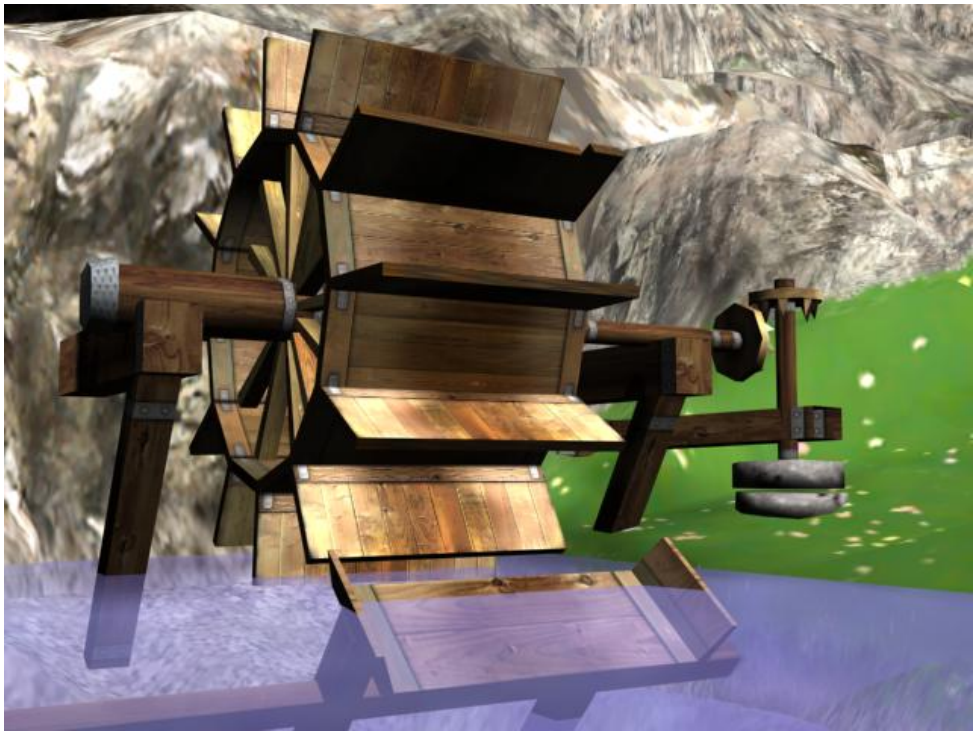


Figure 2.18: The water wheel is connected to the mill stones via an axle and some cogs

Constraints are a requirement for creating interconnected systems. Forces applied to one object are instantaneously applied to all other connected objects. In the example above, when the water turns the paddle, the axle turns, the horizontal cog turns the vertical cog which turns the vertical axle and the mill stones begin to grind.

### 2.1.2 What is Havok Physics 2012?

#### 2.1.2.1 Overview

Now that we have a base understanding of a physics engine, let's look at what Havok Physics 2012 can do. Havok Physics 2012 has been designed from the ground up to be a powerful yet flexible physics simulation system.

The major strengths of Havok Physics 2012 are:

- **Multithreading** : By default, the engine is fully multithreaded across all platforms that support it. This includes collision detection, constraint solving and asynchronous queries.
- **Determinism** : Havok Physics 2012 is fully deterministic in its base use. The documentation will highlight any situations where determinism can be affected.
- **Cross-platform** : Havok Physics 2012 fully supports all leading consoles, old and new.
- **Optimized** : Havok Physics 2012 is optimized across all its supported platforms. Hand-coded assembly, optimised math routines and full multithreaded support help Havok to use each architecture in the most efficient way.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

- **Extendable and Flexible** : Havok Physics 2012 is designed for any degree of customization, allowing it to be used for a wide array of game genres and use cases.

Havok Physics 2012 also provides you with higher level solutions for game genre-specific problems like vehicle simulation (for driving games), human ragdolls (for dynamic simulation of killed enemy characters), physical interaction of keyframed characters within a game environment (to simulate enemies getting hit in different parts of their bodies), character control (for first person or third person games).

In addition, it has a rich product set surrounding the core SDK, including tools for content creation, geometry processing, profiling and debugging (e.g. the Visual Debugger), integration with third-party renderers like 3D Studio Max and Maya (including full source for scripted utilities for easy inclusion in your game's existing art export pipeline), and tweaking and tuning.

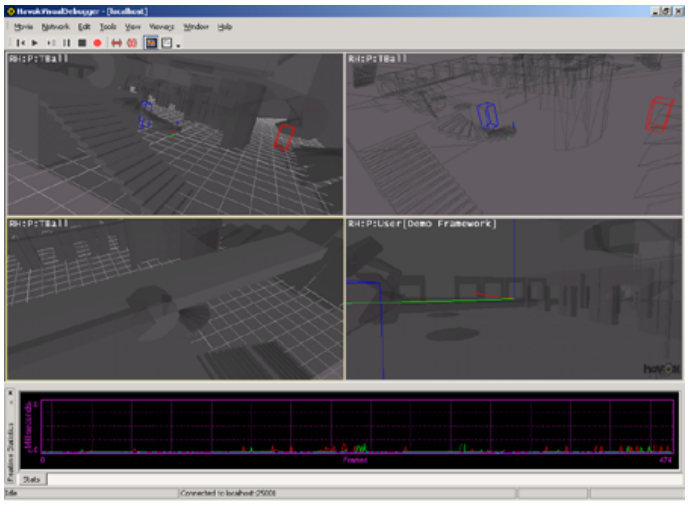


Figure 2.19: The Visual Debugger in action, which is a very useful tool for (remotely) viewing just the physical simulation and studying various in-game performance statistics

Havok Physics 2012 is an open, object-oriented framework. The SDK has been designed to be very modular and allows you to extend or re-implement many of the core components. You can also customize almost any aspect of the physical behavior during the simulation (e.g. change gravity, apply forces and torques to objects, add and remove objects dynamically).

However, Havok Physics 2012 is *not...*

- A packaged Game Construction Kit. The SDK is just that, an SDK. It provides building blocks that can be used and glued together in many different ways. Like any middleware technology, it is a generic technology that can be applied in a variety of game-specific contexts.
- A simple technology. It requires an investment in time to become familiar with its features, its documentation and its interfaces. To use it effectively, it demands a high level of programming expertise and a familiarity with the basic concepts of dynamics.
- A black box. Havok Physics 2012 provides low-level access to core functionality so that you can construct complex physical behaviors that are specific to your game and don't come as standard with the SDK.
- A commercial renderer. Although we provide a simple cross-platform display library, it is used merely as a testbed for rapid prototyping and for deploying Havok demos.

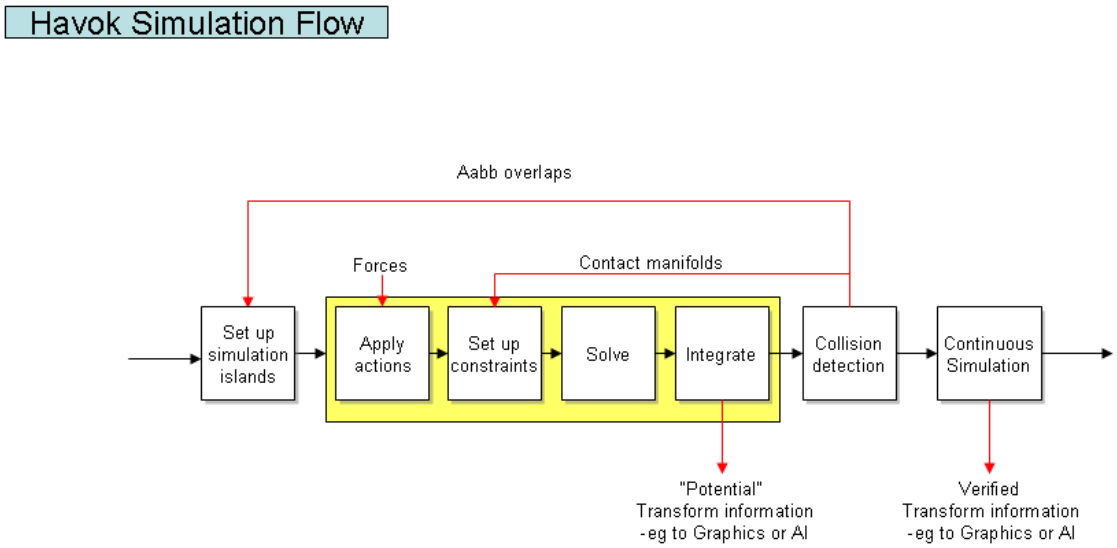
#### 2.1.2.2 Key Concepts

Havok Physics 2012 is built around a few key concepts that are used throughout the documentation and source code:

- Shapes: The basic representation of a physical object.
- Rigid Bodies: A simulated physical object formed by one or more shapes.
- Constraints: These limit the freedom of movement between a pair of rigid bodies in the simulation world.
- Simulation World: The representative world where all rigid bodies and constraints are simulated.
- Callbacks, Listeners and Events: These allow the user to plug in to the simulation and be notified of any number of key events, both high-level (object removal, object deactivation, etc.) and low-level (contact point creation, collision callbacks, and so on). Havok provides a wide array of callbacks and listeners to cater to every need.

#### 2.1.2.3 Simulation Overview

The main Havok Physics 2012 simulation loop performs a specific set of steps each simulation frame. The following diagram illustrates the steps taken to move the simulation forward (and their interdependencies):




**Setting Up Simulation Islands.**

During the physics simulation separate groups of objects are partitioned into **simulation islands** . Simulation islands allow groups of objects to be simulated independently for performance reasons (CPU and memory consumption, and multithreading).

Objects share an island if there are certain physical effects between them. Simulation islands are set up automatically by the system for these groups of objects. **This process is fully automatic and handled internally by Havok.**

In most cases, you will never have to deal with simulation islands, but they can occasionally have an impact on how certain features can be used. This will be explicitly highlighted in the documentation whenever the situation occurs.

Simulation islands play a role in the deactivation of objects, a topic which is covered in detail in a later section.

### Applying Actions

An **action** is an external tool enabling you to control the state of the bodies in the simulation. At each simulation step, every action that has been added to the world will be applied. The Havok Physics 2012 SDK provides a number of commonly used actions (such as the Vehicle Action). Custom actions can also easily be implemented. More information about actions is available in the Actions chapter.

### Setup Constraints

At this stage, the constraints present in the scene are processed and prepared for the solving stage. This includes contact constraints (generated by the system for colliding objects to prevent interpenetration) and constraints explicitly created between objects (such as hinges or ball-and-socket joints).

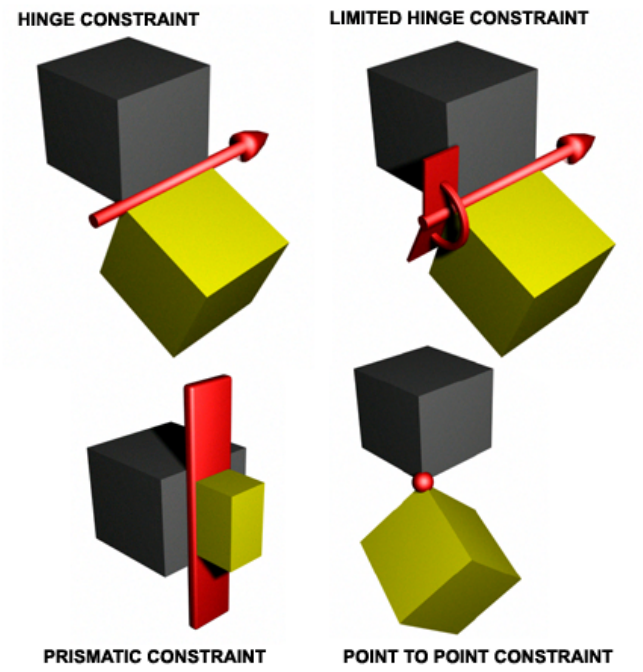



Figure 2.20: A sample of the some of the built-in Havok constraints

### Solve

All the constraints in the simulation world are then processed by the **solver** . The solver is responsible for calculating the necessary changes to reduce error for all constraints in the simulation. Every constraint defines a set of conditions that must be maintained. For each constraint, the "distance" between the current situation and the correct solution defines this error.

In a contact constraint, the error is defined by penetration depth (how much one rigid body is embedded in the other). In a ball-and-socket constraint, this error is the amount the objects are separated.

The solver iteratively processes all constraints to reduce the size of this error. This in turn adjusts the position of the objects involved in the constraint. The results of the solver are passed on to the integrator.

**Integration**

The **integrator** calculates the new motion state (initial and final position and orientation, velocity, acceleration, and so on) for each object in the simulation, taking into account the information provided by the solver. These new positions and rotations can then be used externally (for example, by a renderer to update the corresponding in-game objects).

The motion state of an object stores two transforms relating to its position and orientation at the beginning and the end of the simulation step. It is then easy to calculate the position of the object in between two simulation frames. This is advantageous when running a slow-motion visualization and using one simulation step for a number of visualization frames.

Note that the set of operations [Apply actions, Setup Constraints, Solve, and Integration] are sometimes refered to simply as "integrate". For instance, the method `hkpWorld::integrate()` actually performs all these operations.

**Collision Detection**

The simulation needs to determine whether any of the objects in your scene are colliding/overlapping. In Havok Physics 2012, collision detection is broken into three phases:

- **Broadphase** collision detection, which is a quick approximation that finds any potentially colliding pairs of objects.
- **Midphase** collision detection, which carries out a more detailed approximation of potential collisions.
- **Narrowphase** collision detection, which finally determines if any of these pairs are actually colliding.

If any pair of objects are colliding, the intersection information from the narrowphase is used to create collision agents and collision contact points. Those agents and contact points are used by the simulation solver in the following simulation frame.

The collision detection also solves all "Time Of Impact" (TOI) events that it generates, ensuring that collisions between high-quality objects are solved correctly, and avoiding any bullet-through-paper problems.

You can find out more about working with Havok collision detection, including creating shapes and collision filtering and collision queries, in later sections.

### 2.1.3 Demo Resources

The `demos` folder of your installation includes a number of useful demos that you can use to learn about Havok Physics 2012. Source code is given for each demo, allowing you to quickly find out how to perform important tasks, and providing code that you can cut and paste into your own applications. Helpful comments in the code describe each demo and the tasks performed in it. Together with the

documentation, we hope the demos will help you to familiarize yourself with the SDK and get started with Havok Physics 2012 in your own games.

Figure 2.21: Various screenshots of Havok Physics 2012 demos (shipped with full source code)

#### 2.1.3.1 The Demo Framework

All Havok demos use the same framework, which abstracts the differences between the various hardware platforms supported by Havok and also separates the display and simulation aspects of each demo. This abstraction leaves the physics source in one place so that it is easy to read and understand without distraction from non-essential information.

The framework includes a base example demo class (`hkDefaultDemo`) and a GUI application that can be used to navigate between demos and to play them.

Information on building and interacting with the demos for a chosen platform is available in its separate platform-specific manual.

#### 2.1.3.2 Demo Types

The demos are divided into three broad categories: *API*, *Use Case*, and *Test*. Each *API* demo illustrates a very specific aspect of working with Havok physics, such as setting up a single constraint between two rigid bodies. These will help you to gain familiarity with the feature set of the SDK, the concepts to which these features relate, and the key classes associated with individual features.

In contrast, *Use Case* demos combine multiple features and show how they can be used together, for example in a character or vehicle game. We have made these demos as simple as possible so as to isolate key concepts and show how they relate to one another. *Test* demos are used for demonstrating performance of the SDK under more realistic gameplay scenarios.

#### 2.1.3.3 Running the Visual Debugger Server with the Demo Framework

To use the visual debugger, run the demo framework with a "*-d*" command line parameter to have the demo start a visual debugger server instance.

More information on using the visual debugger (and on using it with the various consoles) can be found in the Visual Debugger section of this help file.

## 2.2 Creating a Simulation

This chapter will take you through the entire process of getting a simulation world up and running. Specifically, we'll explore how to create the physical objects and constraints, then how to populate a simulation world with these objects, and finally how to update the simulation world.

### 2.2.1 Creating Physics 2012 Objects

In this section, we will explain all the physical objects and constraints available to you and how to create them. We'll first explore the collision shapes supported by Havok Physics 2012, then the rigid bodies used to simulate those shapes. Finally, we will examine the various constraints available in Havok Physics 2012.

#### 2.2.1.1 Shapes

Shapes define the collision surface of rigid bodies and thus how they collide with other bodies and respond to collision queries. There are many types of shapes available in Havok Physics 2012, from simple implicit spheres or boxes, to triangle meshes exported from a modeler, or even a hierarchical compound structure using lists of shapes and transformed shapes. Shapes can be shared between or even within rigid bodies, and are usually created and serialized in a tools pipeline then later loaded by the application at runtime.

All shape information is provided by the **hkpShape** class and its subclasses. Among the important virtual functions provided by `hkpShape` are:

- `getType()`: Returns the shape's type, from the `hkcdShapeType` enum. This is used for example by `hkpCollisionDispatcher` to select the most appropriate collision detection algorithm (agent) to deal with each pair of potentially colliding objects. It should also be used to identify and cast to the appropriate shape type when parsing results from a user collision query, for example.

  Shapes can also have one or more "alternate" dispatch types. For example, while `hkpBoxShape` has a type of `hkcdShapeType::BOX`, both `hkpBoxShape` and an `hkpSphereShape` have an alternate type of `hkcdShapeType::CONVEX` since they inherit from the common `hkpConvexShape` parent class. These alternate types are registered dynamically with the dispatcher by `hkpAgentRegisterUtil`.

- `getAabb()`: Returns an Axis Aligned Bounding Box for the shape. This function is used for example by `hkpWorld` to update each object's broad phase AABB, and thus to determine which shapes are overlapping at any time.

- `getMaximumProjection()`: Returns the "furthest" point on the shape surface in a given direction, or an approximation thereof. This is used to help create bounding volumes for the shape.

- `castRay()`: Performs a ray cast against the shape, returning the closest intersection point, as well as additional information about which child shape of a hierarchical shape was hit (if relevant). More information on shape queries is available in the asynchronous queries section.

Shapes can generally be divided into several categories, which will be explored in this chapter.

- Convex shapes: These represent simple convex primitive shapes, which are the leaf shapes in any hierarchy and are used by the narrow phase collision detection algorithms.

- Container shapes: These represent sets of other shapes, allowing for creation of compound/concave shapes. These shapes implement the **hkpShapeContainer** interface via the `hkpShape::getContainer()` method. Each child shape in an `hkpShapeContainer` is identified using a `hkpShapeKey`–a 32-bit value which each container type may interpret in its own way. A shape key can be used by the container's `getChildShape()` method to retrieve a child shape from the container.

- Bounding Volume shapes: These can be used to wrap other shape containers, usually to accelerate queries into those shapes.

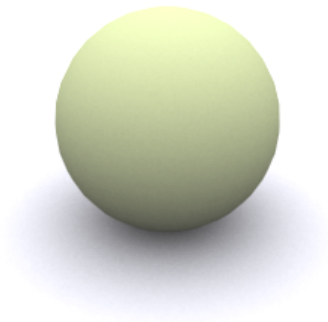
- Height Field shapes: These offer an alternative collision pipeline suited to representing terrain.

### Convex Shapes

**hkpConvexShape** is the parent class for all of the primitive convex shapes described in this section. All `hkpConvexShapes` have `hkcdShapeType::CONVEX` as an alternate dispatch type.

All convex shapes can be thought of at least one vertex plus a "convex radius" that adds an extra shell around those vertices. It is important to note that a large convex radius helps to minimize the work that the collision detector will need to do to, but large values also result in more rounded corners. Further information on the convex shape radius is available in the Optimizing and Tuning section.

### Sphere Shape



**hkpSphereShape** is a sphere *centered at the shape's local origin* with a specified radius. It is essentially a single vertex and a large convex radius, so is very efficient for collision detection.

```
hkpSphereShape* sphereShape = new hkpSphereShape( 2.0f ); // creates a sphere shape with a radius of 2
```

To create a sphere whose center is not the origin, you should make it the child of an `hkpConvexTranslateShape`. See the convex transform shapes section below for more information.

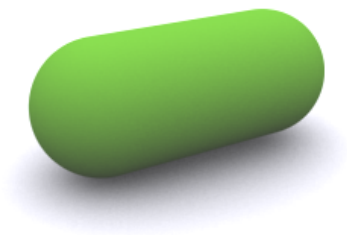
**Box Shape**

**hkpBoxShape** is a simple box *centered at the shape's local origin.* To create an `hkpBoxShape` you must specify its half-extents, as in the following example. An XxYxZ box has the half-extents (X/2, Y/2, Z/2).

```
hkVector4 halfExtent( 1.0f, 2.0f, 3.0f );
hkpBoxShape* boxShape = new hkpBoxShape( halfExtent, 0.0f ); // creates a 2x4x6 box shape, with zero "
  convex radius"
```

To create a box whose center is not the origin, you should make it the child of an `hkpConvexTranslateShape`. See the convex transform shapes section below for more information.

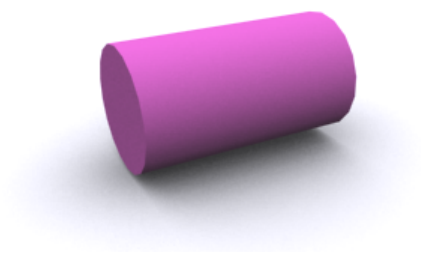
**Capsule Shape**



**hkpCapsuleShape** is a capsule defined by two vertices and a radius. These two vertices are specified in the shape's local space when creating the shape. The specified radius is used as the convex shape radius; therefore capsules are very efficient for collision detection.

```
hkVector4 start( 0.0f, 0.0f, 0.0f );
hkVector4 end  ( 0.0f, 2.0f, 0.0f );
float radius = 0.5f;

hkpCapsuleShape* capsule = new hkpCapsuleShape( start, end, radius ); // creates a capsule with a axis
  between "start" and "end", and the specified "radius"
```

### Cylinder Shape



**hkpCylinderShape** is a cylinder defined by two points and a non-zero cylinder radius. These parameters are specified in the shape's local space when creating the shape. Note that the cylinder radius is separate from the "convex radius".

```
hkVector4 start( 0.0f, 0.0f, 0.0f );
hkVector4 end  ( 0.0f, 2.0f, 0.0f );
float radius = 0.5f;
float optionalConvexShapeRadius = 0.1f;

// creates a capsule with a axis between "start" and "end", the specified "radius", and an optional
  additional "shell" radius
hkpCylinderShape* cylinder = new hkpCylinderShape( start, end, radius /*, optionalConvexShapeRadius */ );
```

### Triangle Shape

**hkpTriangleShape** is a simple triangle, stored in the shape as three `hkVector4` vertices. To create an `hkpTriangleShape`, specify the triangle's three vertices in the shape's local space:

```
hkVector4 v0( 0.0f, 0.0f, 0.0f );
hkVector4 v1( 0.0f, 0.0f, 5.0f );
hkVector4 v2( 5.0f, 0.0f, 0.0f );

hkpTriangleShape* triangleShape = new hkpTriangleShape( v0, v1, v2 ); // creates a triangle shape using
  these 3 vertices
```

An "empty" `hkpTriangleShape` can be created by passing no parameters to the constructor. The triangle details can specified later. The following example uses this approach:

```
hkpTriangleShape* shape = new hkpTriangleShape();

float vertices[] = {
 -0.5f, -0.5f, 0.0f, 0.0f, // v0
  0.5f, -0.5f, 0.0f, 0.0f, // v1
  0.0f,  0.5f, 0.0f, 0.0f, // v2
};

int index = 0;
int stride = 4;
for ( int i = 0; i < 3; i++ )
{
  hkVector4 v( vertices[index], vertices[index + 1], vertices[index + 2] );
  shape->setVertex( i, v );
  index = index + stride;
}
```

Note that triangle shapes are generally not created directly by the user, but rather created on-the-fly by higher-level container shapes.

Triangles also have an "extrusion" parameter. This is an `hkVector4` that defines the direction and length of extrusion for each triangle. By default it is set to a zero length vector. When the extrusion is non-zero, the collision detector implicitly converts the triangle to a convex object represented by the triangle extruded by the vector. Extrusion can help to avoid the "bullet through paper" problem in some situations without resorting to continuous simulation. It does not however guarantee robust results.

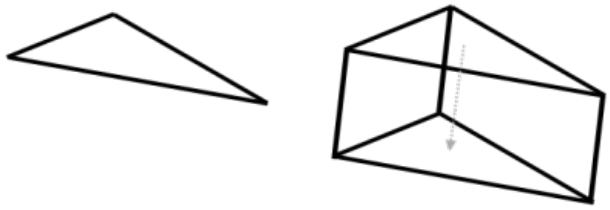
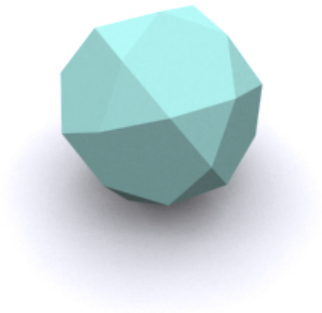


Figure 2.22: The triangle on the left is extruded downwards to obtain the convex shape on the right.

**Convex Vertices Shape**



**hkpConvexVerticesShape** is a generic convex shape represented by any number of vertices. The vertices are specified in the shape's local space. Note that the vertices given in the constructor are not shared, but instead are copied into the shape and stored in an optimized, transposed format. The vertex cloud given in the constructor used does not need to be convex; an appropriate convex hull will be generated from it. For further information please refer to the Convex Hull Generation documentation.

```
int numVertices = 4;

// 4 vertices plus padding
float vertices[] =
{
 -2.0f, 2.0f, 1.0f, 0.0f,  // v0
  1.0f, 3.0f, 0.0f, 0.0f,  // v1
  0.0f, 1.0f, 3.0f, 0.0f,  // v2
  1.0f, 0.0f, 0.0f, 0.0f   // v3
};

// This structure provides the build options
hkpConvexVerticeShape::BuildConfig config;
config.m_convexRadius = 0.1f;

// Creates a convex vertices shape from these 4 vertices and the convex radius.
// By default, the shape will be shrunk during construction so that the resulting "shell" matches the
  input vertices.
hkpConvexVerticesShape* shape = new hkpConvexVerticesShape( hkStridedVertices( vertices, numVertices ),
  config );
```

The `hkpConvexVerticeShape::BuildConfig` structure above contains several options controlling how the shape should be constructed. For example, the default is to "shrink" the vertices such that the resulting shell (including the convex radius) matches the input vertices, but alternatively the radius could instead be added to the input vertices if desired. Please see the `hkpConvexVerticeShape::BuildConfig` header

comments for details.

### Convex Transform/Translate Shape

**hkpConvexTransformShape** and **hkpConvexTranslateShape** are special convex shapes which wrap other convex shapes with an additional transform. This transform allows for shapes to be offset from their origin (this is especially useful in the case of sphere and box shapes), and also allows for convex shape instances within a hierarchy to use different local transforms. There are two types of convex transform shape available:

- `hkpConvexTransformShape` allows for translation, rotation, and scaling (where possible). This is the most general convex transformation available. Non-uniform scaling is supported for most convex shapes, but since the convex radius value cannot be non-uniformly scaled, it is not possible to non-uniformly scale a sphere shape for example.
- `hkpConvexTranslateShape` only allows for translation, but with less code it is slightly faster than `hkpConvexTransformShape` and is preferred where only a translation is required.

Being convex shapes, these transform shapes have their own convex radius value, distinct from the child shape's convex radius value. Normally, the convex radius value is copied from the child shape during construction. However if there is scaling involved in the transform, the convex radius may also be scaled in order to maintain an appropriately scaled convex "shell".

To create a convex transform shape, simply specify the child shape and a transform during construction:

```
// Create a child shape
hkpBoxShape* boxShape = [...];

// Create a translated version of the child shape
{
  hkpVector4 translation( 2.0f, 0.0f, 1.0f );
  hkpConvexTranslateShape* translatedBoxShape = new hkpConvexTranslateShape( boxShape, translation );
}

// Create a fully transformed version of the child shape
{
  hkVector4 translation( -0.9f, 0.7f, 0.0f );
  hkQuaternion rotation( hkVector4( 0.0f, 0.0f, 1.0f ), -0.4f ); // a rotation about the Z axis
  hkVector4 scale( 2.0f, 1.0f, 1.0f );

  hkQsTransform transform( translation, rotation, scale );
  hkpConvexTransformShape* transformedBoxShape = new hkpConvexTransformShape( boxShape, transform );
}
```

**Note:**

These shapes are not designed to have their transforms changed dynamically at runtime. If this happens, the collision detector will create new contact points with appropriate position and normal, but will not consider the convex surface to have any velocity. So the collision response will not be fully correct. Also, if the shape AABB changes, then the broad phase should be updated to reflect any new or removed overlaps (by reinserting the shape into the broad phase using `hkpRigidBody::setShape()` for example).

### Mesh Shapes

Container shapes which support a flat, but potentially large, list of convex primitive shapes are generally called "mesh" shapes in Havok Physics 2012. These are designed to efficiently store a large number of

convex child shapes, for example to represent the static landscape for an entire game level. Since mesh shapes do not store container shapes, they cannot return shape hierarchies as children. However, they can be used as part of a more generic container shape's hierarchy. This is often a good idea as it allows for detailed meshes to be instanced multiple times within a single hierarchy for example.

### BV Compressed Mesh Shape

**hkpBvCompressedMeshShape** is a memory- and performance-optimized container shape for storing a large, static set of triangles and/or other primitive convex shapes. The primitives are stored in a compressed format to minimize the memory footprint, and are wrapped with an internal static bounding volume tree to optimize spatial query performance. The combination of compression and query optimization makes this the ideal shape choice when representing detailed static landscapes.

Since this shape stores all of its primitive geometries internally in a compressed format, it cannot be used to reference external geometry. Also, the primitives and other properties of the shape cannot be modified after the shape is constructed. However, disabling (and enabling) of primitives can be achieved by wrapping the shape in a `hkpStaticCompoundShape`.

Please refer to the "`Demos/Physics2012/Api/Collide/Shapes/BvCompressedMesh`" demo for a detailed example of construction and use of this shape.

**Note:**
The "BV" in the name refers to this shape managing its own Bounding Volume. This is to distinguish it from the deprecated `hkpCompressedMeshShape`, which required a separate bounding volume.

### Construction

A `hkpBvCompressedMeshShape` is constructed by first deriving a `hkBvCompressedMeshShapeCinfo` structure to describe the source geometry. This structure provides several abstract methods which must be implemented to describe:

- The number of triangles and triangle vertices, and the vertex indices of each triangle.
- The number of other convex primitive shapes and pointers to those shapes.
- The collision filter information for each primitive (triangle or convex shape).
- The user data value for each primitive (triangle or convex shape).

The structure also provides several members which control:

- The default convex radius for all child shapes.
- Some tolerance values for the compression process.
- Whether instrumented welding information should be computed for the triangles.
- Whether a map should be built from the input primitive indices to their resulting shape keys.

A default `hkpDefaultBvCompressedMeshShapeCinfo` implementation is provided which uses a `hkGeometry` to provide the triangles, and a simple interface for adding other convex shapes.

A hkpBvCompressedMeshShape is then created by passing your construction info structure to the shape constructor. The construction info and the source geometries can then be disposed of, as the shape copies all geometry internally into its compressed format.

```
hkGeometry geometry = [...];
hkpDefaultBvCompressedMeshShapeCinfo cInfo( geometry ); // using the default construction info
  implementation here
hkpBvCompressedMeshShape* meshShape = new hkpBvCompressedMeshShape( cInfo );
```

Note that the construction process is costly, as it involves compressing the mesh, computing the bounding volume information, and potentially computing the welding information. So in most cases it is highly recommended to construct this shape offline in a tools process and serialize the result.

**Per-primitive data**

Each primitive (triangle or convex shape) in a BV compressed mesh shape has 32 bits available to store per-primitive data. This data is encoded as follows:

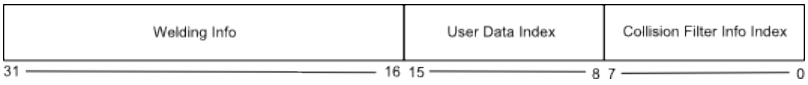



Figure 2.23: Per-primitive data encoding

- Lower 8 bits: Collision filter index. This value is used either as the actual collision filter info if the m_collisionFilterInfoMode construction option was set to PER_PRIMITIVE_DATA_8_BIT, or as an index into the collision filter info palette if PER_PRIMITIVE_DATA_PALETTE was set.
- Next 8 bits: User data index. Similarly to the collision filter index, this value can represent the actual user data value or an index into the user data value palette, depending on the value of the m_userDataMode construction option.
- Upper 16 bits: Instrumented welding information. This value is computed during construction if m_weldingType is set to something different to WELDING_TYPE_NONE. Keep in mind that computing welding information can be an expensive step that will add significantly to the construction time.

As stated above, collision filter information and user data can be stored either directly or in a palette, depending of the value of the m_collisionFilterInfoMode and m_userDataMode construction options respectively.

If the values are in the range [0,255], they can be stored directly in the per-primitive data, by specifying the PER_PRIMITIVE_DATA_8_BIT construction option, which will avoid allocation of extra space in the palettes and, for collision filter info, avoid extra DMAs from SPU to access the final values.

If the values exceed 255, specifying the PER_PRIMITIVE_DATA_PALETTE construction option will build a palette of all the different values, with the per-primitive user data or collision filter info storing the index of the actual value in the palette. Please note that even when using palettes to store values greater than 255, there is still a limit of 256 *unique* values, due to the limit of 256 index values.

Alternatively, for user data, specifying the PER_PRIMITIVE_DATA_STRING_PALETTE option at construction will allow storage of user-supplied strings instead of integers in the palette. This is used by the content tools filter to store material names; see the *Havok Content Tools documentation* for more information.

Per-primitive data is separated from geometry data internally, and is Run-Length Encoded so that adjacent primitives with uniform values will be efficiently stored. For example if no per-primitive data is required at all across the whole mesh, all per-primitive data will be zero and so internally this will compress very well to almost nothing. Due to this compression scheme, per-primitive data cannot be changed after construction. When using palettes for collision filter information or user data though, the value associated to all primitives with the same index can be modified afterwards by accessing the palette values.

**Limitations and known issues**

- BV compressed mesh shape should not be used for dynamic bodies. Any dynamic bodies using it are currently forced to PPU on PlayStation®3 for collision detection. Please use list shapes instead for dynamic bodies.
- Collision filter info and user data are limited to 256 unique values, even when using palettes.
- The number of primitives in a single BV compressed mesh shape is limited to $2^{23}$.

**Extended Mesh Shape**

**hkpExtendedMeshShape** is a shape container that references an arbitrary triangulated mesh and/or a list of other convex shapes. The triangles and convex shapes are referenced by the extended mesh shape rather than being stored, making this a flexible mesh shape for the cases where you wish to reuse geometry that already exists. For example, an existing render geometry could be referenced by this shape to in order to create a matching physics geometry (though not recommended in general as render geometries may be very detailed).

This shape handles both triangle lists and triangle strips, as well as handling degenerate triangles gracefully. Note that it does not handle triangle fans directly as the striding between the vertices is not constant ($\text{vertex}_0$ + $\text{vertex}_{n+1}$ + $\text{vertex}_{n+2}$ for the $\text{n}^{th}$ triangle). The groups of triangles that make up the `hkpExtendedMeshShape` are known as its "subparts". Each triangle's subpart has an `hkQsTransform` member which lets you apply translation, rotation and scaling to your returned vertex data for that subpart. This is a good alternative to using shape hierarchies.

This shape does not maintain any bounding volume structure of its own, so should always be wrapped in a separate bounding volume shape (usually a MOPP shape).

**Storage Details**

Each triangle's subpart structure contains the following information:

- (Vertex Info) a pointer to the vertex data.
- (Vertex Info) the byte offset in memory between consecutive vertices, also known as the vertex striding.
- (Vertex Info) the number of vertices in the subpart.
- (Triangle Info) a pointer to the triangle index data.
- (Triangle Info) whether 8, 16, or 32 bits are used to index the triangles.
- (Triangle Info) the offset between index triplets in the triangle index, also known as the index striding.

- (Triangle Info) the number of triangles formed by the subpart.

The triangle index (or index buffer) is an array which indicate which vertex triplets form triangles. Vertex indexing enables vertex sharing, removing the need to store multiple instances of the same data. It also facilitates the use of triangle strips, where adjacent sets of triangle indices overlap. A triangle strip can be thought of as a window of 3 indices that progresses by one index along the index array for each triangle, in which case the index striding is equal to the size of a single index element, i.e. either 2 bytes or 4 bytes (depending on whether 16-bit or 32-bit indices are being used).

For instance, if you store 16bit indices in the a large unsigned short array, one after each other, as a triangle strip, the index striding is `(1*sizeof(unsigned short))`, which is 2. If you store your indices as 32 bits and say that they represent a triangle list (not a strip), the striding is `(3*sizeof(int))` which is 12.

**Per Triangle Material Tagging**

Each triangle and subpart can hold specific material information. Shape keys are used to index the material information on mesh subparts and triangles.

Each subpart has members `m_materialIndexBase`, `m_materialIndexStridingType` and `m_materialIndexStriding` which associate 8 or 16 bit integer indices with each triangle or for an entire subpart (by setting striding to zero). Note that the striding does not need to match the striding type for interleaved indices.

```
// Simple eight bit indices
hkInt8* indices8 = ...;
part.m_materialIndexBase = indices8;
part.m_materialIndexStridingType = hkpExtendedMeshShape::MATERIAL_INDICES_INT8;
part.m_materialIndexStriding = sizeof(hkInt8);
```

```
// Interleaved eight bit indices
struct Interleaved { hkInt8 materialIndex; hkInt8 morePackedData; } * interleaved;
part.m_materialIndexBase = interleaved;
part.m_materialIndexStridingType = hkpExtendedMeshShape::MATERIAL_INDICES_INT8;
part.m_materialIndexStriding = sizeof(Interleaved);
```

Each subpart also has `m_materialBase`, `m_materialStriding` and `m_numMaterials` members which refer to the material data. Again, the material array may be interleaved. Note that using `m_materialBase` implies enabling collision filtering as the base class `hkpMeshMaterial` contains a filter value.

```
// externally stored materials - we use the materialIndex into an external array
part.m_materialBase = HK_NULL;
```

```
// Interleaved material
struct MyMaterial : public hkpMeshMaterial
{
 // inherited hkUint32 m_filterInfo for triangle collision filtering
 int m_extraData;
};

MyMaterial* interleavedMaterials;
part.m_materialBase = interleavedMaterials;
part.m_materialStriding = sizeof(MyMaterial);
part.m_numMaterials = ...;
```

### Compound Shapes

While mesh shapes are designed to efficiently represent a flat list of convex shapes, other more generic container shapes can have container shapes as children, allowing compound multi-level shape hierarchies to be created. There is no theoretical limit on the complexity of the hierarchy, provided it is not recursive and does not contain a loop. There may of course be performance loss associated with large hierarchies. Practically speaking, shape hierarchies should be limited to 2 levels maximum (for example, a shape container of transformed shape containers). This is especially applicable on the PlayStation®3 as the SPU stack may overflow due to the extra memory costs.

### Transform Shape

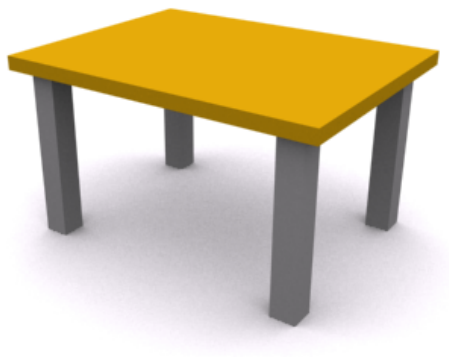
**hkpTransformShape** allows for positioning of child shapes relative to their parents other when creating a compound shape hierarchy. This is similar to `hkpConvexTransformShape`, with the difference being that `hkpTransformShape` supports any type of child shape (not just convex), allowing for containers of transformed containers for example.

However, this shape requires that an extra runtime collision agent be created for each collision and therefore incurs the associated memory and performance costs. So `hkpConvexTransformShape` should always be used instead when the child shape is convex. The performance implications of using transform shapes are explored in more detail in the Optimizing Collision Detection section.

### List Shape

**hkpListShape** is a simple list of `hkpShape`s of any type. It is designed for relatively small numbers of children (i.e. not for landscape shapes), and does not maintain any bounding volume information of its own. Therefore if the list has a large number of children, it is advisable to wrap the list in a bounding volume shape in order to accelerate any queries.

The following example illustrates using an `hkpListShape` to build a simple compound shape for a dynamic body, in this case a table consisting of several distinct box shapes:



The first step in creating the list shape is to create the `hkpShapes` that will become the children:

```
hkVector4 halfExtent1( 2.0f, 1.0f, 0.1f );
hkVector4 halfExtent2( 0.1f, 0.1f, 1.0f );

hkpBoxShape* box1 = new hkpBoxShape( halfExtent1 );
hkpBoxShape* box2 = new hkpBoxShape( halfExtent2 );
```

Then, because the `hkpListShape` constructor needs a pointer to an array of shapes, we also create an array to store the child shapes:

```
hkArray<hkpShape*> shapeArray;
```

Before adding the child shapes to the array we need to reposition them, as otherwise the boxes will all be created at the same position. To do this, we create an `hkpConvexTranslateShape` for each of the table legs; the following snippet shows the creation of one of these. These new transform shapes are added to the array as they are created:

```
// Add the table top shape to the array as it is
shapeArray.pushBack( box1 );

// For each table leg, wrap the other shape in a convex translate shape and store it in the array
hkVector4 offset( -1.0f, -0.5f, -0.5f );
hkpConvexTranslateShape* translateShape = new hkpConvexTranslateShape( box2, offset );
shapeArray.pushBack( translateShape );
```

Finally, the `hkpListShape` is created using the array of shapes:

```
hkpListShape* listShape = new hkpListShape( &shapeArray.begin(), shapeArray.getSize() );
```

`hkpListShape` allows for child shapes to be disabled and re-enabled on-the-fly, however there is an upper

limit of 128 on the number of child shapes that can be disabled.

### Static Compound Shape

**hkpStaticCompoundShape** is a shape container intended to be used for landscapes, buildings or other large static structures that are composed of repeated instances of other shapes. For example, the landscape below is a static compound shape composed of multiple transformed instances of three basic shapes: a tree asset (`hkpBvCompressedMeshShape`), a boulder (`hkpConvexVerticesShape`), and the terrain surface (`hkpBvCompressedMeshShape`).

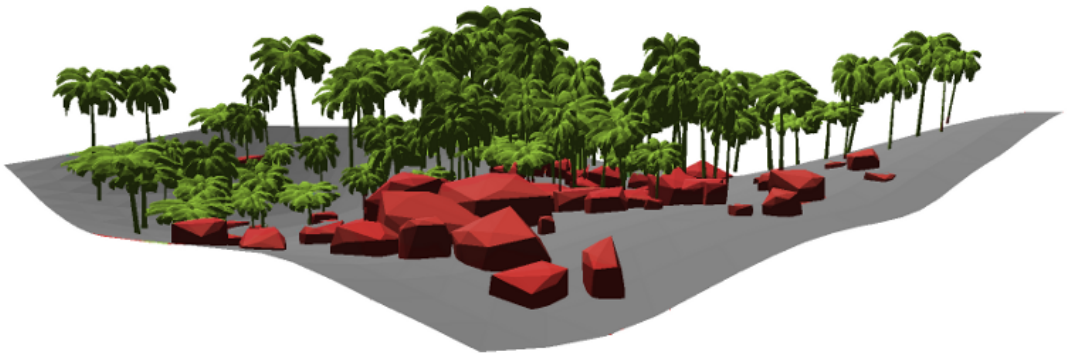


Figure 2.24: A static compound shape instancing a small number of reference shapes many times.

(The above demo can be found under "`Demos/Physics2012/Api/Collide/Shapes/StaticCompound/Construction`".)

In comparison to a list shape, static compound shape offers:

- Reduced runtime memory overhead for nested shape containers (described below).
- Reduced static memory overhead per instance (64 bytes).
- Support for *non-uniform scale* per instance (as well as translation and rotation).
- Ability to *disable any instance or shape key* within any instance, without modifying the child shapes.
- Per-instance mutable user data and collision filter info (with masking).
- An internal bounding volume tree around the instance AABBs, optimized for this use case.

Static compound shape reduces runtime memory overhead for nested containers by presenting itself to the engine as a single flat list of convex shapes (in most use cases), while still allowing the user to treat it as a normal shape hierarchy. By presenting itself as a single flat list, it avoids the runtime performance and memory overhead associated with the nested collision agents that would otherwise be created. It achieves this by encoding several pieces of data in the shape key, and by wrapping leaf shapes in convex transform shapes when necessary. The diagram below demonstrates the concept: the orange colored parts are leaf shapes that each will have a unique shape key, as seen from the top layer static compound shape. This means that shape containers like lists or meshes will not be directly seen by the engine. Each numbered line in the diagram represents an instance, that can have its own transform. The red number marks the instance ID for each instance in a static compound shape.

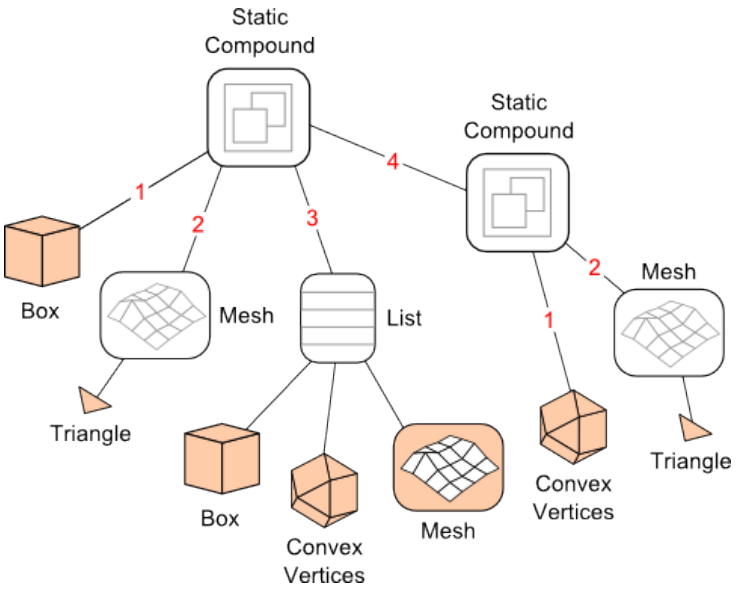



Figure 2.25: Hiding containers within a static compound shape. Shapes shaded in orange correspond to a single shape key in the root shape's shape key space.

Each instance of a child shape in a static compound shape has its own spatial transformation. This transform is represented as an `hkQsTransform`, supporting translation, rotation and non-uniform scaling. If an instance contains a shape container, all its child shapes will be affected by the transform specified in the instance when returned by the static compound shape. Any leaf shape's final transform will be the composition of all instance transforms encountered by walking up the instance tree from leaf to root. However, due to performance considerations, `hkQsTransform` does not support transforms involving skewing. This has the implication that in general, non-uniform scaling can only be safely used with instances of shapes that do not apply any rotations to their child shapes, otherwise skewing is likely to be present in the final transform. In practice however, this skewing factor will be truncated and the resulting transform will be wrong or unexpected, so care must be taken when specifying non-uniform scale in instance transforms.

### Construction

A `hkpStaticCompoundShape` is constructed as an empty shape, then the (external) child shapes are added as instances, each with their own transform. The shape and transform of each instance cannot be modified after it has been added. When all instances have been added, the shape must be baked to finalize its caches and build the internal bounding volume. Note that the static compound shape maintains references to the external child shapes after construction. These *child shapes should not be modified* in any way after adding them to the static compound shape, as the static compound shape maintains caches and a bounding volume based on the child shape configurations.

The code below shows a basic example of creating a `hkpStaticCompoundShape` using two instances of the same `hkpBvCompressedMeshShape` with different transforms. After the final call to `bake()`, no further calls to `addInstance()` are possible, and the child shapes should not be modified.

```
// Create a child shape
hkpBvCompressedMeshShape* meshShape = [...];

// Create some transforms to instance the child at (including non-uniform scale)
hkQsTransform transform1 = [...];
hkQsTransform transform2 = [...];

// Create the hkpStaticCompoundShape and add the instances.
// "meshShape" should not be modified by the user in any way after adding it as an instance.
hkpStaticCompoundShape* staticCompoundShape = new hkpStaticCompoundShape();
int id1 = staticCompoundShape->addInstance( meshShape, transform1 );
int id2 = staticCompoundShape->addInstance( meshShape, transform2 );

// This must be called after adding the instances and before using the shape.
staticCompoundShape->bake();
```

### Collision Filter Info Masking

Both a collision filter info and a collision filter "mask" can be specified for each instance in a static compound shape. These can be changed at any time, both before and after the shape is baked. The mask is applied to any collision filter info that the instance's child shape may provide. In the example of the forest above, each tree is a `hkpBvCompressedMeshShape`. This shape can have collision filter info per triangle. The mask we specify for each tree will be applied to this child collision filter info. In turn, the result of that is combined with the filter info provided by the instance itself. The formula for this is:

```
filterInfo = ( childFilterInfo & instanceFilterMask ) | instanceFilterInfo
```

This structure allows for flexible collision filtering with minimal overhead. Below is an example of setting filter information and filter masks.

```
// We want filter info for all the children of instance 1 to be provded completely by its children
staticCompoundShape->setInstanceFilterInfoMask( instanceId1, 0xffffffff );
staticCompoundShape->setInstanceFilterInfo( instanceId1, 0x0 );

// We want filter info for all child shapes of instance 2 to be group filter layer 2, regardless of its
  children
staticCompoundShape->setInstanceFilterInfoMask( instanceId2, 0x0 );
staticCompoundShape->setInstanceFilterInfo( instanceId2, hkGroupFilter::calcFilterInfo(2) );
```

The following diagram gives an example of how the filter information is computed when a child shape is a container. Instance filter mask and filter information are noted in red and blue respectively. Child shapes' collision filter infos are written in black. The final filter information is shown on the right.

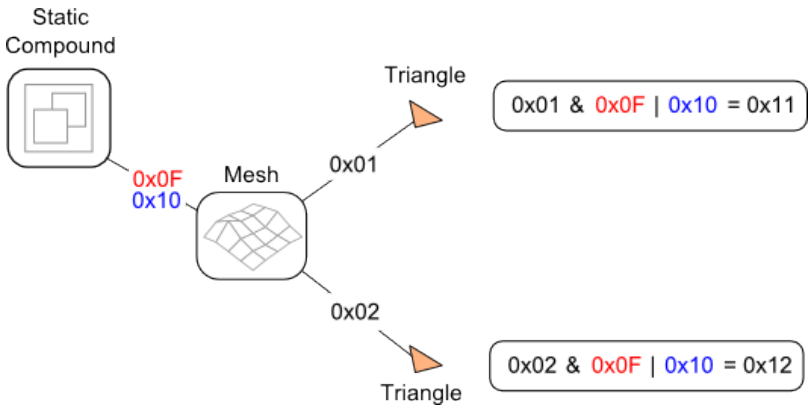



Figure 2.26: Collision filter info masking

### Disabling Instances and Shape Keys

Static compound shape allows for any instance or individual child shape key to be disabled or re-enabled on the fly, *without touching the referenced child shapes*. This means that you can for example break one window pane of a single building that is instanced many times, or break an entire building instance at once. To set an instance or shape key state, simply do the following:

```
// Disable a whole instance
staticCompoundShape->setInstanceEnabled( instanceId, false );

// Disable a single shape key
staticCompoundShape->setShapeKeyEnabled( shapeKey, false );
```

Often, you will have retrieved a static compound shape's shape key from a collision event or query, and will wish to determine the instance that this belongs to (in order to disable it). Static compound shape provides a utility function to achieve this. Sticking to the forest example above, lets assume that we have obtained a shape key for some triangle hit by a ray cast. We now wish to disable the instance representing the tree containing that triangle. This can be accomplished in the following manner:

```
hkpShapeKey hitKey = [...]; // e.g, obtained from ray cast

// Decompose the shape key into: the instance ID; that instance shape's child shape key
int instanceId;
int childShapeKey;
staticCompoundShape->decomposeShapeKey( hitKey, instanceId, childShapeKey );

// instanceId now refers to that instance of the "tree", so let's disable it
staticCompoundShape->setInstanceEnabled( instanceId, false );

// or you may wish to look up something from the referenced "tree" shape...
const hkpShape* childShape = staticCompoundShape->getInstances()[instanceId]->getShape();
[...];
```

The "`Demos/Physics2012/Api/Collide/Shapes/StaticCompound/KeyDisabling`" demo demonstrates this concept, as shown in the screenshot below. A single static compound shape contains two instances of a BV compressed mesh shape (which contains many box shapes). As the ray cast moves around, individual shape keys of one of the instances are disabled.

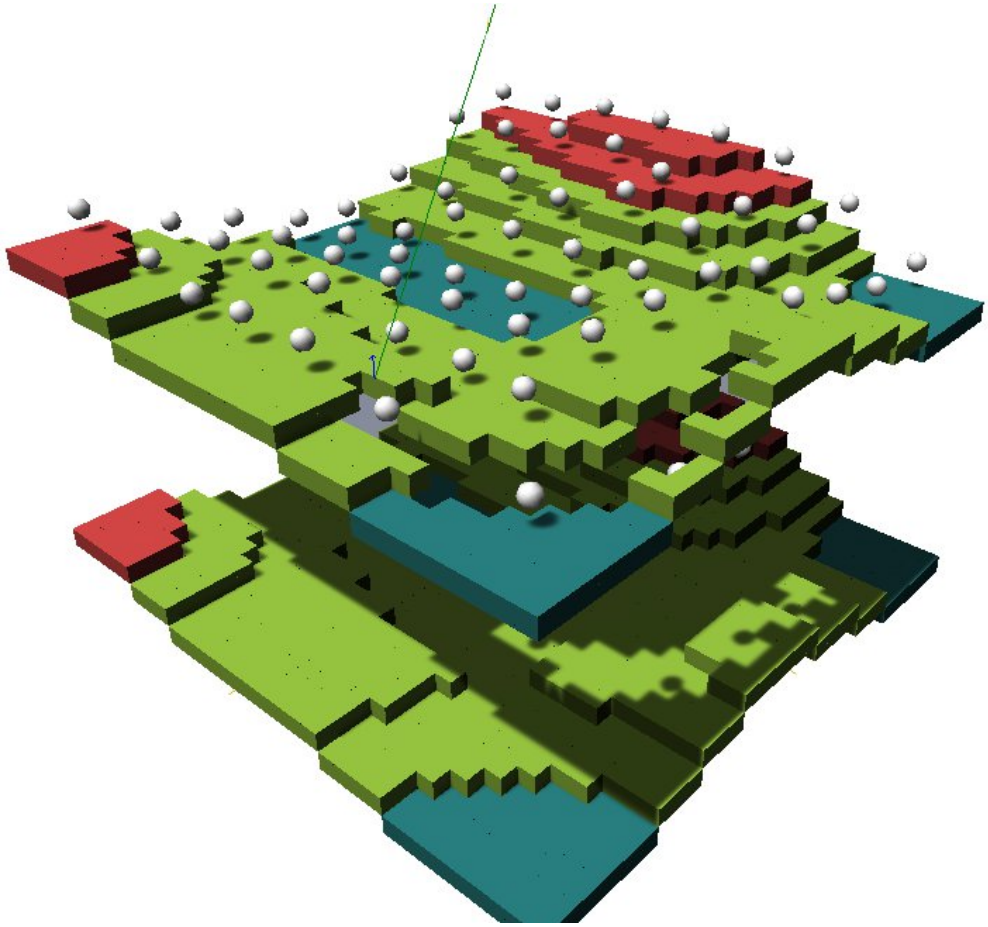


Figure 2.27: Enable and disable unique shape keys in nested static compound shapes

**Limitations and known issues**

- This shape should not be used for dynamic bodies. Any dynamic body found to be using it is currently forced to PPU on PlayStation®3 for collision detection.
- Instance transforms do not support skewing, which will be truncated. A safe way to avoid skewing is to use non-uniform scaling only in instances of shapes that do not apply any additional rotation to their child shapes.
- Non-uniform scaling on instances of shapes that contain triangle meshes with welding information (such as `hkpBvCompressedMeshShape` or `hkpExtendedMeshShape`) will produce incorrect welded contact normals, as the precomputed triangle welding information is only valid as long as any scale applied is uniform. There is currently no support for non-uniform on-the-fly scaling of instrumented welding information.
- Degenerate triangles can be produced by the transformation (especially reducing scale) of otherwise valid triangles. Such degenerate triangles are currently not pruned, and can cause asserts in the collision detector. It is recommended to avoid using reduced scale with mesh instances. Instead, create the mesh at the smallest instance scale and scale it up for the other instances.
- Certain queries used by the shape are more efficient when instances do not have large rotations. It is recommended to avoid using rotations in instances where possible.
- There is currently no support for this shape in the Havok Content Tools.

### Bounding Volume Shapes

Bounding volume shapes define a simple volume around complex shapes (such as mesh shapes or shape collections). These bounding volume shapes can significantly speed up collision detection for complex shapes as it can filter the precise collision detection by a simplified check. If the simplified check results in a non colliding state, no further checks are necessary and we can skip calling the precise collision detection. Shape containers with lots of child shapes should always be wrapped in some form of bounding volume; otherwise queries into the container can be very slow.

There are several bounding volume implementations in Havok Physics 2012:

- `hkpBvShape` shape wraps a single complex shape with a single simple bounding volume shape (user definable). This should only be used for Phantom Callback Shapes.
- `hkpBvTreeShape` creates an optimized spatial tree for a large list of children, providing optimized access to children shapes which are in close proximity of the colliding body.
- `hkpTriSampledHeightFieldBvTreeShape`, which is only used for a specialized implementation of height fields. This shape will be discussed below.
- Some shapes have their own internal bounding volume structure, notably `hkpBvCompressedMeshShape` and `hkpStaticCompoundShape`. These should *not* be wrapped in any other bounding volume.

### Simple Bounding Volume Shapes

An `hkpBvShape` has two `hkpShape` members, one is the encapsulated child shape, the other describes the bounding volume used for the child shape. A collision agent for collisions between an `hkpShape` S and the child shape is created only when S intersects (penetrates) the bounding volume. The agent is deleted if S ever becomes disjoint from the bounding volume. Thus collisions between S and the child shape will only be detected when S also intersects the bounding volume.

Consider a car (with a detailed physics representation) driving over a landscape. By using a box as a bounding volume shape we can speed up collision detection for the complex car as long as the box is not constantly colliding with the landscape. If we expect the box to continuously collide (with the landscape), we still have to perform the whole complex collision work and the bounding volume shape becomes useless and an unnecessary CPU cost.

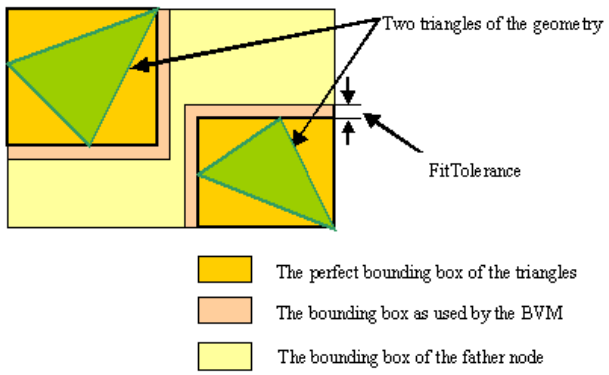
To re-iterate, the primary use of the `hkpBvShape` is for the creation of "phantom" callback shapes which are discussed in detail here. For all other situations, we recommend the use of Bounding Volume Tree Shapes (`hkpBvTreeShape`).

### Bounding Volume Tree Shapes

An `hkpBvTreeShape` adds extra bounding volume information to an `hkpShapeCollection` (such as an `hkpExtendedMeshShape`) to enable fast region queries. In fact while the underlying acceleration structure does not necessarily have to be a tree structure, in practice it usually is one (such as an AABB tree or a BSP tree), hence the name. A bounding volume tree is very useful in situations where the collision detector must check for collisions between a moving object and a large static geometry such as a landscape.

Before all shapes inside the collection are dispatched for collision with an `hkpShape`, a region query can be used to produce a smaller subset of shapes in the collection which may collide with, say, the AABB of that shape. It is also used often to speed up other queries such as raycasts.

If you use an `hkpBvTreeShape` for a landscape, the shapes that make up the landscape are hierarchically grouped in a bounding volume tree. At every node in the tree there exists a bounding polytope, which encapsulates all of its children. The top-level bounding volume contains the entire landscape, while the nodes on the leaf levels encapsulate one geometric primitive, normally a triangle. The fit of this bounding volume can be perfect (as in some AABB trees), or can have an extra margin/tolerance built in (e.g. MOPP).




In the case of a landscape, instead of checking whether the moving object is colliding with each of the triangles in the landscape in turn (which is extremely time-consuming), the bounding box of the moving object is checked against the bounding volume tree: first Havok evaluates whether the moving object is intersecting with the top-level bounding volume, then with any of its child bounding volumes, and so on until the check reaches the leaf nodes. A list of any potentially colliding triangles are then passed to the narrow phase collision detection. You can think of the bounding volume tree as a filter to the narrowphase collision detection system.

We recommend that the `hkpBvTreeShape` be used for every large static shape in the world.

**MOPP Bounding Volume Tree Shape**

**hkpMoppBvTreeShape** is an optimized Havok acceleration tree structure (modeled on the same principles as KDOP tree). It is a fast, memory-optimized structure, primarily meant to be used with triangle meshes.

The `hkpMoppBvTreeShape` is an implementation of the `hkpBvTreeShape` using a MOPP. A MOPP uses a number of optimizations to speed up collision queries with the underlying shape. Each leaf node bounding box in the MOPP tree has a "fit tolerance" that indicates how closely the box fits the enclosed shape. A default fit tolerance is provided, but the fit tolerance can also be manually specified to optimize the tree for specific needs. Users are encouraged to experiment with different values to obtain the best possible results. More information on customizing the fit tolerance can be found in the Optimizing and Tuning section.

The following example illustrates how to use an `hkpMoppBvTreeShape` to create a landscape which has extra bounding volume information. First, an `hkpExtendedMeshShape` is created for the landscape's geometry. Then a single subpart is created, specifying the necessary vertex and index information, and added to the mesh shape. While the mesh shape in this example has only a single subpart, in general use an `hkpExtendedMeshShape` will contain many subparts corresponding to different areas of the landscape.

```
m_mesh = new hkpExtendedMeshShape;

// it is common to have a landscape with 0 convex radius (for each triangle)
// and all moving objects with non zero radius.
m_mesh->setRadius( 0.0f );

{
    hkpExtendedMeshShape::TrianglesSubpart part;

    part.m_vertexBase = vertices;
    part.m_vertexStriding = sizeof(float) * 3;
    part.m_numVertices = numVertices;

    part.m_index16Base = indices;
    part.m_indexStriding = sizeof( hkUint16 );
    part.m_numTriangleShapes = numTriangles;
    part.m_stridingType = hkpExtendedMeshShape::HK_INT16_INDICES;

    m_mesh->addTriangleSubpart( part );
}
```

The MOPP information is then created for the mesh shape using the `hkpMoppUtility::buildCode()` function (which builds an `hkpMoppCode`). The `hkpMoppCode` holds all the actual MOPP tree and bounding volume information and is used whenever the MOPP tree is traversed. In this example, the default fit tolerances are used, but it is important to set these tolerances manually according to the type/scale of data used. See the MOPP fit tolerance part of the Optimizing the narrowphase section.

```
hkpMoppCompilerInput mci;
m_code = hkpMoppUtility::buildCode( m_mesh, mci );
```

Finally, the mesh shape and its code are used to create the `hkpMoppBvTreeShape`.

```
hkpMoppBvTreeShape* moppShape = new hkpMoppBvTreeShape(m_mesh, m_code);
```

The MOPP is generally very low on memory consumption (on average ∼10 bytes per triangle in a mesh shape although this number will vary depending on the parameters used for the tolerance fit ).

**Note:**
In general, it is recommended that only one MOPP landscape is used per scene/level.

### Height Field Shapes

Havok Physics 2012 provides height fields as a method of representing terrain without using triangles or other convex shapes. There are several reasons to consider using height fields for terrain representation instead of a typical triangulated mesh:

- Smaller memory footprint than a standard triangulated landscape.
- No requirement for the shape to store any geometry. It can simply use a height lookup function.
- A collision agent specially designed to do collision detection between height field shapes and normal

convex shapes is as fast as, if not faster than, a bounded-mesh-based approach (depending on the type of height field used).

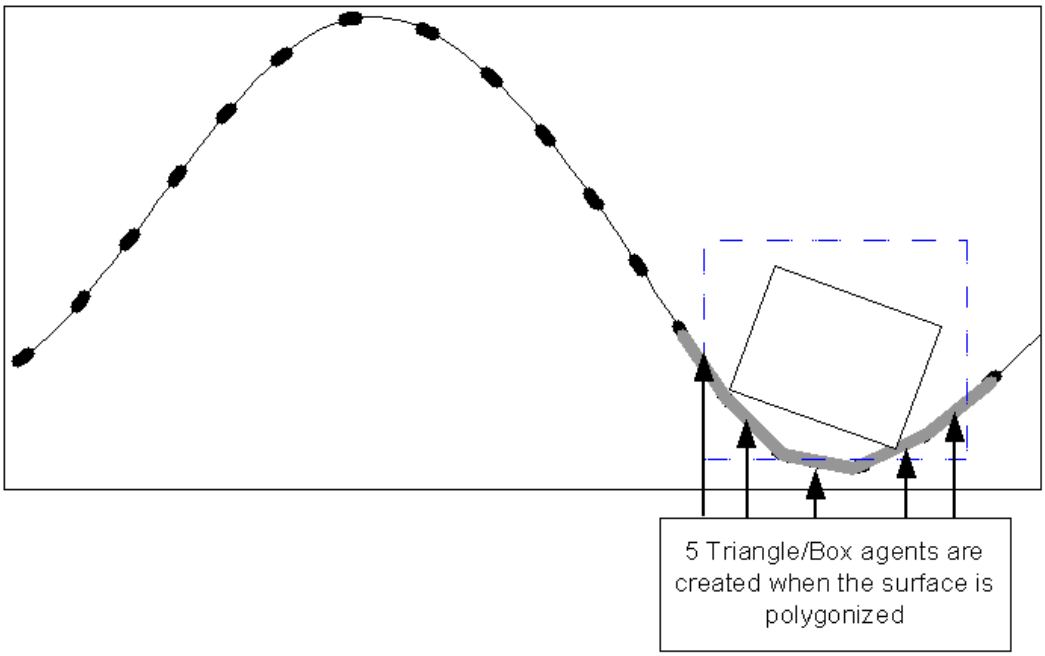
Height fields also come with certain limitations:

- Cannot handle overhangs or any formations other than displacement.
- Limited in both precision and amount of variation in the landscape.

### Understanding Height Fields

The height field shape makes several approximations to keep the CPU and memory footprint required as low as possible. It is important to understand the approximations made and their consequences before choosing to represent a terrain as a height field. However, if these approximations are acceptable, the use of height fields will provide clear improvements in the amount of CPU and memory required for collision detection.

The initial memory benefits of using a height field are immediately obvious. The static memory footprint depends on the underlying data structure used to represent the surface but is often far lower than storing a polygonal representation, e.g. a 16-bit height field stores a surface with just 1 byte per triangle. There are also no bounding volume overheads that would be associated with a triangulated mesh approach. Finally, at runtime, only a single agent is needed for each object that interacts with the height field. If this height field surface were considered at a polygon level then potentially several agents would be created for each interacting object.

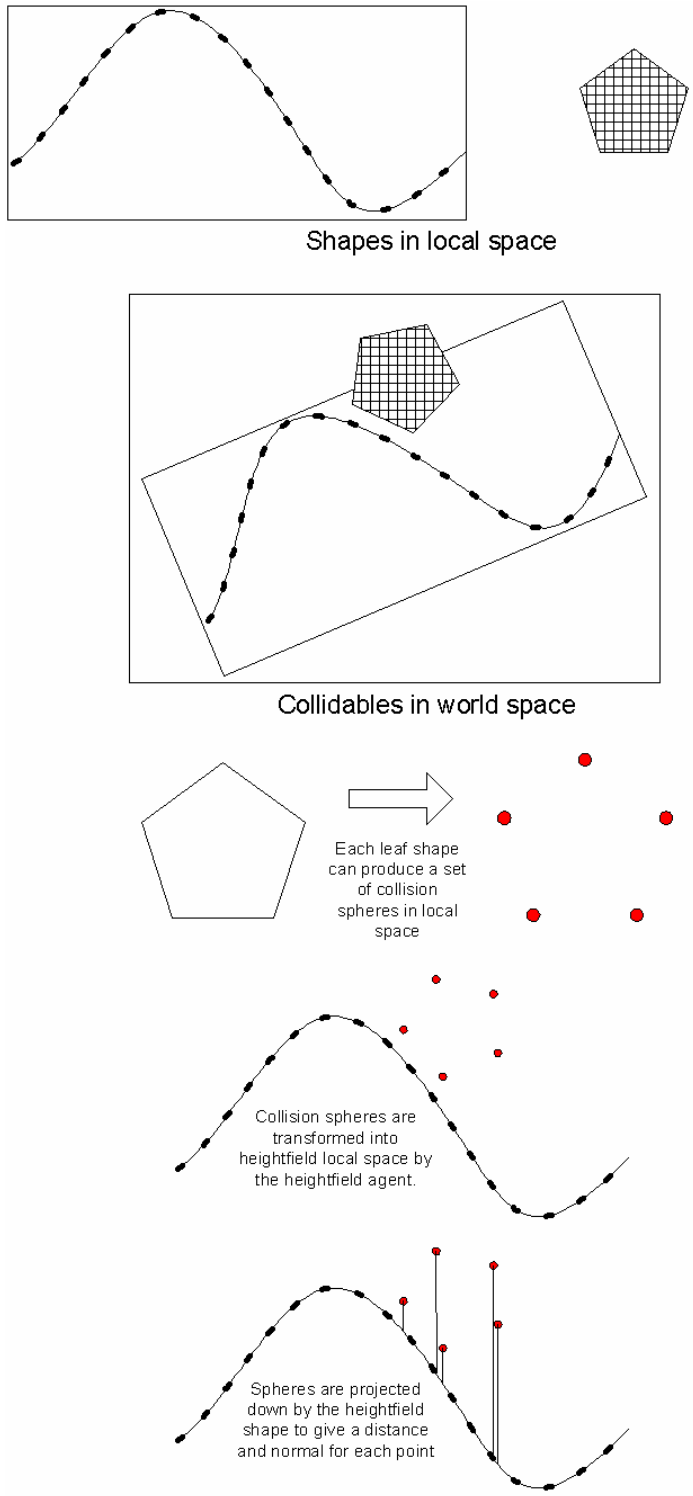



### hkpHeightFieldShape Interface

**hkpHeightFieldShape** is the interface for this surface representation. Its implementation requires providing ray casting support, sphere casting support (see approximations) and the ability to report a

set of distances for a given set of spheres. The spheres are used to collide convex primitive shapes against the height field shape.

The spheres are generated using the `hkpSphereRepShape` interface, which all convex primitive shapes implement. For example, a convex vertices shape implements this by creating a sphere with a small radius at each of the vertices of the convex vertices shape. Since the `hkpSphereRepShape` is used, only a single collision agent is needed to handle all collision detection.




The data flow through the height field collision agent is optimized to avoid any heap allocations. The simplified intermediate sphere representation is created through the interface on the stack and only exists while the agent processes collisions.

The most common implementation is a regularly sampled height field, but on its own, this generic

implementation may be useful when using specialized surfaces (such as parametrically defined spline surfaces). The `UserHeightFieldDemo` demonstrates how to implement this generic height field: instead of assuming a grid-based displacement map, it models the height field as a hollow sphere and does a simple sphere-sphere test in the main `collideSpheres()` method. However, the `castRay()` functionality is not implemented in the base height field class.

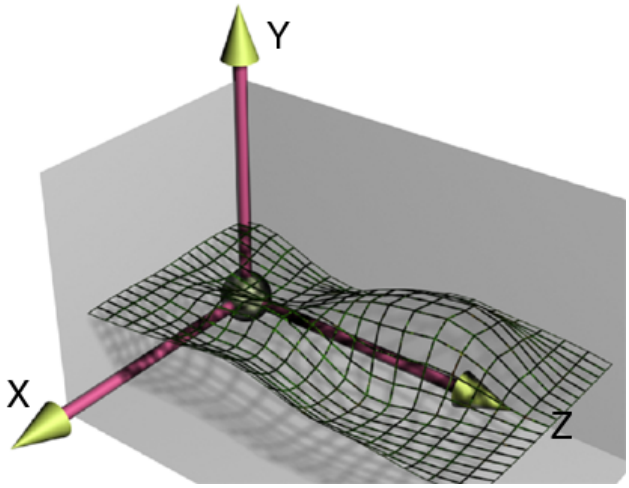
### Comparing Height Field Solutions

Havok provides two height field shape implementations: the sampled height field and the triangulated sampled height field. While the (default) sampled height field is a simple position-based distance lookup table, the triangulated sampled height field is simulated using triangle meshes. Each method comes with its own set of advantages and drawbacks:

| Sampled Heightfield | Triangulated Sampled Heightfield | |
|---|---|---|
| **Precision** | Approximate. | Precise. |
| **Curvature** | Can handle small curvatures only (gently undulating terrain). | Flexible. |
| **Holes** | No support. | Can handle holes or custom collision filters on parts of the height field. |
| **Continuous Collision Detection** | No support, but will prevent any tunnelling due to the one-sided nature. | Supports continuous collision detection, but is also a two-sided collision system (items *can* tunnel). |
| **Speed** | Fastest. | Slower, but as fast as a MOPP based solution. |
| **Memory Usage** | Smallest, very memory cache friendly. | Small, less cache friendly. |
| **Linear Cast Speed** | Fast. | Usually fast, but slow for long objects. |

Table 2.4: Comparison of Sampled and Triangulated Sampled Heightfields

### Sampled Height Field




The above image shows the default configuration for a sampled height field defined in local space. Havok provides an optimized default implementation (`hkpCompressedSampledHeightFieldShape`). The implementation allows the easy reuse of displacement data (if it is shared with the renderer for example). It also contains a highly optimized and exact ray cast implementation that works well for both short and long raycasts.

**Note:**
Havok also provides a reference implementation for debug snapshot purposes called `hkpStorageSampledHeightFieldShape`.

### Sampled Height Field User Implementation

To provide a custom implementation of the sampled height field, a class must inherit from the `hkpSampledHeightFieldShape` class and implement the `getHeightAt()` and `getTriangleFlip()` methods. It must also forward the `collideSpheres()` method to a templated implementation. This forwarding is an optimization that allows Havok to inline the user implementation of the `getHeightAt()` and `getTriangleFlip()` methods. This significantly improves the performance for ray casting and collision generation.

Here is what a user-defined height field class would look like (from the `SampledHeightFieldDemo`):

```
class MySampledHeightFieldShape: public hkpSampledHeightFieldShape
{
  public:
      MySampledHeightFieldShape( const hkpSampledHeightFieldBaseCinfo& ci): hkpSampledHeightFieldShape(ci)

      {
      }

      // My simple height lookup.
      HK_FORCE_INLINE hkReal getHeightAt( int x, int z ) const
      {
          return 0.0f; // return your height here...
      }

      // Assuming each height field quad is defined as four points { 00, 01, 11, 10 },
      // this should return true if the two triangles share the edge p00-p11.
      // Otherwise it should return false if the triangles share the edge p01-p10
      HK_FORCE_INLINE bool getTriangleFlip() const
      {
          return false;
      }

      // Forward to collideSpheres Implementation.
      virtual void collideSpheres( const CollideSpheresInput& input, SphereCollisionOutput* outputArray)
        const
      {
          return hkSampledHeightFieldShape_collideSpheres(*this, input, outputArray);
      }
};
```

`hkpSampledHeightFieldShape_collideSpheres()` is provided by Havok and is templated on the type passed in its first argument. This allows the `collideSpheres()` method for the height field to acquire the height at a given (X, Z) coordinate with a quick inline call.

The `getTriangleFlip()` method is to allow for differences in the tessellation of the quads in the height field. If the quad is defined as 4 points, the triangles can share either of the two opposing sets of vertices. The collision detection system is informed of the order of the vertex data via this call, so that the edges are aligned along the correct diagonal.

### Optional Bounding Volume Hierarchy

A sampled height field supports optional hierarchical bounding volume information in the form of a coarse

min-max quadtree. This is used to speed up raycasts and can make a considerable difference for large height fields.

The simplest way to enable it is to call `buildCoarseMinMaxTree()`. The *coarseness* parameter determines how detailed the quadtree will be. Lower coarseness means it will use more storage and get better performance. The side length of the finest level in the quadtree will be $2^{coarseness}$. For example, if coarseness is 3 (the default) it will store two floats (min and max) for every 8 x 8 vertices at the finest level. The additional storage required for the higher levels of the quadtree adds up to approximately a third of the finest level.

A user implementation of `getCoarseMinMaxImpl()` can be used to control the storage of the min-max quadtree. In that case you need to call `setMinMaxTreeCoarseness()` to set the supported coarseness.

The default behavior is to decide at run-time to use the coarse min-max tree method if tree data has been built, and the legacy method otherwise. However, to reduce code size, any of these two methods can be stripped off (by calling the appropriate register function on SPU, or with the feature exclusion mechanism on other platforms). If you keep only the min-max tree algorithm you have to make sure that the coarse tree data is built for any sampled height field that could be queried.

**Triangulated Sampled Height Field**

To make use of the memory performance of height fields without the approximations that height fields entail in collision detection, sampled height fields must be simulated as triangle meshes. This can be accomplished with just a few extra lines of code:

```
SampledHeightFieldShape* heightFieldShape;

// Create your height field here as normal...

// Wrap the height field in an hkpTriSampledHeightFieldCollection:
hkpTriSampledHeightFieldCollection* collection = new hkpTriSampledHeightFieldCollection( heightFieldShape
  );

// Now wrap the hkpTriSampledHeightFieldCollection in an hkpTriSampledHeightFieldBvTreeShape
hkpTriSampledHeightFieldBvTreeShape* bvTree = new hkpTriSampledHeightFieldBvTreeShape( collection );
```

Internally, the height field is being "wrapped" in a special shape collection and a BV Tree shape. These two shapes work together to present the height field to the collision detector as a bounding volume tree together with a triangle mesh. When an object collides with the height field, the bounding volume shape (`hkpTriSampledHeightFieldBvTreeShape`) will extract all triangle IDs that correspond to the (X, Z) coordinates at the area of collision. The shape collection (`hkpTriSampledHeightFieldCollection`) internally converts these IDs into triangles. The collision detector then uses these triangles for detection with the colliding object. A demo highlighting the differences between the regular sampled height field and the triangulated sampled height field can be found in "`Physics2012/Api/Collide/Shapes/HeightField/TriSampledHeightField`".

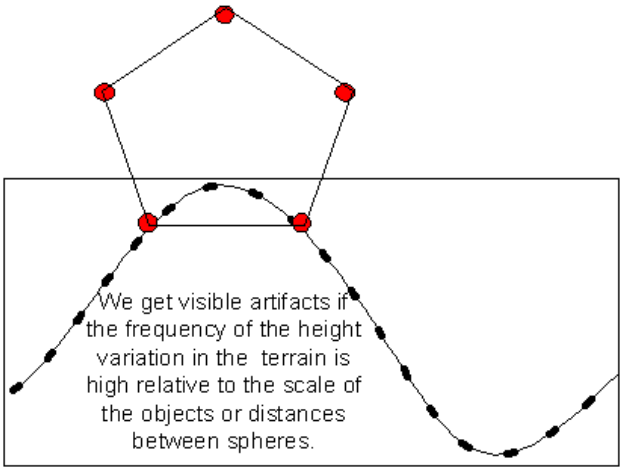
There is one very important performance caveat to be aware of when using triangulated sampled height fields. Performance will likely be prohibitively slow when using long linear shape casts across the landscape. The triangulated sampled height field does not use the same optimized linear cast algorithm as the regular sampled height field. Instead, it builds a 2-dimensional AABB around the linear cast and pulls out all triangles intersecting this AABB.

This does not cause a performance problem for short linear casts (using a character controller on a

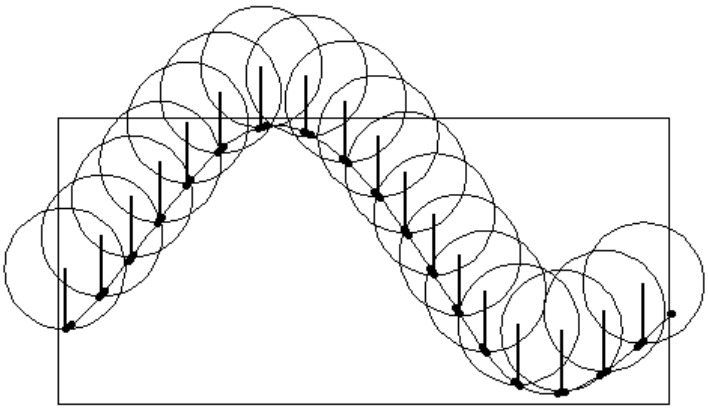
triangulated sampled height field is fine). Raycasting also does not cause a performance problem because the triangulated sampled height field uses the same raycast as the regular height field implementaion.

### Height Field Approximations

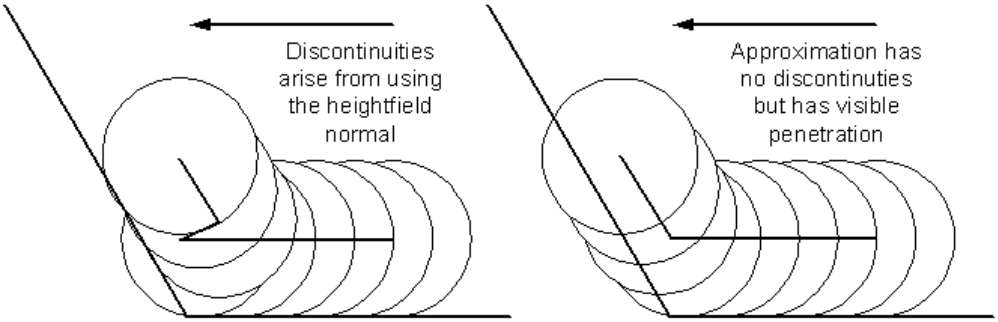
Shapes colliding with the landscapes are approximated using their sphere representation. This simplification of the incoming shape obviously does not take into account edge or face collisions. As the rate of change in the height of the landscape increases, so does the number and scale of collision artifacts. With sheer drops, it is important to note that although objects collide correctly with the cliff face, they will not collide properly at the cliff edge because of the lack of edge/edge collisions. This simplification affects collisions with all shape types (as they are all approximated with spheres).



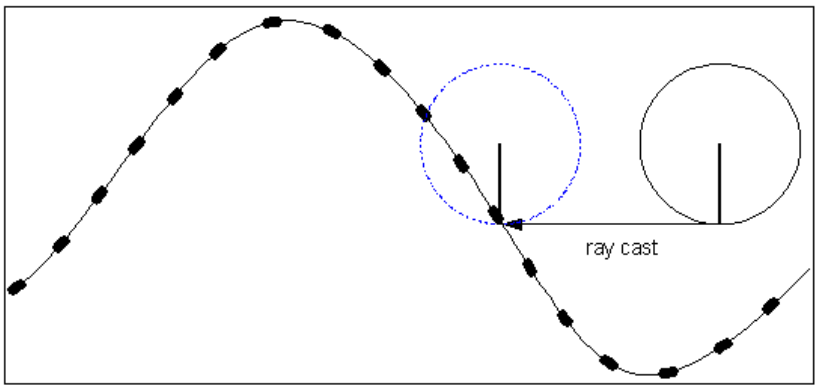

In the `hkpSampledHeightFieldShape::collideSpheres()` implementation, the center of each incoming sphere is projected down along the local y axis to compute the required distance. It is then adjusted back along the Y axis by the radius of the sphere. This adjustment leads to visual artifacts as the slope of the height field or the radius of the sphere increases. Taking the landscape normal into account when making the adjustment would only lead to more serious discontinuities in the height field.



Sphere collision approximation




This approximation is generally only an issue for sphere and capsule shapes (which have a sphere representation of 2 spheres). If this approximation is unacceptable, custom sphere and capsule shapes can be manually implemented to return more spheres in their `hkpSphereRepShape` interface implementation.



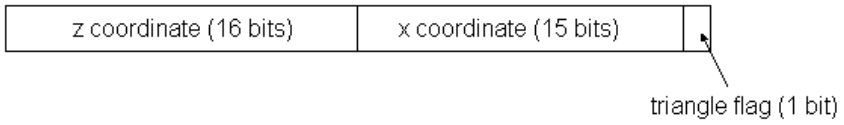

Sphere casting

When performing linear casting, the height field agent casts the spheres for the incoming shape against the height field. It does this through the `hkpHeightfieldShape::castSphere()` method. In the Havok `hkpSampledHeightfieldShape` implementation, this sphere casting is evaluated using the same approximation as above. The algorithm drops along the Y axis and performs a raycast. This obviously suffers from the same artifacts as the above approximation and is probably most visible when using capsule characters with height fields that have steep slopes. Examine the `HeightFieldRayVsLinearCastDemo` to better understand this issue.

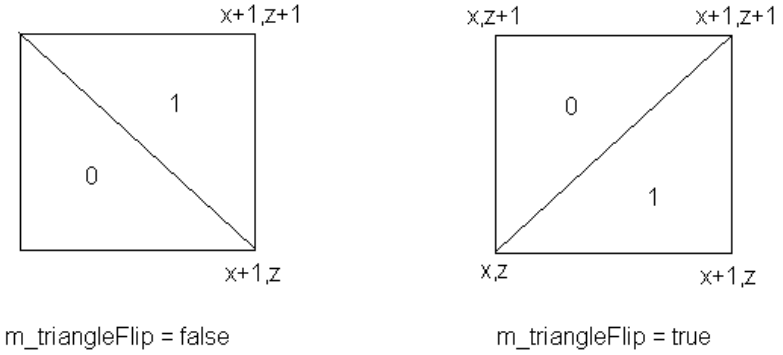
### Height Fields, Shape Keys and getTriangleFlip()

The triangles of the height field map to 32-bit shape keys. This mapping is used in the normal height field. When a ray cast is performed against the height field, it will return a shape key corresponding to the "triangle" of the height field hit by the ray. With the triangulated sampled height field, this shape key is used more extensively: the `hkpTriangleShape` corresponding to the triangle in question can be retrieved (using the standard `hkpShapeCollection` interface).

The 32-bit shape key for a given triangle is broken into 3 components: the X coordinate of the quad containing the triangle, the Z coordinate of the quad containing the triangle, and a 1-bit flag representing one of the two triangles in the quad. The lowest bit is used for the triangle flag. The next 15 bits are used for the X coordinate, and the top 16 bits are used for the Z coordinate:

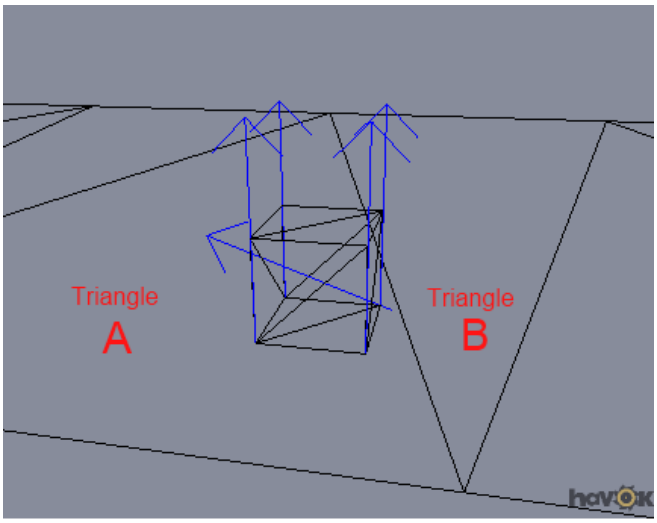



There are two ways the height field can be triangulated (shown below), depending on the value it returns from `hkpHeightfieldShape::getTriangleFlip()`. In each case, the triangles are marked with a "1" or a "0". This will be the value of the triangle flag corresponding to that triangle.




### Contact Point Welding

"Welding" is the term used in Havok Physics 2012 to address the problem of objects bouncing as new contact points are created while transitioning from a collision with one shape to a collision with a neighboring shape. This problem is illustrated by the following diagram:




This picture shows a cube sitting on a triangle mesh. It is in contact with one triangle (triangle A) with 4 contact points (the 4 arrows pointing up). However it is also deemed to be in contact with the triangle slightly to the right (triangle B) as it is within the collision tolerance (the distance within which a contact point is created between two objects). The contact point between the cube and triangle B is shown by the arrow pointing to the left. As the box slides from left to right across the mesh this contact point will cause the box to "hop" to triangle B, leading to unrealistic or unexpected behavior.

Our welding solution aims to address this problem. While a legacy solution is available for reference, our current solution is recommended as it is much faster, more robust and is supported on SPUs. It involves a pre-process step which stores extra information about each edge of every triangle in the mesh. This information is used at runtime to "snap" the incorrect collision normals to the desired direction (more information on snapping below).

This information adds an extra 2 bytes per triangle to the mesh's storage requirements. To aid this new algorithm the triangle winding order needs to be specified. For the best results when using welding, ensure the following conditions are met:

- Use the same winding order for all the triangles in a mesh.
- Avoid T-junctions and degenerate edges.

It is good practice to observe these rules for all meshes, whether or not they will be welded. While the Havok collision detection system does its best to work around inconsistent meshes, they can still occasionally lead to adverse results. Refer to the API demos in "`Physics2012/Api/Collide/Welding`" for more information.

If the welding algorithm finds pairs of triangles that share an edge (even partially) but have different winding, it will produce a warning message for each inconsistent winding pair. This test has been designed so as to produce as few false positives as possible. There is also an option in the content tools export filter to mark those triangles and edges (also disabling the welding at runtime) which you can later view in a special VDB viewer called "Inconsistent Winding Viewer".

Welding also has an impact when dealing with Character Rigid Bodies. These issues are explored in detail both in the character controller chapter, and in the API demo "`Physics2012/Api/Collide/Welding/WeldingIssues`".

### Mesh Welding Utility

Havok supplies a utility that computes this welding information for all Havok meshes currently available. First a MOPP (`hkpMoppBvTreeShape`) must be built around the mesh shape This is usually handled automatically in the tools pipeline, but it can also be created on the fly and discarded after the welding information is calculated. The following function can then be called, passing in the mesh, mopp and triangle winding order. Optionally, we can specify whether to weld open edges:

```
static void HK_CALL hkpMeshWeldingUtility::computeWeldingInfo( hkpShapeCollection* mesh, const
  hkpBvTreeShape* mopp,
                              hkpWeldingUtility::WeldingType weldingType, bool weldOpenEdges = true, bool
                                disableEdges = false  );
```

This welding method will only consider the specific mesh's structure when calculating the welding information. In many cases however, it is preferable to calculate welding information across a series of adjacent meshes.

Havok provides a function which will consider all adjacent shapes instead of just the single shape itself. The one shape function above is actually a wrapper for this generic function. Again, in most cases, taking into account adjacent shapes will lead to much better welding results. Please note that each shape considered needs a MOPP (`hkpMoppBvTreeShape`) built around them (again, usually handled in the tools pipeline).

```
static void HK_CALL hkpMeshWeldingUtility::computeWeldingInfoMultiShape(const hkTransform& meshTransform,
  hkpShapeCollection* mesh,
                              hkpWeldingUtility::WeldingType weldingType, hkArray< ShapeInfo >& shapes,
                                bool weldOpenEdges = true, bool disableEdges = false );
```

The `weldingType` parameter should be `WeldingType::WELDING_TYPE_ANTICLOCKWISE` if all triangles in your mesh when viewed from "outside" the mesh are wound in the anti-clockwise direction. This function builds the welding information and stores it in the mesh shape.

### Snapping Normals

The welding information computed from the mesh is used at runtime to modify the collision normals generated from objects hitting the welded mesh. The welding information stored is actually just the normal of the adjacent triangle for each edge of a triangle.

When an object collides with the mesh in a manner that generates contact points with two adjacent triangles in the same frame, the normal from the conflicting contact is adjusted to generate an appropriate response. It is "snapped" to the first triangle's normal.

In the example above, the "bad" contact normal generated when the box collides with the edge of Triangle B has to be adjusted. It is snapped to the normal of the triangle sharing the edge with triangle B (which is Triangle A). This adjusted normal will allow the box to continue sliding across triangle B correctly.

Here's another example to illustrate snapping. Consider the following two adjacent triangles A and B (sharing an edge) and a box, all viewed from the side. The gaps between the three objects are for illustration purposes only. The box is sliding up triangle A. The blue normal represents the current contact point normal generated on the face of triangle A.

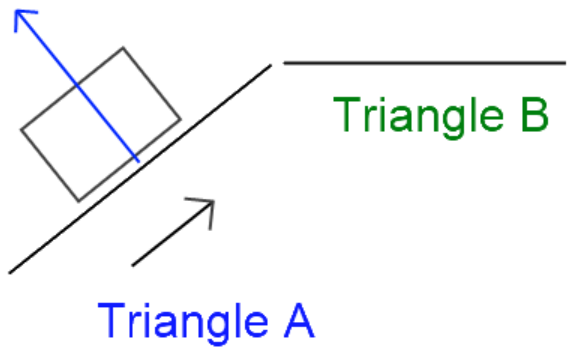



Figure 2.28: Box sliding up Triangle A

As the box continues to slide up triangle A, it will eventually come in contact with the edge of triangle B, generating the dotted green collision normal. The welding algorithm simply takes this normal and "snaps" it to the position represented by the solid green arrow. Again note that this normal corresponds to the normal of triangle A.

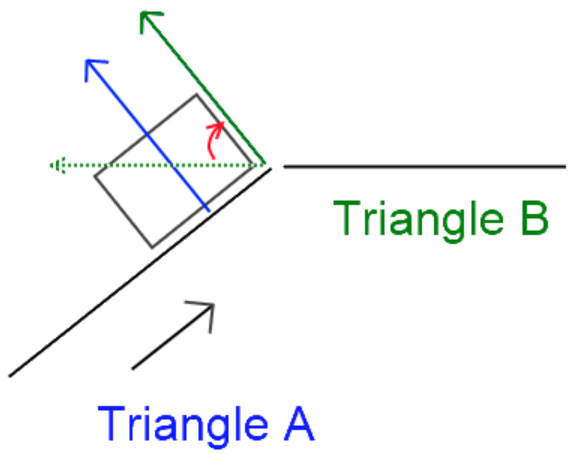



Figure 2.29: Box sliding up Triangle A and hitting edge of Triangle B

**Welding Viewer**

Havok provides a welding viewer (`hkpWeldingViewer`) to assist in the display of welding information in a scene.

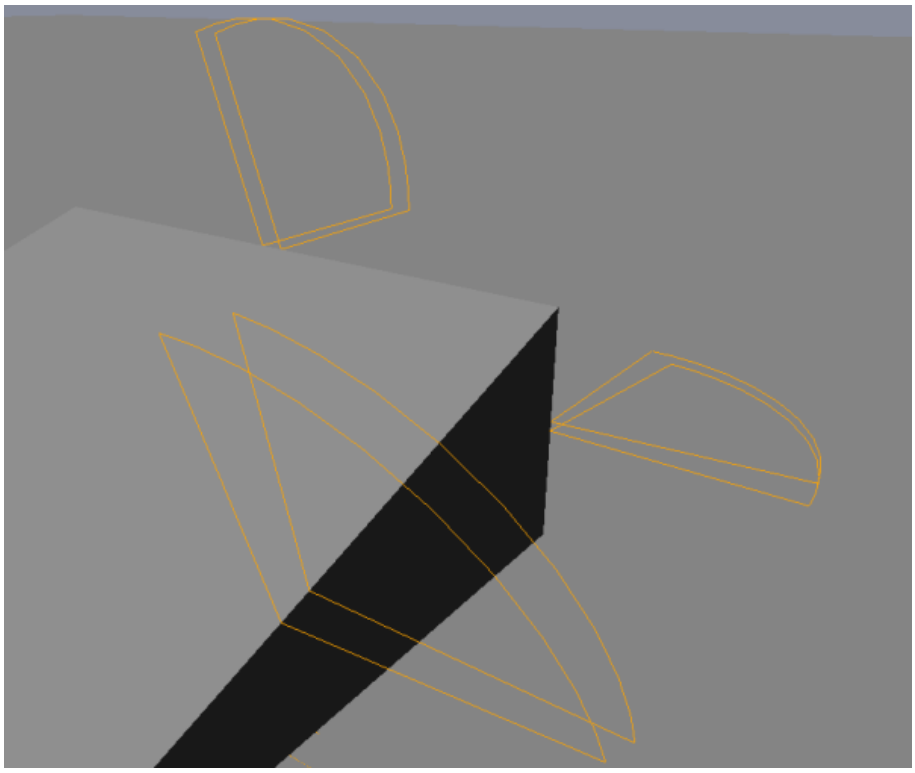


Figure 2.30: Welding Viewer in Action

As observed in the screenshot above, the welding viewer draws orange lines which represent the snap vector along with the triangle's normal.

**Dynamic Welding Solution (Deprecated)**

A deprecated version of welding is also available. As we recommend the use of the newer solution, we provide this information only for reference purposes.

In the deprecated solution, this welding problem is resolved by a run time analysis of all contact points between the convex shape (the box in out example above) and the mesh. Contact points with conflicting normals are evaluated, and a simple heuristic is used to reject the contact points deemed to be most incorrect.

To enable this legacy welding solution, the `hkpWorldCinfo::m_enableDeprecatedWelding` variable must be set to true before creating the simulation world.

Please note that this solution is less robust than our current solution as it may sometime cause objects to tunnel through landscapes (or other mesh shapes). While it works well with convex objects, it does not support SPU simulation nor does it handle TOI collisions.

**Collidables**

**hkpCollidable** is the core collision object in the collision detection system, linking the collision detection information (`hkpShape`) to the dynamics information (`hkMotionState`). While you will rarely need to create an `hkpCollidable` directly, you may still occasionally interact with them. This section gives a brief introduction to collidables.

A `hkpCollidable` primarily links the shape with the position and rotation of a dynamics object. While

hkpShape instances can be shared, each hkpCollidable is owned by exactly one hkpWorldObject (usually either an hkpEntity or an hkpPhantom). When a world object is created, a collidable is automatically created as a runtime link between the collision detection system and the dynamics system. This collidable forms a one-to-one link with the newly created world object. The important collidable member information here is spread across several classes within hkpCollidable, so for convenience they are listed here together. Please note that access to all these variables is still made through the hkpCollidable class interface:

- **Shape** : Defines the hkpShape for this collidable.
- **Motion State** : This member *usually* stores the hkMotionState information for this collidable (and is held in the parent class hkpCdBody). As Havok Dynamics is not 3-dimensional but 4-dimensional (the $4^{th}$ dimension being time), the hkMotionState also includes information about the movement of a rigid body between two physics frames. However, as the collision detector can be used asynchronously to collide two collidables, this variable can also hold an hkTransform (as we only need a position and rotation to test collisions between two shapes). Therefore the hkpCollidable can point to either an hkTransform or an hkMotionState (Note: the internal call to processCollision() expects an hkMotionState pointer).
- **Owner** : The owner pointer of the hkpCollidable will point to the hkpWorldObject it belongs to (which is the base class for all hkpRigidBody, hkpEntity and hkpPhantom objects).
- **Broadphase Handle** (of type hkpTypedBroadPhaseHandle): This member will be set when the collidable is used with the dynamics library. It essentially means that the collidable can be added to the collision detection broadphase (and therefore simulated in the by Havok Dynamics). It in turn holds several other key member variables:
  - **Type** : Indicates whether the collidable is owned by an rigid body or a phantom.
  - **Collision Filter Info** : Specifies filtering information for the object.
  - **Collidable Quality Type** (of type hkpCollidableQualityType): Defines how much processing time we wish to invest to stop this object from penetrating other objects. See the chapter on continuous physics for more information on this.
- **Allowed Penetration Depth** : Defines how deeply this object is allowed to penetrate other objects before continuous collision detection is applied.

### Retrieving the World Object

Collidables (and their parent hkpCdBody class) will often be encountered during callback events or when dealing with collectors. From these results, we usually want to access the owning object of this collidable. To retrieve the owner of a particular hkpCollidable, we can query the hkpCollidable type (with hkpCollidable::getType()). This is either set to hkpWorldObject::BROAD_PHASE_ENTITY (indicating an hkpRigidBody) or hkpWorldObject::BROAD_PHASE_PHANTOM (indicating an hkpPhantom). Then we can safely cast the void* returned from hkpCollidable::getOwner() method. Two useful helper functions are provided to to this: hkpGetRigidBody(hkpCollidable*) and hkpGetPhantom(hkpCollidable*). They will check the type and return a pointer to the corresponding hkpRigidBody or hkpPhantom. If the type does not match they return HK_NULL.

These helper methods become quite useful when dealing with collectors. Collectors return a list of hkpCdBody (the base class of hkpCollidable). We can quickly access the individual sub-shape hit (assuming we're dealing with a compound body) using hkpCdBody::getShape(). To access the root shape however, we need to first retrieve the root collidable (using hkpCdBody::getRootCollidable()). Once we have the root collidable, we can get access to the rigid body (using the hkpGetRigidBody() helper method) or the root shape (using hkpCollidable::getShape()).

**Manually Creating a Collidable**

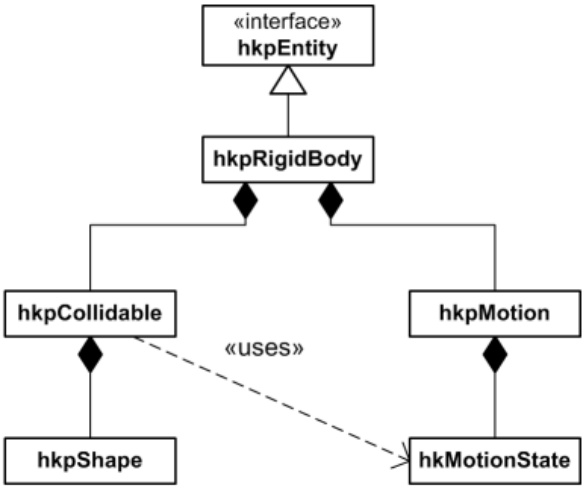
As mentioned above, in most cases collidables will be automatically generated as an embedded class in the `hkpWorldObject` class. However, there are certain situations where we may wish to manually create an collidable. A good example is when we would like to directly collide two objects outside of the dynamics system. This can be accomplished by creating two collidables using the shapes you wish to collide. Using the collision dispatcher (`hkpCollisionDispatcher`), an appropriate collision agent (`hkpCollisionAgent`) can be accessed to dispatch the collision query manually. All of this can be accomplished without ever accessing the dynamics library (and therefore without having to create rigid bodies for simulation).

A collidable can also be created to run a custom query (a linear cast for example) on the current simulation world. Again, only a shape and a transform need to be provided to create the collidable. There is no need to create a rigid body and add it to the world. Please refer to the Interacting with the Simulation chapter for more examples and details.

#### 2.2.1.2 Rigid Bodies

Any real-world object that doesn't change its shape–from a car chassis to a boulder hurtling down a mountainside–can be simulated in Havok Physics 2012 as a *rigid body*. Rigid bodies are central to the Havok Physics 2012 simulation system.

Rigid bodies are modeled using the **hkpRigidBody** class, a subclass of `hkpEntity`.




`hkpRigidBody` has an `hkpMotion` that stores all the information relating to how the rigid body moves, including its mass and velocity. It also has an `hkpCollidable` member that contains the information needed for the body to work with the collision detection system: an `hkpShape` and a pointer to the `hkMotionState` (which is owned by the `hkpMotion`). This motion state pointer points to a structure holding the position and orientation of the rigid body, thus linking the collision detection system to the dynamics system.

**Note:**
The collision detection system is designed to work without the dynamics library, hence the apparent level of indirection (the `hkMotionState` for instance) that exists between the collidable and the rigid body that owns it.

A `hkpRigidBody` is created using a construction info structure, the **hkpRigidBodyCinfo** . Here is a basic example of creating a simple rigid body:

```
// Sphere of radius 2.
hkpSphereShape* sphere = new hkpSphereShape(2.0f);

hkpRigidBodyCinfo rigidBodyInfo;
rigidBodyInfo.m_shape = pShape;

hkpRigidBody* pRigidBody = new hkpRigidBody(rigidBodyInfo);
```

There are many configuration parameters for a rigid body, which can either be setup during construction or edited directly on the rigid body after creation. The most common ones are explored next. Alternatively, you can skip ahead to the next step: adding a body to the simulation world. For a complete list and explanation, refer to the reference manual entry or the header file for `hkpRigidBodyCinfo`.

### Basic Properties

These are the basic properties of a rigid body, controlling its shape, mass and position in the simulation world.

- **hkpRigidBodyCinfo::m_motionType**

  This parameter specifies whether the `hkpRigidBody` is fixed, movable with an inertia type, or keyframed. This defaults to `hkpMotion::MOTION_DYNAMIC`. Refer to `hkpMotion::MotionType` for a full list of motion types and their implications. More information on keyframed rigid bodies is available in the keyframed objects section.

  The motion type of a rigid body can be changed at run time (using `hkpRigidBody::setMotionType()`) in most situations. If an `hkpRigidBody` was created as fixed or keyframed, it cannot be changed to dynamic later during the simulation. However, a dynamic `hkpRigidBody` can be set to fixed or keyframed during simulation and switched back.

- **hkpRigidBodyCinfo::m_position**

  The initial position of the `hkpRigidBody`, specified as an `hkVector4` in world space. By default, this is set to the origin.

- **hkpRigidBodyCinfo::m_rotation**

  The initial rotation of the `hkpRigidBody`, specified as an `hkQuaternion`. By default, this is set to the unit quaternion (the identity rotation).

- **hkpRigidBodyCinfo::m_friction**

  The initial friction value for the `hkpRigidBody`. A body's friction value indicates how smooth its surface is and hence how easily it will slide along other bodies. Friction values generally range between 0 and 1, but can be higher (a maximum of 255). The default value is 0.5. The effects of changing friction values are demonstrated in the `FrictionChangeDemo`.

- **hkpRigidBodyCinfo::m_restitution**

  The initial restitution value for the `hkpRigidBody`. This indicates how "bouncy" the object is–in other words, how much energy it has after colliding with another object. A value of 1 means that the object gets all its energy back after a collision, a value of 0 means that it will stop moving completely. The default value is 0.4. The implementation of the restitution is only a rough approximation, therefore users should experiment with different values to get the desired effect.

Note that there is a hard limit of 1.99 for this parameter. The effects of changing restitution values can be observed in the `RestitutionDemo`.

To replicate the classic example of a ball bouncing off the floor forever, set both the ball and the floor's restitution to 1. Of course, one has to make sure to also remove any linear velocity damping to obtain this exact behavior. For more information on the internal implementation of the restitution, refer to the documentation for `hkContactPointMaterial::setRestitution()`.

### Mass Properties

The following properties manage the motion behavior of the rigid body:

- **hkpRigidBodyCinfo::m_mass**

  The mass of the `hkpRigidBody`, which defaults to 1. This value is specified in kilograms if the world scale being used is meters (the recommended default). A mass of zero implies that the `hkpRigidBody` is fixed and will use a fixed motion type. The distribution of this mass within the body defines how it rotates; this behavior is encoded in the body's inertia tensor.

- **hkpRigidBodyCinfo::m_centerOfMass**

  This variable specifies the center of mass (COM) of the `hkpRigidBody`, in the body's local space. By default, it is set to the local space origin. For boxes and spheres, this is the center of the shape. For other shape types, the COM depends on how the shapes are set up. Refer to the shapes chapter for more details. Note that changing an `hkpRigidBody`'s center of mass does *not* change the position of the `hkpRigidBody`. It is also possible to set this to a point outside of the body.

- **hkpRigidBodyCinfo::m_inertiaTensor**

  The inertia tensor of the `hkpRigidBody`, specified as an `hkMatrix3`. An inertia tensor describes how difficult it is to rotate an object around any given axis. In practice, it's reasonable to assume that most non-fixed objects in Havok will use a box inertia tensor. The inertia tensor used is determined by the rigid body motion type described below. `hkpInertiaTensorComputer` is a helper class which uses shape and body information to calculate an appropriate inertia tensor value. An example is illustrated below:

```
hkpRigidBodyCinfo info;
hkpMassProperties massProperties;

hkVector4 boxSize ( 1.0f, 2.0f, 3.0f );
hkReal boxMass = 10.0f;

hkpInertiaTensorComputer::computeBoxVolumeMassProperties(boxSize, boxMass, massProperties);
info.m_inertiaTensor = massProperties.m_inertiaTensor;
```

  Note that the method shown is for a box. Different methods are provided to calculate the inertia tensor for different types of object. The `hkpInertiaTensorComputer::setShapeVolumeMassProperties()` is a convenient method which handles all shapes.

  Note that depending on the motion type, this inertia matrix gets reduced to:

  - `hkpMotion::MOTION_SPHERE_INERTIA`: The highest value of the inertia diagonal is used, and all outer diagonals are set to 0. This simplification is valid for quite a few objects like spheres or equally sized boxes.

- `hkpMotion::MOTION_BOX_INERTIA`: The diagonal is used and all outer diagonal elements are discarded. This simplification is valid for all symmetric objects. As Havok's solver does not support full inertia, make sure that all your objects are aligned to the local coordinate space.

### Collision Properties

These properties control the behavior of the rigid body during collisions:

- **hkpRigidBodyCinfo::m_shape**

  As explored in the previous chapter, `hkpShape` defines an object's collision detection volume, such as simple shapes (a box or sphere), a compound shape made up of simpler `hkpShape`s, etc. More information about the different shape types and how to create them is available in the Shapes chapter.

- **hkpRigidBodyCinfo::m_collisionFilterInfo**

  This value can be used by collision filters to identify the entity and filter collisions between certain entities. By default, this is set to zero. More information is available in the collision filtering section of the Interacting with the Simulation chapter.

- **hkpRigidBodyCinfo::m_qualityType**

  The collision quality type of the `hkpRigidBody` determines how the body interacts with other objects during collisions, for example whether the body should do "continuous" collision detection to prevent tunneling through other bodies. The quality type, and its impact on collisions is explored in extensive detail in the Continuous Physics chapter. The different types available are enumerated in the object quality types section.

- **hkpRigidBodyCinfo::m_allowedPenetrationDepth**

  This variable defines how deeply a high quality object is allowed to penetrate in normal situations. However, the engine is allowed to breach this value to solve critical situations. The value should be small for small objects (about 5% to 10% of the object's diameter) and can be larger if quality is not a primary issue. If you reduce this value, you will get a higher quality at the expense of some extra CPU. The default value is 0.05 (metres) and works well for objects as small as 20 cm. The topic of allowed penetration depth is discussed in extensive detail in the Optimization chapter.

### Damping and Clamping Properties

The rigid body also exposes several properties to control the damping and clamping of velocities:

- **hkpRigidBodyCinfo::m_linearDamping, hkpRigidBodyCinfo::m_angularDamping**

  These specify the initial damping used for linear and angular movement, respectively. The default linear damping value is zero while the default angular damping value is 0.05. Linear damping reduces the linear movement of the object over time, and angular damping performs the same to the rotation of the object. For example, if the linear damping is set to 0.1f, then every second approximately 10% of the object's linear velocity will be removed (or to be precise: `newVelocity = oldVelocity * exp( -damping*deltaTime)`),

- **hkpRigidBodyCinfo::m_maxLinearVelocity, hkpRigidBodyCinfo::m_maxAngularVelocity**

  Set hard limits for the `hkpRigidBody`'s linear and angular velocities. The default linear velocity limit is set to 200 m/s. If a higher value is needed (for fast projectiles) consider setting `hkpWorldCinfo::m_expectedMaxLinearVelocity` as well. The default angular velocity limit is set 200 radians/s (roughly 32 rotations/second).

There is another clipping mechanism in place which limits the angular velocity of bodies to ∼170 degrees per simulation frame. Those values are actually hard internal integrator limits. A rigid body will never exceed this value.

These parameters should only be used as a safety net to stop the physics from getting crazy. If you wish to limit the velocity of an object for gameplay reasons (say a box slowly falling down due to an attached parachute) then you should use either damping or write a custom action.

**Note:**

Setting the maximum velocity too low can lead to strange effects. Consider for example that the maximum velocity of a box is set to 10 meters/second. If a fast car hits this box, the car won't be able to push the box faster than this 10 meters/second. However, the Havok solver does not know about this limit as these are handled in the integrator. The solver will not work as expected and the box will quickly sink into the car. If continuous physics is enabled between the car and the box, extra CPU resources will be invested in TOI resolving. However the integrator will simply undo the work of the TOI solver.

### User Properties

`hkpRigidBody` provides a user data member which may be used for any purpose, for example to point to a game entity which owns the body. It is also possible to set any number of additional generic user properties on a rigid body (or any object inheriting from `hkpWorldObject`, including phantoms). This done using the `hkpProperty` class. Please refer to its documentation for further details.

### Example

The following snippet shows a slightly more detailed example with a few more configuration parameters. We create a convex `hkpRigidBody` with a mass of 10 and we compute an appropriate inertia tensor for its shape and size. The default values are used for the remaining `hkpRigidBodyCinfo` parameters:

```
int numVertices = 4;

float vertices[] = {
  -2.0f, 2.0f, 1.0f, 0.0f, // v0
   1.0f, 3.0f, 0.0f, 0.0f, // v1
   0.0f, 1.0f, 3.0f, 0.0f, // v2
   1.0f, 0.0f, 0.0f, 0.0f  // v3
};

hkpConvexVerticesShape* pShape = new hkpConvexVerticesShape( hkStridedVertices(vertices, numVertices));

hkpRigidBodyCinfo rigidBodyInfo;
rigidBodyInfo.m_shape = pShape;

// Compute the shapes inertia tensor
hkReal mass = 10.0f;
hkpInertiaTensorComputer::setShapeVolumeMassProperties(pShape, mass, rigidBodyInfo);

// Basic motion and quality types.
rigidBodyInfo.m_motionType = hkpMotion::MOTION_BOX_INERTIA;
rigidBodyInfo.m_qualityType = HK_COLLIDABLE_QUALITY_MOVING;

hkpRigidBody* pRigidBody = new hkpRigidBody(rigidBodyInfo);
```

#### 2.2.1.3 Constraints

Constraints are essentially restrictions on the freedom of movement of objects, allowing you to easily create complex systems such as hinges, body joints, and wheels. Whatever the type of constraint, these elements can help to make the dynamic environment very rich.

This section first introduces you to constraint spaces and constraint construction, and then outlines the different types of constraint that are supported by Havok and shows you how to create and configure them. Some guidelines are also provided for the use and limitations of constraints.

In addition to reading this section, you should take time to look at and play with the demos in the dynamics/constraints folder provided with your Havok Physics 2012 installation, examples of which are used throughout the text. Each of these simple demos illustrates a particular aspect of working with Havok constraints.

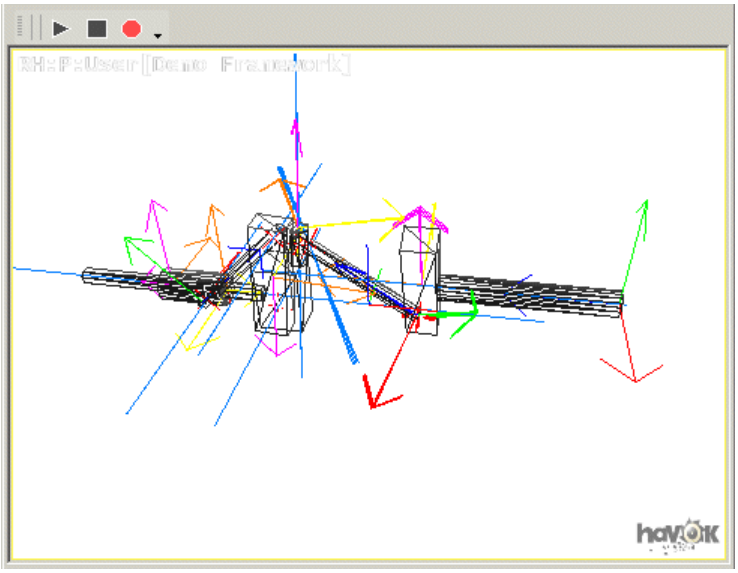



Figure 2.31: Constraints Abound

**Introduction**

**Constraint Spaces**

In rigid body dynamics each body has 6 degrees of freedom:

- 3 translational degrees of freedom.
- 3 rotational degrees of freedom.

Each type of Havok constraint can remove or limit one or more of these degrees of freedom for its constrained bodies.

Different types of constrants are produced depending on the number and type of these limitations: from the simple point-to-point constraint to the more complex ragdoll constraint. For example, in a point-to-point constraint, the constrained objects are completely free to rotate around the constraint pivot point, but have no linear freedom relative to each other in any direction–they are attached together at

the point. In contrast, with a ragdoll constraint, the objects not only have no linear freedom, but the possible relative orientation is also restricted (imagine the rotational limits placed on a shoulder joint ).

In each constraint, these angular and linear limits are defined in terms of the constraint's coordinate system or *constraint space*. Because a constraint restricts the movement of its objects relative to each other, a constraint also maintains a transform from each object's local space to the constraint space.

Each constraint stores the minimum amount of information it needs to construct this transform. For example in a point-to-point constraint, because there are no angular limits, only the pivot point is specified in local space for each object; the objects' orientation is unimportant.

### Constraint Scenarios

Constraints which limit object movement in different ways can result in a number of useful applications. Consider some of these examples:

| | |
|---|---|
| Swinging sign, door, elbow joint, spinning wheel | Limited hinge constraint / Hinge constraint |
| Elevator, mechanical piston | Linear movement limited to a single axis (Prismatic constraint) |
| Shoulder joint, caravan hitch joint | Ragdoll constraint |
| Bell clanger, neck joint | Limited ball-and-socket joint |
| Car wheel | Wheel constraint |

Table 2.5: Examples of Constrained Systems

More complex custom constraint setups can be designed and implemented using the constraint construction kit or constraint atoms. However, the more complex the constraint, i.e. the more degrees of freedom it limits of constraints, the more expensive it is to compute.

### Creating Constraints

Each Havok constraint allows you to specify the rigid body or bodies to be constrained, and the necessary axes and pivot points, in either world space or the objects' local spaces. Some constraints also allow you to specify additional parameters, such as linear or angular limits and friction values.

Specifying axes and pivot points in world space is a quick and simple approach to constraint setup. These are converted to the local space of both objects. However, this approach works only if the objects are already correctly positioned relative to each other in world space. If this is not necessarily the case, you should use the more complex local space setup. You can see an example of this in the section on Ragdoll constraints.

When you create a constraint in world space, the constraint's `isValid()` method is automatically called. This checks that you have provided valid setup information for the constraint (such as unit vectors that are actually unit vectors, perpendicular vectors that are actually perpendicular, and reasonable linear and angular limits). The implementation of this method depends on the constraint type. If the information you provide is invalid, Havok will assert in debug builds.

### Constraint Data, Instance and Runtime

The Havok constraint system encourages data sharing whenever possible. Certain constraints can be pretty large. If you want to simulate 50 ragdolls, it makes sense to share common data between all the ragdoll constraints. Therefore, constraints are split into 3 distinct classes:

- `hkpConstraintData`

  This is the shared data between the constraints. Actually, this class is `hkpConstraintData`

simply an abstract class and implemented by all constraint implementations like `hkpBallAndSocketConstraintData` or `hkpRagdollConstraintData`. This data can be used several times, however all instances using this shared data will be dependent upon it. That means **if you change the values in the data, it affects all instances** . This is particularly important for motors.

- `hkpConstraintInstance`

  This class is used to create an instance of a constraint. An instance has pointers to two rigid bodies, a pointer to a shared `hkpConstraintData` and a priority. Note that this class is not abstract and is not subclassed.

- `hkpConstraintRuntime`

  Some constraints sometimes need run-time data for simulation. An example is the `hkpBreakableConstraintData`, which needs to get access to the impulses applied by its child constraint. This data is stored in the `hkpConstraintRuntime` memory. This runtime memory is allocated by the system when the constraint is added to the world. It is up to the constraint data implementation to decide how and when to allocate the runtime data:

  – The constraint data implementation can decide not to be shared and store the constraint data internally. In this case no external constraint runtime is allocated.
  – The constraint data implementation can decide that it always needs runtime information and always requests an external runtime information allocation.
  – The constraint data implementation can decide that it does not care about the runtime information. It checks the user flag `hkpConstraintInstance::m_wantRuntime` and allocates runtime data only if needed.

  The system tries to organize the runtime information of constraints in a cache friendly way. That means the system will move runtime information around in memory. So you should **never maintain a pointer to a hkpConstraintRuntime** .

Several helper methods are provided to access various information:

- From a `hkpConstraintData` class, you can call `hkpConstraintData::getType()` to retrieve the constraint type and then upcast the abstract interface to the correct implementation.
- The runtime information is accessed by calling `hkpConstraintInstance::getRuntime()`. The returned pointer must then be casted to the appropriate class:

```
// Example to get the detailed runtime information from a ball and socket constraint

hkpConstraintRuntime* rt = myInstance.getRuntime();
hkpBallAndSocketConstraintData::Runtime* runtime = hkpBallAndSocketConstraintData::getRuntime( rt );
hkReal impulseX = runtime->m_solverResults[hkpBallAndSocketConstraintData::SOLVER_RESULT_LIN_0];
hkReal impulseY = runtime->m_solverResults[hkpBallAndSocketConstraintData::SOLVER_RESULT_LIN_1];
hkReal impulseZ = runtime->m_solverResults[hkpBallAndSocketConstraintData::SOLVER_RESULT_LIN_2];
```

### Constraint Atoms

Constraint atoms (`hkpConstraintAtom`) are the basic building blocks used to create the `hkpConstraintData` for a constraint. While `hkpConstraintData` has an intuitive real-world interpretation like a hinge or wheel constraint, the `hkpConstraintAtom` encapsulates lower level mathematical information on how certain degrees of freedom are restricted.

Constraint atoms are used as they allow us to reduce code size by re-using the same types of atoms in different ways to form various types of `hkpConstraintData`.

Constraint atoms can also be used to craft custom constraints. More information on constraint atoms is available in the Understanding Constraint Atoms section.

### Constraint Limits

As you know, constraints can limit one or more degrees of freedom for their constrained bodies.

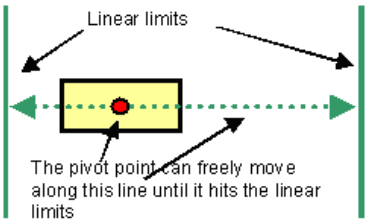



Figure 2.32: Linear limits along one axis

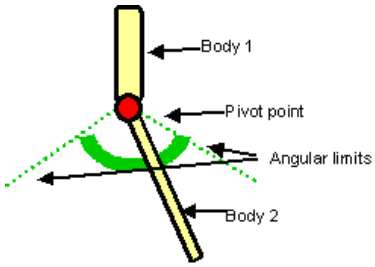



Figure 2.33: Angular limits around one axis

You can specify linear and angular limits for a number of Havok constraints. Specific examples are available in the sections on limited hinges, ragdoll constraints, and prismatic constraints.

A number of Havok constraints have friction limits for certain degrees of freedom (usually unconstrained). For example, when creating a limited hinge, you can specify a friction value for the constrained bodies' angular movement around the hinge axis. Other constraints with friction limits include prismatic constraints and point-to-path constraints.

### Updating Constraint Settings

Once you have created a constraint, you can update its data by calling the appropriate constraint class `set..()` methods. For debug builds, you can call `isValid()` yourself to ensure that the constraint still has valid information.

### Limiting Constraint Force

#### Basics

`hkpBallAndSocketConstraintData`, `hkpHingeConstraintData`, `hkpLimitedHingeConstraintData` and `hkpRagdollConstraintData` allow you to set the maximum linear impulse applied by their ball-and-socket component. To do this use `setMaxLinearImpulse()` and the solver will clip all applied impulses

to that specified limit. One should note that the limit is specified as impulse per solver step.

When a limit is breached a callback is fired to every `hkpContactImpulseLimitBreachedListener` attached to the `hkpWorld`. Havok also provides a default handler in the form of `hkpBreakOffPartsUtil`. If you use this utility, you must also call setBodyToNotify() on the constraints, so that the utility knows which body should receive the impulse-breached notification.

The `hkpBreakOffPartsUtil` converts breached constraints into the "breached contact point" representation so that they can be treated uniformly by the `hkpBreakOffPartsListener`. Please refer to the section on the break off listener for more details.

**Important Note:** The impulse applied by the solver while in the `METHOD_STABILIZED` mode is not clamped to the value set by `setMaxLinearImpulse()`, because in some cases, impulse clamping can add instability. For more details, please see the section on ball-and-socket stability below.

#### Details

Havok uses impulses and not forces to specify the strength of the constraint. This is intuitive for contact points, where the highest forces come from impacts. Impacts are also usually applied entirely within one solver step. For constraints, impulses are distributed over several solver steps and several frames. Therefore one should note that the maximum impulse assigned to constraints may need to be scaled and also may depend on the frame rate and solver settings.

When using hkpBreakOffPartsUtil, the maximum impulse is specified by calling the methods `hkpBreakOffPartsUtil::markEntityBreakable(entity, maxImpulse)` or `hkpBreakOffPartsUtil::markPieceBreakable(entity, childShapeKey, maxImpulse)`. You can also control the ratio between the limit for contacts and limits for non-contact constraint added later by calling `setMaxImpulseFactorForConstraints(entity, maxImpulseFactorForConstraints)`.

See BreakOffPartsDemo for further reference.

#### Advanced Constraint Parameters

The `hkpConstrainInstance::m_priority` determines whether a constraint should be considered in processed TOI events (PSI and TOI priorities). Higher-priority (`TOI_HIGHER`) constraints may be used to override normal-priority (PSI or TOI) when the two are in conflict. The topic of TOI events and priorities is explored in detail in the Continuous Simulation section.

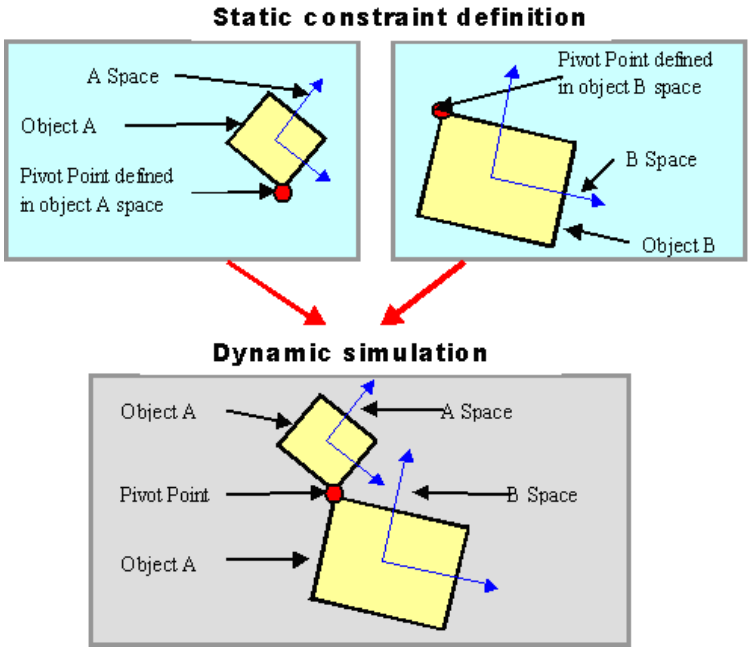
#### Constraint Types

#### hkpConstraintData

The constraint classes in Havok all inherit from the abstract base class `hkpConstraintData`. This includes a number of methods for accessing and modifiying basic properties, and updating the constraint.

#### Ball-and-Socket Constraints

A ball-and-socket constraint (also known as a point-to-point constraint) forces its objects to try to share a common point in space. The objects can move freely about this space, but always have a point in common. This point is the constraint's pivot point:




When creating the constraint, the pivot point has to be defined in the object space of each object involved. During runtime the constraint tries to apply forces to the objects so that the two pivot points defined by the two objects match. Using the `setInWorldSpace()` method, you can also define the pivot point in world space, which is then internally transformed into the body space of each object. Setting the pivot point in this manner works only if both objects are already positioned in world space in the way you want them to be constrained.

The following snippet shows how to use a ball-and-socket constraint to connect two objects by defining the pivot point in world space. When the simulation is run, the bodies spring together in order to try and share this common point in space, and will maintain this common point throughout simulation. The complete code for this example is available in the `BallAndSocketDemo`.

```
hkVector4 halfSize(0.5f, 0.5f, 0.5f);
hkVector4 size;
size.setMul4(2.0f, halfSize);
hkVector4 position(size(0), -size(1) -0.1f, 0);
hkpBallAndSocketConstraintData* bs;
{
  // Set the pivot
  hkVector4 pivot;
  pivot.setAdd4(position, halfSize);
  pivot(0) -= size(0);  // Move pivot to corner of cube

  // Create the constraint
  bs = new hkpBallAndSocketConstraintData();
  bs->setInWorldSpace(rb0, rb1, pivot);
  hkpConstraintInstance* instance = new hkpConstraintInstance(rb0, rb1, bs);
  m_world->addConstraint( instance );
  bs->removeReference();
  instance->removeReference();
}
```

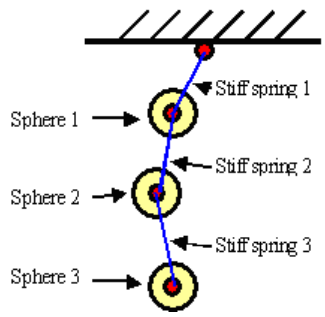
You can use a series of these constraints between adjacent bodies to form a chain.

**Important Note:** The ball-and-socket constraint can be simulated in a more stable manner by setting its solving method to `hkpConstraintAtom::METHOD_STABILIZED`. Please see the section below on ball-and-socket stability for more details about this.

### Stiff Spring Constraints

The stiff spring constraint is similar to the point-to-point constraint except for one important detail: it holds the constrained bodies apart at a specified distance, as if they were attached at each end of an invisible rod. A stiff spring requires a point in the body space of each of two bodies, and a constant distance by which these two points must be kept separated. Each body is free to rotate around its point.

You can easily create chains of objects using stiff springs, as in the diagram below:




You create a stiff spring between two objects using an `hkpStiffSpringConstraintData`, as in the code example below. The points can be specified in world space, as shown, or in the bodies' local space. By default, the stiff spring's length is the original distance between these points in world space. A different rest length can be set using the `setSpringLength()` function.

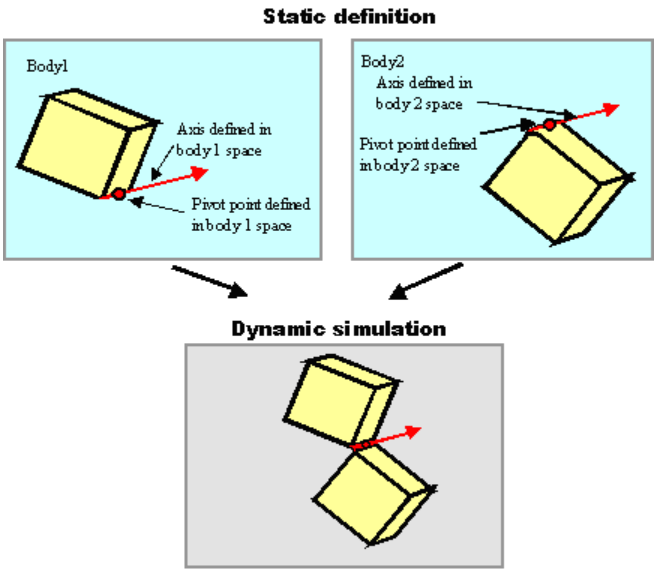
You can see the complete code for this example in the Stiff Spring demo.

```
hkpStiffSpringConstraintData* spring;
{
  // Create constraint
  spring = new hkpStiffSpringConstraintData();
  spring->setInWorldSpace(rb0->getTransform(), rb1->getTransform(), position1, position2);
  hkpConstraintInstance* instance = new hkpConstraintInstance(rb0, rb1, spring);
  m_world->addConstraint( instance );
  spring->removeReference();
  instance->removeReference();
}
```

You can use `hkpStiffSpringConstraintData` to create a bouncy spring by making it malleable. Refer to the section on malleable constraints for more details.

### Hinge Constraints

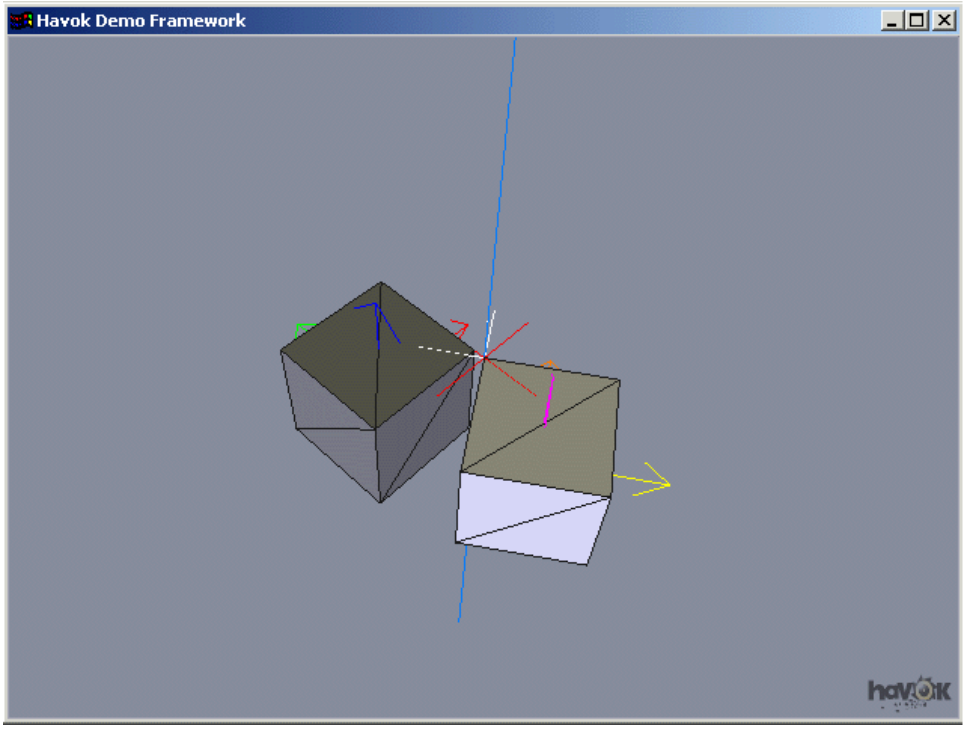
The hinge constraint allows you to simulate a hinge-like action between two bodies. You specify a line segment in body space for each body, with a position and a direction. The two lines then attempt to match position and direction, thereby creating an axis around which the two bodies can rotate.




The following snippet from HingeDemo illustrates how to constrain two cubes (one fixed and one movable) to be hinged together. To set up the constraint, you need to specify the point around which the hinge is created and the common axis of rotation through this point (aligned with the Z axis in our example). You can do this in world space using the `hkpHingeConstraintData::setInWorldSpace()` method, provided of course that the cubes are correctly positioned.

```
hkpHingeConstraintData* hc;
{
  // Set the pivot
  hkVector4 pivot;
  pivot.setAdd4(position, halfSize);
  pivot(0) -= size(0);  // Move pivot to corner of cube

  hkVector4 axis(0, 0, 1);

  // Create constraint
  hc = new hkpHingeConstraintData();
  hc->setInWorldSpace(rb0->getTransform(), rb1->getTransform(), pivot, axis);
  m_world->createAndAddConstraintInstance( rb0, rb1, hc )->removeReference();
  hc->removeReference();
}
```

When you run the demo, you can see the movable cube rotating around the hinge axis, shown here as the vertical line in the middle of the picture.




**Important Note:** The ball-and-socket part of the hinge constraint can be simulated in a more stable manner by setting its solving method to `hkpConstraintAtom::METHOD_STABILIZED`. Please see the section below on ball-and-socket stability for more details about this.

**Limited Hinge Constraints**

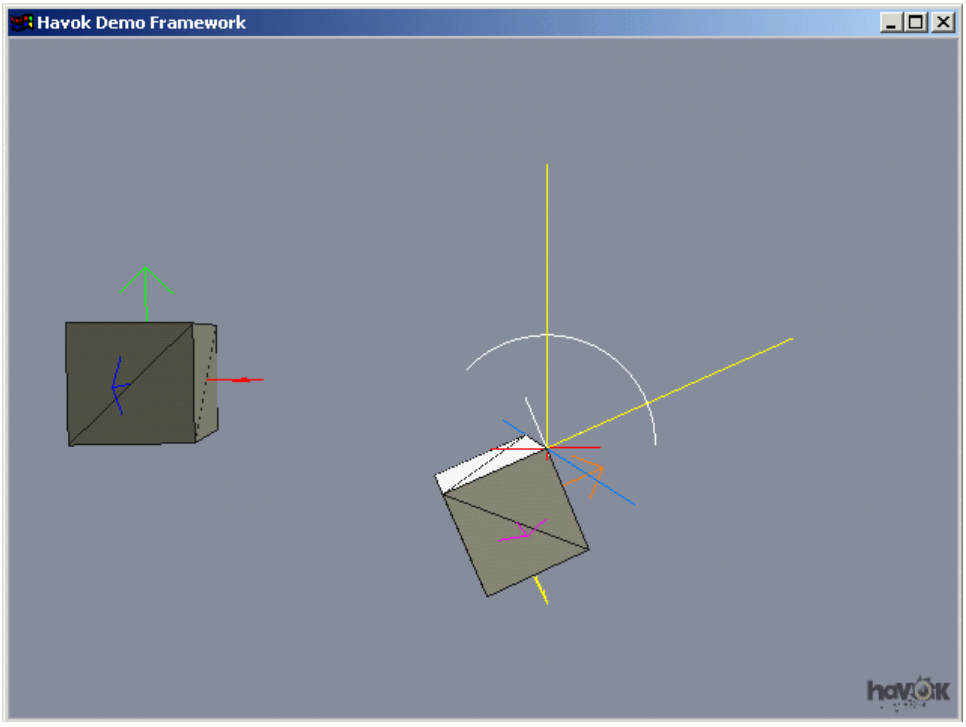
An `hkpLimitedHingeConstraintData` is similar to a hinge constraint, but also allows you to specify the maximum angle the hinged bodies can rotate. This can be useful for preventing interpenetration. Adding limits to a hinge requires another axis, perpendicular to the hinge's rotation axis. You specify the angular limits relative to this new axis, which is created automatically for you when you use the `setInWorldSpace()` function. The default limits are -PI and PI.

As mentioned previously, you can also provide a friction value for the constrained bodies' movement around the hinge axis. The default friction value is zero.

The following example, from LimitedHingeDemo, shows the creation of a limited hinge between two boxes (like the hinge example before), one box is movable while the other is fixed. The angular limits are specified using the `setMinAngularLimit()` and `setMaxAngularLimit()` methods. Note that the total range of movement can exceed 2*PI (a full rotation).

```
hkpLimitedHingeConstraintData* lhc;
{
  // Set the pivot
  hkVector4 pivot; pivot.setAdd4(position, halfSize);
  pivot(0) -= size(0);  // Move pivot to corner of cube

  hkVector4 axis(0.0f, 0.0f, 1.0f);

  // Create constraint
  lhc = new hkpLimitedHingeConstraintData();
  lhc->setInWorldSpace(rb0->getTransform(), rb1->getTransform(), pivot, axis);
  lhc->setMinAngularLimit(-HK_REAL_PI/4.0f);
  lhc->setMaxAngularLimit(HK_REAL_PI/2.0f);
  m_world->createAndAddConstraintInstance( rb0, rb1, lhc )->removeReference();
  lhc->removeReference();
}
```

You can see the angular limits in the following screenshot from the demo. The movable cube is free to rotate around the hinge axis within the limits indicated by the white arc.




**Important Note:** The ball-and-socket part of the limited hinge constraint can be simulated in a more stable manner by setting its solving method to `hkpConstraintAtom::METHOD_STABILIZED`. Please see the section below on ball-and-socket stability for more details about this.

### Fixed Constraints

A fixed constraint leaves no degree of freedom between the two constrained bodies. You can think of it as a ball-and-socket constraint that would permit no angular movement between the bodies. Thus the constraint will keep them in the same position relative to each other throughout the simulation, as if they were fixed together.

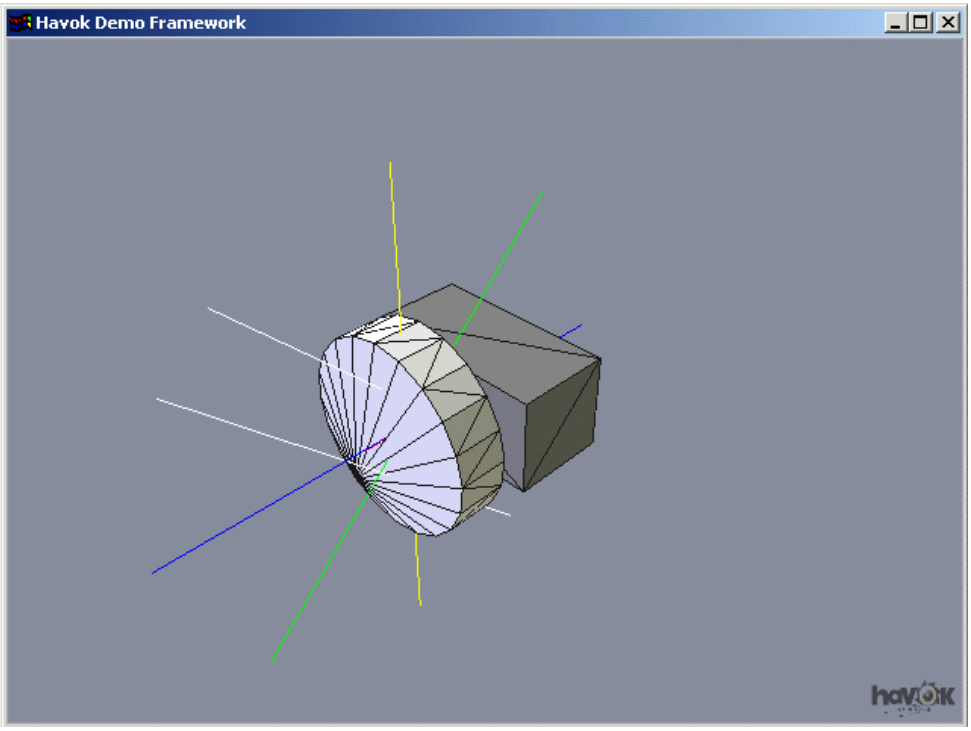
To create a fixed constraint you must specify a common pivot point in the two bodies' space (using `setInBodySpace()`) or give the pivot's absolute position and the two bodies' transform (`setInWorldSpace()`); the process is similar to creating a ball-and-socket constraint. Once constrained, the bodies' position and rotation relative to this pivot point will be kept constant.

**Wheel Constraints**

The wheel constraint is essentially a hinge with an extra free translation/suspension axis. This constraint allows you to attach a wheel to another object, for instance a car chassis. The rotation axis provides the axle of the wheel, allowing it to spin freely without limits. The suspension axis works relative to the chassis and allows movement along the direction of this axis. You can provide damping and stiffness values for the suspension, as well as limits for the movement of the wheel in the suspension direction, when creating the constraint.

The `hkpWheelConstraintData` lets you dynamically orient the rotation axis relative to the chassis to allow for steering, using the `setSteeringAngle()` function. To do this, the wheel constraint needs an additional steering axis, which you can use to specify this angle. In many cases, the suspension and the steering axis will be the same axis, although you can specify different axes when creating the constraint if desired.

These axes and limits are shown in the following screenshot from `WheelDemo`. The rotation axis is shown in blue, the steering axis in yellow, the suspension axis in green, and the suspension limits are shown as two white lines.




The snippet below (also from `WheelDemo`) illustrates how to create a wheel using a wheel constraint. As you can see, you can use the `hkpWheelConstraintData::setInWorldSpace()` function to specify the following:

- The two rigid bodies (first the wheel, then the chassis),

- The pivot point for the rotation axis, in world space.
- The rotation axis, in world space.
- The suspension axis, in world space.
- The steering axis, in world space.

You can also, of course, specify the point and axes in body space using `setInBodySpace()`. Note that if you use this function, you only need to set the pivot point and rotation axis in both spaces–the steering and suspension axes are specified in chassis space only.

The example also shows setting suspension limits, suspension stiffness, and suspension damping for the wheel.

```
hkVector4 suspension(1.0f, 1.0f, 0.0f);
suspension.normalize3();
hkVector4 steering(0.0f, 1.0f, 0.0f);
steering.normalize3();

m_wheelConstraint = new hkpWheelConstraintData();

m_wheelConstraint.setInWorldSpace( m_wheelRigidBody->getTransform(), m_chassis->getTransform(),
  m_wheelRigidBody->getPosition(), axle, suspension, steering );
m_wheelConstraint.setMaxSuspensionLimit( .2f );
m_wheelConstraint.setMinSuspensionLimit( -.5f );

m_wheelConstraint.setSuspensionStrength( 0.01f );
m_wheelConstraint.setSuspensionDamping( m_world->m_solverDamp * 0.25f );

m_world->createAndAddConstraintInstance( m_wheelRigidBody, m_chassis, m_wheelConstraint )->removeReference
  ();
```

During the game `setSteeringAngle()` is used to change the steering angle depending on input from the user. The following snippet shows the code to turn the wheel to the left.

```
const hkReal maxAngle = (HK_REAL_PI * .25f);
const hkgPad* pad = environment->m_gamePad;

if( (pad->getButtonState() & HKG_PAD_DPAD_LEFT) )
{
  m_steeringAngle += 1.25f * environment->m_deltaTime;
  // Clip steering angle to a max of PI/4 radians
  if( m_steeringAngle < maxAngle)
  {
      m_wheelConstraint->setSteeringAngle( m_steeringAngle );
  }
  else
  {
      m_steeringAngle = maxAngle;
  }
}
```

**Note:**
In general, the Havok Vehicle Kit is a better way to construct four wheeled vehicles than using wheel constraints, especially for racing or fast moving vehicles. Take a look at the Vehicle Physics chapter for more information.

### Pulley Constraints

Pulley constraint allows you to apply leverage between two dynamic objects. It represents a rope and pulley mechanisms where the pulleys have fixed positions in world, and the rope's ends are connected to dynamic bodies. By setting the leverage ratio you set the proportion of the forces applied by the rope onto the connected bodies.

To initialize a pulley constraint you need to call either `hkpPulleyConstraintData::setInBodySpace()` or `hkpPulleyConstraintData::setInWorldSpace()` passing the transforms of the linked bodies, the connection points where the rope is attached to the dynamic bodies, and the global positions of the fixed pulley blocks. Below is a simple setup code:

```
hkpPulleyConstraintData* pulley = new hkpPulleyConstraintData();
hkReal leverageRatio = 3.0f;

pulley->setInBodySpace(body0->getTransform(), body1->getTransform(),
                       bodyPivot0, bodyPivots1,
                       worldPivots0, worldPivots1,
                       leverageRatio);

hkpConstraintInstance* constraint = new hkpConstraintInstance(body0, body1, pulley);

world->addConstraint( constraint);
constraint->removeReference();
pulley->removeReference();
```

Both of the initializing functions calculate the rope length corresponding to the positions of the linked bodies and assuming the rope is fully stretched. The length of the rope and the leverage ratio can be modified later using the functions below:

```
pulley->setRopeLength(9.0f);
pulley->getLeverageOnBodyB(2.0f);
```

Given an input force on one side, a pulley mechanism applies that force multiplied by `leverageRatio` on the other side. However, one also has to apply `leverageRatio` times the displacement to move the other end by a requested distance. The current rope length is computed as the sum of the distance from the first body to the first pulley attachment point and the distance from the second body to the second attachment point multiplied by the leverage ratio.

The pulley gets unstable when one of the attached bodies gets very close to the pulley block. Note also that the pulley is not a fixed-distance constraint, therefore it does not restrict the bodies from coming too close to the pulley attachment points.

This constraint is different from most other constraint for two reasons:

- The pulley attachment points must be specified in the fixed global space, and cannot be linked to a moving body.

- The constraint applies different forces to each of the bodies.

### Ragdoll Constraints

Ragdoll constraints allow you to realistically simulate the behavior of body joints, such as hips, shoulders, and knees.

Once you have decided what degree of movement a joint should have, you can model it by specifying limiting values for the ragdoll constraint. This constraint also requires you to specify which of its rigid bodies is the reference body for the joint; this is used by the constraint as a reference when defining the limits for the motion of the attached body. For example, when you move your upper arm, your forearm always moves with it. So when modeling an elbow joint, you would specify that the upper arm was the reference body, making the forearm the attached body.

It is worth noting that ragdoll constraints (and limited hinge constraints) provide support for joint friction using a `setMaxFrictionTorque` function. The value passed to this function is not a parametric friction value between 0 and 1 but is dependent on the masses of the constrained bodies. To stiffen up a constraint try setting the friction high enough that the joint is completely stiff and then reduce the value until the desired behaviour is achieved.

The following sections will introduce you to the possible ragdoll constraint limits and explain how to set up a ragdoll constraint using the Havok Physics 2012 SDK.

### Ragdoll Constraint Limits

There are a number of limitations that need to be considered when dealing with ragdolls.

### Choosing the Parameters

An important part of creating a realistic ragdoll joint is choosing appropriate values for the various constraint limits. Before looking at the code, try stretching your right arm out with the palm facing downwards in such a way that if you turn your head 90 degrees to the right, you are looking straight along it. Imagine a pivot point at your shoulder. How would you model the freedom that your arm has to move using a ragdoll constraint?

First of all, you need to find the **twist axis** . This is easy in this case, as it is the axis running the length of your arm.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

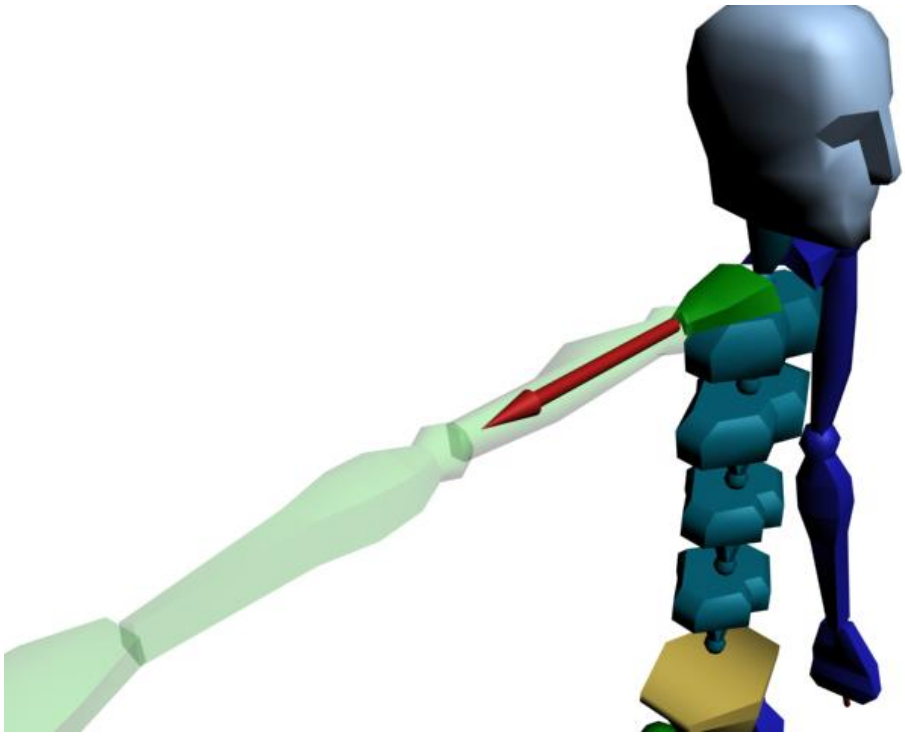


Figure 2.34: Twist Axis

Now with your torso immobile, see how far your upper arm can twist around this axis relative to your torso. You can probably twist your upper arm forwards by about 5 degrees and backwards by the same amount (your lower arm and wrist can twist further than this). The **twist limits** for your joint will need to reflect this.

Next you need to figure out your arm's available movement space. Again with your arm outstretched, make the widest arc that you can with your arm. Try to imagine a circular cone that includes all of the areas your arm reached. We will prune this cone later so it takes into account the areas you couldn't reach.

We specify the cone of movement using the **cone angle** .

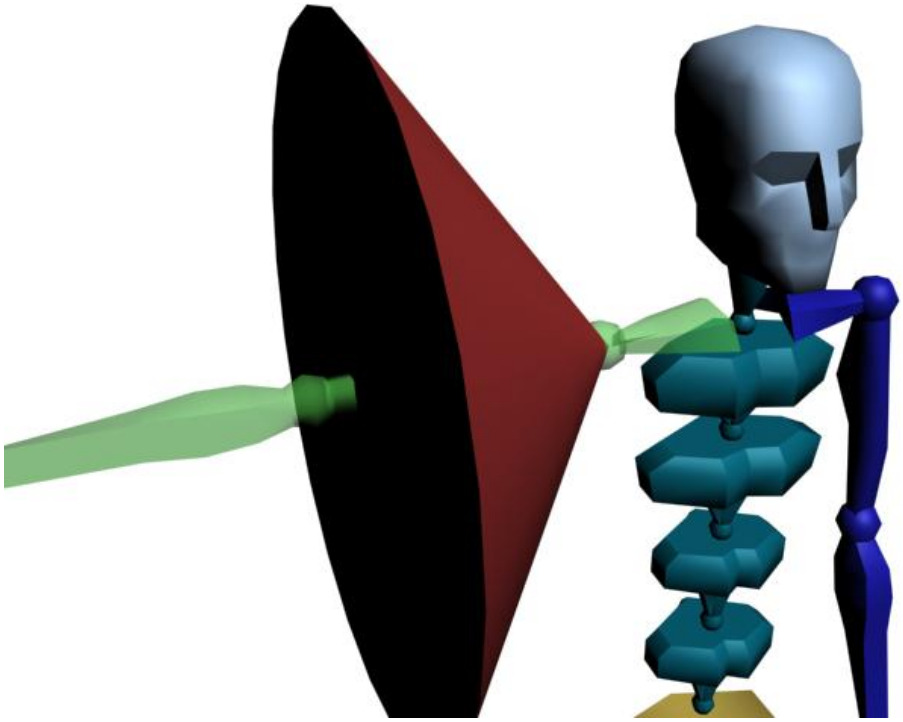


Figure 2.35: Cone of Movement

In the diagram, the cone is centred around the arm so there is equal movement upwards as downwards. However, you can probably swing your arm upwards until it is not quite vertical, and then around in an unbroken arc forward until it is pointing downwards. However, you will (almost certainly!) not be able to move your upper arm very far backwards relative to your torso. So, we need to tightly restrict its backward movement by clipping the cone of movement.

Havok uses two separate cones aligned around a common axis to prune the movement cone. The common axis for these clipping cones is refered to as *the plane normal*. To model the shoulder, let's align your coordinate system so that the plane axis is pointing straight forward from the shoulder.

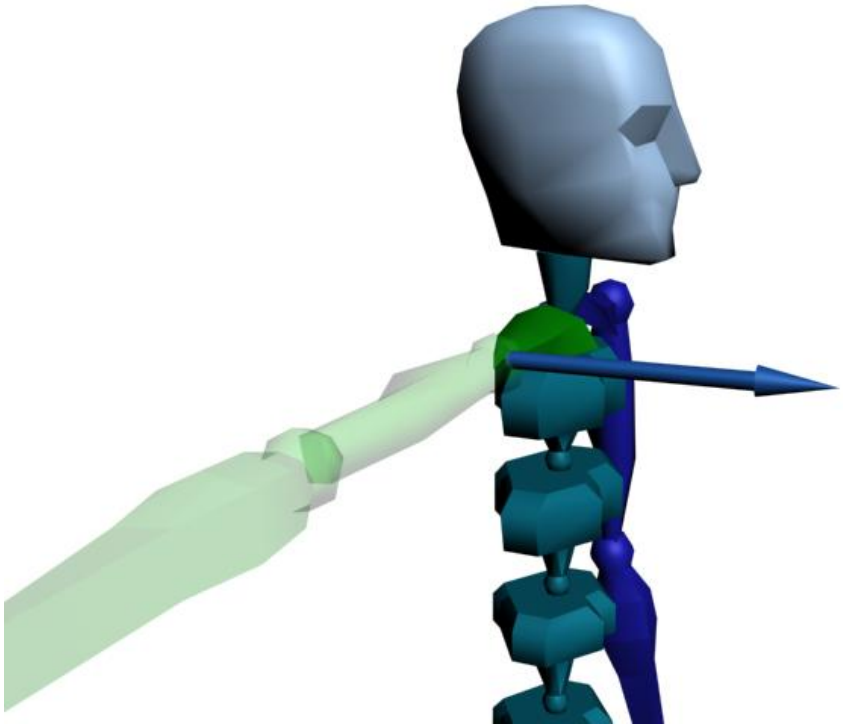


Figure 2.36: Plane Axis

We can specify a wide cone at the back so that it significantly clips the arm's movement. The front cone is small so that it does not affect the arm's movement at all. The specification of these cones is done using the **plane max** and **plane min** angles, which are explained below.

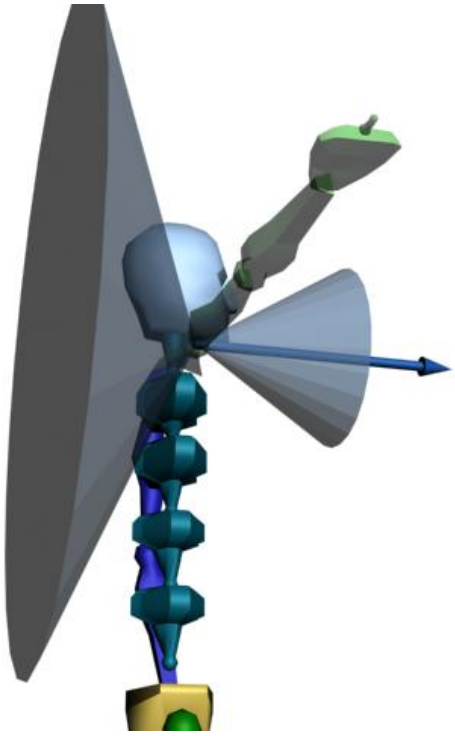


Figure 2.37: Plane Cones

Here is the restriction of the movement given by the plane cones we've described.

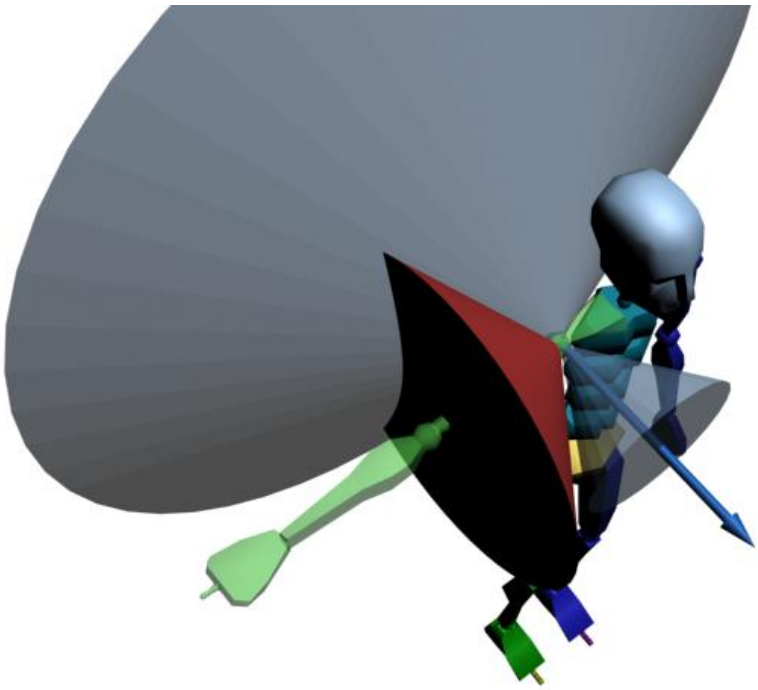


Figure 2.38: Plane Cones restricting Cone of Movement

The resulting movement is significantly restricted now in the backwards direction.

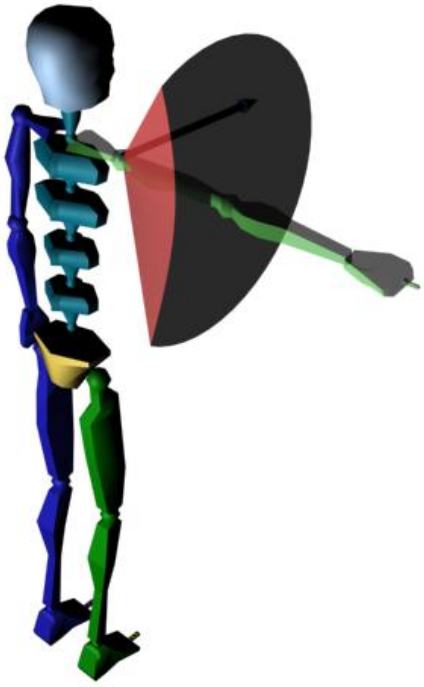


Figure 2.39: Resulting Cone of Movement

### Ragdoll Axes

You restrict how a ragdoll constraint's attached body can move relative to the reference body using the constraint's cone, plane, and twist limits. These are specified relative to the constraint's three axes.

As you know, you can define a three dimensional frame of reference using an origin and three axes. The ragdoll constraint has a set of transformation matrices defining the transform from the objects' space to the constraint space. The constraint space frame of reference for a rag doll constraint is constructed by defining:

- The origin as the pivot point of the constraint between the reference and attached bodies.
- The first axis as the twist axis of the ragdoll constraint; this is the axis around which the attached body can twist relative to the reference body.
- The second axis to be the ragdoll's plane normal, also known as the constraint's plane axis.
- The third axis to be the cross product of the twist axis with the ragdoll's plane normal.

The following diagram shows a ragdoll constraint's reference body, attached body, and axes, viewed from above. The twist axis crossed with plane axis is pointing out of the page. As you can see, the twist axis is aligned along the longest axis of the attached body. This is usually the case when modeling body joints, as this is the axis that is constrained within the constraint's cone of movement. You will see more clearly how this works in the cone and plane limits sections ahead.

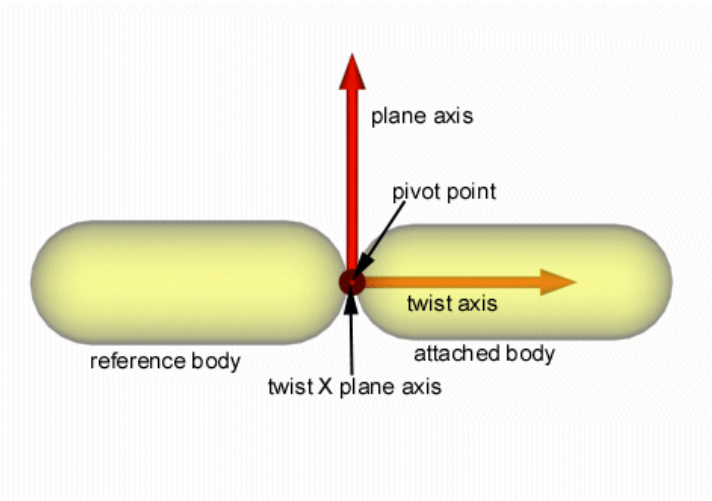



Figure 2.40: Ragdoll constraint bodies and axes, top view

Note that a ragdoll constraint's plane axis has to be at right angle to its twist axis, as shown in the diagram.

### Twist Limits

The ragdoll constraint's twist limits define the relative permitted rotation around the twist axis between the two constrained bodies. The angular limitations around the twist axis, i.e. the twist limits, should be set in the range [-pi/2..0] for the minimum and [0..pi/2] for the maximum. You specify the twist limits relative to the twist X plane axis. The following diagram should help you visualize the effect of these

limits. In this example, the attached body can twist relative to the reference body only within the range indicated in blue.

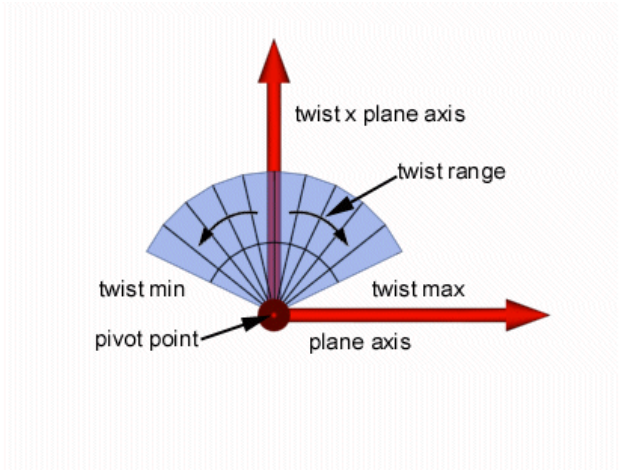



Figure 2.41: Twist limits, front view

**Cone Angle**

The ragdoll constraint's cone of movement is circular and has its point at the pivot point. It limits the rotational movement of the attached body so that its twist axis can only move relative to the reference body *within* that cone. The cone is circular at all times and its size is defined using the cone angle, which is specified relative to the twist axis. The cone angle should be set in the range [0..pi].

The following illustration should help you visualize how this limit works. You can see the attached object's cone of permitted movement and the constraint axes, viewed from the side and in perspective. In each case, the attached body's twist axis is free to move within the space indicated by the green cone.

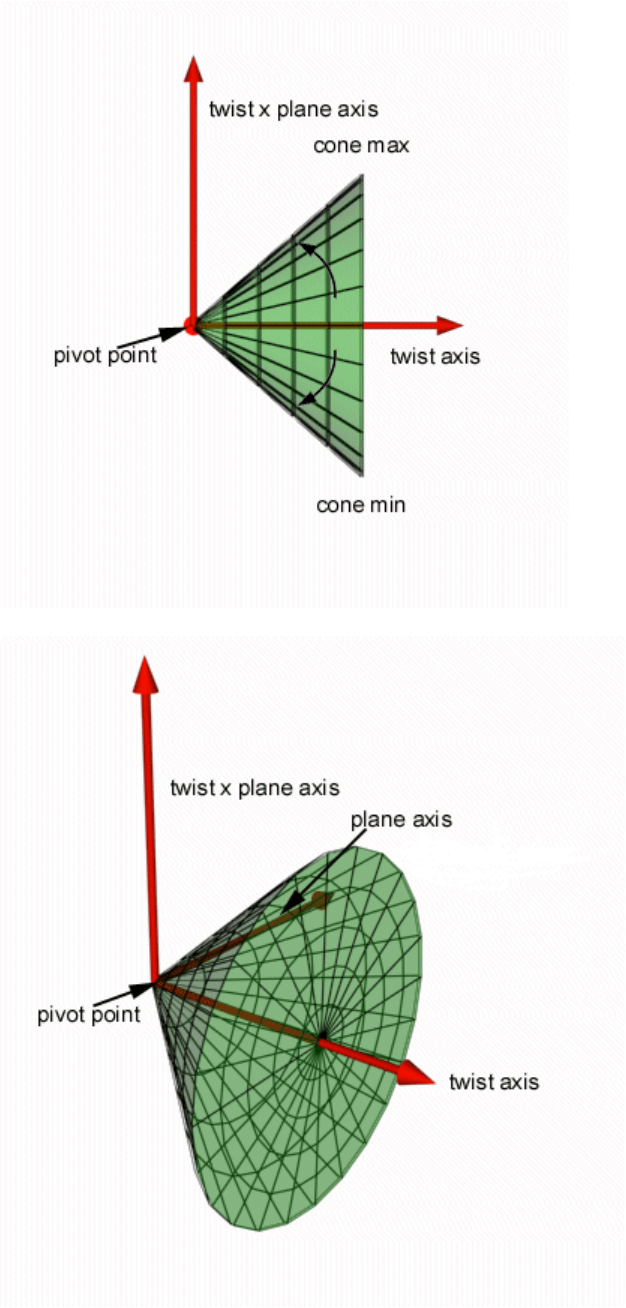



Figure 2.42: Cone of movement, side and perspective views

Alternatively, you can specify minimum and maximum angles for the cone. If these values are different, this has the effect of rotating the circular cone around the plane axis (the method you use to do this actually moves the twist axis in the reference body's space).

**Plane Limits**

You can use a ragdoll constraint's plane limits to clip the cone of available movement specified with the cone limits. As you know, one of the axes you need to define for a ragdoll constraint is its plane axis. This is the plane normal to a plane in the twist axis/twist X plane axis space. The constraint's plane limits indirectly specify the radii of two circular cones with their points at the constraint's pivot point,

known as the plane minimum cone and plane maximum cone. Both cones are aligned with the plane axis.

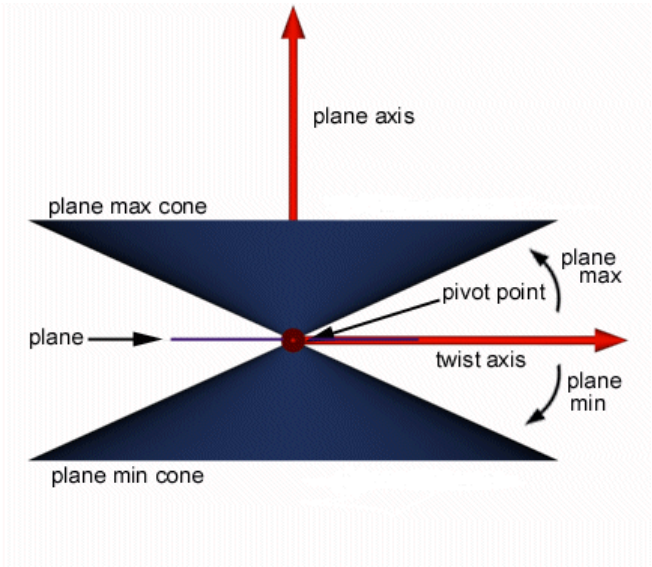



Figure 2.43: Plane and plane cones, top view

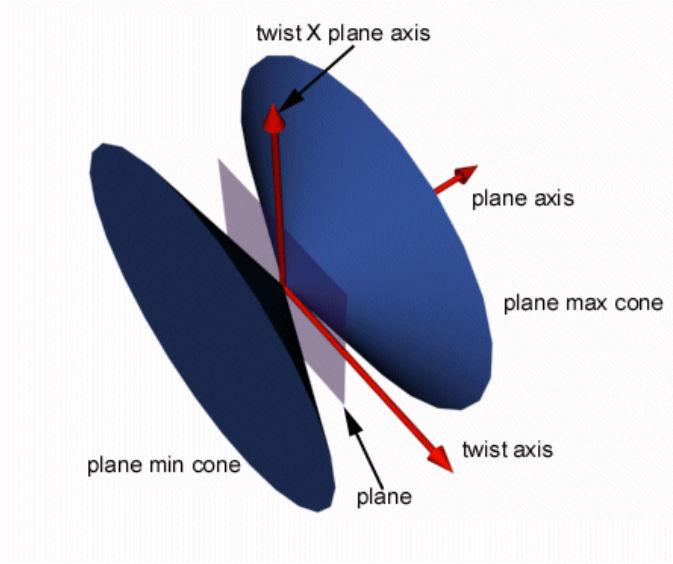



Figure 2.44: Plane and plane cones, perspective view

As you can see in the above diagrams, the plane max cone extends in the plane normal direction while the plane min cone extends in the negative plane normal direction. These cones limit the movement of the constrained body so that it can only move *outside* these cones. The plane limits should be set in the range [-pi/2..0] for the plane min value and [0..pi/2] for the plane max value, and are specified relative to the plane–large plane values produce small plane cones, and vice versa. While the illustrations show plane cones of equal size, it is of course possible to have differently sized plane cones by specifying different minimum and maximum values. Note that a minimum or maximum plane limit of 0 means that the corresponding plane cone becomes a plane, restricting all movement of the attached body in that direction.

You can see how plane values affect the constrained object's permitted movement space in the following

illustrations. As in the previous section, the attached object is free to move within the area represented by the green cone.

Let's first look at what happens when large plane values are provided. As you can see, in this example the cones do not intersect the main cone of movement, and hence the plane values have no effect on the attached body.

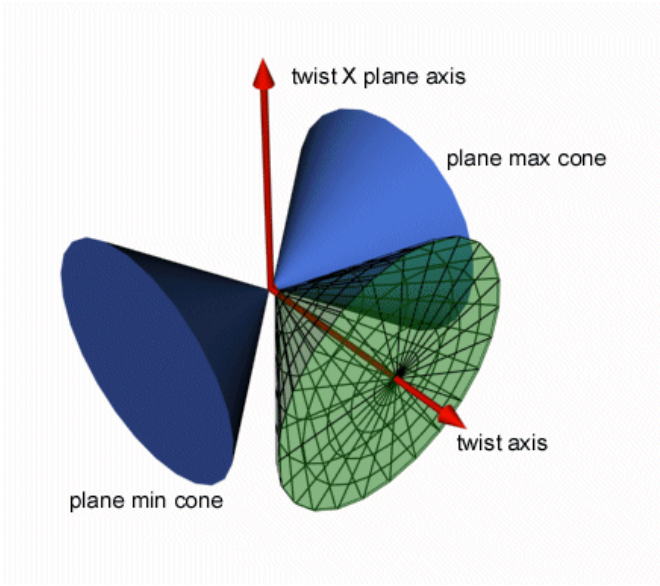



Figure 2.45: Plane cones not intersecting with movement cone

In the following example, however, lower plane values are specified and the cones intersect the main cone of movement. The subsequent diagrams show the constrained object's available movement space with these plane limits, the green cone having been clipped by the plane cones.

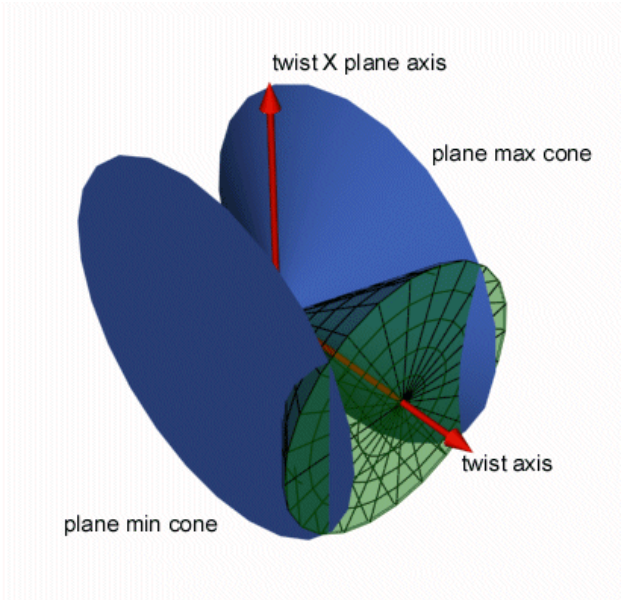



Figure 2.46: Plane cones intersecting with movement cone

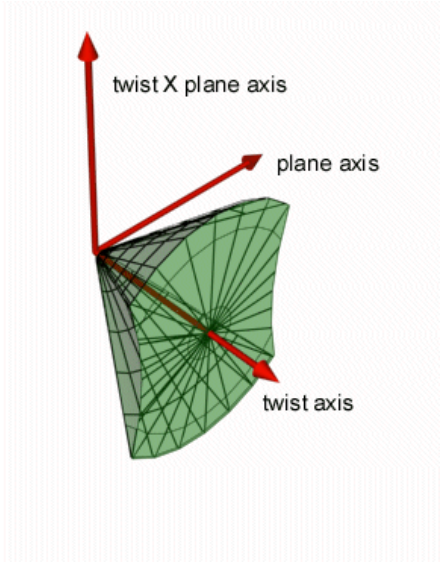



Figure 2.47: Permitted movement space

**Creating Ragdoll Constraints**

Now that you have been introduced to the ragdoll constraint's axes and limits, let's look at how to create a ragdoll constraint using the Havok SDK. In the following example, you're going to create a shoulder joint. Some suitable limit values are provided, but you can of course try your own parameters until you are happy with the movement of the joint.

**Creating the Shoulder Joint**

First you need to create two rigid bodies to act as the torso and arm, or reference body and attached body, respectively. Once these are created and positioned in world space, they can be passed to one of the `hkpRagdollConstraintData`'s `setIn...Space()` methods. In this example, you're going to specify the constraint details in body space. Hence, the torso and arm are passed to `setInLocalSpace()`–the first body passed to the method becomes the attached body, the second, the reference body.

You also use this function to specify the following information for each object, expressed in its own local space:

- the constraint pivot point
- the twist axis
- the plane axis

You don't need to specify the twist X plane axis, as it is calculated for you using the twist and plane axes. Note that when specifying the axes in local space, it should be possible to align the local axes. For instance, you shouldn't specify a different angle between the twist and plane axes for each object, as you may then get unpredictable behavior from the constraint. In the example, the objects are actually lined up correctly in world space, so the axis information is the same for each body.

Once you have set up the constraint's axes and pivot point, you can then specify the constraint limits using the appropriate methods. Because you want a cone with unequal values, you use the

setAsymmetricConeAngle() function to set the cone limits after setting up the axes. Finally, as with other constraints, your new ragdoll constraint must be added to the world.

```
hkpRagdollConstraintData* rdc;
{
  hkReal planeMin =  HK_REAL_PI * -0.05f;
  hkReal planeMax =  HK_REAL_PI * 0.5f;
  hkReal twistMin =  HK_REAL_PI * -0.025f;
  hkReal twistMax =  HK_REAL_PI *  0.025f;
  hkReal coneMin  =  HK_REAL_PI * -0.3f;
  hkReal coneMax = HK_REAL_PI * 0.45f;

  rdc = new hkpRagdollConstraintData();

  rdc->setPlaneMinAngularLimit(planeMin);
  rdc->setPlaneMaxAngularLimit(planeMax);
  rdc->setTwistMinAngularLimit(twistMin);
  rdc->setTwistMaxAngularLimit(twistMax);

  hkVector4 twistAxisA(1.0f, 0.0f, 0.0f);
  hkVector4 planeAxisA(0.0f, 0.0f, 1.0f);
  hkVector4 twistAxisB(1.0f, 0.0f, 0.0f);
  hkVector4 planeAxisB(0.0f, 0.0f, 1.0f);
  pivotA.set(-1.5f, 0.0f, 0.0f);
  pivotB.set(1.5f, 0.0f, 0.0f);
  rdc->setInBodySpace(pivotA, pivotB, planeAxisA, planeAxisB, twistAxisA, twistAxisB);
  rdc->setAsymmetricConeAngle(coneMin, coneMax);

  m_world->createAndAddConstraintInstance( rb1, rb0, rdc )->removeReference();
  rdc->removeReference();
}
```

### Where to Use Ragdoll Constraints

Ragdoll constraints are more expensive to simulate than, for example, hinges, and should be used only where the special angular limits offered by ragdoll constraints are necessary. You can often simulate knee and elbow joints adequately using simple hinges, saving the ragdoll constraints for joints with more complex movement such as shoulders and hips.

**Important Note:** The ball-and-socket part of the ragdoll constraint can be simulated in a more stable manner by setting its solving method to hkpConstraintAtom::METHOD_STABILIZED. Please see the section below on ball-and-socket stability for more details about this.

### Prismatic Constraints

A prismatic constraint is a constraint between two rigid bodies that allows translation movement along a single axis only. All three degrees of rotations as well as the remaining two translation axes are fixed. Prismatic constrainst are useful for implementing "sliders", i.e. one dynamic rigid body moving along another fixed rigid body in a given direction (the movable axis of the constraint).

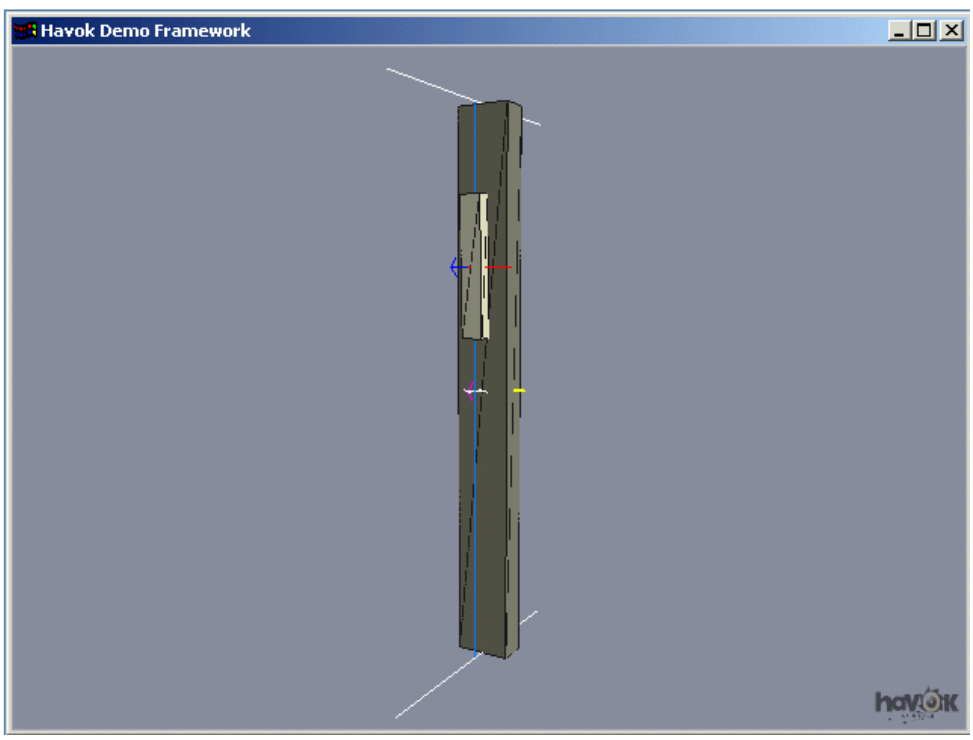
hkpPrismaticConstraintData is used to specify the two rigid bodies and the axis for the direction of movement. As with the other constraints, the axis is specified either in world space or in each body's space, in each case providing a position (pivot point) and a direction for the axis. A minimum and maximum linear limits can also be specified, along with a friction value for the attached body's movement along this axis. By default there are no linear limits and no friction.

**Note:**
The pivot point of the prismatic constraint should be on or inside the attached body and not the reference body. A pivot point in the reference body may produce strange/sponge-like behavior.

The following code snippet from PrismaticDemo shows you how to set up a prismatic constraint:

```
hkpPrismaticConstraintData* pris;
{
  hkVector4 axis(0.0f, 1.0f, 0.0f);

  // Create constraint
  pris = new hkpPrismaticConstraintData();
  pris->setInWorldSpace(rb1->getTransform(), rb0->getTransform(), position, axis);
  pris->setMaxLinearLimit(10.0f);
  pris->setMinLinearLimit(-10.0f);
  m_world->createAndAddConstraintInstance( rb1, rb0, pris )->removeReference();
  pris->removeReference();
}
```

The following illustration shows the prismatic constraint created in the example. The small box is free to move along the blue axis within the limits indicated by the white lines.




### Prismatic Constraint Linear Limits

If a prismatic constraint between two dynamic bodies has a linear limit, when the attached body hits this limit, large angular effects may occur and cause the constraint to become unstable. If this occurs, make sure the pivot point is specified inside the attached body, and also increase the mass of the reference body to many times that of the attached body. Note however that normally it is desirable for constrained objects to have similar mass. Increasing the damping values for the bodies may also help if this occurs.

### Point-to-Path Constraints

The point-to-path constraint allows you to constrain two bodies so that one is free to move along a specified path relative to the other. You could, for example, use a point-to-path constraint to create a ski chairlift, where a chair is conveyed along suspended from a cable.

When creating a point-to-path constraint, you must provide two rigid bodies as well as the path the first body should follow. You specify the path using an `hkpParametricCurve`. `hkpLinearParametricCurve` is an implementation of `hkpParametricCurve` that represents a piece-wise linear curve. It has a number of useful methods that you can use to set up your path, including `addPoint()`, which allows you to add a point to the path.

The point-to-path constraint allows you to specify an orientation type for the attached body's movement. The possible orientation types are:

- `HK_DO_NOT_CONSTRAIN_ORIENTATION`, where the attached body's orientation relative to the path is not restricted by the constraint. This is the default orientation type.
- `HK_CONSTRAIN_ORIENTATION_ALLOW_SPIN`, where the attached body is allowed to spin around the path that it's traveling along, but cannot change its orientation in any other way.
- `HK_CONSTRAIN_ORIENTATION_TO_PATH`, where the attached body's orientation is constrained to the path.

Note that for the `HK_CONSTRAIN_ORIENTATION_TO_PATH` orientation type to work, the `hkpParametricCurve` must have its `getBinormal()` method implemented. This should return a normalized vector that defines the "up" direction for the path. A very naive implementation is provided in `hkpLinearParametricCurve`.

You can also provide a friction value for the body's movement along the path. By default, this friction limit has a value of zero.

The following examples from `PointToPathDemo` show you how to set up a point-to-path constraint. In this demo, the attached body is only free to move along the specified helix curve.

First, let's look at how you create the path's curve. As you can see, the `hkpLinearParametricCurve::addPoint()` method is used to build up the helix:

```
m_helix = new hkpLinearParametricCurve();
const hkReal twists = 3.0f;
const hkReal height = 15.0f;
const hkReal radius = 10.0f;
hkVector4 pt;
for (hkReal t = 1.0f; t > 0.0f; t -= 1.0f / 300.0f)
{
    const hkReal angle = t * HK_REAL_PI * 2.0f * twists;
    pt = hkVector4(hkMath::cos( angle ) * radius * ( 1.0f + t), t * height, hkMath::sin( angle ) * radius
      * (1.0f + t) );

    m_helix->addPoint( pt );
}
```

Once you have created the rigid bodies and path, you use the `hkpPointToPathConstraintData::setInWorldSpace()` method to specify the attached body, the

reference body, the pivot point in world space (which specifies the position of the path) and the path curve for the constraint. If you use `setInBodySpace()`, you can set the pivot in each body's space instead. This example also sets an orientation type and friction value.

```
hkpPointToPathConstraintData* p2Path;
{
  p2Path = new hkpPointToPathConstraintData();

  // constrain angular DOFs so that it can only spin around the path
  p2Path->setOrientationType(hkPointToPathData::HK_CONSTRAIN_ORIENTATION_ALLOW_SPIN);

  p2Path->setInWorldSpace(rb1->getTransform(), rb0->getTransform(), rb1->getPosition(), m_helix);
  p2Path->setMaxFrictionForce( .1f );

  m_world->createAndAddConstraintInstance( rb1, rb0, p2Path )->removeReference();
  p2Path->removeReference();
}
```

### Rack-and-Pinion and Cog-Wheel constraints

The `hkpRackAndPinion` and `hkpCogWheel` constraints allow you to create simple mechanical devices. The rack-and-pinion constraint transforms rotational movement of one body (a pinion) into a translational movement of another body (a rack). This constraint can also function as a screw-and-nut constraint. The cog-wheel constraint binds the rotational movement of two bodies.

The `hkpRackAndPinion` and `hkpCogWheel` constraints are 1-dimensional constraints only. They only control relative velocity along chosen axes, but they do not control how the bodies are oriented or fixed relatively to each other. This is to give you the freedom of constructing custom setups.

The `hkpRackAndPinion` parameters let you specify the rotation pivot and axis for the pinion and shift axis for the rack, and the radius of the pinion. The rotation direction of the pinion is then determined by the direction of its axis and the sign of the radius value. The actual point where force is applied between the moving parts is calculated by offsetting the pinion pivot by the pinion's radius.

This constraint can also function as a screw-and-nut constraint (when rotation and shift axes are parallel). In that use case, the pinion radius parameter is interpreted as screw pitch (linear translation resulting from a single rotation of the screw).

The `hkpCogWheel` constraint lets you specify the rotation axes for the wheels and their radii. The interaction point of the cog wheels is then dynamically calculated based on the wheels' orientation. You can dynamically change the radius of the cog wheels of a `hkpCogWheelConstraintData`. To do this however you should provide references to all `hkpConstraintInstance`s that use the data. This is needed to reset the runtime data of the constraints. The runtime data stores relative orientations of the cog wheels to prevent them from snapping, or losing rotation, when one is quickly accelerated.

### Creating Complementing Constraints

To create a working setup you need to additionally attach bodies to each other, or to other elements of a mechanism. One would usually use hinge constraints and prismatic constraints.

Both constraints have helper functions that let you quickly create a matching prismatic or hinge constraint. The helper functions use the axes and pivots from the `hkpRackAndPinion` or `hkpCogWheel` constraints. The functions only work on bodies already positioned in the world. The helper methods also let you specify a third body that the additional hinge/prismatic constraints should connect to. This way

you can attach elements of the rack-and-pinion to a fixed world, or to other dynamic elements.

For convenience, the `hkpPrimaticConstraintData` can be converted to a cylindrical constraint using the `hkpPrismaticConstraintData::allowRotationAroundAxis()` method.

The CogsDemo contains simple examples using `hkpRackAndPinion` and `hkpCogWheel` constraints. It also contains a more complex model of a differential mechanism built from cog-wheel and hinge constraints.

**Breakable Constraints**

A breakable constraint (as the name suggests!) is a constraint that breaks when a specified threshold is exceeded. For example, you could use a breakable constraint to make a door that breaks off its hinges if it's kicked hard enough. `hkpBreakableConstraintData` is a wrapper around one of the other constraint types that allows you to make the constraint breakable.

The following example from `BreakableConstraintDemo` illustrates how to create a breakable hinge. As you can see, you first create a hinge as usual:

```
// set pivot
hkVector4 pivot  ( 0.5f, -0.6f, 0.5f );
hkVector4 axleDir( 0.0f,  0.0f, 1.0f );

// Create hinge constraint
// Do not add it to the world, only add the "wrapped" breakable version below
hkpHingeConstraintData*  axle = new hkpHingeConstraintData();
axle->setInWorldSpace(rb0->getTransform(), rb1->getTransform(), pivot, axleDir );
```

However, rather than adding the hinge to the world directly, you pass both the hinge and the world to the constructor of a new `hkpBreakableConstraintData`. Then you specify the threshold at which the constraint will break using `hkpBreakableConstraintData::setThreshold()`:

```
// Create breakable constraint wrapper
hkpBreakableConstraintData* breaker = new hkpBreakableConstraintData( axle , m_world );
breaker->setThreshold( 10 );

// Remove reference to "axle" since the BreakableConstraint now owns it.
axle->removeReference();
```

Breakable constraints also allow you to specify whether you want the breakable constraint to be removed from the world when it breaks. You do this using `hkpBreakableConstraintData::setRemoveWhenBroken()`:

```
breaker->setRemoveWhenBroken(false);
```

Finally, once you have set up the breakable constraint, you add it to the world:

```
// Add the breakable constraint to the world
m_world->addConstraint( constraint );

constraint->removeReference();
breaker->removeReference();
```

When you run `BreakableConstraintDemo`, the hinged blocks remain hinged until a falling third block strikes them. The force of this collision exceeds the breakable threshold and the hinged blocks are separated.

### Breakable Listeners

If you want to be notified when a breakable constraint breaks, you can implement the `hkpBreakableListener` interface. The `constraintBrokenCallback()` method is called by the simulation each time a constraint breaks, passing the listener a reference to the constraint, the force that broke it, and whether the breakable constraint has been removed from the simulation.

### Malleable Constraints

You can make a constraint softer or harder by making it into a malleable constraint. This changes the virtual mass used by the system when solving the constraint. This is useful, for instance, if you want to create a bouncy spring using the `hkpStiffSpringConstraintData`.

You make a malleable constraint by wrapping an existing constraint in an `hkpMalleableConstraintData`, in the same way as you create a breakable constraint. Then you can use the `setStrength()` function before adding the malleable constraint to the world.

### Other Constraint Types

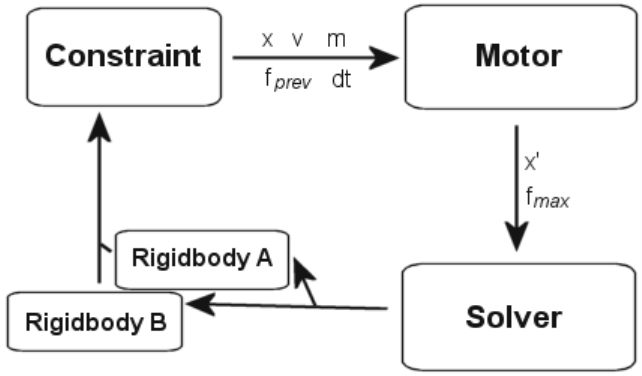
Several less commonly used constraint types exist which are not described in detail here. Refer to `Source/Physics2012/Dynamics/Constraint/Bilateral` and classes derived from `hkpConstraintData`.

### Enabling Motors in Constraints

Havok also provides "powered" versions of some of its constraints, namely the limited hinge, ragdoll and prismatic constraints. Each degree of freedom for these constraints can be controlled by a chosen motor. The motor implements a force law to apply calculated forces to that degree of freedom. These forces work to move that degree of freedom from its current position to a desired target position. For example, you can model powered machinery or muscles using motors.

### Understanding Motors

Along with the constraints specified above, motors can also be added to custom constraints created with the constraint construction kit. You need to explicitly add your desired motors to the powered constraints.




The diagram above shows the basic feedback loop for the motors. The state variables passed to the motor are in 1D "motor space": they are the dynamic values of the bodies converted into the one dimension that the motor works on. The constraint is responsible for doing this conversion and calling the motors with this data.

Position ( x ) and velocity ( v ) are relative between the two bodies connected by the motor. You can just treat the input and output values as if they described a particle on a line. The mass ( m ) is the effective mass that is "felt" by the motor. For example, if you apply 2 units of force and the mass is 1 unit with a delta time of 1 unit, you should get a change in constraint space velocity of 2 units.

The motor does not apply forces directly, it tells the solver what it would like done and the solver applies the forces to the rigid bodies. The outputs from the motor (the inputs to the solver) are the new motor space position you are trying to reach and a maximum force that the solver can apply to get there.

You convert a simple applied force to the motor outputs by setting the maximum force (*f_max*) equal to the applied force, and the target position (*x'*) equal to that force multiplied by the timestep squared and divided by the mass. Setting the target position follows from the basic laws of physics: `f = ma`, `a = dv/dt`, `a*dt = v = dx/dt`, `(f/m)*dt*dt = x`. This slightly more complicated interface to the outputs is provided to allow the motors maximum control over the stability and behavior of the system.

**Using Motors**

A number of motors are provided for you. The `hkpPositionConstraintMotor` is a good motor for blending keyframed animation targets with an articulated figure. It uses an implicit spring-damper model to calculate the target for the current step, then allows the solver to act as a stiff driven constraint to get to that target.

You may have a very specialized game-specific force law in mind for your motors, i.e. a motor that gets stronger as the distance from the target increases, up to a point, then becomes weaker past that point. To write your own motor you should use the `hkpCallbackConstraintMotor` and provide a custom `CalcMotorDataCallbackFunc()` callback function.

The following code block demonstrates how to implement an ideal spring. The force exerted by an ideal spring is proportional to the displacement of the spring from its equilibrium position.

```
// Create a callback function
void HK_CALL SampleCallbackFunc(const hkpCallbackConstraintMotor& motor, const hkpConstraintMotorInput*
  input, hkpConstraintMotorOutput* output)
{
  // We're not interested in achieving a specific velocity
  output->m_targetVelocity = 0.0f;

  // We compute relative targetPosition for the motor i.e. our spring's rest point.
  // This is computed directly from the given input.
  output->m_targetPosition = input->m_deltaTarget + input->m_positionError;

  // These are desired spring constant and dumping.
  // Those values need to be converted into simple tau and damping parameters used by the solver.
  hkReal springConstant = 100.0f;
  hkReal springDamping  = 5.0f;

  hkReal invMass = 1.0f / input->m_virtualMass;
  hkReal dt = input->m_stepInfo->m_substepDeltaTime;

  // Forces applied by the solver are proportional to tau and virtual mass of involved bodies.
  // Spring force is independent of the bodies' masses therefore we divide tau by their virtual mass.
  // To convert tau to solver space we multiply it by dt^2
  hkReal tau = springConstant * dt * dt * invMass ;
  output->m_tau = hkMath::clamp( tau, 0.0f, 1.0f );

  // Solver damping is computed analogically to tau.
  // To convert damping to solver space we multiply it by dt
  hkReal damp = springDamping * dt * invMass;
  output->m_damping = hkMath::clamp( damp, 0.0f, 1.0f );

  // Explicitly specify minimum and maximum forces that the motor can apply.
  output->m_maxForce = motor.m_maxForce;
  output->m_minForce = motor.m_minForce;
}

// Create a callback motor which uses the above function.
// This motor can be then used to power a constraint.
hkpCallbackConstraintMotor* ccm = new hkpCallbackConstraintMotor( hkpCallbackConstraintMotor::
  CALLBACK_MOTOR_TYPE_HAVOK_DEMO_SPRING_DAMPER, SampleCallbackFunc );
```

### Powered mode of Hinge and Ragdoll Constraints

Motors of a ragdoll constraint can move the constraint's attached body to a specified target position, set using the constraint's `setTargetFromRelativeFrame()` or `setMotorTargetAngle()` function. You specify the target as a relative orientation between the attached and reference bodies. The constraint will then use three motors to try to achieve this desired pose target.

The ragdoll constraint's motors work differently than the motors for the rest of the constraints. Ultimately we are trying to match two orientations using the motors, rather than directly powering a degree of freedom to a target with a motor. The current position passed in will be the distance from the zero point of the motor. So all you need to do with the powered rag doll is set its orientation target; you don't need to access the motors directly.

The three motors apply torques around the twist, cone and plane axis. You can set the twist, cone and plane axis motors to be separate motors or they can all point to an instance of the same motor. Motors can be shared between axis and between different constraints. You must however take care that you don't share motors if you have implemented your own motors which contain non-shareable data.

**Note:**
The motor strength for powerered hinge and ragdoll constraints is affected by the global solver settings. When these solver settings are made stiffer, the motor strength correspondingly becomes stronger (and vice-versa).

### Tuning and Tweaking Constraints

#### Solving Constraints

Havok deals with each constraint added to the simulation as part of an overall constraint system. As a result, constraints are relatively stable and stiff, although they can get slightly softer with a lower simulation frequency or differences in mass. The constraint system also includes contact constraints (which are used internally by Havok to prevent objects from interpenetrating), as well as the constraint types described earlier.

The overlap information for any pair of objects provided by Havok's collision detection system is an error in satisfying their contact constraint. Similarly, when a pair of constrained objects become separated, the measure by which they are separated is an error in satisfying their ball-and-socket constraint.

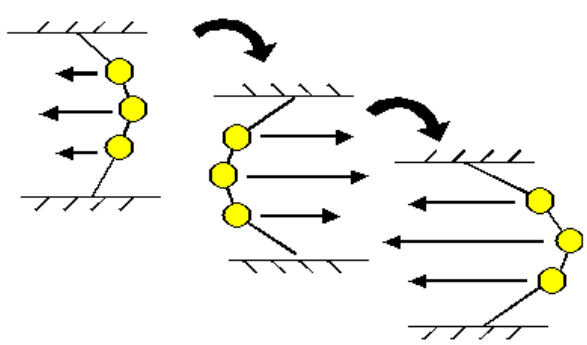
Errors in satisfying constraints can be counter-acted by moving the objects to a new position. This is done very carefully, however, as moving an object to deal with one error can introduce another error elsewhere in the system.

This problem is dealt with internally by the constraint solver. The solver is responsible for calculating the necessary changes to reduce error for all constraints in the simulation in each simulation step. This information is passed on to the integrator, which uses it to calculate the new motion state (position, orientation, velocity, acceleration, etc.) for each object in the simulation. These new positions can then be used to update various game state information (such as updating the game display of the objects).

#### Reasons for Instability of Constraints

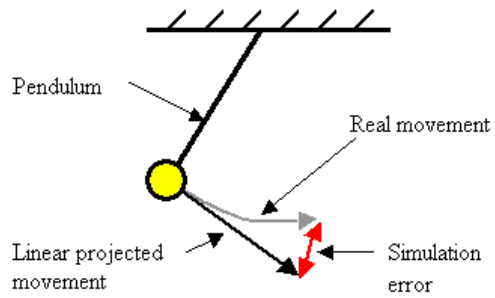
Simulating constraints is a hard problem to solve. For practical reasons a constraint solver has certain stability limitations:

- The forces involved in solving a constraint system can be too high and lead to frequency of object movements higher than the simulation frequency. For example, imagine simulating a swinging short rope that gets pulled harder and harder. In a physical simulation with 30 physical steps per second pulling the simulated rope can lead to strange artifacts like instability or soft constraints.



Swinging exploding rope (with three balls)

- The constraint solver solves a set of linear equations. In the physical simulation, however, angular movement leads to non-linear behavior. If the time step is relatively big compared to the angular velocities of the objects, the error introduced may lead to instability.




### Getting Stable Constraints

To make constraints more stable, try following these simple rules. Start with the first rule and only if you do not get your expected results, try the next ones.

- Avoid low rotation inertia. Low inertia leads to high angular velocity, which in turn introduces large simulation errors . In particular, long and thin objects have one axis with a very low inertia. Try to artificially increase the inertia around this axis. Very small objects also have low rotational inertia.

  The following rules are useful here:

  - In a multi body system, the minimum rotation inertia of one object should not be less then 1/10th of the maximum rotation inertia of the same object.
  - If a thin object is constrained between two heavy objects, the minimum rotation inertia of any object should not be less then 1/4 of the maximum rotation inertia of the same object.

- Use the functionality provided by the `hkpConstraintStabilizationUtil` to pre-stabilize the constrained rigid bodies system.

- If your constraints appear to be very soft, try to avoid constraining light objects between two heavy objects, or between a fixed object and a heavy object. Make the light object at least as heavy as a 1/10th of the mass of the heavier objects.

- If you constrain a very large number of objects together, you may need to decrease the tau value of all constraints involved. Use the following rule: tau = $\sim$ 1.0/sqrt(number of constraints in a system).

- Decrease the damping of the constraints slightly (to 0.6).

- Increase simulation frequency by increasing the number of substeps.

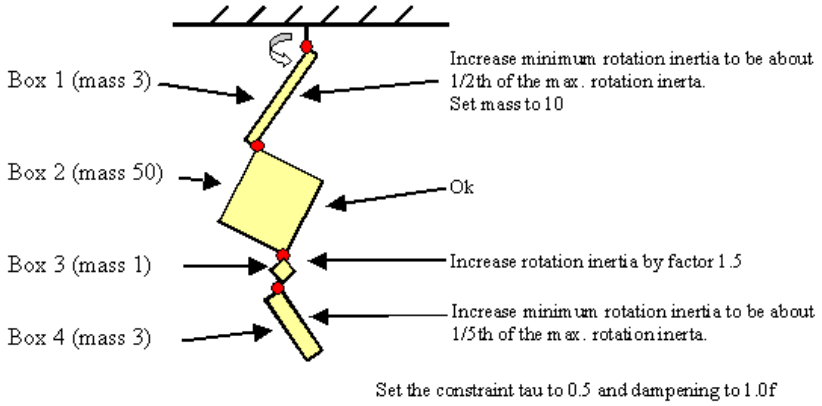
- Increase solver tau and damping parameters. (See `hkpWorldCinfo::SolverType`)

- Contact Havok support.

**Note:**

These rules are rather strict. If you care more about physical accuracy, then you can either experiment with your system to find the real limits, or, if possible, just increase the number of substeps.

The following example shows a chain of objects with some hints to get a very stable solution:




**Stability of the hkpBallSocketConstraintAtom**

The stability of the ball-and-socket constraint atom (present in the ball-and-socket, hinge, limited hinge and ragdoll constraints) can be improved by opting for a special solving method, i.e. calling `setSolvingMethod(hkpConstraintAtom::METHOD_STABILIZED)`. This tells the constraint solver to use a block solving method for the ball-and-socket atom, enforcing the constraint simultaneously on all 3 axes.

The 3D block solver generally produces a much stiffer constraint, but still does not get rid of all stability issues. The latter are still induced by large mass ratios, big distances from the body to the constraint pivot (i.e. force arms), and in general, anything that affects the angular motion of the constrained bodies.

Consider a pair of bodies constrained by a ball-and-socket. Knowing that the angular part of their motion is proportional to the squared arm length, the constrained pair can easily be made unstable by simply moving the constraint pivots sufficiently far away from the centers of mass. Due to discretization and numerical errors, solving such a configuration will most likely cause the constraint to blow up. To prevent that, the solver can be allowed to dynamically alter the inertia tensors, in order to numerically stabilize the problem.

The amount of inertia alteration that can be applied by the solver is controlled by the `m_inertiaStabilizationFactor` parameter of the `hkpBallSocketConstraintAtom` class. It must be inside the interval [0, 1] and has the effect of modifying the physics problem to be solved into a stable one. For a value of 0, no alteration is applied and the solver moves the bodies with their original inertias. However, for a value of 1, the solver has full liberty and alters the inertia tensors as much as needed in order to reach a stable solution.

The dynamic inertia alteration inside the solver can produce visual artifacts when combined with other constraint atoms that still perceive the constrained bodies with their original inertias (for instance, a ball-and-socket chain colliding with itself or the ground). In this case, it is best if the inertia alteration is performed beforehand, outside the simulation. This can be done either manually, by calling one of the functions provided by the utility class `hkpConstraintStabilizationUtil`, or automatically from the export pipeline, by using the *Create Constraints* filter UI.

Note that in order to properly compute a stable inertia tensor for a given rigid body (in off-line mode), it has to be already properly constrained. Also, if a rigid body has had its inertia tensor stabilized off-line, it is generally useless to allow the solver to dynamically alter it, so the `m_inertiaStabilizationFactor` should be set to zero in this case.

The performance of the stable ball-and-socket solver has been evaluated on a test scene consisting of 500

ball-and-socket constraints. The results can be found in the table below:

| **Platform** | **Average METHOD_OLD *Integrate* timer** | **Average METHOD_STABILIZED *Integrate* timer** | **Slowdown / Speedup** |
|---|---|---|---|
| Win32 | 0.32 ms | 0.43 ms | 25% slower |
| Xbox 360 | 0.62 ms | 0.69 ms | 10% slower |
| PlayStation®3 | 0.31 ms | 0.29 ms | 6% faster |

Table 2.6: Ball-and-Socket constraint solver: performance comparison for 500 constraints.

The stabilized ball-and-socket atom solver assumes some quantities have been pre-computed (performance optimization). The solver is instructed to compute these quantities whenever an enabled `hkpSetupStabilizationAtom` atom is encountered. This is an important fact to be remembered: if you are building custom constraints that include the ball-and-socket atom with the solving method set to `METHOD_STABILIZED`, the constraint must include an enabled `hkpSetupStabilizationAtom` atom. Since the pre-computed quantities depend on arm lengths, this atom must generally be placed after the constraint pivots have been set (or modified).

The stable ball-and-socket atom solver does not clamp the maximum applied impulse by default, due to instabilities added by this techinque in some cases. It is however possible to limit the magnitude of the angular impulses applied by setting the `hkpSetupStabilizationAtom::m_maxAngle` variable to the maximum angle the solver is allowed to rotate a body. By default, the maximum angle is set to a very high value so it has no effect.

One may note the existence of the `hkpBallSocketConstraintAtom::m_velocityStabilizationFactor` parameter inside the ball-and-socket constraint atom. This currently deprecated and is preserved only to maintain functionality of the previous solving method (i.e. `hkpConstraintAtom::METHOD_OLD`). It has no effect while in the stable solving method.

#### Stability of Angular Limit Constraint Atoms

Angular limit constraint atoms can lead to some stability issues when the limit angles are set too high. Internally, when decomposing the quaternion into its axis angle representation during constraint solving, we introduce instability. This is simply a general mathematical problem that cannot be avoided. As the angle limit approaches the position 180 degrees away from target angle, the constraint system can simply not know which direction is the correct direction to fix the error in.

If you encounter stability issues relating to the use of angular limit constraints, you can address it in several ways:

- The first technique is simply to lower the angular limits to avoid approaching the singularity case. This may not always be possible.
- Lower the angular limit stiffness factor of the affected constraint (`hkpAngLimitConstraintAtom::m_angularLimitsTauFactor`). This will make the error-correction algorithm less aggressive in its changes every step.
- Consider using the ragdoll constraint instead of constraint atoms. The ragdoll constraint has stability algorithms built into it, essentially dynamically adjusting tau values to compensate for instability situations.

#### Velocity Stabilization Factor (deprecated) in hkpBallSocketConstraintAtom

The parameter `hkpBallSocketConstraintAtom::m_velocityStabilizationFactor` controls the

amount of dampening applied by the solver while in the `hkpConstraintAtom::METHOD_OLD` mode, and has a default value of 1.0. It works by modifying the constraint's setup so that it opposes the relative velocity of the constraint bodies. As side effect you will notice that the positional constraints may be violated when the constraint is looked at from a close distance.

The velocity stabilization factor can be increased further for simple setups that tend to not stabilize.

For specific solver settings and more complicated systems (such as unusual ragdoll setups), the stabilization factor may introduce jitter, and should be reduced or set to 0.0.

**CPU Requirements**

All constraints in Havok are O(n) constraints. This means that simulating 100 constraints requires 10 times more CPU than simulating 10 constraints. As a result, there is no worse case CPU requirement if many objects are constrained together.

In principle, the more degrees of freedom a constraint restricts, the more expensive it gets. A hinge with one degree of freedom restricts the remaining 5 degrees and is more expensive than a point-to-point constraint.

**Constraint Limitations**

In general, the Havok Vehicle Kit is a better way to construct four wheeled vehicles than using wheel constraints, especially racing or fast moving vehicles. Take a look at the Vehicle Kit chapter for more information.

**Increasing Inertia Tensors for Ragdoll (e.g. hkpInertiaTensorComputer::optimizeInertiasOfConstraintTree())**

Simulating a dynamics system of only linear equations is pretty simple. However, many of the problems when simulating dynamic objects are due to angular effects. A simple trick to reduce angular effects is to increase the inertia of the rigid bodies.

Examples:

- A rope of bodies blows up.
- A ragdoll jitters a lot.
- The ragdoll limits are too soft.
- A ragdoll explodes.
- An object causes too many TOI events.
- Ragdoll motors are too soft.

All of these problematic situations can be improved by increasing the inertia of the bodies concerned.

So we want to:

- increase inertia to make things more stable but...

- ...without changing the behavior to the extent that a user can see the inertia tensor increase (e.g. extremely slow rotating objects).

Most of our use cases include constrained rigid bodies. Such constrained bodies cannot move entirely freely. As a result the virtual inertia tensor of each rigid body is much higher than its real inertia.

For example, say we have a human arm: .6m*.1m*.1m (using a box) with a real inertia diagonal of (0.0066, 0.12, 0.12). The inertia around the X axis is 18 times smaller than the other inertia elements. Now consider two of those arms: rotate one by 90 degrees and glue them together (fixed constraint) at their center of mass. The resulting inertia tensor diagonal now looks like (.1266, .1266, .24) (both inertias added). If we modify the original inertia of one arm by 1000% (which seems a lot) from .0066 to 0.066, the combined inertia will only go up to (.186,.186,.24).

In short: Changing the inertias of individual objects (especially the smaller ones) in a constrained system only has a minor effect on the behavior seen by the user, but makes motors and limits much better/stiffer.

The `hkpInertiaTensorComputer::optimizeInertiasOfConstraintTree()` method tries to increase the inertia of the inner bones of a ragdoll as much as possible to improve motor, limit and friction stability, but not so much that the combined ragdoll inertia tensor increases significantly. As a result the motors should handle much better, without drastically modifiying the behavior of the ragdoll as a whole.

### Constraint Chains

#### Introduction

Constraint chains are useful for simulating long chains of linked bodies, which would or might fail to be properly simulated by Havok's iterative solver.

The advantages are:

- **Ragdolls behave more solidly** : Characters limbs form chains of bodies of varied mass. In such cases, a low mass bone in the middle of the chain (e.g. a shoulder plate) will largely reduce accuracy for other heavy bones located further away from the torso. This can lead to ragdoll limbs stretching and separating when ragdolls are thrown against obstacles, or dragged forcefully.
- **Better motor accuracy for powered ragdolls** : When powering a ragdoll with animation information, Havok's iterative solver may not deliver an accurate match to the requested pose. Creating the ragdoll with powered chain constraints gives greatly improved results.
- **Much better results when simulating of ropes** : Ropes can be simulated with a large number of bodies, i.e. more than 20 and up to a few hundred.

#### How to Use Chains

Chains are treated by the Havok engine almost identically to all the other constraints; they are, however, slightly more complex to setup. We now describe how to setup constraint chains, but also refer to the example demos available.

To setup a constraint chain you need to initialize a `hkpConstraintChainData` and `hkpConstraintChainInstance` object, in the same manner as with normal constraints. The constraint instance must be added to the world using `hkpWorld::addConstraint( myChainInstance )`.

The `hkpConstraintChainData` holds a list of constraint information specific to the type of the chain. Each constraint information holds data analogous to data held by a regular Havok constraint data (such as pivot points). `hkpConstraintChainInstance` holds a list of linked entities. There is a one-to-one match between the `ConstraintInfo`s and the list of entities.

The constraint data's list is populated by calling `hkpConstraintChainData::addConstraintInfoInBodySpace()` and the instance's list calling `hkpConstraintChainInstance::addEntity()`.

Havok supports three types of chain data:

- `hkpStiffSpringChainData`: the simplest and most effective chain in terms of CPU usage. It is however also much less stable than the `hkpBallSocketChainData` described below. One may consider using this chain in conjunction with 'stabilized' motions for the bodies in the chain, or with objects with increased inertia tensor values.
- `hkpBallSocketChainData`: the preferred constraint allowing for quick and easy setup of ropes. This is also the preferred chain to use for non-powered ragdolls. However, as chains do not support constraint limits, you will need to add additional limit constraints on top of the chains. There are two stripped-down constraints available for that purpose: `hkpHingeLimitsData`, and `hkpRagdollLimitsData`.
- `hkpPoweredChainData`: the chain to use when you want to create complex machines, crane arms, or ragdolls. Note that when adding chains to a ragdoll you can overlap the chains. You just need to ensure that the overlapping chains to not apply contradicting constraints to the linked bodies. Those chains can also be driven by Havok's animation data.

**Interface to the Chains**

The constraint infos of `hkpStiffSpringChainData` and `hkpBallSocketChainData` contain only pivot points and, in the former one's case, spring length.

To setup constraint infos in `hkpPoweredChainData` which need to be additionally specified are:

- The target orientation of the linked bodies (`hkQuaternion` *m_aTc*): it specifies the relative orientation of the second (child) body in the first (parent) body's space. The second body is the one added to the chain instance's list later.
- The three motors to be used:
  - The motors used to enforce the 3-dimensional orientation constraint.
  - The motors available for use with the `hkpPoweredChainData` are `hkpPositionConstraintMotor` and `hkpVelocityConstraintMotor`.

You may also make use of the following additional parameters (although they should only be used after a good understanding of the above parameters):

- `hkBool` *m_switchBodies*: this allows you to reverse the parent-child relation. When this flag is set to true, the *m_aTc* variable represents the first body's target orientation in the second body's space.
- `hkQuaternion` *m_bTc*: this allows you to specify a reference frame for the child body. The motor's space is aligned to that reference frame. This is useful when you want to vary motors' strengths or disable one of them and allow rotation along a specific axis. (See the "Motor Chain Demo/Test constraints space alignment" demo).

All `hkConstraintChainData` have their own *m_tau*, *m_damping*, and *m_cfm* settings, independent of the global `hkpWorld`'s solver settings.

**Note:**

`hkpConstraintChainInstance` also references a `hkpConstraintChainInstanceAction`. This is an internal action and it is added to the `hkpWorld` together with a `hkpConstraintInstance`. The action's purpose is to allow for the `hkpConstraintChainInstance` to keep track of its entities (as the `hkpConstraintInstance`'s interface only can reference two bodies). This action ensures that all the entities will be treated as a group by Havok and always be simulated in the same simulation island.

### Limitations of Constraint Chains

- Powered chains do not support limits. However, you can use the `hkpHingeLimitsData`, and `hkpRagdollLimitsData` to use limits with chains.
- You cannot remove or disable a constraint in a chain easily.
- The chains are not handled during TOI events. As a workaround you can create normal constraints on top of the chain constraints and flag those as TOI constraints.
- The chain is solved perfectly, however it still works in an iterative solver. If any body in a chain is constrained by a non-chain constraint, this will introduces softness at that body. Based on the random order of constraint solving, this softness can either appear in the chain or at the other constraint:
  - A: a nice stable stretching between each pair or
  - B: if you pull harder, a unstable stretching, where some bodies jitter between several positions.
  - The chain constraints solve A perfectly, but they still suffer from B. And because the forces are much higher, B will be more likely. To reduce this effect, **increase inertia** (and mass) of the jittering bodies.
- Powered chain constraints can be slow. Most of the time, chains are really fast. However, if you have a powered chain, where many motors have their maximum force limits breached, these chains become slow. For each breached motor, we have to recalculate the whole LU decomposition of the chain again. If there are 200 bodies and 100 motors are breached, the system has to perform 200*100*n matrix6 multiplies.
  - As a workaround, if your motor is very weak and you expect many maximum forces to be breached, do not use powered chains, but use ball socket chains instead with a normal powered ragdoll constraint on top.

### What Demos to Look At

Demos using the constraint chains technology:

`BallAndSocketRopeDemo`: This demo shows a rope with weights at each end that you can drag around and crash into obstacles.

In the `Physics2012/Api/Constraints/Chain` directory:

- `SwingingRope`
  - Ball-and-Socket chains variant:
    - Illustrates stable chains of 100+ bodies.

        - The demo also displays a rope solved with the old Havok constraints for comparison.
    - Stiff spring chains variant: note that these ropes are much less stable then their ball-and-socket counterparts

- `Bridge`: A bridge of a hundred or so elements constructed with two ball-and-socket chains.
- `ChainNet`: Demos imitating cloth, using array of bodies interconnected with a number of chains constructed at 90 degree angles.
- `MotorChain`:
    - Illustrates accuracy of motors in the chain. You can modify the strength of all the motors and drag the boxes with the mouse, to get a better feeling for the behavior.
    - Velocity control.
    - Enforce hinge.
    - Test constraints space alignment.
- `Spider`: Uses the chains to power the spider legs.

`Animation/Api/Ragdoll/Blendtest`

- This shows a powered ragdoll fed with an animation. Old Havok constraints hardly follow the requested animation. The new chains yield a very accurate match. Note how the characteristics of the movement change as you control the maximum forces of the ragdoll's motors. There's also a variant which illustrates how to use ragdoll and hinge limit constraints on top of constraint chains.

**Performance Issues**

The computational complexity of chains is linear in respect to the number of connected bodies. `hkpPoweredChainData` is several times more expensive than `hkpBallSocketChainData`, and `hkpBallSocketChainData` is several times more expensive than `hkpStiffSpringChainData`. Use of large chains in large simulation islands may slow down the solver due to memory caching issues.

Chains have a considerable memory overhead and store their intermediate results together with other solver data. When using `hkpPoweredChainData` be aware that their performance will drop noticeably when motors' maximum forces are breached.

**Powered Chain Mapper**

The `hkpPoweredChainMapper` facilitates manipulation of systems of bodies linked by a number of `hkpPoweredChainData` instances. The mapper facilitates modification of target orientations for motors, which is useful when feeding the constraint chains with animation data. It also allows you to set maximum forces of the motors during simulation, which allows you to influence the characteristics of interaction with the environment.

**Setup**

The mapper is intended to be initialized from a previously setup physical system consisting of rigid bodies and normal Havok constraints (which connect exactly two entities, each), e.g. a ragdoll.

```
// Set options for the mapper building function
hkpPoweredChainMapper::Config config;
config.m_createLimitConstraints = true;
config.m_cloneMotors = true;

// Supply all the chains that we want to use
hkArray<hkpConstraintInstance*> allConstraints = theConstraintsThatWeWantToUse;

// Specify a list of start-end point pairs
hkArray<hkpPoweredChainMapper::ChainEndpoints>& pairs;
hkpPoweredChainMapper::ChainEndpoint& pair = pairs.expandOne();
pair.m_start = someEntityStart;
pair.m_end = someEntityEnd;

// Optionally you can pass an array which will be filled with constraints
// that were not used to construct chains.
hkArray<hkpConstraintInstance*>* unusedConstraints;

// Create the mapper
hkpPoweredChainMapper* mapper = hkpPoweredChainMapper::buildChainMapper(config,
                                                                        allConstraints,
                                                                        pairs,
                                                                        unusedConstraints);

// Use the utility to add all the mapper's consraints to the world
hkpPoweredChainMapperUtil::addToWorld(world, mapper);
```

To create a mapper you need to specify an initial set of normal constraints which simply tell the mapper creation function which bodies can be linked in a sequence and a list of pairs of `hkpEntity`, which specify the start and end points for each chain created. All the constraints which are in the created chains must be of the 'powered' type, i.e. have motors which can be extracted by the `hkpConstraintUtils::getConstraintMotors()` method. Finally, all the specified start-end entity pairs must also be valid.

**Note:**

If there's more than one constraint connecting one pair of bodies, then it is unclear which constraint's information will be used when creating the chain.

### Usage

As the chains may overlap, a single link index of the mapper may refer to more than one link in a number of chains. Setting a property of one link modifies information in all the related chains at once. The index-to-link mapping is based on the ordering of the original constraints in the list passed to the mapper-building function. The n$^{th}$ link connects the same bodies as the n$^{th}$ constraint in that original list.

```
// To weaken a constraint you have to set all its three motor's forces to new values,
// as you can assign a separate motor to each dimension. If you use the same motor for all three
// dimensions then there is no need for the loop
const hkReal newStrength;
for (int i = 0; i < 3; i++ )
{
    mapper->setForceLimits(linkIndex, i, -newStrength, newStrength);
}

// And to set a new target orienation
hkQuaternion requestedOrienation = hkQuaternion::getIdentity();
mapper->setTargetOrientation(linkIndex, requestedOrientation);
```

### 2.2.2 Simulating a Physics 2012 World

Every Havok Physics 2012 simulation has at least one simulation world, an instance of the **hkpWorld** class. `hkpWorld` acts as a container for all physical objects in the simulation. It is also used to step the simulation forward in time.

#### 2.2.2.1 Creating a World

The **hkpWorldCinfo** structure contains many parameters used during construction of the simulation world. Please refer to the reference manual entry for `hkpWorldCinfo` for a complete description of all parameters. Let's take a look at some of the main parameters that can be set:

- **Continuous vs Discrete Simulation**

  You can specify whether to use continuous (`SIMULATION_TYPE_CONTINUOUS`) or discrete (`SIMULATION_TYPE_DISCRETE`) collision detection for the simulation using the `hkpWorldCinfo::m_simulationType`. Continuous simulation is the default and recommended setting for collision detection. Discrete simulation should only be used if performance is a significant problem. Multithreaded simulation (`SIMULATION_TYPE_MULTITHREADED`) also makes use of continuous collision detection.

  Please note in multithreaded mode that Havok provides multithreaded access checks to ensure that the world is correctly synchronized when access is needed (usually during collision callbacks or asynchronous queries).

- **Gravity**

  An `hkpWorld`'s gravity is applied to all its simulated objects. The default value for this parameter reflects "real-world" gravity (-9.8 in the Y direction, which is Havok's default up axis). Alternative gravity settings can be used for your game world: you can define gravity along an alternative axis (your engine might use the Z axis as up instead of Y, the Havok default). In space, you would define gravity to be 0 along all three axes. Gravity is specified as an `hkVector4`. More complicated gravity models can be implemented using Actions.

- **Solver**

  The Havok simulation uses a solver to calculate the necessary changes to reduce error for all constraints in the simulation. As a number of different parameters control the solver, the `hkpWorldCinfo` has helper methods to allow you to set the solver types. Each solver type provides a different speed/accuracy balance for the simulation. The solver behavior depends on how many iterations the solver goes through in each step when calculating the changes, and the tau and damping values used by the solver.

  You can read more about the solver's role in the simulation in the Solver Introduction. As a rule, the higher the number of iterations, the better the accuracy. If you are feeling adventurous you can set the solver parameters directly, otherwise see the reference manual entry for `hkpWorldCinfo` for a complete list of solver types.

The following example shows how to create an `hkpWorld`. As you can see, you simply create the `hkpWorldCinfo`, assign the values you want to its members, and then pass it to the `hkpWorld` constructor.

```
hkpWorldCinfo info;
info.m_gravity.set( 0,-9.8f,0);
info.setBroadPhaseWorldSize( 150.0f );
info.setupSolverInfo( hkpWorldCinfo::SOLVER_TYPE_4ITERS_MEDIUM );
info.m_simulationType = SIMULATION_TYPE_CONTINUOUS; // Default.

m_world = new hkpWorld( info );
```

Once you have created the `hkpWorld`, you can use `hkpWorld` functions to get and set all the `hkpWorldCinfo` values.

### Broad Phase

Each `hkpWorld` maintains a broad phase structure, which is mainly used to determine which entities may overlap and therefore warrant additional mid or narrow phase work during collision detection. It also provides additional functionality such as detecting entities which have "left the world", or performing entity visibility queries.

- **Type**

  A number of broad phase implementations are available. The choice that gives the best performance will vary depending on the use case. In general, it is a good idea to try out the various broad phase types to see which behaves best for a particular case. The options are as follows:

  - `BROADPHASE_TYPE_SAP`

    This uses 3-axis sweep-and-prune and is the default option. It provides fast incremental updates which happen during normal rigid body simulation. However, adding and removing bodies, in addition to collision queries, can become slow for densely populated worlds.

  - `BROADPHASE_TYPE_TREE`

    This uses a spatial tree to provide fast insertions and queries, particularly for densely populated worlds. However, incremental updates are slower than the sweep-and-prune option.

  - `BROADPHASE_TYPE_HYBRID`

    This uses a combination of the tree and sweep-and-prune options to provide fast updates and fast queries. This is the recommended option for general use cases. However, there are some performance and memory overheads required to keep both structures synchronized. An additional 72 bytes is required per entity and, compared to using sweep-and-prune by itself, in the worst case the tree broad phase will add about 10% to the CPU/PPU cost when an entity is added and removed, and about 20% to the broad phase update.

- **Size**

  The size of the broad phase box can be configured by setting `hkpWorldCinfo::m_broadPhaseWorldSize`. The default is a box with sides of length 1,000 (the equivalent of a 1 kilometer cube if you using meters, the recommended scale). It is important to realize that, for all broad phase types that use sweep-and-prune, there are performance implications to not setting the broad phase to a size appropriate for your scene. You can find out more about the broad phase and the effect of changing its size in the Optimizing Broad Phase Performance section.

- **Border Callback**

  The `hkpBroadPhaseBorder` class (provided in the `hkpUtilities` module) can be used to monitor objects leaving the broad phase. This is not used by default; you need to create an instance of it yourself. `hkpBroadPhaseBorder` has a `maxPositionExceededCallback()` function that's called

whenever an object leaves the broad phase world. By default, it removes the entity from the world but you may implement your own version.

- **User Objects**

  The tree broad phase supports the insertion, update and removal of user objects (i.e. `hkpPhantom`). User objects are not part of the physics simulation, but are reported by all world queries (ray casting, linear casting, etc.). In addition, they do not have any impact on the physics simulation speed, thus making them ideal for a lot of tasks, such as frustum culling of game objects (see demo: "`Physics2012/Api/Collide/Broadphase/Culling/UserObjects`").

  User objects related methods:

  - `addUserObjects(int numObjects, hkpBroadPhaseHandle*const* handles, const hkAabb* aabbs);`

    Add a set of user objects to the tree broad phase. Handles and AABBs have to be provided.
  - `updateUserObjects(int numObjects, const hkpBroadPhaseHandle*const* handles, const hkAabb* aabbs, const hkVector4* velocities=HK_NULL);`

    Update a set user objects from the tree broad phase. Handles and new AABBs have to be provided and, optionally, velocities can be provided to optimize update for moving objects.
  - `removeUserObjects(int numObjects, hkpBroadPhaseHandle*const* handles);`

    Remove a set of user objects from the tree broad phase. Only handles have to be provided.

- **Convex Query**

  The tree broad phase also has the ability to perform convex queries. When given a set of planes defining a convex region (not necessarily closed), it will enumerate all entities inside or crossing this region. This can be used for frustum culling (see demo: "`Physics2012/Api/Collide/Broadphase/Culling/Rigid bodies`"). **Note** : this convex query is not currently supported on PlayStation®3 SPU.

### Advanced Simulation Parameters

There are also a number of advanced parameters for the simulation that can be tweaked. While the default values are fine in most situations, they can be modified to obtain a specific behavior. We will cover here some of the commonly used parameters,

- **Collision Tolerance**

  The collision tolerance is an optimization setting. Contact points between two objects are created once the distance between them drops below the collision tolerance. This variable allows you to specify a global collision tolerance value for the `hkpWorld`. The default is 0.1 (or 10 cm if you are using the recommended scale of meters). You can find out more about this value and its effect in the Collision Tolerance section of the Optimization chapter.

- **Deactivation**

  Most objects in a normal scene are stationary and thus they do not need to be updated in the same manner as moving objects. A lot of the data can persist across simulation steps. The world can identify objects that should be deactivated by monitoring the relative movement they make as time progresses. It monitors the movement based on two frequencies to reduce the chance of deactivating objects that are oscillating the same as the sampling frequency. The `hkpWorldCinfo::m_enableDeactivation` flag turns the monitoring on and off.

  To save on CPU usage, we check each object not every frame but every $4^{th}$ frame (high frequency checks) and as an extra backup deactivation check every $16^{th}$ frame (low frequency check). If all the objects in an independent group of objects are flagged for deactivation, this group will

deactivate. Normally this group is one island, however for performance reasons the engine might merge independent groups into a 'sparse' island.

Further details on deactivation are available in the Optimization chapter.

- **Linear Cast Accuracy**

  Linear casting, also known as shape casting, is the ability to sweep a convex shape through the world and find the points of collision. This can be very useful for accurate proximity checks for characters and movement planing. The casting is an iterative algorithm and so the max number of iterations is exposed to allow for performanceâĘŤaccuracy tweaking along with the minimum distance tolerance value for the cast, *m_iterativeLinearCastEarlyOutDistance*. If you change the collision tolerance then this number should also be adjusted, usually to 10% of the collision tolerance.

  Please refer to the Linear Casting chapter for a complete understanding of this collision query and the world parameters that affect it.

- **Continuous Simulation Parameters**

  Continuous simulation and collision detection use a group of internal safety and optimization parameters. They are mainly derived from a few parameter exposed in `hkpWorldCinfo`.

| Parameter | Purpose, Values |
|---|---|
| `m_simulationType` | Selects type of simulation (as discussed earlier). Available values are `SIMULATION_TYPE_DISCRETE`, `SIMULATION_TYPE_CONTINUOUS` (default), `SIMULATION_TYPE_MULTITHREADED`. |
| `m_expectedMaxLinearVelocity` | Assumed maximum linear velocity for all bodies. This does not set the linear velocity in the objects. If later on, a rigid body is significantly faster than this value, it might tunnel through other objects even if its *qualityType* is set to critical. It defaults to 200 meters/second. |
| `m_expectedMinPsiDeltaTime` | Minimum expected physical simulation delta time. Defaults to 30 Hertz. |
| `m_contactPointGeneration` | Allows contact points to be created at negative distance, which speeds up the simulation but doesn't work well with small objects. For small object this setting may cause TOI events being generated massively (instead of regular constact points created before interpenetration of objects) and therefore cause a drop in frame frequency. This setting is enabled when hkpWorld::m_contactPointGeneration is set to `hkpWorldCinfo::CONTACT_POINT_REJECT_DUBIOUS` or `hkpWorldCinfo::CONTACT_POINT_REJECT_MANY` (the default setting). |

Table 2.7: Continuous Simulation Initialization Parameters

### Units and Scaling

Havok is designed to work at the scale of objects that we interact with in the real world (cars, people, buildings etc). By default, the Havok engine works in **meters** and **kilograms** . A velocity of length 1 implies a velocity of 1 meter per second, and a unit cube is one cubic meter.

If your existing game engine uses an arbitrary scale internally, a common method should be used to scale (up or down) the values the game engine passes to Havok at runtime (e.g. by using custom macros). This ensures that the physics simulation is processed using values within the preferred range while causing minimal change to the existing game code.

Be very careful with scale, particularly when using 3D modelers. Often modelers have their own mechanisms for displaying units in dialog boxes (e.g. 3ds Max allows specification of the units used and automatically converts all values displayed to those units. However, internally it always works in inches, including when exporting geometry).

While it is possible to change the Havok scale, it is **not recommended** . If you are considering this option, please contact support. The `WorldScaleAndOffset` demo also illustrates how the choice of world scaling affects the simulation.

There are several hard coded numerical tolerances in the engine which are not exposed to the user. The unit scale of the engine cannot be changed by a large factor (e.g. centimeter units or kilometer units cannot be used). However, by suitably adjusting some externally available parameters, unit scaling can be changed on a smaller scale (to obtain a unit scale in feet for example). Most of these parameters are in the `hkpWorldCinfo` structure. In each case these parameters must be scaled by the required factor. In the case of feet units, the values must be set to 3.28 times their default values. The default values are found in `hkpWorldCinfo`.

*Simulation Parameters:*

- *m_gravity*
- *m_expectedMaxLinearVelocity*
- *m_broadPhaseWorldAabb.m_min*, *m_broadPhaseWorldAabb.m_max*

*Collision Detection Internal Tolerances:*

- *m_collisionTolerance*
- *m_iterativeLinearCastEarlyOutDistance*

In addition to these `hkpWorldCinfo` parameters, the following shape-based values must also be changed:

- The *radius* of any convex shapes created, either on construction or using the method `setRadius()`.

#### 2.2.2.2 Populating a World

Now that our simulation world is created, we need to add elements such as rigid bodies, phantoms, constraints, actions and listeners to take part in the simulation. `hkpWorld` provides a number of functions for adding and removing different types of simulation element and for updating their behavior.

**Rigid Bodies**

**Adding and removing Rigid Bodies**

All simulated physical objects (such as a rigid body) are known as *entities*, and inherit from the base class `hkpEntity`. It provides a number of functions and data members common to all entities.

To add a rigid body to the simulation world, we use `hkpWorld::addEntity()`, as in the snippet below.

```
hkpRigidBody* pRigidBody = new hkpRigidBody(...);
m_world->addEntity( pRigidBody );
pRigidBody->removeReference();
```

Here, `removeReference()` is called after the `hkpRigidBody` has been added to the `hkpWorld` because rigid bodies are reference counted (`hkpRigidBody` inherits from `hkReferencedObject`) and when added to the world, the world keeps a pointer to the `hkpRigidBody` and increases the body's reference count.

Calling `removeReference()` here gives full ownership of the body to `hkpWorld`. (More information about Havok's reference counting is available in the Havok Base Library chapter.)

To remove a `hkpRigidBody` from the simulation, call:

```
hkpWorld::removeEntity( pRigidBody );
```

This decrements the reference count, deleting the body if there are no other references to it.

There is some CPU overhead associated with adding and removing bodies. If many bodies are being added/removed at the same time, then the more efficient `hkpWorld::addEntityBatch()`, `hkpWorld::removeEntityBatch()` functions should be used.

### Activation State

A second (optional) parameter can be provided to `hkpWorld::addEntity()` which specifies whether the `hkpRigidBody` is added in an active or inactive state. The default `HK_ENTITY_ACTIVATION_DO_ACTIVATE` value always activates the body. The `HK_ENTITY_ACTIVATION_DO_NOT_ACTIVATE` value adds the body in the inactive state. However, note that the body will be instantly activated by the simulation if it overlaps with another active body.

Rigid bodies can also become deactivated over time after they are added in the world. Please refer to the Activation section for more information.

When removing an `hkpRigidBody` from an inactive `hkpSimulationIsland` containing other objects, you may want to manually reactivate the island. If a box is removed from the bottom of a stack in an inactive island, the rest of the stack will remain floating until you manually activate the island. This can be done by activating the body you want to remove before you remove it.

### Changing the Motion State

`hkpRigidBody` provides a number of functions that you can use to apply physical impulses, forces, and torques to a rigid body, as well as changing its velocity.

Changing the motion type allows you to turn a simulated `hkpRigidBody` into a keyframed one and vice versa, or a dynamic body into a fixed one (and vice versa). This is accomplished with a call to `hkpRigidBody::setMotionType()`. When changing the motion state you must pass activation information (like in `addEntity()` function). The parameter doesn't matter when changing into fixed state.

For more information on keyframed rigid bodies, please refer to the keyframed objects section.

### Converting to a Fixed Object

Static objects do not collide with other static objects, therefore if you change the state of the object to or from static, the Havok engine has to requery the collision detection engine, which incurs a significant CPU hit. To avoid this CPU hit you may suppress the collision detection query by passing the `HK_UPDATE_FILTER_ON_WORLD_DO_NOT_QUERY_FOR_NEW_PAIRS` parameter to the function. Use this feature with caution.

**Warning:**
Use the *checkBroadPhase* parameter with caution as it disables creation of new collision agents. Therefore, it is safe to use this parameter when changing motion type to fixed if we expect the body to keep all its agents except the ones linking to other fixed bodies. It is not safe to use this parameter changing the motion type back to dynamic, when the body needs to collide immediately with fixed bodies in proximity.

### Converting to a Dynamic Object

Having a dynamic object requires having an inertia tensor. As static or keyframed objects do not have this information, you cannot (in theory) convert static or keyframed objects to dynamic ones.

In practice we can actually use a trick: create the object as a dynamic one first, then set its state to be static or keyframed before adding it to the world. In this case the object actually stores its initial inertia tensor, which it uses when it is converted back to dynamic.

Changing an object's motion state is particularly useful when applying actions. You can find out more about this in the Actions chapter.

### Setting Positions and Rotations

The rigid body's world space position and rotation can be directly set using `hkpRigidBody::setPosition()` and `hkpRigidBody::setRotation()`, respectively. Both can be updated simultaneously by setting the body's transform (using `hkpRigidBody::setTransform()`) or using the helper `hkpRigidBody::setPositionAndRotation()`.

Setting a body's position and orientation directly should be reserved for specific situations (such as initial positioning, or when warping an object to another location). It is generally encouraged to change a body's transform using more indirect, yet simulation-friendly means such as changing the velocities or applying forces and impulses. There are two reasons for this:

- Setting the transform is a critical operation that triggers a costly broad phase update.
- Updating the transform directly can lead to interpenetration artifacts if an object is suddenly placed inside another.

### Changing Velocities

The `hkpRigidBody::setLinearVelocity()` and `hkpRigidBody::setAngularVelocity()` functions are used to specify new linear and angular velocities for a rigid body during simulation.

The following code from the `MarbleAction` (used in `PyramidDemo`) shows the marble's "brake". The `setLinearVelocity()` and `setAngularVelocity()` functions are used to set the marble's velocity to zero when the brake button is pressed.

```
if (m_brakePressed)
{
    hkVector4 zero;
    zero.setZero4();
    rb->setLinearVelocity(zero);
    rb->setAngularVelocity(zero);
    setBrakePressed(false);
}
```

### Applying Forces and Torques

Applying a force to an `hkpRigidBody` over time has an effect on the velocity of an object (accelerating or decelerating it). Applying a torque has a similar affect on the angular velocity of the object.

You can apply a force to an `hkpRigidBody`'s center of mass or to a specified point using its `applyForce()` functions. Similarly, you can apply a torque to a body with `applyTorque()`.

Any forces and torques that you specify for an `hkpRigidBody` during a simulation step are applied immediately as impulses. The current's step time length is needed to calculate the value of impulses, therefore you must also pass the *hkStepInfo* parameter to the function.

You specify a force as an `hkVector4`, where the values of the X, Y, and Z elements indicate the strength of the force in the corresponding axis direction, in world space. For instance, if you want to accelerate the body in an upward direction, you specify a value in the y element of the `hkVector4`, as in the example below. As with the linear impulse, the change in the body's linear velocity after the simulation step is inversely proportional to the body's mass.

As a force causes a change in the velocity of a body over time, the change in the object's velocity is dependent on the simulation delta time value. For example, if you apply an upward force to an object every 0.1 seconds (or 10 times a second) and then apply the same upward force to the same object every 0.0166 seconds (or 60 times a second), you would end up with slightly different results within the simulation. This is an artifact of numerical integration.

See `hkpRigidBody`::applyTorque() for more details on how that function operates.

In the following code from the `FountainDemo`, upward forces are applied to all objects that fall inside the fountain:

```
for (int i = 0; i < m_phantom->getOverlappingCollidables().getSize(); i++ )
{
    hkVector4 force( 0,70,0 );
    hkpRigidBody* rb = = hkpGetRigidBody(m_phantom->getOverlappingCollidables()[i]);
    if (rb)
    {
        rb->applyForce( stepInfo, force );
    }
}
```

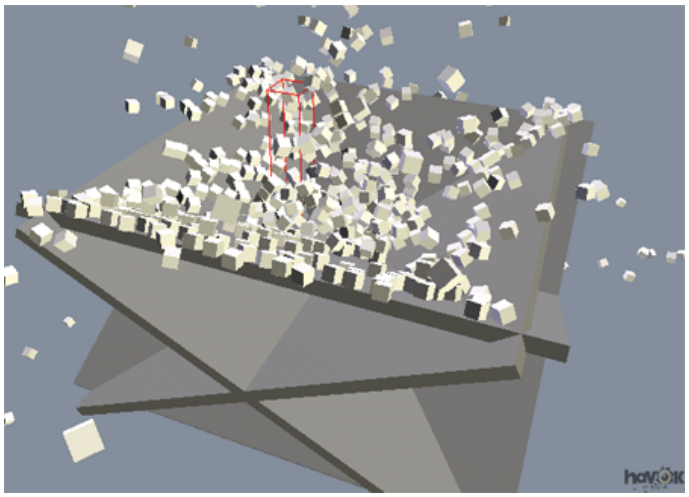


Figure 2.48: The Fountain example in action

**Impulses**

Applying an impulse has the immediate effect of changing the rigid body's velocity - think of a car driving into a wall! An impulse delivers all of the specified force to an object instantly (think of it as a force which has been pre-multiplied by a time step). If you had a box in a room and wanted it to jump, you would apply an upward impulse to it. If you wanted it to float, you would apply a force to it over time.

The `hkpRigidBody` class has several functions for applying impulses to rigid bodies. You can use `applyLinearImpulse()` to apply an impulse at the object's center of mass, or `applyPointImpulse()` to apply a linear impulse at a specified point. The `applyAngularImpulse()` function lets you apply an instantaneous change in angular velocity (in world space) around the center of mass.

For a linear impulse you specify the impulse as an `hkVector4`, where the values of the X, Y, and Z elements indicate the strength of the impulse in the corresponding axis direction, in world space. So, for instance, if you want to change the body's velocity in an upward direction, you specify a value in the Y element of the `hkVector4`, as in the example below. The change in linear velocity is inversely proportional to the body's mass (the difference in velocity is the impulse multiplied by the inverse mass).

You also specify an angular impulse as an `hkVector4`. However, in this case the normalized vector specifies the axis around which you want the body to move, and the magnitude of the vector specifies the strength of the impulse. For an angular impulse, the difference in angular velocity is inversely proportional to the body's inertia (or to be precise, proportional to the inverse of the inertia tensor matrix).

If you apply an impulse to a point other than the center of mass, it changes the body's linear velocity as described for a linear impulse at the center of mass, but also affects its angular velocity. The change in angular velocity depends on how far the point is from the center of mass as well as the body's inertia.

The following example, again from the `MarbleActionDemo`, shows the use of an upward impulse to make the marble jump when the jump button is pressed. As you can see, the body's own mass is used when specifying the impulse, meaning that this method will have similar results with marbles of different masses.

```
if (m_jumpPressed && ((m_lasttimeCalled - m_lastJump) > 1.0f))
{
    m_lastJump = m_lasttimeCalled;
    hkVector4 imp(0, m_body->getMass() * 6, 0);
    m_body->applyLinearImpulse(imp);
    setJumpPressed(false);
}
```

The `MarbleAction` also uses impulses to roll and turn the marble. You can see the complete `MarbleActionDemo` and `PyramidDemo` in the demos provided with your Havok installation. In addition to exploring the code, try playing the game and knocking down the pyramid with the marble!

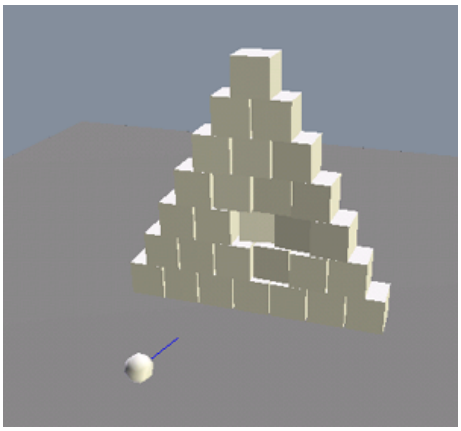


Figure 2.49: The Pyramid example in action

### Keyframed Objects

Keyframed objects are manually controlled rigid bodies whose motion can't be affected by collisions with other objects. A keyframed object is treated by the simulation as if it had infinite mass. However, unlike a fixed object, a keyframed object can be moved around.

A keyframed body with a velocity of zero behaves exactly like a fixed `hkpRigidBody`. However, if you set a velocity for it, it will move at that velocity. Objects will bounce off and roll over a keyframed object but will not affect its motion, and impulses and forces will not change its velocity.

Keyframed objects are useful for modeling the behavior player controlled objects (such as the main character), or moving platforms or even pre-animated cinematic scenes (like the tumbling motion of a vehicle after it explodes).

**Note:**
You can also directly set the position of a keyframed object, but great care must be taken to avoid accidentally causing a major interpenetration between it and another rigid body.

### Changing the Motion Type of an Object

Dynamic objects can be converted into keyframed (or fixed) objects and back during the simulation, allowing you to gain or relinquish full control over the motion of objects on the fly within the physics simulation. This can be useful for any number of scenarios (such as a character picking up and dropping objects for example).

To change the motion type of an object, call `hkpRigidBody::setMotionType(hkpMotion::MotionType)` with the desired motion type. To convert a dynamic object into a keyframed object, call `hkpRigidBody::setMotionType(hkpMotion::MOTION_KEYFRAMED)`. To return it to its movable state, use `setMotionType(hkpMotion::MOTION_DYNAMIC)`.

For more examples, refer to the SetKeyframed demo.

**Note:**
As mentioned earlier, **you cannot convert a body that was originally created keyframed or fixed into a movable object** .

### Collision with Fixed Objects

By default, collisions between keyframed objects and fixed objects are **disabled** . If you are really interested in the contact point data, you can enable it by editing the `hkpCollisionDispatcher::m_collisionQualityInfo` table. However, this is not recommended as it will generate many useless contact points, which only cost CPU and memory.

The `KeyframedInteractions` demo illustrates the behavior of keyframed objects under various scenarios.

### Keyframed vs. Fixed

You should favor using fixed objects over keyframed objects if they will never be moved. Fixed objects are more efficient as Havok can perform certain optimizations when they are added to the world (as it knows that they will always remain fixed). Fixed objects are also faster to use as they do not affect simulation islands.

### hkpKeyFrameUtility

`hkpKeyFrameUtility` contains helper functions to calculate and apply velocities to keyframed bodies so that they will follow a chosen position/orientation and velocity.

This can be particularly useful when using a keyframed object with a character controller. The keyframed object's position and orientation could correspond to the game character's position and orientation, allowing the character to realistically interact with the physics world. You can find out more about using keyframed objects in this manner in the Character Control chapter.

### applyHardKeyFrame()

This function changes the linear and angular velocities of a body so that it will move to a particular position and orientation in a specified time. You must provide the target position (as an `hkVector4`), the target rotation (as an `hkQuaternion`), the *inverse* delta time (as an `hkReal`) and the keyframed body. This will immediately update the body's velocities to position the body in the desired positions.

### applySoftKeyFrame()

Using a soft key frame applies the desired changes in a more approximate "blended" manner than the hard key frame approach.

The function allows you to specify more detailed acceleration guidelines for moving the keyframed body. As well as the keyframed body and the inverse delta time, you need to provide the following:

- **KeyFrameInfo**

  A KeyFrameInfo struct contains the current keyframe position, orientation, angular velocity, and linear velocity. It also provides functions for calculating the angular and linear velocities if you only have the current and target positions and orientations. The position must be specified as the position of the center of mass of the object rather than the local origin. See the reference manual entry for `hkpKeyFrameUtility` for more information.

- **AccelerationInfo**

  This structure specifies the following:

  - **Linear position factor**
    This governs how quickly you want the body to reach the target position. You specify this as 1/time. So, for instance, if this value is 2, then `applySoftKeyFrame()` will set the body's velocity so that it will reach the position of the next keyframe in half a second. Valid values are between 0 and 1/deltaTime.
  - **Angular position factor**
    This governs how quickly you want the body to reach the target orientation. You specify this as 1/time. So, for instance, if this value is 2, then `applySoftKeyFrame()` will set the body's velocity so that it will reach the orientation of the next keyframe in half a second. Valid values are between 0 and 1/deltaTime.
  - **Linear velocity factor**
    This governs how quickly you want the body to reach the target linear velocity. You specify this as 1/time. So, for instance, if this value is 2, then `applySoftKeyFrame()` will set the body's velocity so that it will reach the linear velocity of the next keyframe in half a second. Valid values are between 0 and 1/deltaTime.
  - **Angular velocity factor**
    This governs how quickly you want the body to reach the target angular velocity. You specify this as 1/time. So, for instance, if this value is 2, then `applySoftKeyFrame()` will set the body's velocity so that it will reach the angular velocity of the next keyframe in half a second. Valid values are between 0 and 1/deltaTime.
  - **Max linear acceleration**
    If the linear velocity changes required by `applySoftKeyFrame()` exceed this acceleration, they are clipped against this value.
  - **Max angular acceleration**
    If the angular velocity changes required by `applySoftKeyFrame()` exceed this acceleration, they are clipped against this value.
  - **Max allowed distance**
    If the distance between object and keyframe gets bigger than the max allowed distance, the object is immediately "warped" to the correct keyframe position/orientation, and corresponding velocities.

### Constraints

Once you have created a constraint, you can add it to the simulation world using the `hkpWorld::addConstraint()` method:

```
hkpBallAndSocketConstraintData* bsData = new hkpBallAndSocketConstraintData();
hkpConstraintInstance* bsInstance = new hkpConstraintInstance( bodyA, bodyB, bsData );

m_world->addConstraint( bsInstance );
bsData->removeReference();
bsInstance->removeReference();
```

Or using a convenience function in `hkpWorld`:

```
hkpBallAndSocketConstraintData* bsData = new hkpBallAndSocketConstraintData();
m_world->createAndAddConstraintInstance( bodyA, bodyB, bsData )->removeReference();
bsData->removeReference();
```

Similarly, you can remove constraints using `removeConstraint()`.

Like entities, shapes, and actions, constraint data and instances are referenced objects. A call to `removeReference()` must be made once you are finished with them (usually after adding them to the world, as in the example above). Havok will delete each constraint once it is no longer in use. More information about reference counting is available in the Havok Base Library chapter.

### Actions

Actions provide a simple and effective mechanism to control the *behavior* of entities during a Havok simulation. They can range from simple (such as an "anti-gravity" action to make an object float) to complex (such as an action that controls the flight of a helicopter).

In addition to reading this section, take time to look at and play with the actions used in the dynamics/actions demos and the `PyramidDemo`, as well as the example actions provided in `hkpUtilities/actions`.

### About Actions

#### hkpAction::applyAction()

`hkpAction` is an abstract class. Every child class of `hkpAction`, no matter how simple or complex, has an `applyAction()` function that performs the "work" of the action. Once an `hkpAction` is added to the simulation world, `applyAction()` is automatically called during every simulation step.

**Note:**
Each `hkpAction` is automatically applied so long as its bodies are active. If bodies becomes inactive, any attached actions are not applied. If the bodies re-activate, the actions are once again applied to the bodies.

The `applyAction()` function is passed an `hkStepInfo` data structure by the simulation each time it is called. `hkStepInfo` stores the simulation's current delta time and inverse delta time, and can be queried by the action. For example, in the `UnaryAction` demo the force applied by `AntiGravityAction` varies depending on the simulation time step:

```
inline void AntiGravityAction::applyAction( const hkStepInfo& stepInfo )
{
    // Apply the antigravity force to the body.
    // The force ranges from [-2 * gravity to 0], modulated by a sine wave as a function of time
    // so sometimes it is stronger than gravity, sometimes less strong.
    m_currentTime += stepInfo.m_deltaTime;

    // Scaling factor that varies over time.
    hkReal scale = 1.0f + hkMath::cos(2 * m_currentTime);

    // Vector representing force to be applied.
    hkVector4 force;

    // Apply the antigravity force in the opposite direction to gravity. Its strength is
    // determined by multiplying m_gravity (also a vector) by scale.
    force.setMul4( -1 * scale, m_gravity);

    // This function call actually applies the force to the rigidBody.
    m_body->applyForce( force );
}
```

### Action Types

All actions implement the `hkpAction` interface. The Havok SDK provides three classes derived from `hkpAction` that can be used when creating custom actions. They are:

- `hkpUnaryAction`

  For actions that apply to a single rigid body. The motor action (`hkpMotorAction`) described later in this chapter is an example of a **unary** action. It can be used to spin a rigid body around an axis.

- `hkpBinaryAction`

  For actions that apply to pairs of rigid bodies. The spring action (`hkpSpringAction`), also described later in this chapter, is an example of a **binary** action. It can be used to attach two rigid bodies together.

- `hkpArrayAction`

  For actions that apply to many rigid bodies. A magnet could be represented by an `hkpArrayAction`. Care must be taken to use the `hkpArrayAction` only when appropriate. See the section on multiple actions versus single actions below.

These classes provide basic functionality for actions and have data members for the rigid bodies each action applies to. `hkpAction` also has a virtual function, `entityRemovedCallback(hkpEntity* entity)` triggered when the entity associated to this action is removed from the simulation world. In all three implementations, `entityRemovedCallback()` removes the action from the simulation if *any* of the rigid bodies it is applied to are removed.

This behavior may be undesired, especially for an `hkpArrayAction`. To avoid this, re-implement `entityRemovedCallback()` for `hkpArrayAction`, or derive directly from `hkpAction`.

### Adding and Removing Actions

To add an action to the simulation, use `hkpWorld::addAction(hkpAction* action)`:

```
// Add antiGravityAction to the world. From now on antiGravityAction will automatically
// be applied to its body (boxRigidBody) during integration so long as boxRigidBody is
// active. If boxRigidBody becomes inactive, antiGravityAction is not applied. If
// boxRigidBody reactivates, antiGravityAction again starts applying to it.
// antiGravityAction can also be removed using hkpWorld::removeAction(antiGravityAction)
// after which it is no longer included in simulation.

/// Add the action.
m_world->addAction( antiGravityAction );

// After addAction() m_world references antiGravityAction so the local reference can
// safely be removed.
antiGravityAction->removeReference();
```

As `hkpAction` inherits from `hkReferencedObject`, the simulation world adds a reference to each action during the `addAction()` call. It is therefore safe to call `removeReference()` on the local `hkpAction` pointer after addition to the world.

To remove an action from the simulation, use `hkpWorld::removeAction(hkpAction* action)`:

```
m_world->removeAction( antiGravityAction );
```

`hkpWorld` calls `removeReference()` on `hkpAction` in `removeAction()`. If the action's reference count reaches zero at this point the action will be deleted as normal. Find out more about Havok reference counting in the Havok Base Library chapter.

#### hkpAction::getEntities()

`hkpAction` has a pure virtual `getEntities()` function. For classes derived from `hkpAction`, `getEntities()` is used to retrieve the bodies the `hkpAction` applies to. When an action is added to the simulation world, the world queries the first body the `hkpAction` applies to, to determine what simulation island the body is in. It then adds the action to the same island. All bodies an `hkpAction` is applied to are in the same simulation island. More information about simulation islands is available in the Optimization chapter.

Implementations of `getEntities()` are provided in the derived classes `hkpUnaryAction`, `hkpBinaryAction`, and `hkpArrayAction`. In all three implementations all the bodies each action is applied to are returned, be it one, two or an entire array of bodies. The following code sample shows the implimentation of `hkpUnaryAction::getEntities()`:

```
void hkpUnaryAction::getEntities( hkArray<hkpEntity*>& entitiesOut )
{
    entitiesOut.pushBack( m_entity );
}
```

#### Multiple Actions versus Single Actions

It is perfectly acceptable to have multiple actions apply to an entity. It is also fine for an entity to have an action with multiple functionality but this does not give an appreciable performance gain.

### Actions and Simulation Islands

It is important to note that an action that applies to multiple entities merges those entities' simulation islands. This may result in a performance loss if some of the merged simulation islands would otherwise be inactive. In particular, actions that affect all the entities in a world should be avoided as these will merge all the simulation islands! A separate action per entity is definitely a good idea in this situation.

Actions with more than one body are an association between those bodies and so affect the partitioning of simulation islands. As such you can not change the bodies in an action once the action has been added to the world. If you want to change the bodies, query the action for its properties and recreate another action from that information.

### Actions versus Constraints

Often it is not clear whether a game feature should be better represented by an action or a constraint. In general, actions are faster to simulate than constraints, but less stable e.g. the stability of dashpots is more sensitive to simulation frequency (large timesteps can make simulation unstable). Find out more about constraints in the Constraints chapter.

### Actions and Determinism

It is quite easy to interact with the physics in a way that produces non-deterministic behavior in a multithreaded environment. This is a distinct possibility when working with actions.

Actions typically apply impulses to objects upon which they act. If an action applies an impulse to an object outside its simulation island (i.e. an object that is not in the list of entities reported by `getEntities()`), then undetermined behavior will result (because the object may currently be processed in a separate thread simultaneously).

Furthermore, if an action depends on a phantom's overlap information, it is not safe to check this information in multithreaded mode. Any other simulation island being processed in another thread may penetrate the phantom during the `hkpAction::apply()` call, which leads to undetermined behavior.

The `hkpVehicle` is an action which does both of these things. For this reason there is a flag `hkpWorldCinfo::m_processActionsInSingleThread`, which, when set to true, causes all actions to be processed in a single thread before other threads can process simulation islands. See the section on Synchronization Issues for help detecting similar situations that can lead to non-deterministic behavior.

### Example Actions

The Havok SDK provides a number of ready-made actions. These actions are provided with full source code, and are useful examples of what actions can do. These actions can be found in the `Physics2012/Utilities/Actions` directory of your installation.

### Dashpots

### Linear Dashpot

A linear dashpot (`hkpDashpotAction`) is a simple action that is used to attach two rigid bodies together. Its `applyAction()` function applies forces to the rigid bodies in an attempt to keep the bodies' attachment points in the same relative positions. If other strong forces or impulses are applied to the rigid bodies the dashpot may momentarily "break" but it will attempt to force the bodies back to their specified positions in the next simulation step.

The following `hkpDashpotAction` properties can be specified:

- **Rigid Bodies**

  The two rigid bodies the dashpot will act on.

- **Attachment Points**

  The attachment points for each rigid body. The points can be specified in either world space or the rigid bodies' local space, and do not need to be inside the volume of either rigid body.

- **Strength**

  The strength of the restoring force between the two attachment points. A good initial range is 0.5–1 times the mass of the attached rigid bodies. The default strength value for both linear and angular dashpots is 1.0.

- **Damping**

  This value affects how quickly the oscillation of the dashpot stabilizes. It governs the impulse applied to the connected bodies due to their relative velocities. Low values create springy behavior, while high values reduce any oscillations very quickly with the size of the oscillations getting much smaller each timestep. A good initial value for the strength is 0.1. The default damping value for linear and angular dashpots, based on the default strength, is 0.1.

For similar behavior to a linear dashpot but with greater stability, use a ball and socket constraint.

#### Angular Dashpot

An angular dashpot (`hkpAngularDashpotAction`) is the rotational equivalent of a linear dashpot. An angular dashpot aligns two rigid bodies so that they maintain the same orientation, with an optional offset. The dashpot's angular offset value is a quaternion defining the orientation of the first body relative to the second. In its `applyAction()` function, the angular dashpot works to maintain this offset between the two bodies. By default, this is the identity rotation.

As with `hkpDashpotAction`, the dashpot pivot points can be specified in either world space or each rigid body's local space. `hkpAngularDashpotAction` also provides damping and strength values.

#### Springs

Use an `hkpSpringAction` to join two rigid bodies with a spring. The `hkpSpringAction::applyAction()` function applies forces to the bodies in an attempt to maintain the spring's rest length.

`hkpSpringAction` has the following properties:

- **Rigid Bodies**

  The two rigid bodies that the `hkpSpringAction` will act on.

- **Attachment Points**

  The attachment points for each rigid body. The points can be specified in either world space or the rigid bodies' local space, and do not need to be inside the volume of either rigid body.

- **Rest Length**

  A rest length for the spring. The default value is 1.

- **Strength**

  This governs the force the spring will apply to each attached body for each unit difference between its current length and its rest length. The default value is 1000.

- **Damping**

  This governs how quickly the oscillation of the spring stabilizes. A spring without damping will continue to oscillate the same amount forever. A heavily damped spring will reduce any oscillations very quickly. The default damping value is 0.1.

- **Extension and Compression Flags**

  You can use these values to specify whether the spring acts under extension or compression or both.

For similar behavior to `hkpSpringAction` but using a constraint, use a malleable stiff spring constraint.

### Motors

The `hkpMotorAction` acts as a simple motor that can spin a rigid body. In each simulation step the action applies torque to the rigid body until it reaches a specified angular velocity. To create an `hkpMotorAction`, provide the following information to its constructor:

- *body*: the rigid body to spin.
- *axis*: the desired angular axis of spin for the motor.
- *spinRate*: the rate of spin about this axis, specified in radians per second.
- *gain*: specifies how fast the desired velocity should be reached. The default value is 2.

The following sample shows how to create a motor that applies to the rigid body, myBoxRigidBody:

```
hkVector4 axis( 0.0f, 0.0f, 1.0f );
hkReal spinRate( 5.0f );
hkReal gain( 5.0f );

hkpMotorAction* motorAction = new hkpMotorAction( myBoxRigidBody, axis, spinRate, gain );
m_world->addAction( motorAction );
```

### Wind Action

The `hkpWindAction` will apply a force to a rigid body simulating the drag-like effect of wind. This force is sensitive to the shape and orientation of the rigid body it affects. The action makes use of the `hkpWind` utility to calculate and apply the correct forces (explained below).

The following sample illustrates how to create a Wind Action and apply it to a rigid body. Refer to the WindActionDemo for more details.

```
// Create the hkpWind structure first. hkpPrevailingWind is a simple implementation of hkpWind.
hkpWind* air = new hkpPrevailingWind( hkVector4( 2.0f, 0.0f, 0.0f ) );

// Create and add the action.
hkpWindAction* action = new hkpWindAction( body, air, 0.1f );
m_world->addAction( action );
```

### The hkpWind Utility

The `hkpWind` utility defines an interface through the pure virtual `getWindVector()` method. Implementations of `hkpWind` are vector fields which can be queried for the wind vector at a given world position.

The length of the wind vector should not be thought of as a wind velocity (though it may be a function of the velocity of a simulated wind). Instead, it is the magnitude of the force which should be applied to a unit of surface area perpendicular to its direction. The algorithm which applies wind to rigid bodies takes into account the orientation of surfaces which are not perpendicular to the wind.

In addition to support for external vector fields, the `hkpWind` utility provides an implementation of resistance to motion which is sensitive to shape and orientation. The user provides a *resistanceFactor* (intuitively, a measure of viscosity). The object's motion is opposed by a wind force of strength `speed * resistanceFactor`.

### Applying Wind to Rigid Bodies

As well as using the `hkpWindAction`, the call to apply wind to rigid bodies can also be performed directly by the user (for example, in a `hkpWorldPostSimulationListener`) or using the provided `hkpWindRegion` class. To apply wind directly, the `hkpWind` methods `applyWind()`, `applyResistance()` and `applyWindAndResistance()` should be used as appropriate.

### Approximations

An accurate calculation of the forces to apply to a convex vertices shape can be expensive. The oriented bounding box of these shapes can be used instead to simplify calculations. This is done by providing a non-zero *obbFactor* to the `applyWind()`, `applyResistance()` and `applyWindAndResistance()` methods. The *obbFactor* applies to all `hkpConvexVerticesShape`s in the rigid body.

### An Example: hkpPrevailingWind

The `hkpPrevailingWind` class is provided as an example wind implementation. It has a base wind direction. It can also maintain several oscillators to enable wind to vary over time.

### Notes

- Wind is sampled once per shape. Thus, it is suitable for winds which do not vary significantly across the length of shapes.
- Shapes do not shade other shapes from the wind. This is true of nested shapes and also of the triangles in meshes and MOPPs.
- Applying resistance to rigid bodies of extremely low density can lead to instability. A rule of thumb is that a resistance factor $R$ should not apply to body of mass when `m <= R * Amax * dt` where $m$ is the mass of the body, $Amax$ is its maximum projected area and $dt$ is the time step.
- A `hkpWindAction` applies wind to a rigid body only while the rigid body is active.

### Creating a Custom Action

This section explains how to create a custom action. The example action applies a gravity-like force to a rigid body in each simulation step. As a result, the body "orbits" in an ellipse around a point during

the simulation. The user of the action is able to specify the rigid body, the point around which the body will orbit, and the strength of the gravitational force.

Read the motion state section to find out more about applying forces to rigid bodies before exploring this example. For more details on the `hkVector4` functions used in the example, see the Havok Math Library chapter and the `hkVector4` section of the Reference Manual. Refer to the Serialization chapter for help on serializing custom objects.

### Action Type

First, decide which of the `hkpAction` classes to derive from. As the new gravity action applies to a single rigid body, it should inherit from `hkpUnaryAction`. `hkpUnaryAction` provides some basic functionality for the action and a data member for the entity that it is applied to (`m_entity`).

```
// This action applies a "gravity" force to a satellite body, attracting it to a fixed point.
class GravityAction: public hkpUnaryAction
```

### Construction

We must be able to specify the rigid body, the point around which it will orbit, and the strength of the force applied to it. As seen in the constructor below, the rigid body is passed on to the `hkpUnaryAction` constructor so that it becomes the action's `m_entity` member.

```
inline GravityAction::GravityAction(hkpRigidBody* satellite, const hkVector4 &gravityCenter, const hkReal
  gravityConstant)
                      : hkpUnaryAction(satellite),
                        m_gravityCenter(gravityCenter),
                        m_gravityConstant(gravityConstant) { }
```

### Implementation of applyAction()

Now to make the action do something! The implementation of `applyAction()` should apply a force to `m_entity`, acting towards the specified gravitational center. The magnitude of the force should to take into account both the mass of the entity (so that the action will behave in the same way regardless of the rigid body's mass) and the distance between the body and the center.

First, we find the direction in which the force should be applied:

```
hkVector4 dir;

// Sets dir to be the difference of m_gravityCenter and getRigidBody()->getCenterOfMassInWorld().
dir.setSub4( m_gravityCenter, getRigidBody()->getCenterOfMassInWorld() );
```

The `dir` vector now points from the body's center of mass to the specified gravity center.

Next, we calculate the distance between the body and the point, and a unit direction. This example uses `normalizeWithLength3()`, which normalizes the vector as well as returning the distance. This is useful as the normalized vector is needed later in the example to create the force.

```
hkReal distance = dir.normalizeWithLength3();
```

Then we calculate the necessary magnitude of the force. To get the correct effect, it needs to be proportional to the body's mass, and inversely proportional to the squared distance: `F = (GMm)/d^2`.

```
// Use a version of the gravitational formula:
//
// F = (GMm)/d^2
//
// If G and M are fixed (rolled into m_gravityConstant), then F = m_gravityConstant * m / d^2
// i.e. the force is proportional both to the mass, and the inverse distance squared.
hkReal magnitude =  m_gravityConstant * getRigidBody()->getMass() / (distance * distance);
```

Next, we create the force as a vector acting along the correct direction (using the normalized direction vector) and with the correct magnitude.

```
hkVector4 force;
force.setMul4(magnitude, dir);
```

Finally, we apply the force to the rigid body.

```
// Apply the gravity force.
getRigidBody()->applyForce( force );
```

Note that this action's work does not require time information from the simulation, so the `hkStepInfo` structure is not needed.

Below is the complete `applyAction()` function for the `ExampleGravityAction`.

```
inline void GravityAction::applyAction( const hkStepInfo& stepInfo )
{
    hkpRigidBody* rb = getRigidBody();

    hkVector4 dir;

    // Sets dir to be the difference of m_gravityCenter and rb->getCenterOfMassInWorld().
    dir.setSub4( m_gravityCenter, rb->getCenterOfMassInWorld() );

    hkReal distance = dir.normalizeWithLength3();

    // Use a version of the gravitational formula:
    //
    // F = (GMm)/d^2
    //
    // If G and M are fixed (rolled into m_gravityConstant), then F = m_gravityConstant * m / d^2
    // i.e. the force is proportional both to the mass, and the inverse distance squared.
    hkReal magnitude = (m_gravityConstant * rb->getMass()) / (distance * distance);

    hkVector4 force;

    // Set force to be dir multiplied by magnitude.
    force.setMul4(magnitude, dir);

    // Apply the gravity force.
    rb->applyForce( stepInfo.m_deltaTime, force );

    if ( distance < 8.0f )
    {
        rb->setLinearDamping( -0.1f );
    }
    else
    {
        rb->setLinearDamping( 0.0f );
    }
}
```

### Instantiation

Now that the class for the action is written, we create an instance of it, providing the target rigid body, center of gravity and force strength.

```
hkpRigidBodyCinfo satelliteInfo;

// Set up satelliteInfo...
// ...
hkpRigidBody* satellite = new hkpRigidBody(satelliteInfo);

m_world->addEntity(satellite);

hkVector4 centreofgravity(0.0f, 0.0f, 0.0f);
GravityAction* gravityAction = new GravityAction( satellite, centreofgravity, 50.0f );
```

### Adding the Action to the Simulation

Finally, we add the action to the simulation using `addAction()`. Remember to call `removeReference()` once the action is added.

```
m_world->addAction( gravityAction );

// After addAction() m_world references antiGravityAction so the local reference can safely be removed.
gravityAction->removeReference();

satellite->removeReference();
```

#### Important Points when Using Actions

#### Top Tips

Some important points to remember when using actions are:

1. Actions are **only applied to active bodies** .
2. Actions are normally **removed from the simulation** when their associated rigid bodies are removed.
3. Actions often need to **take time into account** when determining how large a force/implulse they should apply to each body.
4. Actions that act on more than one body **merge the simulation islands** the bodies are in.

#### Interpenetration

When using actions, be careful not to inadvertently cause interpenetration.

Interpenetration happens when one body is moved inside another, thus compromising the physical realism of the simulation. Havok uses its solver and special constraints to try to prevent this from happening. However, if an action is added that directly sets the position or orientation of a body (e.g. using `hkpRigidBody::setPosition()`) then it risks causing an interpenetration that cannot be easily resolved.

For instance, if an action moves a body around by changing its position, it is possible that it will eventually move the body into something that is fixed (like the ground). Havok will detect this and attempt to recover physical consistency. However, in the next simulation step, the action will be called again, which may again try to put the body back into the ground.

A safer way to move a body is to use either forces or impulses, as in the example classes provided with the Havok SDK. The difference between a force and an impulse is that when a force is applied it has the effect of accelerating or decelerating the body. Just like a car, if it is traveling at 100 mph and brakes it will take some time before it comes to a complete stop.

Impulses change the velocity of a body directly (more like hitting a tree than applying a brake!). To change the orientation of a body apply forces and impulses off center, or apply torque and angular impulses. More information about applying forces, torques, and impulses is available in the "Change the motion state" section.

### 2.2.2.3 Stepping the Simulation

As mentioned previously, the `hkpWorld` is used to step the simulation forward in time. You do this by calling `hkpWorld::stepDeltaTime()` or `hkpWorld::stepMultithreaded()`, passing it the desired delta

time value, in seconds. At a fixed frame update of 60 Hz, the delta time would be 0.0166f. Very small or large delta values are not advised, and values that vary a lot reduce the accuracy of the solver. The Simulation Overview section describes the tasks carried out by the physics engine in each step.

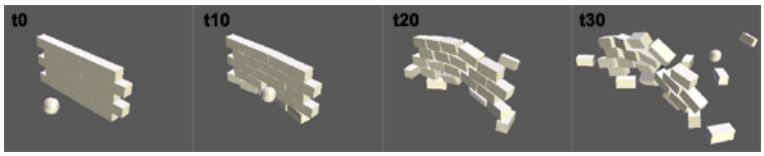



Figure 2.50: Snapshots of a Havok simulation being stepped forward.

### Multithreaded vs Singlethreaded Stepping

Havok Physics 2012 simulation fully supports multithreading; this is the default behavior of the engine. A full discussion on multithreaded stepping (including Havok Physics 2012 multithreading specifics) is available in the multithreading chapter in the Common documentation. This includes instructions on setting up and configuring the multithreading parameters.

To force the simulation to use singlethreading, set the simulation type to `hkpWorldCinfo::SIMULATION_TYPE_DISCRETE`.

### Synchronous Stepping

To step the world simulation forward at 60 Hz as described above, you can simply call either of the following:

```
// singlethreaded
m_world->stepDeltaTime( 0.0166f );

// or multithreaded
m_world->stepMultithreaded( /*hkJobQueue*/, /*hkThreadPool*/, 0.0166f );
```

After this call, the positions and velocities are all updated based on the timestep and all the other simulation variables. At this point, the results of the simulation can be used in-game, for rendering, or any other purposes. The best time to apply impulses and other forces from your game to the physical world is just before this call.

### "Half" Stepping

The call `hkpWorld::stepDeltaTime()` performs several steps in order to advance the simulation. It is possible to spread out these steps over multiple frames if performance is an issue. Note that this can only be done in singlethreaded simulations, i.e. `hkpWorldCinfo::SIMULATION_TYPE_DISCRETE` and `hkpWorldCinfo::SIMULATION_TYPE_CONTINUOUS` (on a single thread). These functions are:

```
// Integrate the world forward in time - solves all constraints, and advances all transforms to the next
  timestep
m_world->integrate( 0.0166f );

// Perform collision detection
m_world->collide();

// Perform continuous simulation
m_world->advanceTime();
```

An example of "half" stepping the world using these functions is given in the demo `Physics2012/UseCase/Fracture/City/FractureDemo`. It performs integration and collision detection on alternate frames. It does so as follows:

```
void stepPhysics( physicsDeltaTime )
{
   hkTime timeForDisplay;
   if (m_halfStepCounter++ %2 == 0)
   {
      m_world->integrate( physicsDeltaTime );
      timeForDisplay = m_world->getCurrentTime() + physicsDeltaTime * .5f;
   }
   else
   {
      m_world->collide();
      m_world->advanceTime();
      timeForDisplay = m_world->getCurrentTime();
   }

   synchronizeTransforms( timeForDisplay );
}
```

In the above code, on alternate frames, we either call `integrate()` or `collide()` and `advanceTime()`. This allows us to split the physics processing time between rendering frames, i.e. we render at 60 Hz, but we actually simulate at 30 Hz. Note that you have to synchronize the transforms of the display objects using a time greater than the world current time after `integrate()`, because the world current time is not updated until `advanceTime()`. To synchronize a transform you call `hkpRigidBody::approxTransformAt( timeForDisplay )`. Doing this gives you a smooth interpolation of transforms in the display frames.

The following diagram shows how these three calls look in the context of the block diagram presented in the introduction section.

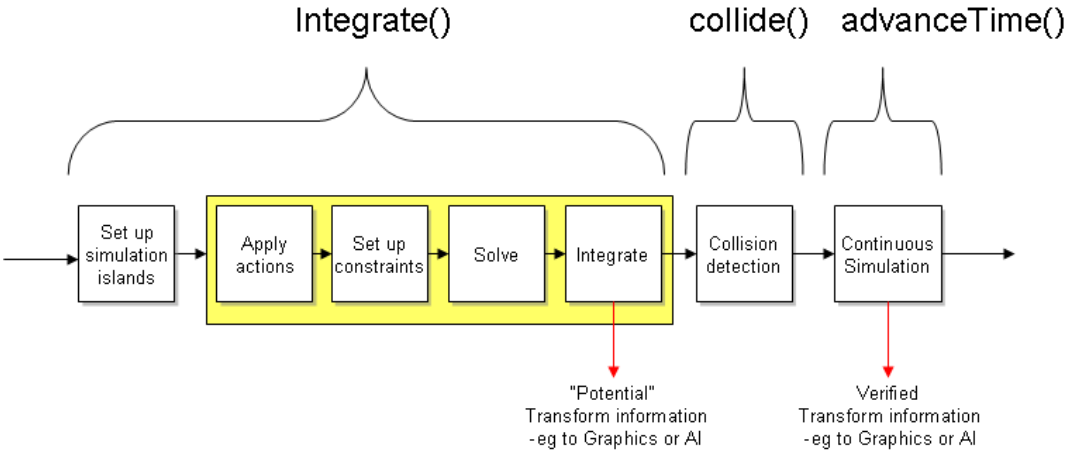



Figure 2.51: Integrate(), collide() and advanceTime()

The `advanceTime()` function performs continuous simulation. If the simulation is of type `hkpWorldCinfo::SIMULATION_TYPE_DISCRETE`, no continuous simulation is performed by `advanceTime()`, however it is still essential to call the function in order to advance the current time of the world.

**Asynchronous Stepping**

Note that in the above case, we are stepping physics and graphics in parallel. This is our recommended approach because it keeps things simple, and is the most predictiable in terms of CPU load, and, particularly in multithreaded simulation, the most efficient way to simulate the objects.

However, there could be reasons (e.g. very varying framerate, slow framerate or running the physics at a much lower frequency) to decouple physics and framerate and step the physics asynchronuously. To do this, several more functions are provided in `hkpWorld`. These are `hkpWorld::setFrameTimeMarker()`, `hkpWorld::isSimulationAtMarker()` and `hkpWorld::isSimulationAtPsi()`. Please note that using asynchronous stepping throughout your game engine is challenging and only recommended for advanced users.

The code that you use to step the world asynchronously is:

```
world->setFrameTimeMarker( frameDeltaTime );

world->advanceTime();
while ( !world->isSimulationAtMarker() )
{
    HK_ASSERT( 0x11179564, world->isSimulationAtPsi() );

    {
        // Interact from game to physics
    }

    world->stepDeltaTime( physicsDeltaTime );

    if (world->isSimulationAtPsi() )
    {
        // Interact with physics: physics data to game data
    }
}
```

This code is best explained in conjunction with a diagram.

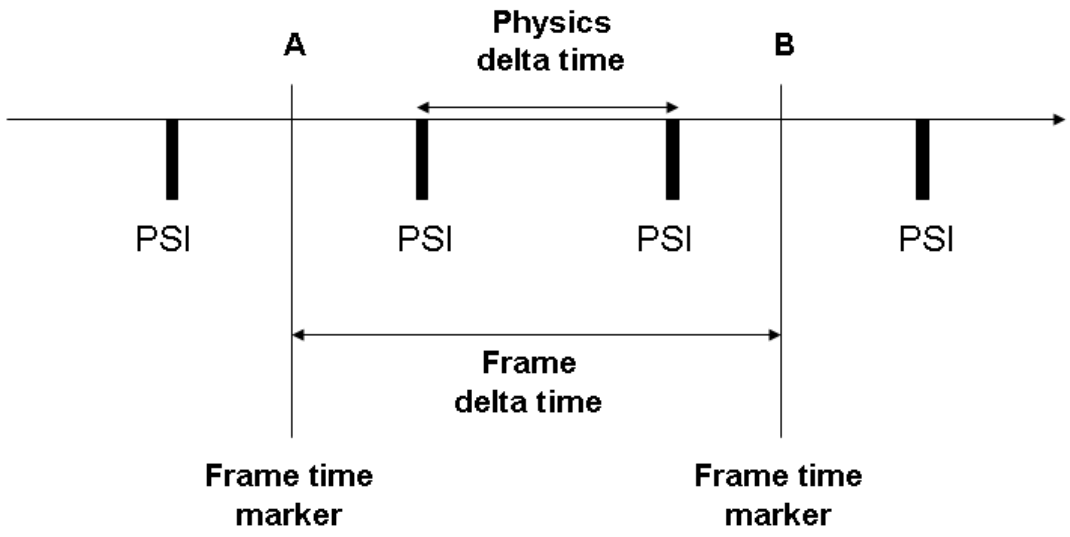



Figure 2.52: Integrate(), collide() and advanceTime()

The above diagram shows a picture of simulation time as the physics progresses in discrete timesteps, called PSIs. In synchronous simulation, i.e. when you are simply calling `stepDeltaTime( physicsDeltaTime )`, the simulation simply moves from one PSI to the next. However, it is possible to "simulate" for any length of time by setting "frame time markers". These are set with the `hkpWorld::setFrameTimeMarker( frameDeltaTime )` call and are shown in the diagram above. Assuming that the world current time is at time "A", if you call `setFrameTimeMarker( frameDeltaTime )` you will set a marker at time "B". Here we are assuming that we have just completed an asynchronous step. The world current time (obtained by `hkpWorld::getCurrentTime()`) will be different from the world current PSI time (obtained using `hkpWorld::getCurrentPsiTime()`). The PSI time is always the same or greater than the current time. This initial situation is illustrated below:

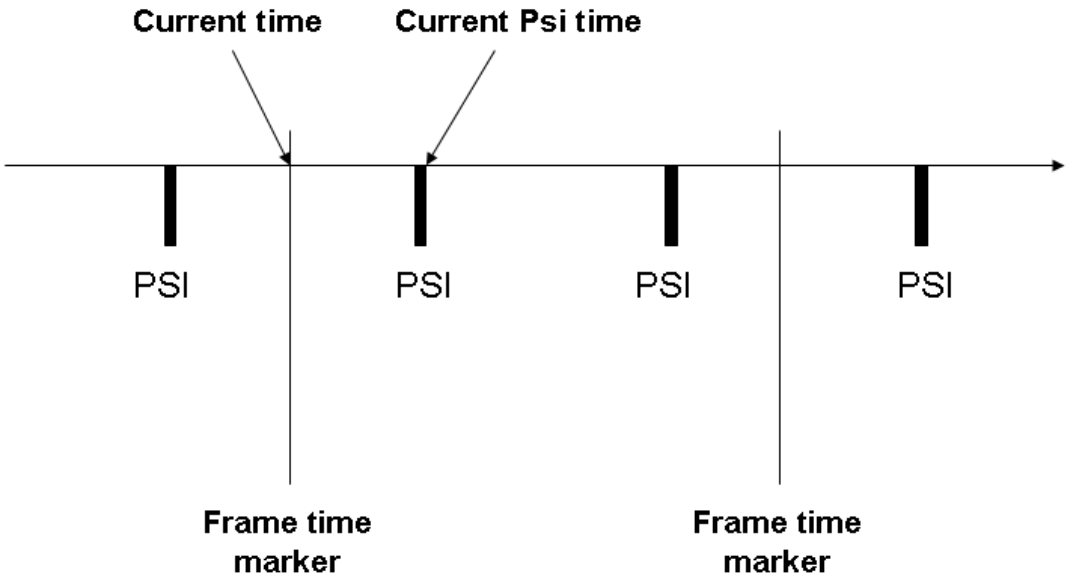



Figure 2.53: Integrate(), collide() and advanceTime()

After setting the new marker, you must then call advanceTime(). This will advance the current time to either the current PSI time or the frame time marker, whichever is earlier. If you are running a continuous simulation, this will solve all TOIs between the current time and the time it advances to. In our example, after `advanceTime()` is called, the time bar will look like this:

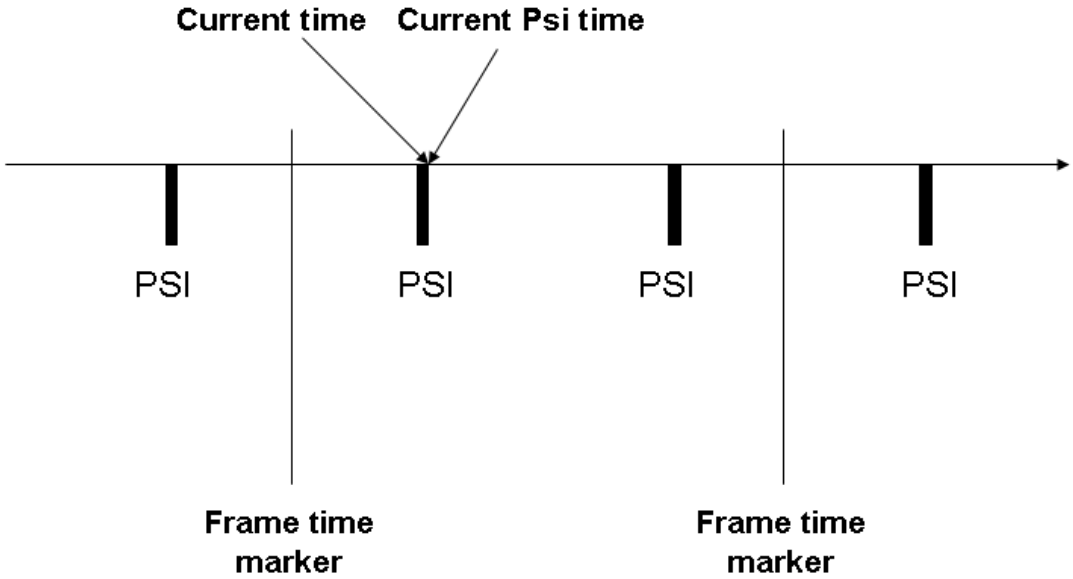



Figure 2.54: Integrate(), collide() and advanceTime()

You must then check if the simulation has reached the frame time marker, using `hkpWorld::isSimulationAtMarker()`. If it has not then you must step a PSI step, with `stepDeltaTime()`. Before calling `stepDeltaTime()`, you should write code to pass information from your game to the physics. The call to `stepDeltaTime()` will advance the simulation, and the

current time, to either the current PSI time or the frame time marker, whichever is earlier. Actually, `stepDeltaTime()` simply calls `integrate()`, `collide()`, and `advanceTime()` in order. After this call you need to check if the simulation is at a PSI (rather than "mid-step" at a frame marker) and if it is, execute your code to interact from your physics to your game. After `stepDeltaTime()` the time bar will then look as follows:

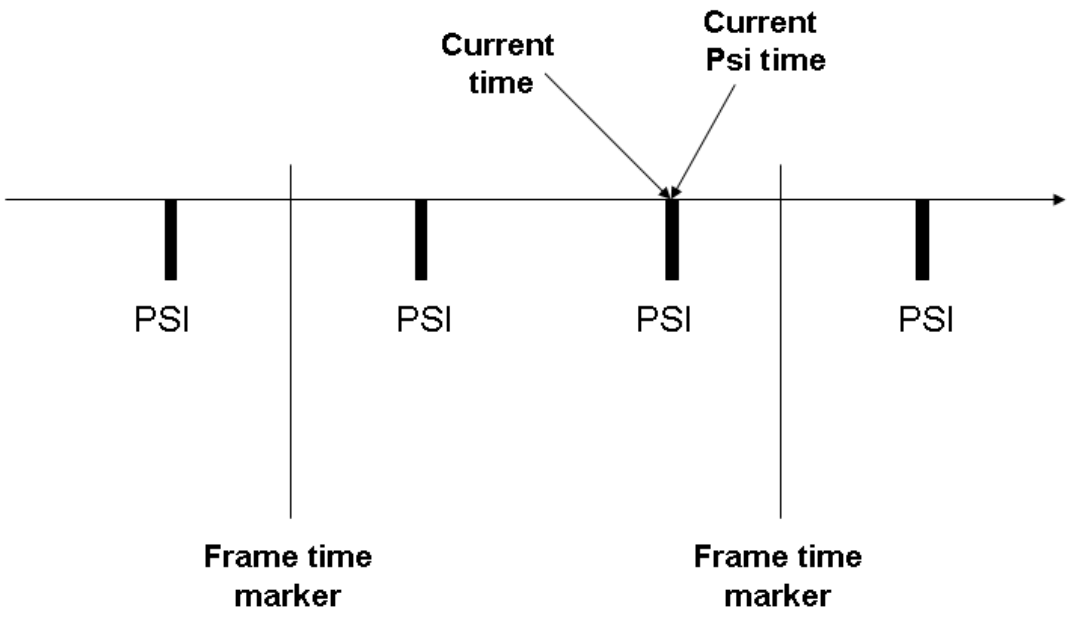



Figure 2.55: Integrate(), collide() and advanceTime()

At this point `isSimulationAtMarker()` still returns false, so another PSI step must be performed by calling `stepDeltaTime()`. After this call it can be seen (see the diagram below) that the current PSI time advances to the next PSI marker (this happens in the `integrate()` and `collide()` parts of `stepDeltaTime()`) and the current time only advances as far as the frame time marker (this happens in the `advanceTime()` part of `stepDeltaTime()`). At this point the simulation has been advanced to the desired frame time so the loop terminates.

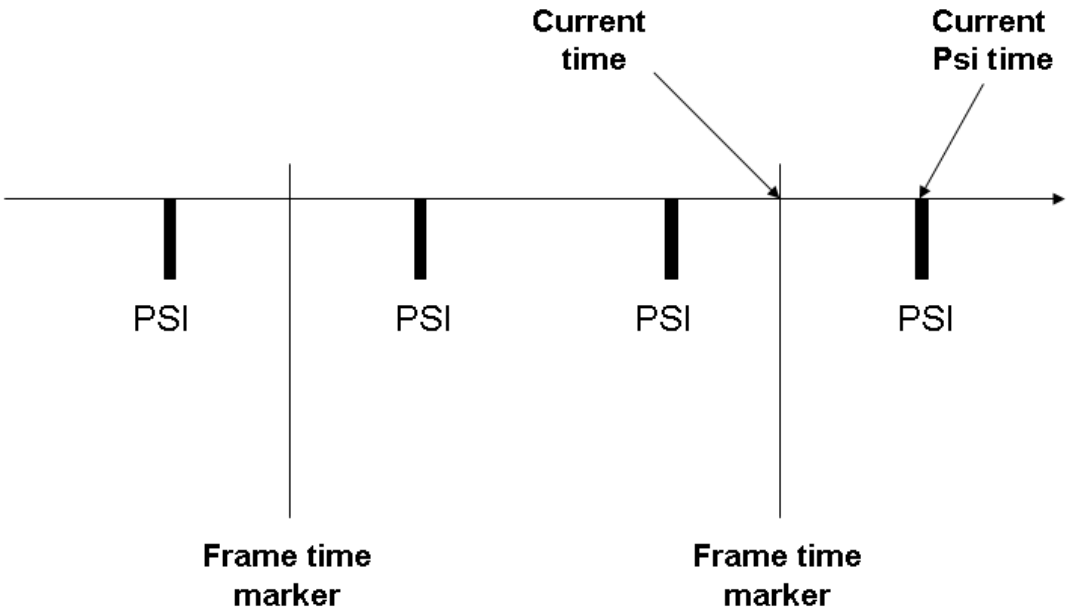



Figure 2.56: Integrate(), collide() and advanceTime()

It is important that you do game/physics interaction when the world current time is at a PSI rather than when it is between PSIs (as it is at the end of the loop). It is fine to render the objects at this position, because the system will interpolate positions and orientations using the `hkpRigidBody::approxCurrentTransform()`, or `hkpRigidBody::approxTransformAt()` functions. However, applying impulses and forces to these objects at this time is problematic. As well as having two positions and orientations at unique times, each object has a single transform, representing its transform at its "current simulated time". For some objects this transform will be at the current PSI time, while some will have a transform at the current time (depending on whether they have been involved in continuous simulation. Also to accurately apply a force to the object, you need to apply the force for the remainder of the timestep.

You can avoid these problems by interacting with the physics when the simulation is at a PSI. This is shown by the comments in the code block above. This code block is actually the implementation of the utility function `hkpAsynchronousTimestepper::stepAsynchronously()`. While you can use this function, when it returns it will typically give you a current time out of sync with the current PSI time, so it is not great to interact with the physics at this point. Rather, we recommend you take the implementation and integrate it into your main game loop, and add physics interaction code at the points specified.

**Variable Time Step**

While it is recommended that the simulation be stepped forward with the same time step every frame, Havok does support variable time steps (as certain game situations occasionally require this flexibility).

Varying times steps can introduce some unexpected behavior if it is not carefully done. In general, the simulation will behave fairly stably as long as the time step does not vary more than 10% every frame. Varying the time step will also affect the stiffness of the simulation, introducing a different feel from frame to frame. Minimizing the variation of the time step will also minimize the change in stiffness.

**Postponing of Operations**

All of the critical functions in `hkpWorld` use a locking mechanism. When the engine performs integration, collision detection or handles TOI events the world is internally locked. This allows the user to call any functions (e.g. `addEntity()`, `removePhantom()`, `updateCollisionFilterOnEntity()`) at any time and they will be stored on an internal todo-list and executed at the earliest possible time when the `hkpWorld` is unlocked.

These postponed functions are executed in the order they are received, with each call fully executed (including recursive calls) before the next function is processed.

The following functions are considered critical (all are members of `hkpWorld`, except where noted otherwise):

- `addEntity()`
- `removeEntity()`
- `addEntityBatch()`
- `removeEntityBatch()`
- `addConstraint()`
- `removeConstraint()`
- `addAction()`
- `removeAction()`
- `hkpWorldOperationUtil::mergeIslands()`: Used internally.
- `addPhantom()`
- `removePhantom()`
- `addPhantomBatch()`
- `removePhantomBatch()`
- `entityUpdateBP()`: Updates `hkpRigidBody`'s broad phase entry. Used by functions affecting a body's position/orientation and therefore its `hkAabb` representation, e.g. `hkpRigidBody::setPosition()`, `hkpRigidBody::setRotation()`.
- `hkpRigidBody::setMotionType()`
- `hkpRigidBody::setShape()`: This entails removing and re-adding the rigid body to the simulation.
- `updateFilterOnEntity()`
- `updateFilterOnPhantom()`
- `updateFilterOnWorld()`

### Updating game objects after the step

Once the simulation step (usually `hkpWorld::stepDeltaTime()`) has been completed, the positions and velocities of the simulated rigid bodies have been updated and can be fed into the game objects. For example, the rigid body's transform is accessed using `hkpRigidBody::getTransform()`.

After the simulation step is the best place to keep game objects synchronized with their physical counterparts. The best time to apply impulses to objects in the physical world is right before the simulation is stepped forward.

When half-stepping or asynchronously stepping, the rigid body transforms may not necessarily be synchronized. In these situations, use the `hkpRigidBody::approxCurrentTransform()` method to calculate an approximate transform at the current simulation time.

The transform approximation methods can also be useful for (approximately) predicting where objects will be one or more frames in the future. This can be especially useful in games with multiplayer modes where you must predict the positions of objects being passed back and forth over a network.

#### 2.2.2.4 Synchronization Issues

The Havok SDK is not thread safe (except for the memory manager). This means that care should be taken when accessing and modifying world objects on multiple threads.

Havok SDK is deliberately not thread safe because:

- Creating a fully thread safe engine would result in a very slow engine.
- A thread safe engine does not mean that the game using the engine is thread safe: For example, you cast a ray and want to delete the object the ray is hitting. Let's assume Havok was thread safe and you are running multiple threads. So you cast the ray into the "thread safe" engine and identify an object, then you make a call to delete the hit object. Unfortunately between the call to cast the ray and delete object, another thread was scheduled and deleted the object as well. As a result the engine crashes.

As such, most synchronization in the simulation is left explicitly to the user. Instead of exposing semaphores which control when it is safe to perform different operations, we provide safety markers that simply warn and assert when synchronisation has to be enforced.

- When you call `hkpWorld::stepDeltaTime()`, Havok Physics 2012 internally manages the synchronization of the simulation by splitting the work into discrete jobs and distributing these among the available threads. The constraint solving and actions are distributed per island. In callbacks you have restricted access to different objects. For example, if you apply an impulse to a random object in an action (not in the same island) then this is not appropriately synchronized–another thread may be writing to that object.
- when you are performing asynchronous queries which are only reading from the engine (read only access) then several threads can use the engine simultaneously. If one thread tries to access an operation which requires a write the engine will assert in debug mode. In this case the user is responsible for synchronizing all access to the Havok engine.

Havok has a debug utility built in that helps to identify missing or wrong synchronization. If multi-threaded simulation is enabled, the engine will guard all functions to the physics system.

There are two seperate situations where clients should take care of correctly synchronizing access to the physics engine: during asynchronous queries and during callbacks.

### Synchronizing Asynchronous Queries

Asynchronous queries are performed outside of the Havok simulation step. To enable access to checked functions, you should synchronize your game engine before accessing Havok. In this case you then can tell the simulation world that you have synchronized by calling one of the following functions:

- `hkpWorld::markForWrite()`
- `hkpWorld::markForRead()`
- `hkpWorld::unmarkForWrite()`
- `hkpWorld::unmarkForRead()`

These functions should be used before and after read and write accesses to Havok data.

Examples:

- A single thread is creating and populating an `hkpWorld`. As the thread knows it is the only possible user of `hkpWorld` it calls `hkpWorld::markForWrite()`, just after the `hkpWorld` is created. Once the `hkpWorld` is populated, it calls `hkpWorld::unmarkForWrite()`.
- In our previous raycast/delete object example, we have to create a single critical section object in our game guarding the physics access. Whenever we want to access the physics, we enter our critical section, mark the physics for write, do our work, unmark the physics and leave the critical section.

### Synchronizing Callbacks

In a physics callback we cannot synchronize access to the physics, as we are already inside the Havok simulation step (the `hkpMultiThreadedSimulation` is responsible for this). However many of the critical functions are actually thread-safe as they are not executed immediately but put on a thread safe command queue (as discussed earlier). Many of the other functions also have a thread safe version. E.g. `hkpRigidBody::activate()` is not thread safe, but `hkpRigidBody::activateAsCriticalOperation()` is.

In debug mode, the engine will assert if you call the wrong function.

**Note:**
**Never call** `hkpWorldObject::markForWrite()` in a callback, as doing so simply bypasses the synchronization checks.

### Warning

If possible, do not call the `hkpRigidBody::markForWrite()` functions, try to use the `hkpWorld` version instead. If you call `hkpWorld::markForWrite()` you are indicating that you have exclusive read write access to the entire engine. If you have to call `hkpWorld` or `hkpRigidBody::markForWrite()`, be absolutely sure that you have synchronized your thread before calling this function.

Havok's debug access checks are very helpful, but if you call `markForWrite()` without doing proper synchronization you will have disabled the engine checks without solving the problem (it's like disabling an assert). Eventually the engine will crash in a non-deterministic way.

## 2.3 Interacting with a Simulation

In the previous chapter, we explored how to set up the simulation world and the simulation objects. Havok builds on this by providing a number of tools and features to interact with the simulation and customize its behavior. This chapter will examine the following:

- **Collision filtering** : A powerful yet flexible system to customize which collisions are processed by the engine.
- **General listeners** : Listeners provide information about every important event that can occur during a Havok simulation (such as adding and removing of an object, or the deactivation of a simulation island).
- **Collision listeners** : These listeners provide information about collisions and contact points as they occur, allowing users to modify physical behavior or trigger game logic.
- **Response modifiers** : These allow for selective behavioral tweaking of how constraints and contact points are solved.
- **The query methods in hkpWorld and hkpPhantom classes** : These perform asynchronous queries (such as ray casting, linear casting and low-level collision queries) using the collision detector.

### 2.3.1 Collision Filtering

Collision filtering allows the user to decide which entities can collide with each other and, conversely, which collisions should be ignored. There are many use cases for filtering: you may wish to disable collisions between objects constrained together (such as collisions between the upper and lower arm of a ragdoll); or you may wish to report overlaps between a phantom and only a certain type of bodies. Collision filtering will allow you to easily and flexibly define which collisions to process, and which to ignore.

Havok's filtering system relies on two key pieces of information:

- **The Collision Filter** : The filter decides whether to allow a collision between any pair of objects. The `hkpCollisionFilter` class defines the filter interface.

  A physics world can have at most one collision filter, and that filter affects all collisions in the world. To set the filter, call the `hkpWorld::setCollisionFilter()` method.
- **Collision Filter Info** : Each rigid body, shape and collision query provides a 32-bit "collision filter info", which is used by the collision filter to decide whether each collision involving that object is allowed.

  A collision filter system is of course free to ignore the collision filter info of the objects colliding, but in practice it will need to use this information (or other shape-specific information) to decide whether to filter or allow a collision.

This system of having objects provide user-specified information to a global filter implementation is powerful yet flexible. Havok Physics 2012 provides a default "group" filter implementation and this is recommended for most use cases. This implementation is described in detail below. We will also describe some details of how to implement a custom filter.

Before exploring the default Havok solution, let's consider a very simple collision filtering system to illustrate the basics. A very basic collision filter system is one that decides to allow collisions between two objects only if they have the same collision filter info. If their collision info is different, then the collision is filtered. The `UserCollisionFilterDemo` illustrates this basic filter (and the creation of a simple user-defined collision filter). The demo creates two fixed boxes, and a moving box (see picture below). A value of 0 is assigned to the `m_collisionFilterInfo` of the top-most (white and falling) box, and the bottom-most (fixed) box. A value of 1 is assigned to the (fixed) box in-between. The `isCollisionEnabled()` method of the user-defined collision filter created in the demo simply checks the `m_collisionFilterInfo` of the pair of `physics_collidables` for equality; if the values are equal, collision is enabled. If not, collision is disabled. As expected, the moving box falls through the first fixed box, but collides with the second:

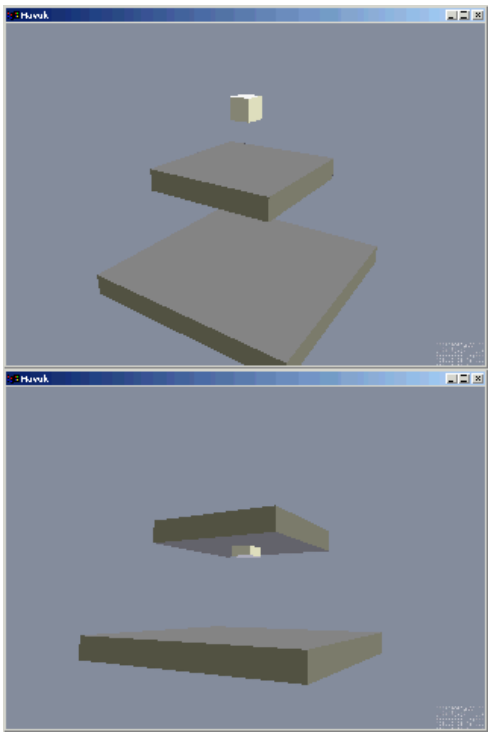


Figure 2.57: The User Collision Filter demo: before, and after

#### 2.3.1.1 Filter Types

The `hkpCollisionFilter` class actually implements four base filter classes. These broaden the range of filtering possible and allow for increased flexibility by splitting the filtering into four distinct collision types:

- `hkpCollidableCollidableFilter`: Handles collidable vs collidable tests. This is the most common usage, allowing us **to filter broad phase results** . Please note that from a filter point of view, phantoms and rigid bodies are identical (as they are both collidables).
- `hkpShapeCollectionFilter`: Handles shape collection vs shape tests. This is used to **filter the narrow phase results** of a collision between a shape container and a convex shape.
- `hkpRayCollidableFilter`: Handles ray cast vs collidable tests. This filters the results of a **ray cast against objects in the broad phase** during world ray casts.

- `hkpRayShapeCollectionFilter`: Handles ray cast vs shape container tests. This is used to filter a **ray cast against children of a shape container** during world or shape ray casts.

The `hkpCollisionFilter` contains four overloaded `isCollisionFilterEnabled()` methods, one corresponding to each base class. This class can be used as the base for custom collision filters.

#### 2.3.1.2 Default Group Filter Implementation

**hkpGroupFilter** is the default Havok Physics 2012 collision filter implementation, providing a powerful and flexible fixed-function filter. It classifies objects using layers, system groups and subsystem IDs, with their interactions defined by a set of user-specified rules.

**Layers and System Groups**

There are two key concepts in the `hkpGroupFilter`. The first is that of **Layers** which correspond to general categories of objects. By default collisions are enabled between all layers but can be disabled for any given layer combinations. For example, if we were to choose keyframed objects, debris objects, dynamic objects, fixed (landscape) objects as categories, we may want to disable collisions between dynamic objects and debris or between keyframed objects. `hkpGroupFilter` stores this information as a symmetric table of layer combinations like the following:

| Dynamic | Keyframed | Debris | Fixed | |
|---|---|---|---|---|
| **Dynamic** | enabled | enabled | - | enabled |
| **Keyframed** | enabled | - | - | - |
| **Debris** | - | - | - | enabled |
| **Fixed** | enabled | - | enabled | - |

The second concept is **System Groups** which allow fine grained filtering between closely related objects. When two bodies collide their system groups are checked before their layers. If both belong to the same system group, the layer check is skipped and they will collide unless the collision has been explicitly disabled for their **Subsystem IDs** . Each body's collision filter information contains its own subsystem ID and the subsystem ID of another body within the same system group it will not collide with. When two bodies in the same system group collide, the collision will be ignored if *either body* matches the "no collide" ID of the other. That is, the ignore IDs need not be symmetric and thus moderately complex rules can be constructed with only a pair of IDs per body.

This mechanism is used to selectively disable collisions *within* a group of bodies without affecting collisions with bodies in other systems. A frequent use case is in systems of constrained bodies such as ragdolls where all bodies will have the same system group and collisions between pairs connected by constraints will be disabled via their subsystem IDs.

There is one special system group (zero) which is reserved and can be thought of as "*no group*". All bodies with zero system group are considered to be in separate groups. Note as well that an object can only be a member of a *single layer* and a *single system group* at a time.

**Filter Logic**

The algorithm basically dispatches to either a layer check or a subsystem check. If both bodies are in the same system group (bearing in mind group 0 never matches) we use the subsystem check. Otherwise we use the layer check.

The entire filtering rule set is illustrated by this flow chart:

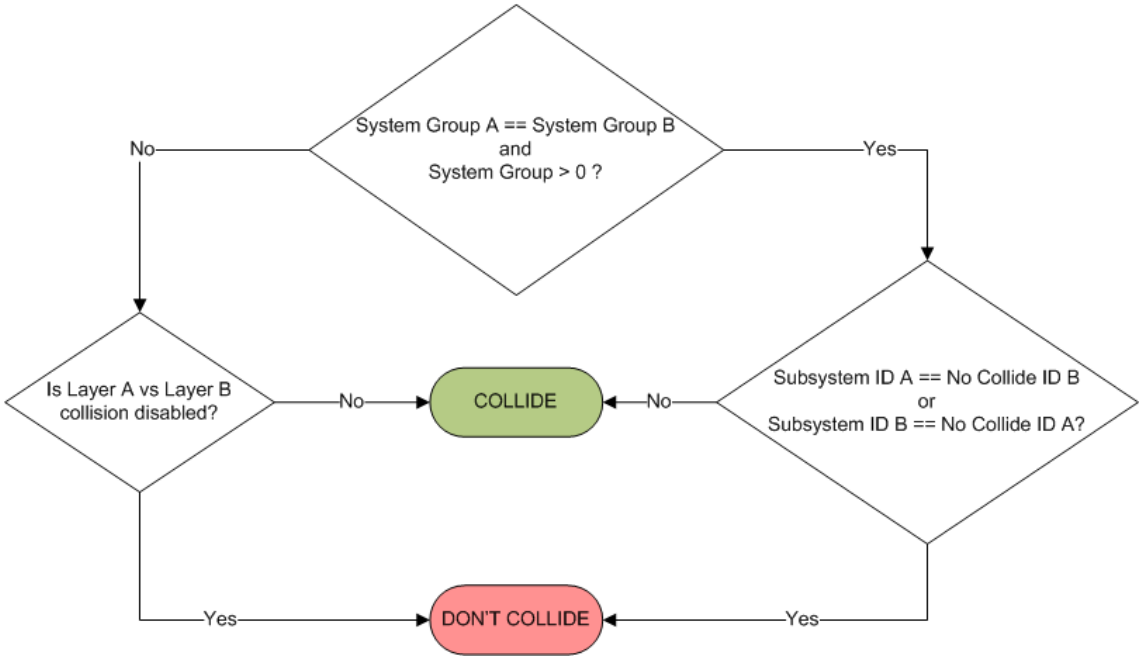



Figure 2.58: Group Filter Decision Flow Chart

### Working with hkpGroupFilter

As the system group and layer information is all packed into one 32-bit collision filter information structure (more details here), Havok provides several helper utilities to simplify the generation and manipulation of these structures. Please refer to the reference for the `hkpGroupFilter` class for full details.

### Generating the Collision Filter Info

A simple method is to generate (or more technically to "pack") the collision filter information from a provided collision layer and system group:

```
hkUint32 collisionFilterInfo = hkpGroupFilter::calcFilterInfo( LAYER_ID, systemGroup );
```

We note that the full method declaration allows us to optionally provide a subsystem ID and corresponding subsystem ID with which not to collide:

```
hkUint32 hkpGroupFilter::calcFilterInfo( int layer, int systemGroup = 0, int subSystemId = 0, int
  subSystemDontCollideWith = 0);
```

The `hkpGroupFilter` also provides a number of helper methods to extract the different components forming the collision filter info (system group, layer, etc), for example:

```
int hkpGroupFilter::getLayerFromFilterInfo( hkUint32 filterInfo );
```

```
int hkpGroupFilter::getSystemGroupFromFilterInfo( hkUint32 filterInfo );
```

### Generating New System Groups

The rationale behind system groups is presented with a use case: Consider an environment with two ragdolls (A and B), and a static landscape for them to fall on. We want the ragdolls to collide with the landscape, and with each other. However, we don't want the body parts of an individual ragdoll to collide with each other (we assume here that the constraint limits will correctly enforce the disjointedness that we want).

Recall that objects in **different** system groups will always collide (unless collision between their layers has been disabled). We can therefore achieve the desired behavior by using one (non-zero) system group for all the bodies of ragdoll A and a second system group for all the bodies of the ragdoll B.

We can obtain an unused system group number by calling:

```
int systemGroupForOneRagdoll = groupFilter->getNewSystemGroup();
```

This increments the internal system group counter of the `hkpGroupFilter`, and returns a fresh system group. Using this system group, we calculate an appropriate collision filter info for the ragdoll:

```
info.m_collisionFilterInfo = hkpGroupFilter::calcFilterInfo( RAGDOLL_LAYER_ID, systemGroupForOneRagdoll );
```

We then assign this filter info to the ragdoll and repeat the same steps for second ragdoll.

### Enabling/Disabling Collisions Between Layers

The collision rules between collision layers are stored in the group filter itself. These must be set up explicitly (as by default all layers are set to collide with each other). Disabling collisions between two collision layers is straightforward (remembering that collision layers are numbered from 0 to 31):

```
groupFilter->disableCollisionsBetween( USER_COLLISION_LAYER_1, USER_COLLISION_LAYER_2 );
```

The corresponding call to enable collisions between layers is:

```
groupFilter->enableCollisionsBetween( USER_COLLISION_LAYER_1, USER_COLLISION_LAYER_2 );
```

We can also enable/disable collisions between multiple layers in a single call using several bitfield-based methods. One useful method is to disable collisions between all layers, as follows:

```
groupFilter->disableCollisionsUsingBitfield( 0xfffffffe, 0xfffffffe );
```

### Accessing and Setting an Object's Filter Info

Finally, an object's collision filter info can be accessed at runtime either through its Rigid Body (`hkpRigidBody::getCollisionFilterInfo()`) or its Collidable (`hkpCollidable::getCollisionFilterInfo()`).

The object's collision filter info can also be changed at runtime using either `hkpRigidBody::setCollisionFilterInfo()` or `hkpCollidable::setCollisionFilterInfo()`.

**Note:**
Changing an entity's collision filter info at runtime will not immediately propagate through the simulation world. The simulation world needs to be notified of the changes and how to propagate them. Please refer to the Updating Collision Filters section.

### hkpGroupFilter Details

The Group Filter has been designed to reflect a balance between speed and flexibility. The entire information for system groups, layers and subsystem IDs is encoded into a 32-bit integer:

- 5 bits for the Collision Layer (allowing for 32 possible layers).
- 16 bits for the System Group (allowing for 65536 unique system groups).
- 5 bits for the Subsystem ID (again allowing 32 possible subsystem IDs per system group).
- 5 bits for the corresponding Subsystem ID not to collide with.

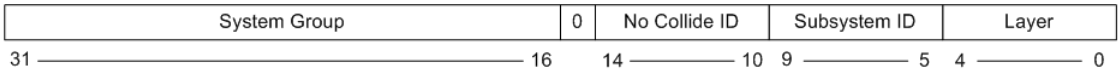



Figure 2.59: Bitfield Layout of Collision Filter Information

Internally, the `hkpGroupFilter` uses masks and bitwise operations to handle system operations in an efficient manner. This makes for a very fast collision filtering system while providing enough flexibility to handle the demands of most possible complex scenarios encountered in games and simulations.

### Collision Filtering Use Cases

We already explored a very basic use case scenario to explain the basics of collision filtering. In this section, we will consider a few more examples to illustrate the versatility of the Havok collision filtering and the `hkpGroupFilter`.

### Using the Subsystem ID

A common use case of the subsystem ID is disabling collisions between pairs of constrained bodies. The best example here is a ragdoll where we want the limbs to collide with each other–we don't want the arms to go through the chest or the legs through each other–but not with their neighbor as they usually overlap a little and collisions would cause jitter and instability with the constraints (e.g. the forearm should not collide with the upper arm).

To achieve this we will calculate the ragdoll's bodies' collision filter infos in such a way that:

- All ragdoll bodies are in the same collision layer (e.g. `COLLISION_LAYER_RAGDOLL`).
- All ragdoll bodies belong to the same system group and no other bodies belong to it. We can obtain a unique system group from the group filter.
- Each ragdoll body has a unique subsystem ID.
- Each ragdoll body has the subsystem ID of their "parent" body in the constraint chain as the "no collide" ID.

```
// Obtain a new system group that we will use for all the bodies in the ragdoll
int ragdollSystemGroup = groupFilter.getNewSystemGroup();

// Caculate the filter info for the root body (e.g chest) with subsytem ID 0
hkUint32 chestFilterInfo = hkpGroupFilter::calcFilterInfo(COLLISION_LAYER_RAGDOLL, ragdollSystemGroup, 0,
  0);

// Caculate the filter info for a body directly constrained to the root (e.g. neck) with subsystem ID 1
hkUint32 neckFilterInfo = hkpGroupFilter::calcFilterInfo(COLLISION_LAYER_RAGDOLL, ragdollSystemGroup, 1, 0
  );

// Caculate the filter info for a body constrained to the previous one (e.g head) with subsystem ID 2
hkUint32 headFilterInfo = hkpGroupFilter::calcFilterInfo(COLLISION_LAYER_RAGDOLL, ragdollSystemGroup, 2, 1
  );

// Enable collisions within the ragdoll layer if they were previously disabled
groupFilter->enableCollisionsBetween(COLLISION_LAYER_RAGDOLL, COLLISION_LAYER_RAGDOLL);
```

Since the subsystem ID check is symmetric it is only necessary to disable collisions in one direction between a pair of bodies. This allows us to consecutively disable collisions between every pair of constrained bodies in the ragdoll while allowing collisions between all non-constrained pairs.

Of course, another way of achieving the same result is by using the `hkpConstrainedCollisionFilter` (discussed below), but the subsystem ID solution is cheaper in terms of performance cost.

**Example Group Filter Setup**

For a more fleshed out example group filter setup, please examine <`hkpCollide/Filter/Group/hkpGroupFilterSetup.h`>, which representative of a typical game scenario: the filter is set up with a static landscape layer, a dynamic object layer, a player layer, an AI object layer, a "keyframed" (i.e., non-physical) object layer and normal debris/fast debris layers.

One can derive the collision rules from looking at the code, but in short:

- Collisions between keyframed objects and other keyframed objects are disabled.
- Collisions between keyframed objects and static objects are disabled.
- Collisions between fast debris objects and other fast debris objects are disabled.

This filter setup is used by most Havok demos using any type of filtering. You can refer to the following demos for more detailed examples: The use case `DoorsDemo` and the use case `ArmsConveyorDemo`.

#### 2.3.1.3 Other Havok Filter Implementations

Aside from the default `hkpGroupFilter`, Havok provides a number of other useful filters. These filters are found across several different libraries (`hkpDynamics`, `hkpCollide`). This layout is chosen in order to minimize dependencies between the libraries. These filters include:

- `hkpNullCollisionFilter` (`hkpCollide`): This filter simply disables all collision filtering. All collisions will be accepted when this filter is used.
- `hkpCollisionFilterList` (`hkpCollide`): This filter holds a list of collision filters. For each collision it receives, it will test each of its filters' `isCollectionEnabled()` methods. If any of the collision filters return false, then the collision is filtered. If all return true, then the collision is accepted.
- `hkpPairCollisionFilter` (`hkpDynamics`): This simple filter enables collision filtering on a pair-by-pair basis. Each pair of entities in the simulation world can have collisions enabled (or disabled) on a individual basis. Please refer to the class documentation and to the `PairwiseCollisionFilterDemo` for more information.
- `hkpConstrainedCollisionFilter` (`hkpDynamics`): This filter disables collisions between two entities if they are connected by a constraint. It also supports a second optional filter which will be tested first making this a useful filter to use on top of a "standard" filtering system (like `hkpGroupFilter`). This filter is also very efficient in terms of updating and querying its list of constrained objects. Please refer to the code documentation for further details.

There are also a number of custom built collision filters available in a number of Havok Demos (search for classes that inherit from `hkpCollisionFilter` in the Havok Demo code base).

#### 2.3.1.4 Custom Filtering

If the default Havok filter implementations do not meet your or requirements, the collision filtering system can be further customized to address these concerns.

First, a short overview of the full Havok filtering pipeline.

**Stages of Collision Filtering**

Filtering happens at several stages of the collision detection pipeline:

1. The collision dispatcher: Havok ignores collisions between objects if the dispatched `hkpCollisionQualityInfo` is invalid. This is the default for collisions between fixed and keyframed objects.
2. The collision filter: Havok tests the collision against the active collision filter implementation. The user implementation of the `hkpCollidableCollidableFilter` is queried (if it is set).
3. The agent "filter": Havok generates an agent for the collision pair. If an appropriate agent cannot be found and created, then the collision is ignored.

**Collision Filtering by Type**

As mentioned earlier, the `hkpCollisionFilter` handles four distinct types of collisions. In most cases, the same filtering logic is used for all four types of collisions. However, this flexible system allows any number of customizations based on the behavior we wish to obtain. A particular interesting case to handle is collisions between Shape Collections (because the order child shapes are tested against each affects the result).

**Filtering on the Fly**

In some cases we may not know in advance whether two objects should collide or not. We would like to decide at the point of collision whether these two objects actually do collide. If they don't, then the collision must be filtered.

Consider a bullet hitting a thin wall. It may either collide with the wall or break it and pass through. We wish to make this decision at the instant the bullet collides (based on any number of parameters such as angle of collision, distance, thickness of wall, etc.).

In such cases, changing the collision filter will not help as the effects are delayed (as discussed in the Updating Collision Filters section). We can instead use the ability to disable contact points in the contact point callback. The contact will be ignored until the next simulation step (so that we can safely change the collision filter info of the object we no longer want to collide with, or delete it outright, when the simulation step has completed).

**Changing the Collision Quality Rules**

While it is not recommended, it is possible to change the rules used to filter collisions based on the Collision Quality info (stage 1 of the stages of collision filtering). These are the rules that define for example that collisions between fixed and keyframed objects are disabled.

The table defining these rules is `hkpCollisionDispatcher::m_collisionQualityTable`. Please be advised that editing this table will fundamentally change how collisions are handled internally.

#### 2.3.1.5 Shape Collection Issues

Handling collisions between Shape Collections presents interesting challenges. As each sub-shape can in turn have its own unique collision filter information, the order in which the collision filtering system processes these sub-shapes against each other is very important.

Shape vs Shape Collections filtering queries are actually handled as their own unique case. (As discussed earlier, Havok makes a distinction between four possible types of filtering queries).

Consider two shape collections A and B. If $A_1$ and $B_1$ are two subshapes of A and B respectively, then following filtering test occurs: the collision filter info of subshape $B_1$ is tested against the collision filter of the **root parent** of $A_1$ (which in this example is A). The subshape $A_1$'s filter is never tested against either $B_1$'s filter info or its parent B's filter info!

Therefore, depending on the on the order of the shapes in the collision pair, different filtering tests will occur. When one of the shape collections is **static** (such as a landscape), Havok ensures that its child shapes collision filters will always be tested (i.e. fixed shape collections will always be the B shape in our example). However, **when both shapes are dynamic** , the ordering of the shape pair cannot not be known ahead of time.

Therefore, we must take great care that the same filtering results will be obtained regardless of the

ordering. In other words, testing between $A_1$ and B should produce the same filtering decision as between A and $B_1$.

#### 2.3.1.6 Refreshing Collision Caches when Filtering is modified

Changing either the filter or the filter info will not immediately update the filtering behavior. Havok Physics 2012 caches various contact and collision information internally. The creation or deletion of contacts will not necessarily happen in the same simulation frame (and one certainly shouldn't rely on this). As such, the collision system must be explicitly notified of any filtering changes in order to update its cached information and create or delete the appropriate contacts.

This situation will be encountered often if the collision filter is only set up *after* entities have been added to the simulation world. Entities are considered "active" in the simulation world as soon as they are added to it. This entails that contact points may potentially already have been created for entities that are deemed to be colliding.

Adding (or changing) a collision filter afterward means that an update must be forced on the entire world to bring it to a consistent state in terms of filtering. This can be accomplished with the following API call:

```
void hkpWorld::updateCollisionFilterOnWorld(  hkpUpdateCollisionFilterOnWorldMode updateMode,
                                              hkpUpdateCollectionFilterMode updateShapeCollectionFilter);
```

The collision filter info for an `hkpCollidable` can also be updated (or, indeed, for the entire system). If a new collision filter info is set in the collidable, it will only be used for new overlaps. Existing collisions will not be affected. Therefore whenever changing the filter info, Havok must be notified to reevaluate some of its internal cached data. The API calls to update the collision filter info on collidables are:

```
void hkpWorld::updateCollisionFilterOnEntity(  hkpEntity* entity,
                                               hkpUpdateCollisionFilterOnEntityMode updateMode,
                                               hkpUpdateCollectionFilterMode updateShapeCollectionFilter );
```

```
void hkpWorld::updateCollisionFilterOnPhantom( hkpPhantom* phantom,
                                               hkpUpdateCollectionFilterMode updateShapeCollectionFilter );
```

The parameters of these methods specify what type of updates to perform:

1. The simulation world will perform a broadphase region query and update all the corresponding pairs using the updated collision filtering information:

| Parameter | Value |
|---|---|
| hkpUpdateCollisionFilterOnEntityMode | HK_UPDATE_FILTER_ON_ENTITY_FULL_CHECK |
| hkpUpdateCollisionFilterOnWorldMode | HK_UPDATE_FILTER_ON_WORLD_FULL_CHECK |

2. The simulation will only remove existing collision pairs that have become invalid. No new pair will be generated. This is much faster than option 1:

| Parameter | Value |
|---|---|
| hkpUpdateCollisionFilterOnEntityMode | HK_UPDATE_FILTER_ON_ENTITY_DISABLE_ENTITY_ENTITY_COLLISIONS_ONLY |
| hkpUpdateCollisionFilterOnWorldMode | HK_UPDATE_FILTER_ON_WORLD_DISABLE_ENTITY_ENTITY_COLLISIONS_ONLY |

3. The simulation world will also check for filter changes within shape collections. This must be used if the filter change occurred on an object colliding with the subparts of a landscape (i.e. a `MeshShape`):

| Parameter | Value |
|---|---|
| hkpUpdateCollectionFilterMode | HK_UPDATE_COLLECTION_FILTER_PROCESS_SHAPE_COLLECTIONS |

4. The simulation world will not check for filter changes within shape collections:

| Parameter | Value |
|---|---|
| hkpUpdateCollectionFilterMode | HK_UPDATE_COLLECTION_FILTER_IGNORE_SHAPE_COLLECTIONS |

As changing a collision filter is a major operation for the engine, it cannot be performed while the engine is locking its internal data structures. Calls to the `updateCollisionFilter` methods may be delayed automatically until the next safe time to execute them. This is typically the case when a filter is changed within a callback.

Further information on dealing with these situation (changes during callbacks), refer to the Filtering on-the-fly section.

**Note:**

Changing the collision filter using the above functions is a **slow operation** . All the collision agents are destroyed for the entity (or phantom) in question, and then the broadphase is re-queried for all possible collisions. Finally, the changes are propagated back through the system to recreate the agents with the updated filter information. Calling `updateCollisionFilterOnWorld` is nearly equivalent in work to removing all your objects from the world, and re-inserting them; **do not call this at runtime if at all possible** .

### 2.3.2 General Listeners

This section introduces Havok listeners and events, and describes how to use the listener base classes provided with Havok Physics 2012.

Listeners allow you to react to specific events that occur within the Havok simulation. These events can be anything from objects colliding, constraints being created, or entities being activated.

Unlike an action, which has a function that is called every single simulation step, listeners have functions that are only called when a particular event occurs. This makes them much more suitable for responding to infrequent changes of state, or for asynchronous events.

To register interest in an event, you simply add the appropriate listener to the simulation. When the event occurs, the specified callback code will be executed.

**Note:**

Collision Listeners are a special class of listeners which are explored in detail in their own chapter.

#### 2.3.2.1 Callbacks and Events

A listener in Havok simply implements a particular callback function that is called by the simulation when a corresponding event takes place. For example, the `hkpActionListener::actionAddedCallback()` method is called when an action is added to the simulation world.

Each callback allows the simulation to pass some useful information about the event to the listener, which the listener can then use when responding to the event. The type of information provided depends on the callback. For instance, an `entityAddedCallback()` is just passed a reference to the newly added entity, while an `hkpContactListener`'s `contactProcessCallback()` is passed a structure containing details of the event, including the `hkpCollidable`s involved and the collision detection results.

**Note:**

The world can be locked so that some operations are queued and not executed until a later point (see `hkpWorld::lockCriticalOperations`). This is often the case when callbacks are issued and may lead to subtle unexpected behavior: if you try to remove an entity, and that removal is deferred, then the entity may subsequently cause callbacks to be raised on listeners until it is actually removed.

#### 2.3.2.2 Listener Types

Havok provides you with a number of base listener classes, each with virtual callback methods that you can implement to respond to particular types of event. You can create listeners that respond to events for a particular entity, or your listener can respond to events for the entire Havok world. Implementations of some of these listener interfaces are also used internally by Havok. The following sections introduce you to the Havok listener classes and tell you how to add these types of listener to the simulation.

**Note:**

Certain listeners are only triggered in singlethreaded simulations. These are highlighted in a dedicated section.

**World Listeners**

The following listener types can be added to the simulation world, using the appropriate `hkpWorld add..Listener()` function. You can see some examples of world listeners in the `WorldListenerApi` demo.

**Island Activation Listeners**

Listeners inheriting from `hkpIslandActivationListener` are notified when any of the world's simulation islands are activated or deactivated. Both the `islandActivatedCallback()` and `islandDeactivatedCallback()` are passed a reference to the relevant simulation island. You add and remove this type of listener using `addIslandActivationListener()` and `removeIslandActivationListener()`.

**Constraint Listeners**

Listeners inheriting from `hkpConstraintListener` are notified when constraints are deleted, or added or removed from the world. The `constraintDeletedCallback()`, `constraintAddedCallback()`, and `constraintRemovedCallback()` are all passed a reference to the appropriate constraint by the simulation. You add and remove this type of listener using `addConstraintListener()` and `removeConstraintListener()`.

Please note the order that constraint callbacks are triggered is not guaranteed to be deterministic. To ensure determinism, callbacks should be queued until the end of the physics step, where they should be sorted before being applied. The `BattlefieldMoppDemo` illustrates this technique as applied to collision callbacks (which experience a similar problem with determinism).

#### World Deletion Listeners

Listeners inheriting from `hkpWorldDeletionListener` are notified when the `hkpWorld` is deleted. The `worldDeletedCallback()` is passed a reference to the `hkpWorld`. You add and remove this type of listener using `addWorldDeletionListener()` and `removeWorldDeletionListener()`.

#### World Post Collide Listeners

Listeners inheriting from `hkpWorldPostCollideListener` are notified after all collision detection has been performed for all entities in the world. The `postCollideCallback()` is passed a reference to the `hkpWorld`, together with an `hkStepInfo` that stores the simulation's current delta time and inverse delta time. You add and remove this type of listener using `addWorldPostCollideListener()` and `removeWorldPostCollideListener()`.

#### World Post Integrate Listeners

Listeners inheriting from `hkpWorldPostIntegrateListener` are notified after the end of the `hkpWorld::integrate()` call. The `postIntegrateCallback()` is passed a reference to the `hkpWorld`, together with a `hkStepInfo` that stores the simulation's current delta time and inverse delta time. This type of listener can be added and removed using `addWorldPostIntegrateListener()` and `removeWorldPostIntegrateListener()`. **Note that this listener is not triggered in multi-threaded simulations** .

#### World Post Simulation Listeners

Listeners inheriting from `hkpWorldPostSimulationListener` are notified after each time the `hkpWorld`'s `simulate()` function is called; they are fired after all the other callbacks in a given simulation step. `simulate()` is a protected function called inside `stepDeltaTime()`, which you use to step the simulation forward. The `postSimulationCallback()` is passed a reference to the hkpWorld. You add and remove this type of listener using `addWorldPostSimulationListener()` and `removeWorldPostSimulationListener()`.

A typical use of this callback might be to update the visual representations of the active rigid bodies in the games scene graph automatically, rather than performing the call after the simulation step call.

#### Island Post Collide Listeners

Listeners inheriting from `hkpIslandPostCollideListener` will receive the callback after all collision detection has been performed for all entities in that island. This type of listener allows you to perform similar collision filtering, etc., as the `hkpCollisionListener`s for `hkWorld`s and `hkpEntity`s but work at a different level of granularity, i.e. `hkSimulationIsland`s.

#### Island Post Integrate Listeners

Listeners inheriting from `hkpIslandPostIntegrateListener` with this callback will receive the callback after `integrate()` is called for the `hkpSimulationIsland`. Again, it is useful for situations where you want to filter or operate at a granularity of islands. Note that this callback is not raised when the simulation type is `MULTITHREADED`. This is because the integration and collision detection are interleaved

to improve multithreaded resource utilization. **Note that this listener is not triggered in multi-threaded simulations.**

#### Action Listeners

The `hkpActionListener` has three callbacks:

- `actionDeletedCallback()`: Called when an action is deleted, note that hkpActionListener subclasses must implement this function.
- `actionRemovedCallback()`: Called when an action is removed from the hkpWorld.
- `actionAddedCallback()`: Called when an action is added to the hkpWorld.

The `actionDeletedCallback()`, `actionAddedCallback()`, and `actionRemovedCallback()` are all passed a reference to the appropriate hkpAction by the simulation. You add and remove this type of listener using `addActionListener()` and `removeActionListener()`. An example of this listener can be found in the "World Listener" demo (`Physics2012/Api/Dynamics/World`).

#### Entity Listeners

Entity listeners can be useful for maintaining the state of various internal game data structures (such as scene graphs, etc.). They provide a mechanism of tracking when entities are added to the world, removed from the world and deleted.

The `hkpEntityListener` has three callbacks:

- `entityDeletedCallback()`: Called when an entity is deleted. Note that classes inheriting from `hkpEntityListener` must implement this method.
- `entityRemovedCallback()`: Called when an entity is removed from the hkpWorld.
- `entityAddedCallback()`: Called when an entity is added to the hkpWorld.

Each callback is passed a reference to the relevant `hkpEntity`. An `hkpEntityListener` can either be added to a particular entity (to receive events for only that particular entity), or added to the `hkpWorld` (to received events for all entities in the simulation world).

When implementing the `hkpEntityListener` interface, the `entityDeletedCallback()` method must be implemented. It must call `removeEntityListener()` to remove the listener from the entity's listener queue.

Entity listeners are added to and removed from the simulation world using the relevant `addEntityListener()` and `removeEntityListener()` methods on `hkpWorld` and `hkpEntity`.

`hkpEntityListener` callbacks are also used internally by Havok (for example, `hkpCharacterProxy` is also a `hkpEntityListener`).

#### hkpEntityActivationListener

Listeners implementing the `hkpEntityActivationListener` interface are notified when a particular entity is activated or deactivated. You add or remove an `hkpEntityActivationListener` using the relevant `hkpEntity`'s `addEntityActivationListener()` and `removeEntityActivationListener()` methods.

#### Phantom Listeners

Phantom listeners operate in a similar fashion to entity listeners, allowing the user to be notified of phantom addition, removals and deletions. The phantom overlap listener also provides notifications of when object enter and leave the space defined by the phantom. For more information on usage, please refer to the reference manual entry for the `hkpPhantomListener` and the `hkpPhantomOverlapListener`.

#### Break Off Listener

The `hkpBreakOffPartsListener` is used in conjunction with the `hkpBreakOffPartsUtil` to handle breaking off child shapes in a rigid body. The `hkpBreakOffPartsListener` has a single callback (`breakOffSubPart()`) which is triggered when the breaking threshold of the child shape has been exceeded during a collision.

The `hkpBreakOffPartsUtil` is used to both mark the pieces that can be broken off and to remove the broken off child shapes from the parent list shape of the object affected. To mark sub-shapes as breakable:

```
breakOffPartsUtil->markPieceBreakable( entity, subShapeKey, impulseThreshold );
```

To safely remove the broken off shape in from the parent list shape:

```
hkpBreakOffPartsUtil::removeKeysFromListShape( entity, keysToRemove, numKeys);
```

For complete examples of the `hkpBreakOffPartsUtil`, refer to `BreakOffPartsDemo` and `BreakOffExtendedMeshDemo`.

**Note:**
The `hkpBreakOffPartsUtil` is small utility to provide limited breaking functionality to the Havok Physics 2012 product. For a full range of breaking and destruction ability, Havok offers a separate Destruction product.

#### Single-Threaded Listeners

Several of the listeners are only triggered in single-threaded mode, so special care should be taken when running multithreaded simulations. These single-threaded-only listeners are:

- hkpWorldPostIntegrateListener
- hkpIslandPostIntegrateListener

### 2.3.3 Collision Listeners

The Havok library provides a number of notification mechanisms for collision events. Users can plug in at a very low level (using the contact listener interface) or leverage high-level utilities (such as the Trigger Volume and Phantom Callback Shape).

Finally, Havok also provides a number of commonly used collision response tools using the response modifier interface. These are useful when the primary goal is to **react** to collision events rather than

**intercepting** them.

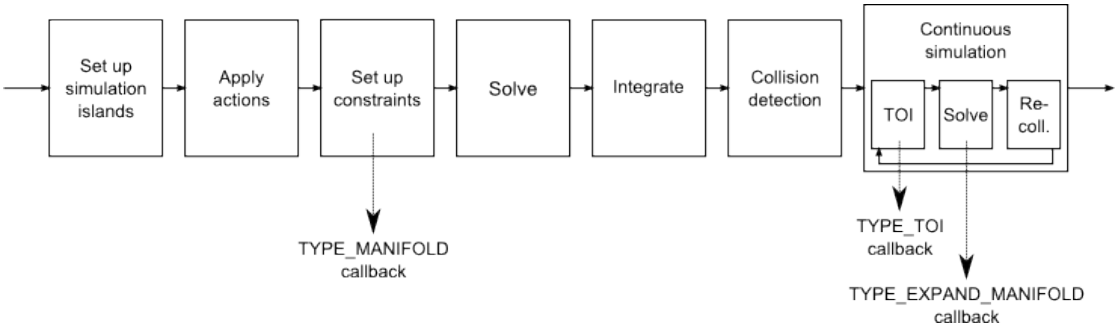
#### 2.3.3.1 The Contact Listener Interface

Using the `hkpContactListener` interface, you can obtain various information about collisions. It works on all platforms without modification. The listener can either be attached to an entity (where it will be informed about collisions involving this entity) or attached to the world (where it will be informed about all collisions). The `hkpEntity::addContactListener()` or `hkpWorld::addContactListener()` methods will register a listener.

The listener provides an interface with several callback methods which can be implemented. By far the most important is `contactPointCallback()`.

**contactPointCallback() and hkpContactPointEvent**

The `contactPointCallback()` receives a `hkpContactPointEvent` which contains details of a specific contact point between two colliding entities. The callbacks can be triggered at different stages in the physics step:




In the order in which they may occur, these are:

**TYPE_MANIFOLD:** An `hkpContactPointEvent` of this type is fired near the beginning of the physics step, before contact points are used by the solver. This is the most common type of contact point and represents a collision happening at the PSI (physical synchronous instance) corresponding to the start of the frame.

**TYPE_TOI:** An `hkpContactPointEvent` of this type represents a TOI (time of impact) event occurring in continuous physics. One of the bodies must be marked with a high collision quality for this to be fired.

**TYPE_EXPAND_MANIFOLD:** An `hkpContactPointEvent` of this type represents a contact point between bodies in the immediate vicinity of a TOI.

**Maintaining Determinism**

Contact points callbacks (along with Actions) are the main potential causes of non-determinism in Havok. All `contactPointCallbacks` are fired by CPU threads. Events of `TYPE_TOI` and `TYPE_EXPAND_MANIFOLD` are fired singlethreaded and in deterministic order.

However, **events of TYPE_MANIFOLD should not be assumed to occur in deterministic order** . They may be fired simultaneously from different threads, but callbacks for contact points involving the same **dynamic** entities will not be fired simultaneously. Note that this does not hold for **fixed** entities.

To ensure determinism, the callback operations should be queued until the end of the physics step, where they should be sorted before being applied. The `BattlefieldMoppDemo` illustrates this technique. Alternatively, use the end-of-step contact point callbacks and request that they be fired deterministically.

**Callback Frequency**

By default, `contactPointCallbacks` are fired only for new contact points. However, you can request that they be fired for all existing contacts too by setting the `hkpSimpleConstraintContactMgr::m_contactPointCallbackDelay` or, in advance of the collision, setting one of the entities' `m_contactPointCallbackDelay` values.

This value sets the frequency of callbacks for the full contact manifold. When a new collision between two bodies is detected, the contact manager's `m_contactPointCallbackDelay` value is set to the minimum of the entities' `m_contactPointCallbackDelay` values (which has a default of 65535).

Each time the collision is processed (at least once a frame, but can be more when it is involved in continuous physics), the manager's `m_contactPointCallbackDelay` value is decremented. A full set of `contactPointCallbacks` is fired when it reaches zero and the value is re-initialized. Note that all attached listeners receive the callbacks.

When callbacks for the full contact manifold are fired, you can use the event's `m_firstContactForFullManifold` and `m_lastCallbackForFullManifold` members to distinguish the first and last callback issued for the contact points in the manifold.

For an example of using callbacks for the full contact manifold, see the `ContactStateDemo`.

**Shape Keys**

By default, the `hkpContactPointEvent` does not include information about which subshapes of the entities contain the contact point. However, when an entity's `hkpEntity::m_numShapeKeysInContactPointProperties` is set to greater than 0, space is set aside for storing shape keys which can be used to traverse the shape hierarchy. (Note that for TOI events the sum total number of shape keys for both colliding bodies is limited by `hkpContactPointProperties::HK_NUM_EXTENDED_USER_DATAS_IN_TOI_EVENT`.)

Once this is set, you can build a `hkpShapeKeyPath` from the event and use the associated `Iterator` to access the shape keys. It provides pointers to the shapes themselves. However, it is only safe to use this method when the entity's `m_numShapeKeysInContactPointProperties` is big enough for any branch in the shape hierarchy. If a full set of shape keys is required, then the entity's `m_numShapeKeysInContactPointProperties` should be set to the maximum depth of any branch in the entity's shape hierarchy. The constructor of a rigid body will do this automatically if `hkpRigidBodyCinfo::m_numShapeKeysInContactPointProperties` is set to -1. As this is done using a costly traversal of the shape hierarchy, It should generally not be used at run-time.

To traverse the shape hierarchy from the root shape to the leaf, you can use the the iterator's `getShape()` method to access the shape it points to in conjunction with the `next()` method to advance the iterator down one shape, as shown in the example below.

```
// Access the shape hierarchy of body designated by bodyIdx.
hkpShapeKeyPath path( event, bodyIdx );
for (hkpShapeKeyPath::Iterator it = path.getIterator(); it.isValid(); it.next())
{
    const hkpShape* = it.getShape();
    // Do something with the shape.
}
```

However, the pointer returned by `getShape()` can be invalidated by subsequent calls to `next()` or if the iterator is destroyed. To get the shapes in a more permanent storage, you can access all the shape pointers at once by calling the `getShapes()` method from `hkpShapeKeyPath`. This method requires that you supply enough storage for all the shapes in an array of `hkpShapeBuffer`.

```
const int MAX_SHAPES = 10;
const hkpShape* shapes[MAX_SHAPES];
hkpShapeBuffer buffers[MAX_SHAPES];

int numShapes = 0;
path.getShapes( MAX_SHAPES, buffers, shapes, numShapes );
```

Please note the following caveats when traversing the hierarchy of shapes:

- The shape keys may terminate on a `hkpConvexTranslateShape` or `hkpConvexTransformShape`. The hierarchy may itself continue with a sequence of `hkpConvexTranslateShapes` and `hkpConvexTransformShapes` before the convex leaf shape is reached.
- If a `hkpSampledHeightFieldShape` is involved in a collision, the shape keys will lead to the height field and will **not** indicate which part of the height field was hit. If the height field is wrapped in a `hkpTriSampledHeightFieldCollection` or a `hkpTriSampledHeightFieldBvTreeShape`, then the shape keys lead to a child `hkpTriangleShape`.

For an example on the use of shape keys in contact point callbacks, see the `ShapeKeysInCallbackDemo`.

**collisionAddedCallback, collisionRemovedCallback and hkpCollisionEvent**

The contact manifold (i.e. the set of contact points occurring at the next PSI) is ready at the end of the physics step. However, the `contactPointCallbacks` of `TYPE_MANIFOLD` are not fired until the beginning of the next simulation frame.

In cases where the collision information is needed at the end of the simulation step (to trigger game logic which cannot tolerate a frame delay), use the collision callbacks to be notified which entities are colliding at that stage.

If `hkpWorldCinfo::m_fireCollisionCallbacks` is set to true, the addition and removal of contact constraints will trigger `collisionAddedCallback()` and `collisionRemovedCallback()` to be triggered correspondingly. These are fired singlethreaded, but not necessarily in a deterministic order.

The callbacks provide `hkpCollisionEvents` containing pointers to the colliding entities and the contact manager responsible for the collision. As long as an unfinished collision exists between two entities at the end of the step, their contact manifold will contain contact points in it.

**Note:**

When using the `hkpCollisionEvents` with continuous physics, if a `collisionAddedCallback` and a `collisionRemovedCallback` for the same collision are triggered in the same frame, it is possible that no collision has actually occurred. In this situation, a collision has occurred if and only if a `contactPointCallback` was fired.
The `hkpTriggerVolume` illustrates how to handle this subtlety. When we know that continuous physics will not interfere with a collision, then the occurrence of a `collisionAddedCallback` is sufficient to know that a collision occured. This assumption is made in the `BattlefieldMoppDemo`.

### End-of-step contactPointCallbacks

The collision callbacks can be used to determine when two entities have points in their contact manifold. It is straightforward to manually write the code to examine the contact manifold, but Havok provide a standard implementation.

By using the methods `hkpContactListener::registerForEndOfStepContactPointCallbacks()`and `hkpContactListener::unregisterForEndOfStepContactPointCallbacks()`, the listener will receive a set of `contactPointCallbacks` of `TYPE_MANIFOLD_AT_END_OF_STEP` at the end of the simulation step (more precisely, these will be triggered by a `hkpWorldPostSimulationListener`).

These are almost identical to the `TYPE_MANIFOLD` events which are fired early in the next step. The significant differences are:

- The separating velocity of the contact is not set.
- They are fired singlethreaded.

The `EndOfStepCallbackDemo` presents a simple example of end-of-step callbacks.

Registering too many collisions for end-of-step callbacks will result in a performance hit, as all registered collisions are stored in an array and processed by a single thread.

### Determinism

As firing the callbacks in deterministic order requires additional operations (e.g. sorting and merging arrays) which naturally incur a performance hit, determinism is disabled by default.

Callback determinism can be enforced like this:

```
hkpCollisionCallbackUtil* util = hkpCollisionCallbackUtil::findCollisionCallbackUtil( world );
util->m_endOfStepCallbackUtil.m_deterministicOrder = true;
```

For an example, see the `DeterministicCallbacksDemo`.

### Disabling Contact Points

Contact points can be disabled by setting the `hkContactPointMaterial::CONTACT_IS_DISABLED` flag in their `hkpContactPointProperties::m_flags` member. A disabled contact point has no physical effect, but is still updated by the collision system and can be re-enabled by clearing the flag. For an example of how to make use of disabled contact points, see the `ShapeKeysInCallbackDemo` or the `hkpTriggerVolume` utility.

### Response Modifiers and contactPointCallback

A common use case for callbacks is to modify the collision response using a response modifier (see the Response Modifier section).

**Prior to using a response modifier in events of TYPE_MANIFOLD, one or both of the colliding entities must be marked with a flag** in their `hkpEntity::m_responseModifierFlags` member (the flag values are defined in `hkpReponseModifier`).

Furthermore, when modifiy the collision response manually within a contact point callback, the functions in `hkpAddModifierUtil` must be used instead of those in `hkpResponseModifier`.

This flag is required because the physics simulation must allocate the extra memory necessary for the use of response modifiers prior to separating the simulation work into multithreaded jobs. Due to this memory increase, response modifiers should be used conservatively.

### Modifying a Body's Velocity within a Callback

The velocity of a colliding body can be modified in a `contactPointCallback` (e.g. by applying an impulse or a force). In order for this to have an effect, the modification must be preceded with a call to `hkpContactPointEvent::accessVelocities()` and followed by a call to `hkpContactPointEvent::updateVelocities()`.

The `UserCollisionResponseDemo` illustrates an example of this feature.

### Modifying a Contact Normal within a Callback

Contact normals should be modified using `hkpContactPointEvent::setContactNormal()` rather than setting the contact point normal directly, as there are several contact parameters which must be recalculated whenever the normal is changed. This utility function calculates those parameters based on the provided normal.

### Keyframed Objects Reporting

By default, Havok does not report collisions between keyframed objects and static objects (or other keyframed objects). As a consequence collision and contact point callbacks are not triggered for interactions between these two types of objects.

If you wish for these callbacks to be triggered nonetheless, you must change the collidable quality of the keyframed object from `HK_COLLIDABLE_QUALITY_KEYFRAMED` to `HK_COLLIDABLE_QUALITY_KEYFRAMED_REPORTING` (or set it in `hkpRigidBodyCinfo::m_qualityType` when creating the object). Simply call the following method on the keyframed object.

```
keyframedObject->setQualityType(HK_COLLIDABLE_QUALITY_KEYFRAMED_REPORTING);
```

The object will now be notified of all contacts and TOI-collisions. Please also refer to the `ToiCollisionsForKeyframedBodiesDemo` for an example.

The subject of collidable quality type is explored in further details in the chapter on continuous physics.

#### 2.3.3.2 Trigger Volumes

The hkpTriggerVolume offers a way of detecting rigid bodies in contact with a particular region of space. It is implemented using a hkpRigidBody and an attached hkpContactListener which records, but disables, collision events.

It offers an alternative to the hkpPhantomCallbackShape. Its main advantages is that it picks up continuous events (it can identify a body which enters and leaves in the same frame, provided that the appropriate quality types are used) and is supported on SPU. Its main drawback is that it is more expensive performance-wise.

**Detecting Interactions between Trigger Volumes and Rigid Bodies**

To use a hkpTriggerVolume, we must create a subclass which implements one or both of its triggerEventCallback() methods. The callbacks are usually fired by a postSimulationCallback, but can also be triggered when the trigger's body is removed from the world, or manually by calling updateOverlaps().

The event parameter of a triggerEventCallback can have several values which fall into two categories:

- Entering events: ENTERED_EVENT and EVENTED_AND_LEFT_EVENT.
- Leaving events: LEFT_EVENT, ENTERED_AND_LEFT_EVENT and TRIGGER_BODY_LEFT_EVENT.

The event values are designed to support the following pattern, which handles the situation where a body enters and leaves the trigger volume in the same frame:

```
void MyTriggerVolume::triggerEventCallback( hkpRigidBody* body, EventType type )
{
  if ( type & ENTERED_EVENT )
  {
      // Handle entering logic here.
  }
  if ( type & LEFT_EVENT )
  {
      // Handle leaving logic here.
  }
}
```

The hkpTriggerVolume::m_overlappingBodies array holds the list of bodies currently in contact with the hkpTriggerVolume. As it is updated after the callbacks are fired, it can be queried during a callback to discover the state of the hkpTriggerVolume at the end of the previous frame.

**Detecting Interactions Between Trigger Volumes and Character Proxies**

The interaction between hkpTriggerVolumes and other hkpRigidBodys is handled by a hkpContactListener during the physics step. However, hkpCharacterProxys are integrated outside the physics step and do not cause the callbacks in hkpContactListener to fire. Instead, we've added explicit support for hkpTriggerVolumes to the hkpCharacterProxy's integrate method. (See the hkpCharacterProxy section.)

The hkpTriggerVolume has an additional triggerEventCallback() which takes a hkpCharacterProxy

as an argument.

**Trigger Volumes and the Box-Box Collision Agent**

Havok does not recommend the use of a `hkpBoxShape` as the shape for a Trigger Volume. An equivalent `hkpConvexVerticesShape` should be used instead.

The `hkpBoxShape` is discouraged as the box-box agents occasionally do not regard two overlapping boxes as being in contact (due to numerical accuracy issues). This does not have particularly serious consequences for normal rigid bodies, since the collision is typically picked up in the subsequent frame, and resolved. Moreover, box-box agents are only used for low quality collisions (high quality collisions are handled by the predictive GSK agent).

Unfortunately, the situation is aggravated when these agents are used with the `hkpTriggerVolume`, since the failed collision detection might cause it to issue a `LEFT_EVENT`. If important game logic is run by within `triggerEventCallback()`, then this may have unfortunate consequences for the consistency of the game.

Although the problem cases are very rare, we recommend that boxes aren't used as trigger bodies if a box-box agent is registered (debug code in the `hkpTriggerVolume` will identify this situation and issue a warning). The remedy is either not to register the box-box agent, or replace the `hkpTriggerVolume`'s `hkpBoxShape` by an equivalent `hkpConvexVerticesShape`.

#### 2.3.3.3 Phantoms as Detectors

While phantoms are primarily used for world queries, they can also be used as object detectors. Before using phantoms, you should note two important drawbacks:

- Phantoms do not perform continuous quality collision detection.
- Phantoms only receive callbacks for AABB overlaps. You will have to query the results further to obtain the narrowphase results.

If these drawbacks do not hamper you, then phantoms can be a useful and fast alternative detector. If continuous detection is important, we recommend trigger volumes. For directly obtaining narrowphase results, we suggest using contact listener callbacks.

#### 2.3.3.4 Phantom Callback Shape

Havok provides an alternate solution to the Trigger Volume for caching user collision queries: the phantom callback shape (`hkpPhantomCallbackShape`). **The phantom callback shape should not be confused with phantoms** . This is a specialized shape (`hkpShape`) that simply detects other overlapping objects.

As this `hkpShape` is part of the physics collision detection system, it can be used like any `hkpShape`. For example, it can be the single shape forming a rigid body, or as part of a hierarchy of shapes forming a list shape. An `hkpPhantomCallbackShape` has no physical effect in a scene (that is to say its collisions **do not create contacts** which affect the body which owns it), but it will trigger events when other shapes penetrate it.

The `hkpPhantomCallbackShape` can be useful in a number of game scenarios. For example, a vehicle in a city landscape: an `hkpPhantomCallbackShape` in the form of a triangle can be attached to the front of the car to detect if there are any collidable objects in front of it (such as another car, or some scenery). The information received by the phantom callback shape can be fed back into the AI system to avoid potential collisions.

Figure 2.60: Example Phantom Callback Shape object detection

The `hkpPhantomCallbackShape` is typically used with a bounding volume shape (`hkpBvShape`). The phantom shape (defining the collision area) is set as the child shape of an `hkpBvShape` (which defines the bounding volume of the collision shape).

If an object is found to collide with the bounding volume, a mid and narrowphase collision check is carried out between the child shape and the colliding object. If these two overlap, the callbacks will be triggered, otherwise no event will be generated.

**Implementation Example**

The `PhantomEventsDemo` illustrates the typical usage of a phantom callback shape. The `PhantomEyesDemo` also provides an alternate example using multiple phantom callback shapes.

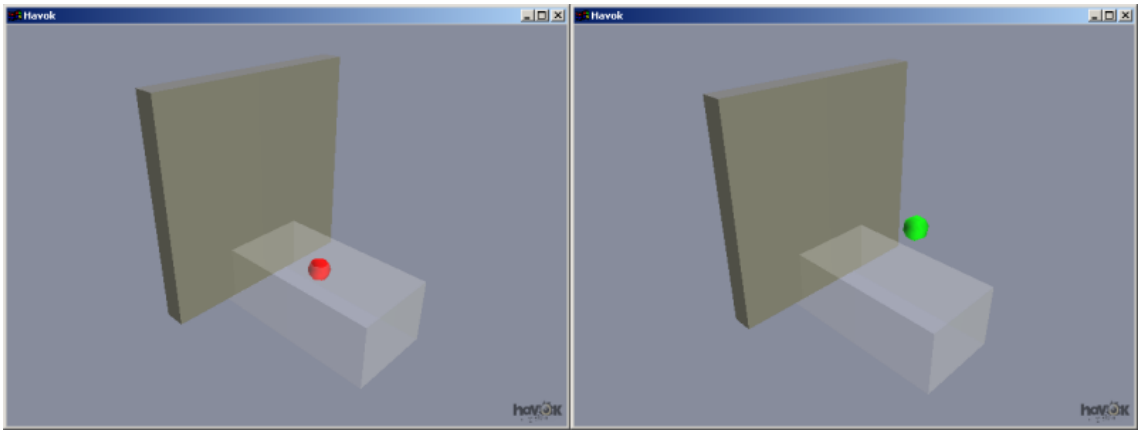



Figure 2.61: An example from the demo framework using a phantom callback shape

In this example, a `hkpBvShape` is used. the bounding volume is defined by the transparent `hkpBoxShape` and the child shape is a `hkpPhantomCallbackShape` implementation:

```
// MyPhantomShape is the demo implementation of the hkpPhantomCallbackShape.
MyPhantomShape* myPhantomShape = new MyPhantomShape();
hkpBvShape* bvShape = new hkpBvShape( boxShape, myPhantomShape );
boxShape->removeReference();
myPhantomShape->removeReference();

boxInfo.m_shape = bvShape;

hkpRigidBody* boxRigidBody = new hkpRigidBody( boxInfo );
bvShape->removeReference();

m_world->addEntity( boxRigidBody );
```

Whenever a shape enters or leaves the space occupied by the `hkpBoxShape`, a corresponding phantom event will be raised. Within these entry and exit event methods (given below) we color the sphere to indicate whether or not it is inside the phantom volume. An upward impulse is applied to the body if it is leaving the phantom volume with a downward velocity. This impulse causes it to bounce back up through the box again whenever it "falls out of the bottom".

```
// hkpPhantom interface implementation
virtual void phantomEnterEvent( const hkpCollidable* collidableA, const hkpCollidable* collidableB, const
  hkpCollisionInput& env )
{
    // the color can only be changed once the entity has been added to the world
    hkpRigidBody* owner = hkpGetRigidBody(collidableB);

    // the "Collidables" here are "faked" so it's necessary to get the owner first in order
    // to get the "real" collidable!
    HK_SET_OBJECT_COLOR((hkUlong)owner->getCollidable(), hkColor::rgbFromChars(255, 0, 0));
    //HK_SET_OBJECT_COLOR((int)&collidableB, hkColor::rgbFromChars(255, 0, 0));
}

// hkpPhantom interface implementation
virtual void phantomLeaveEvent( const hkpCollidable* collidableA, const hkpCollidable* collidableB )
{
    // the color can only be changed once the entity has been added to the world
    hkpRigidBody* owner = hkpGetRigidBody(collidableB);

    // the "Collidables" here are "faked" so it's necessary to get the owner first in order
    // to get the "real" collidable!
    HK_SET_OBJECT_COLOR((hkUlong)owner->getCollidable(), hkColor::rgbFromChars(0, 255, 0));

    // If moving out AND falling down, apply impulse to fire it towards "wall"
    if(owner->getLinearVelocity()(1) < 0.0f)
    {
        hkVector4 impulse(-50.0f, 220.0f, 0.0f);
        owner->applyLinearImpulseAsCriticalOperation(impulse);
    }
}
```

### 2.3.4 Response Modifiers

Response modifiers allow users to selectively alter the behavior of how bodies respond to collisions and constraints. Response modifiers take effect during constraint solving–which is multithreaded–and are generally efficient to use, and in the case of collisions are recommended over custom contact point callback implementations when the desired behavior is identical. For example, it is better to use the provided soft

response modifier than to manually alter contact points in a contact callback to achieve the same effect.

An example use case for response modifiers would be a loose car door. In this scenario, a car's door is attached to its chassis via a hinge constraint, and the door is swinging out from the chassis while driving. Both bodies have realistic masses, so that they can realistically interact with other bodies in the world. However, if the open door hits some static geometry in the world you may not want the motion of the chassis to be affected at all by that collision. The best solution for this use case is to use a "mass changer" response modifier on the constraint between the door and chassis, telling the chassis to have infinite mass for that particular constraint. This means that the motion of the door won't be able to affect the motion of the chassis, but the motion of the chassis will still affect the motion of the door.

Note that since contact points are a type of constraint, response modifiers can be applied to collisions in effectively the same way that they can be applied to "normal" constraints.

#### 2.3.4.1 Adding and removing modifiers

Response modifiers are applied by creating a `hkpConstraintModifierAtom` and applying it to a constraint using `hkpWorldConstraintUtil::addModifier()`. For example:

```
hkpModifierConstraintAtom* atom = new hkpMassChangerModifierConstraintAtom();
atom->m_factorA = 1.0f;
atom->m_factorB = 10.0f;

hkpConstraintInstance* constraintInstance = [...];    // see below
hkpConstraintOwner* constraintOwner = constraintInstance->getSimulationIsland();
hkpWorldConstraintUtil::addModifier( constraint Instance, constraintOwner, atom );
```

To remove a modifier, call:

```
hkpWorldConstraintUtil::removeAndDeleteModifier( constraintInstance, constraintOwner, atom->getType() );
```

**Warning:**
At most one modifier of each type can be active for any single pair of bodies.

#### Retrieving the constraint instance

For "normal" constraints (ball-and-socket, hinge, etc.) the `hkpConstraintInstance` pointer required by the functions above is readily available. For collisions, the constraint instance must be retrieved via the contact manager between the two bodies involved. Contact managers are created and destroyed internally by the engine (specifically, by the collision agents), and can be accessed via a body's linked collidables and agents.

```
// For collisions
hkpSimpleConstraintContactMgr* contactManager = [...];
hkpConstraintInstance* constraintInstance = contactManager->getConstraintInstance();
```

In practice, the common method of applying response modifiers to contact points is to use a collision listener, which already has access to the relevant contact manager. Therefore, Havok Physics 2012 includes utilities for the most common contact point response modification use cases, using contact point listeners to add the response modifiers whenever new contacts are created. These utilities are described next.

#### 2.3.4.2 Contact point modifier utilities

These utilities take the form of entity collision listeners that automatically apply response modifiers to any contact constraints created by the engine. Each utility keeps an extra reference to itself and deletes itself when the body it's attached to is deleted.

**Warning:**
If a response modifier is intended to be used during a contact manifold callback (`TYPE_MANIFOLD`), then it is a requirement that at least one of the two colliding entities has previously been flagged for response modification via their `hkpEntity::m_responseModifierFlags` member. In this situation it is also a requirement to make use of the `hkpAddModifierUtil` methods instead of those in `hkpResponseModifier`. More information is available in the contact point callback section.

**Mass Changer Utility**

This utility changes relative masses of interacting bodies. This, for example, allows the solver to treat some bodies as if they are keyframed when colliding against selected other bodies or limit mass ratios in collision response.

To apply this utility use the sample code below.

```
hkpCollisionMassChangerUtil* util = new hkpCollisionMassChangerUtil(body0, body1, body0InvMassMultiplier,
  body1InvMassMultiplier);
util->removeReference();
```

`bodyXInvMassMultiplier` parameters are the mass scaling parameters. The new mass is equal to the product of the original mass and the inverse of the parameter value.

The more advanced application of this utility allows you to separately control the mass multiplier and the inertia tensor multiplier. Using this feature you can easily pin a body at a given location but allow it to spin freely, or do the opposite: fix its orientation and force it behave like a particle in collisions.

Also when scaling the inertia tensor you have the option to scale the inertia non-uniformly. You can scale the tensor independently along the three principal axes of the body. One could find this useful when e.g. trying to prevent vehicles from spinning during collisions–you can scale up the inertia along the front-to-back and left-to-right axes only, allowing the vehicle to easily react around the vertical axis.

To use the above feature, call the constructor with full vectors that specify inverse mass & tensor multipliers.

```
hkVector4 body0InvMassAndInertiaMultiplierVector(invInertiaAlongX0, invInertiaAlongY0, invInertiaAlongZ0,
  invMass0);
hkVector4 body1InvMassAndInertiaMultiplierVector(invInertiaAlongX1, invInertiaAlongY1, invInertiaAlongZ1,
  invMass1);
hkpCollisionMassChangerUtil* util = new hkpCollisionMassChangerUtil(body0, body1,
                                    body0InvMassAndInertiaMultiplierVector,
                                      body1InvMassAndInertiaMultiplierVector);
util->removeReference();
```

Note that using the mass changer utility will violate the preservation of momentum law.

**Center of Mass Changer Utility**

Similarly to the collision mass changer utility, this utility changes the mass properties of colliding objects.

To apply the utility specify the two objects for which it is to be applied, and specify the displacement vectors for the center of mass of each body. The vectors are in the local space of the corresponding body.

```
hkpCenterOfMassChangerUtil* util = new hkpCenterOfMassChangerUtil(body0, body1, massDisplacementInLocal0,
  massDisplacementInLocal1);
util->removeReference();
```

When displacing the center of mass away from the contact point, a larger part of the collision's energy will be transformed into angular impulse. Also the body with its center of mass displaced will appear to have lesser mass.

Note that using the center of mass changer utility will violate the preservation of momentum law.

An good example of when it can be useful to change the center of mass is with a vehicle. A low center of mass generally improves the handling of a vehicle in a physics simulation. However, collisions will introduce unrealistic torques to the vehicle leading to odd behavior. Using the center of mass changer utility can address this problem.

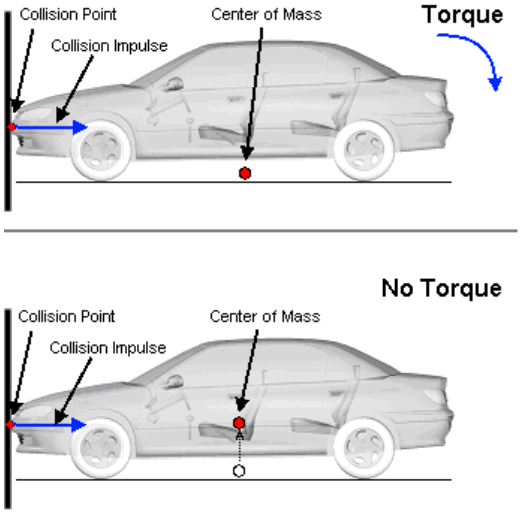



Figure 2.62: Center of Mass Changer on a Vehicle

### Soft Contact Utility

This utility scales the forces applied during collisions between two specified bodies. This allows an effect of soft and bouncy contacts.

To apply this utility use the sample code below.

```
hkpSoftContactUtil* util = new hkpSoftContactUtil(body0, body1, forceScale);
util->removeReference();
```

Note that `body1` can be set to `HK_NULL` to enable the utility for all rigid bodies colliding against `body0`.

Note that this utility only works when the accurate collision response is disabled by setting `hkpWorldCinfo::m_contactRestingVelocity = HK_REAL_MAX`.

Note that this utility should not be used for pairs of bodies which use continuous collision detection, as it may cause significant performance hits. (As the forces during TOI event handling are scaled as well, this may result in a large number of consecutive TOI events being recorded for one pair of bodies.)

### Surface Velocity Utility

This utility simulates movement of the surface of a body. By specifying the velocity of the surface you can simulate moving surfaces like conveyor belts or imitate movement of objects moved by a river's flow.

```
hkVector4 surfaceVelocity(0, 0, 1.0f);
hkpSurfaceVelocityUtil* util = new hkpSurfaceVelocityUtil(baseRigidBody, surfaceVelocity));
util->removeReference();
```

Note: this utility has limited application for dynamic or even keyframed bodies, as it only allows you to specify the velocity in the world's space upon the creation of a contact point. Therefore when the body changes the orientation, its surface velocity will remain unchanged in relation to the world's space instead of the body's space.

Note that the surface velocity should be tangent to the contact plane, otherwise you may get unexpected penetrations or bounciness at the contact.

### Filtered Surface Velocity Utility

This is an expansion of the functionality of the surface velocity utility. This utility gives you methods to enable or disable the surface velocity feature between the base body and any other body at any time. Look at the "`Physics2012/Api/Dynamics/Rigid Bodies/Surface Velocity/Filtered`" demo to see how the utility is combined with a Phantom Callback Shape to achieve a localized "conveyor belt" functionality.

Below is a snippet showing how the utility is created and how to enable and disable its functionality for a chosen body. When enabling the utility for a body the `propertyIdForEnabledEntities` is added to it. This property should be set to a **unique ID** .

```
hkVector4 surfaceVelocity(0, 0, 1.0f);
int propertyIdForEnabledEntities = 100;
filteredSurfaceVelocityUtil = new hkpFilteredSurfaceVelocityUtil( baseRigidBody, surfaceVelocity,
  propertyIdForEnabledEntities);

filteredSurfaceVelocityUtil->enableEntity(someEntity);
filteredSurfaceVelocityUtil->disableEntity(anotherEntity);

filteredSurfaceVelocityUtil->removeReference();
```

#### Viscose Surface Utility

This utility influences static friction between objects. It always allows for a slight relative movement between bodies in contact. It is useful in resolving situations where an object gets squished in and stuck in a narrow gap. In such situations normally friction might prevent the object from being moved at all. A typical application might be a car stuck between two buildings.

Note that you have to call `hkpViscoseSurfaceUtil::makeSurfaceViscose()` before you add the entity to the world.

```
hkpViscoseSurfaceUtil::makeSurfaceViscose(hkpRigidBody* entity);
world->addEntity(entity);
```

#### Break Off Parts Utility

`hkpBreakOffPartsUtil::LimitContactImpulseUtil` is used in conjunction with the Break Off Parts Listener to set the maximum impulse value on any contacts using the Break Off Parts Listener. When that maximum impulse is exceeded within the solver, the listener is notified. For usage details and examples, refer to the Break Off Parts Listener documentation.

### 2.3.5 Collision Queries

Havok provides facilities to asynchronously query the collision detection system in various ways. Asynchronous queries are performed by the user, outside of the simulation step (as opposed to callbacks which are triggered within the simulation step, as discussed above). There are many uses for asynchronous queries. Simple examples would include line-of-sight checks (to drive AI), or finding objects within a volume (to apply explosion damage). More advanced examples would include moving a character proxy without having to place the character in the full physics system, but while still avoiding interpenetration.

There are four primary ways to query the simulation world (`hkpWorld`):

**Ray Casting** :

```
// Casts a ray into the world and get a callback for every hit
// Note: it uses the hkpWorld::getCollisionFilter() for filtering
// This function uses a ray-cast function for traversing the broadphase, so for
// long raycasts it is more applicable than hkpAabbPhantom::castRay
void hkpWorld::castRay(const hkpWorldRayCastInput& input, hkpRayHitCollector& collector );
```

**Linear Casting** :

```
// Casts a shape within the world.
// The castCollector collects all potential hits.
// Note that the distance stored within the castCollector is a hitFraction (between 0 and 1.0) and
// not a distance.
// This function uses an AABB-cast function for traversing the broadphase, so for
// long linear casts it this will perform better than hkpShapePhantom::linearCast
// The [optional] startPointCollector returns all the closest points
// at the start point of the linear cast. If you do not want this functionality, pass HK_NULL as the
// "startPointCollector".
void hkpWorld::linearCast( const hkpCollidable* collA, const hkpLinearCastInput& input,
  hkpCdPointCollector& castCollector, hkpCdPointCollector* startPoCollector = HK_NULL );
```

**Get Closest Points** :

```
// Get the closest points to an hkpCollidable (= [shape,transform,filterInfo] )
// Note: If you have call this function every step for a given object, use the hkpShapePhantom::
  getClosestPoints()
void hkpWorld::getClosestPoints( const hkpCollidable* collA, hkpCollisionInput& input, hkpCdPointCollector
  & collector );
```

**Get Penetrating Shapes** :

```
// Get all shapes which are penetrating the collidable
void hkpWorld::getPenetrations( const hkpCollidable* collA, hkpCollisionInput& input,
  hkpCdBodyPairCollector& collector );
```

These queries are **completely asynchronous** , and **no caching** is done whatsoever. In each case, the broadphase is first queried for all possible overlapping candidates, and the query function is then performed on each of these candidates.

The ShapeQueryDemo shows off all four of these query methods as well as comparing the performance of various strategies. This demo should be the first point of reference when implementing these queries.

#### 2.3.5.1 Phantoms

Every collision query which can be performed on the simulation world (`hkpWorld`) can also be performed on a phantom (`hkpPhantom`).

Phantoms are an optimization structure used to limit the scope of an asynchronous query by limiting the set of objects queried to those overlapping the phantom's bounding volume. Phantoms perform this caching through a pair of broadphase callbacks (`addOverlappingCollidable()` and `removeOverlappingCollidable()`) which update a phantom whenever a shape has entered or left its area of interest. By restricting the query area, the `hkpPhantom` effectively limits the amount of processing necessary when performing an asynchronous query.

**Note:**
While it may be tempting to use phantoms as an object tracking tool, this is discouraged. Detecting when entities enter or leave certain spaces in the simulation world is already handled by the Trigger Volume utility and the phantom callback shape. Both provide better support and a complete feature set for this functionality.

Phantoms fall into two different categories, those that can be used for **raycasts** (`hkpAabbPhantom`), and those that are used for **shape casts** (`hkpShapePhantom` and its child implementations).

### hkpAabbPhantom

The AABB phantom operates by maintaining a list of overlapping objects. It only examines the broadphase representation of objects it encounters when deciding whether or not an overlap has occurred. The overlapping list is therefore updated very quickly. This caching system in turn benefits short raycasts as the query can immediately perform a shape level raycast on each of the currently overlapping shapes.

In contrast, using `hkpWorld::castRay()` for short raycasts requires that a list of overlapping candidates be built each time before performing the shape level casting. Building this list each time is a slow and inefficient process. For long raycasts, however, the world variant is preferred. A very large phantom object would be prone to significant amounts of updating.

### Creating an hkpAabbPhantom

The following details for an `hkpAabbPhantom` when creating it:

- **AABB**

  The *m_min* and *m_max* parameters specify the AABB that forms the phantom area. Each parameter is an `hkVector4`. By default, these are both set to the origin.

- **Collision Filter Information**

  This can be used by collision filters to identify the phantom. By default, this is set to zero. See the Collision Filter section for more information on its possible values.

The following example shows the creation of the phantom area from the Fountain example:

```
hkAabb info;
info.m_min = hkVector4( -2.0f, -10.0f, -2.0f );
info.m_max = hkVector4( 2.0f, 10.0f, 2.0f ) ;
m_phantom = new hkpAabbPhantom( info );
```

### hkpShapePhantom

The second type of `hkpPhantom` is the `hkpShapePhantom` which is the finer-grain relative of the `hkpAabbPhantom`. This phantom uses a shape to define its bounding volume and the user can subsequently perform asynchonous queries to detect which objects' shapes are overlapping with the phantom's shape at the narrowphase level.

Unlike the `hkpAabbPhantom`, the `hkpShapePhantom` does not have a raycast functionality. The `hkpShapePhantom` is primarily designed to be used with a given shape with respect to other shapes in the region. Instead of a ray cast, this phantom has a linear cast, which is used to project the entire shape volume into the world. A linear cast is also known as a shape cast.

The hkpShapePhantom has two primary query methods: getClosestPoints() and getPenetrations(). Both methods take collision point collectors that process the collision data as it is computed by the collision algorithms.

**Note:**
Unlike the hkpAabbPhantom, the hkpShapePhantom does not store an explicit AABB for the broadphase. Instead it computes an AABB from its hkpShape and associated transform.

The hkpShapePhantom has two default implementations: hkpSimpleShapePhantom and hkpCachingShapePhantom. The two varieties reflect opposite sides of the usual memory versus speed trade off, with the caching type boasting greater speed at the cost of increased memory usage.

The key difference between the two is that hkpCachingShapePhantom caches the agents required to process the collision data. hkpSimpleShapePhantom just makes explicit asynchronous calls to the collision detector without storing much state information. The simple type is most useful when memory is tight, or if the phantom moves significant distances every frame (invalidating the caches).

If memory is plentiful and speed is more important then the hkpCachingShapePhantom should be used instead. For simple phantom shapes (such as spheres or capsules), where collision queries are fast, caching does not increase performance significantly and the non caching version should be preferred.

### Using Phantoms

#### Adding and Removing Phantoms from the World

Add and remove hkpPhantoms using hkpWorld's addPhantom() and removePhantom().

#### Querying a Phantom

A hkpAabbPhantom's list of overlapping entities can be queried using its getOverlappingCollidables() method. This returns an array of collidables. The hkpShapePhantom is queried using getClosestPoints() and getPenetrations().

#### Moving a Phantom

An hkpAabbPhantom can be moved after being added to the hkpWorld by calling setAabb(), passing it the new AABB's min and max hkVector4s. The following example is from the PhantomObjectDemo, in which the phantom area is moved in a circle:

```
hkReal circleParam = m_time * 0.3f;
hkReal si = hkMath::sin( circleParam ) * radius;
hkReal co = hkMath::cos( circleParam ) * radius;
hkAabb aabb;
aabb.m_min = hkVector4( -phantomSide * 0.5f + si, 0.6f, -phantomSide * 0.5f + co);
aabb.m_max = hkVector4(  phantomSide * 0.5f + si,  20.0f,  phantomSide * 0.5f + co );

// To "move" a Phantom since it is (in the current API) just an AABB, we use the
// setAabb() method to explicitly set its new location in the World. It
// automtically updates its own list of entities which it currently overlaps.
m_phantom->setAabb( aabb );
```

The moved hkpAabbPhantom automatically updates its list of overlapping entities.

hkpShapePhantom is moved by setting the transform in the phantom. Call setTransform() or just setPosition(). If the phantom will be both linear cast and moved, it is more efficient to do both operations at the same time as a linear cast alters the AABB in the broadphase regardless. Performing both operations separately will update the AABB in the broadphase twice. The setPositionAndLinearCast() function will perform both of these operations together without incurring the extra AABB update cost.

### Customizing Phantoms

Phantoms can be further customized to filter out certain objects using their addOverlappingCollidable() and removeOverlappingCollidable() methods. The code below (from the PhantomObjectDemo) filters out any fixed objects, to ensure that only movable objects are contained within the phantoms list of collidable objects:

```
void hkMovingRBCollectorPhantom::addOverlappingCollidable( hkpCollidable* c )
{
    hkpRigidBody* rb = hkpGetRigidBody(c);

    // Ignore other phantoms and fixed rigid bodies.
    if ( (rb != HK_NULL) && !rb->isFixed() )
    {
        hkpAabbPhantom::addOverlappingCollidable( c );
    }
}

void hkMovingRBCollectorPhantom::removeOverlappingCollidable( hkpCollidable* c )
{
    hkpRigidBody* rb = hkpGetRigidBody(c);

    // hkpAabbPhantom::removeOverlappingCollidable() protects its m_overlappingCollidables member,
    // regardless of whether collidables are present in it or not. However this check needs to be
    // symmetric with the one above, if we want symmetry in callbacks fired within
    // hkpAabbPhantom::removeOverlappingCollidable() and hkpAabbPhantom::addOverlappingCollidable().
    if ( (rb != HK_NULL) && !rb->isFixed() )
    {
        hkpAabbPhantom::removeOverlappingCollidable( c );
    }
}
```

hkpPhantom is the base class for phantom objects. A custom hkpPhantom class can be created by extending one of the provided phantom classes and using one of the free 10 user phantom types (HK_PHANTOM_USER0 through _USER9) and return it in the virtual getType() method.

The following methods must be implemented for a custom phantom:

- addOverlappingCollidable() and removeOverlappingCollidable(): These callbacks are called by the simulation when a new object enters or leaves the phantom's AABB.
- calcAabb(): Calculates the phantom's AABB.

### Phantoms and Determinism

A possible source of nondeterminism is the order of the items returned by hkpAabbPhantom::getOverlappingCollidables(), hkpShapePhantom::getPenetrations(), hkpSimpleShapePhantom::getCollisionDetails() and hkpCachingShapePhantom::getCollisionDetails().

This nondeterminism can arise if entities from two different simulation islands enter or leave a phantom

in the same simulation step.

To guarantee that the items returned by these methods are in a deterministic order, the calls can be proceeded by a call to `hkpPhantom::ensureDeterministicOrder()` which sorts the items, if necessary, before returning them. A single warning is issued if one of these methods is called while the items are in a potentially nondeterministic order.

#### 2.3.5.2 Ray Casting

To recap the Phantom section, there are two ways to raycast in Havok Physics 2012: either using an `hkpAabbPhantom`, or through the global `hkpWorld::castRay()` method. The `hkpAabbPhantom` restricts the test area to the shape that is defined by the phantom, and should generally be used for shorter rays. The `hkpWorld::castRay()` method is more appropriate for long rays as it will call `castRay()` on each possibly intersecting `hkpShape` in the broadphase.

Each `hkpShape`'s `castRay()` function finds the first intersection between the shape and a finite length ray defined in the shape's local space. The result (if the ray hits) is stored in an `hkpShapeRayCastOutput` structure. This includes the ray parameterization value (which specifies the relative position of the intersection point along the ray) and the surface normal at the point of intersection. If the ray has hit a subshape in a shape collection, the identifying shape keys of that shape are also provided. Certain speed ups are possible when casting multiple rays from the same point in the world; for more information please see the `OptimizedWorldRaycast` demo.

**Notes on using castRay()**

Calling `castRay()` on a shape will perform a raycast in the local space of the shape and return `true` or `false` to indicate whether or not a hit was registered. We provide both an input `hkpShapeRayCastInput` structure and a collector `hkpShapeRayCastOutput` structure; these are used to pass and return the details of the raycast (more information on input and collector structures). The output from a `castRay(...)` call for any given shape follows a strict set of rules which determine whether or not a hit is registered. These rules are outlined below:

- A start point of a ray that exactly coincides with the surface of an object and that does not intersect the object does **not** return a hit (unless the object is a `hkpTriangleShape`). This means that a path constructed as a series of rays will hit each object surface only once.
- An end point of a ray does not return a hit.
- Rays that are on a surface of the object and are parallel to it do not generate hits. The only exception to this is for an `hkpCapsuleShape`. If a ray is touching the exact geometric surface of a capsule, a call to `castRay()` will return a hit.
- If the start point of the ray is inside the object no hit will be registered.
- A hit will only be returned if the new `hkpShapeRayCastOutput::m_hitFraction` is less than the current *m_hitFraction* value. Note: The `hkpShapeRayCastOutput` constructor calls its `reset()` method which sets the value of *m_hitFraction* to 1.0f.
- If a hit is found the `hkpWorldRayCastOutput::hasHit()` method will return true and `hkpWorldRayCastOutput::m_hitfraction` will be less than 1.0f.

In addition to these rules there is one more very important rule:

- The convex radius of triangles and convex shapes (`hkpConvexShape::m_radius`) is ignored **for performance reasons** . The only shapes for which the convex radius is used are: `hkpSphereShape`, `hkpCapsuleShape`, `hkpBoxShape` and `hkpCylinderShape`. All other shapes do not use this additional element, most notably the triangle shape (and thus the `hkpMeshShape`) and `hkpConvexVerticesShape`.

Please note that although the radius is ignored for convex vertices shapes, we can still obtain approximately the same results. With convex vertices shapes, the ray is cast against planes (represented by plane equations). These plane equations can be adjusted to improve the results.

### Raycast Hit Hierarchy

Ray casts return the full path of `hkpShapeKeys` from the root shape to the leaf shape, allowing you to identify exactly which leaf shape was hit. Traversing all the shapes hit by a raycast in a hierarchy of shape is achieved using the `hkpShapeKeyPath` and its `Iterator`. First, construct the `hkpShapeKeyPath` from the output of the raycast and create an `Iterator` through the `getIterator()` function. The iterator initially points to the root shape, and each call to `next()` will go down the hierarchy to the next child shape hit by the ray.

### Shape Hierarchy Traversal

```
// This function walks from the root down a shape hierarchy.
// Use the appropriate constructor to build the hkpShapePath
hkpShapeKeyPath path ( worldRayCastCollector.getHit() ); // If using a collector.
hkpShapeKeyPath path ( worldRayCastOutput );             // If using the hkpWorldRayCastOutput.
hkpShapeKeyPath path ( rootShape, shapeRayCastOutput );  // If using rootShape->castRay(...).

for (hkpShapeKeyPath::Iterator it = path.getIterator(); it.isValid(); it.next())
{
    const hkpShape* = it.getShape();
    // Do something with the shape.
}
```

For an example on the use of shape keys with ray casting results, see the `WorldRayCastDemo` and the `ShapeRayCastDemo`.

Please note that the pointer returned by `getShape()` can be invalidated by subsequent calls to `next()` or if the iterator is destroyed. To get the shapes in a more permanent storage, you can access all the shape pointers at once by calling the `getShapes()` method from `hkpShapeKeyPath`. This method requires that you supply enough storage for all the shapes in an array of `hkpShapeBuffer`.

```
const int MAX_SHAPES = 10;
 const hkpShape* shapes[MAX_SHAPES];
 hkpShapeBuffer buffers[MAX_SHAPES];

 int numShapes = 0;
 path.getShapes( MAX_SHAPES, buffers, shapes, numShapes );
```

### Filtering Raycasts

Raycasts also make use of the Havok collision filtering system. If your world object's filter information is set up correctly, then you can use the filter attributes of `hkpWorldRayCastInput` or

hkpShapeRayCastInput to set the filtering information of the raycast. The raycast will only return objects that pass this filter.

See the demos in "Physics2012/Api/Collide/Ray Casting" for an example and more details. Please refer to the Havok collision filtering chapter for a full overview of the filtering system.

#### 2.3.5.3 Linear Casting

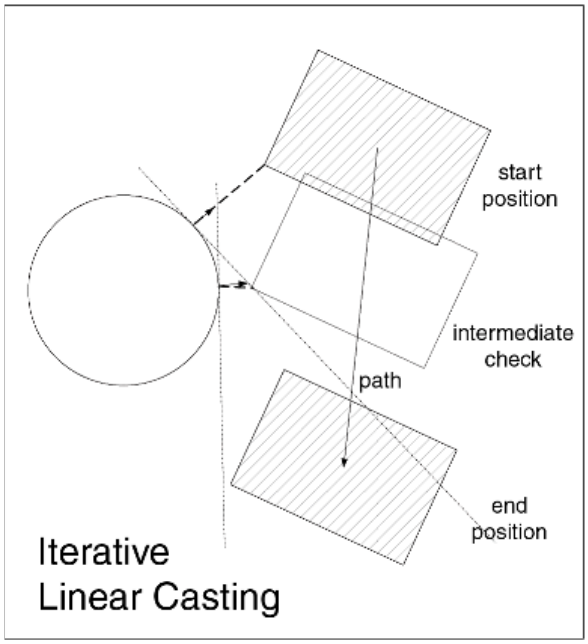
Linear casting is the ability to sweep a convex shape through the world and find the points of collision along the swept path. This is very useful for accurate proximity checks for characters and movement planning. Note that the **shape radius** is also considered so you may want to set the radius of the cast shape to zero.

```
void hkpWorld::linearCast( const hkpCollidable* collA, const hkpLinearCastInput& input,
  hkpCdPointCollector& castCollector, hkpCdPointCollector* startPoCollector = HK_NULL );
```

Please refer to the ShapeQueryDemo for further examples setting up a linear casts.

**The Iterative Linear Casting Algorithm**

Linear casting is an iterative algorithm. We describe the algorithm below to give you an intuitive understanding of the associated parameters.




The image above shows a box swept along a path. The path forces it to come in close proximity with a sphere. First the closest distance between the objects is calculated. This normal (or separating plane) between the objects is also computed (using getClosestPoints()). The initial distance and path are projected onto the normal. Havok then computes the fraction which will bring the closest feature on the

box up to the plane defined by the normal and the closest point on the sphere. We then recheck the separating plane and continue. In this example the box can complete the path before a third check is required.

The linear caster allows you to control:

- `hkpCollisionAgentConfig::m_iterativeLinearCastMaxIterations`: The maximum number of iterations per cast.
- `hkpCollisionAgentConfig::m_iterativeLinearCastEarlyOutDistance`: The minimum distance the object must travel before exiting the algorithm.
- `hkpLinearCastInput::m_maxExtraPenetration`: The maximum distance the object is allowed to increase its current penetration before a hit is reported.

The last parameter in particular is important for handling worst case behavior. This happens when an object is cast parallel to a surface it is resting on. This extra penetration prevents numerical instability in the algorithm and allows it to complete with fewer iterations. This use case happens frequently when you use the character proxy so it is important that this parameter is non-zero.

#### 2.3.5.4 Closest Points Queries

To perform a closest point query, which will return all the closest points to the shape specified:

```
void hkpWorld::getClosestPoints( const hkpCollidable* collA, hkpCollisionInput& input, hkpCdPointCollector
  & collector );
```

The closest point query can also directly be performed on two specific shapes. First, the appropriate function must be requested from the dispatcher:

```
// Get the shape type of each shape (this is used to figure out the most appropriate
// getClosestPoints(...) method to use)
hkpShapeType a = m_bodies[i]->getCollidable()->getShape()->getType();
hkpShapeType b = m_bodies[j]->getCollidable()->getShape()->getType();

// Ask the collision dispatcher to locate a suitable getClosestPoints(...) method
hkpCollisionDispatcher::GetClosestPointsFunc getClosestPoints = m_world->getCollisionDispatcher()->
  getGetClosestPointsFunc( a, b );
getClosestPoints( *m_bodies[i]->getCollidable(), *m_bodies[j]->getCollidable(), input, collector );
```

**Closest Points and Welding Information**

By default the `getClosestPoints()` function returns contact point whose normals do not take welding information into account. Welding information can be returned if desired. It allows the closest points query to better approximate the results from the collision detection. This may be desirable in certain situations.

The diagram below shows all the closest points between a sphere and a small mesh when normals are not welded.

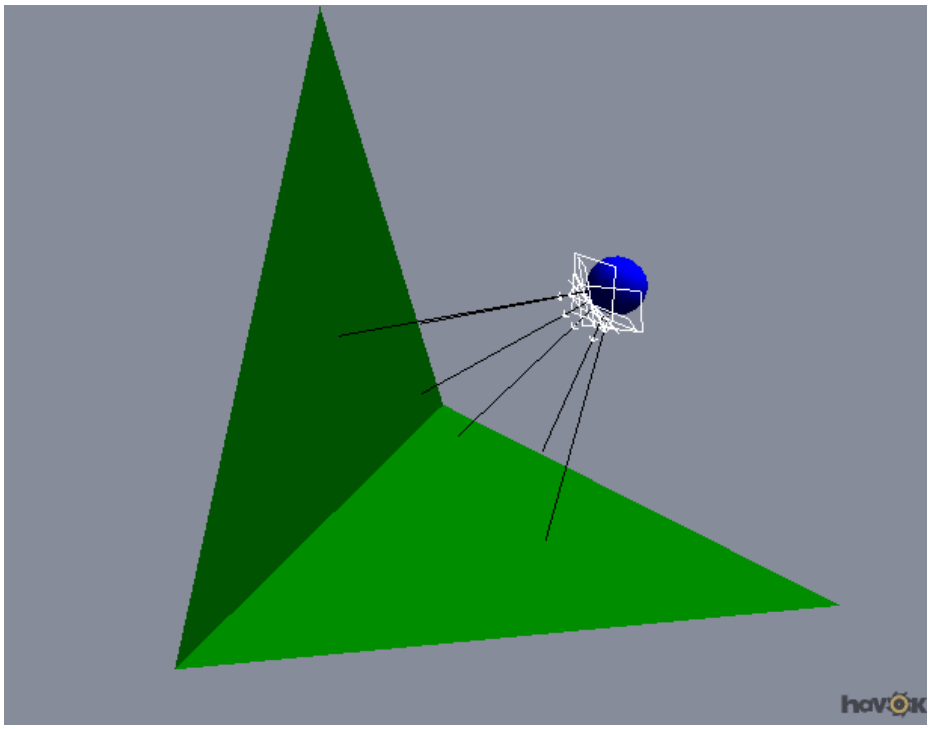


Figure 2.63: Closest points without welding information

To weld normals, the parameter `m_weldClosestPoints` must be set to true:

```
hkpCollisionInput input = *m_world->getCollisionInput();
input.m_weldClosestPoints = true;
```

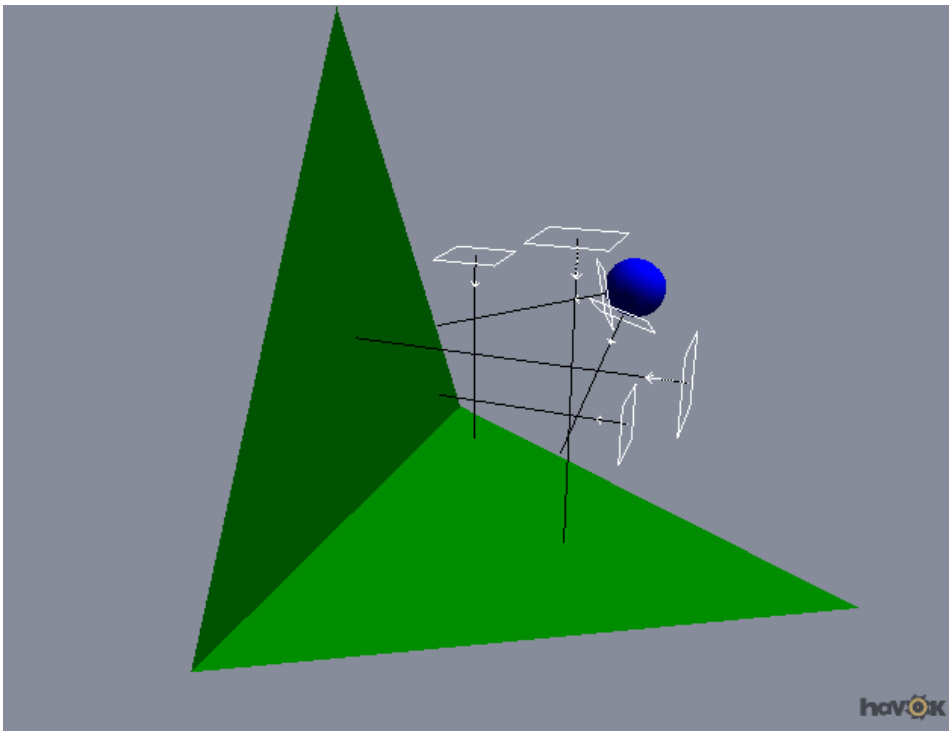


Figure 2.64: Closest points with welding information

Since welded normals no longer point from one body to the other, a consequence of welding the normal is that the corresponding closest point on one of the bodies cannot be deduced.

Please refer to the section on welding for further details on the welding process.

#### 2.3.5.5 Penetrating Shapes Queries

The penetrating shapes query simply returns the list of shapes that are currently overlapping the shape specified:

```
void hkpWorld::getPenetrations( const hkpCollidable* collA, hkpCollisionInput& input,
  hkpCdBodyPairCollector& collector );
```

**Note:**
The Penetrating Shapes Query is not supported on SPUs. We recommend the use of the Closest Points query as an alternative.

#### 2.3.5.6 Inputs and Collectors

**Collision Query Inputs**

Each of the collision queries requires a configuration input. The raycasts use a `hkpWorldRayCastInput`, the linear casts use a `hkpLinearCastInput` and the closest points and penetrating shapes queries both

use the hkpCollisionInput. For both the hkpLinearCastInput and the hkpCollisionInput, it is important to take into consideration the collision tolerance.

If these query methods are accessed directly using the world object the collision input used will be that of the world. To perform the query with a tolerance different than that of the world it must explicitly be set to the required value:

```
hkpCollisionInput input = *m_world->getCollisionInput();
input.setTolerance( 100.f );
```

The input structures contain various parameters to affect the behavior of the collision queries. For details, please refer to the reference manual entry for each. For various examples, refer to the ShapeQueryDemo.

### Collectors

The result of most user collision queries is returned within a collector. The collectors themselves define interfaces. Implementation of these interfaces can perform various processing on the returned results as virtual methods are triggered everytime a result is found. The collectors provide further control of how the query is performed. Examples of the collectors available:

| Collector base | Implementations | Meaning |
|---|---|---|
| hkpCdPointCollector | hkpAllCdPointCollector, hkpClosestCdPointCollector | Used to collect collision detection points for queries returning contact point information. |
| hkpCdBodyPairCollector | hkpAllCdBodyPairCollector, hkpFirstCdBodyPairCollector, hkpFlagCdBodyPairCollector | Used to collect information whether two objects penetrate or not. |
| hkpRayHitCollector | hkpAllRayHitCollector, hkpClosestRayHitCollector | Used to collect ray cast hit information. |

See the Reference Manual for more details on all the collectors. The source for several default collector implementations is provided in the base Havok Physics 2012 package (hkpCollide/Query/Collector).

To illustrate what can be done with a collector, consider the hkpRayHitCollector interface. When raycasting, every hit triggers a call to a virtual function in this interface. Implementations of the hkpRayHitCollector can store the hits, sort them or process them in place. The hkpRayHitCollector also stores the "early out" distance. The raycast algorithms will use this early out distance to exit early and not return ray hits which are further away.

If, in the callback the hkpRayHitCollector sets this distance to 0 (meaning a hit at the very start of the ray was found), no more ray-casts will be performed. So, by implementing a collector, you can get: all hits returned, the closest hit returned, or any hit returned (or any other similar behavior).

All the collectors return a list of either hkpCdbody or hkpCollidable. From these structures, it is always possible to retrieve the root Collidable, and from there the shape and rigid body involved in the collision. For more information, please refer to the Collidable documentation.

### Collectors and Determinism

We've already explored earlier that the order of phantoms matters for asynchronous queries. It affects how collectors return the data they collect. While the closest hit collector will return the ordered results (as long as all hit fractions are unique), other collectors such as the all point collector will hold unsorted results. Meanwhile, the fixed buffer collector may return out of order results if the result buffer is exceeded.

#### 2.3.5.7 Multithreading Queries

It is possible to process collision queries by using parallel worker threads or SPUs (on PlayStation®3). The currently supported collision queries are:

- Ray Cast query
- Linear Cast query
- Get Closest Points query
- World AABB query
- MOPP AABB query

We will take a closer look at the available types in a moment, but first some information on the basic concepts.

**Commands, Tasks and Jobs**

A command represents one specific query (e.g. one ray to be cast). A job combines one or more commands with some common data (for example the MOPP we want to perform AABB queries against). Most of the available job types can take an arbitrary number of commands. Combining several commands into a single job minimizes the overhead for each command.

Internally, a job groups its commands into independent tasks, which are then handed off to worker threads or SPUs to be processed in parallel. You are able to control how many commands will be grouped together by setting the `m_numCommandsPerTask` variable manually. For example, if you will only have 10 commands for a job it makes sense to reduce the number of commands per task to 5 or less so that at least two tasks can be processed in parallel.

**Batching Queries**

To get the most out of multithreaded collision queries you should consider batching all your queries instead of sending them off one by one. Grouping several queries into one job minimizes the performance overhead incurred by each query. Also try to put all different jobs together onto the queue before starting the multithreaded processing; this way the worker threads or SPUs can be better utilized.

**Available Collision Query Types**

Let's have a closer look at the actual query types available. These are all defined in `Source\Physics2012\Collide\Query\Multithreaded\CollisionQuery\hkpCollisionQueryJobs.h` and `Source\Physics2012\Collide\Query\Multithreaded\RayCastQuery\hkpRayCastQueryJobs.h`. Note that each job has a corresponding command. The job takes an array of these commands to be processed.

- **hkpWorldRayCastJob**

  Casts one ray into the broadphase.

- **hkpWorldLinearCastJob**

  Casts one collidable into the broadphase.

- **hkpWorldGetClosestPointsJob**

  Calculates the closest point(s) between one collidable and all the surrounding objects that are located within a certain radius.

- **hkpShapeRayCastJob**

  Casts one ray against an arbitrary number of collidables.

- **hkpPairLinearCastJob**

  Casts one collidable against another collidable.

- **hkpPairGetClosestPointsJob**

  Calculates the closest point(s) between two collidables.

- **hkpMoppAabbJob**

  Performs an AABB query against a MOPP.

As you can observe, there are 3 query types that try to emulate the `hkpWorld` queries mentioned in the previous section. These world queries will query the broadphase collision detection first to find any potentially affected objects. On PlayStation®3 these world jobs benefit from a significant performance increase as querying the broadphase is very fast on SPUs.

The 4th query type (penetration queries) is not implemented as its functionality can easily be handled by the Closest Point query.

**Note:**

All these query jobs are non-blocking and can be processed in parallel as they only require read access on the simulation world. If you will require write-access to the world (to add or remove objects for example), it is important to delay these operations to a point after the jobs have been processed in order to not stall the multithreaded work.

### Setting It Up

Simple starter examples are provided in the "`Demo\Demos\Physics2012\Api\Collide\AsynchronousQueries`" folder hierarchy. These examples all end with "`MultithreadingApiDemo`". Here's a small summary of what needs to be done to start any of the available multithreaded collision queries:

- The first step is to set up the multithreading environment. For further information on this step, refer to the chapter on Creating the Simulation.
- The next step is to create the queries themselves in the form of commands.
- Each job must be passed a job header (`hkpCollisionQueryJobHeader`).
- Now set up the jobs by passing in the requisite commands and any additional parameters to its constructor. Note that each job requires a dedicated semaphore (`hkSemaphoreBusyWait`), which is released once all the job's commands have finished.
- At this point you can specify which queue the job should be processed on by specifying the `hkJob::m_jobSpuType` variable.
- Next, put the jobs onto the job queue. The job queue will put the job in the appropriate queue as specified by `hkJob::m_jobSpuType`. You are able to specify onto which queue (PPU or SPU) ideally the job should go to.
- Finally, start the multithreaded process.

- Each job will signal its 'finished state' by releasing its dedicated semaphore. You may use the time in between for any independent computation that might need to be done.
- Once a job's semaphore has been released, you are free to process its results.

## 2.4 Optimizing a Simulation

Havok Physics 2012 is designed to perform fast, stable, memory friendly and complex simulation of a physics world under a number of varying scenarios. However, depending on the complexities of scenarios, you may experience unexpected or undesired behavior.

One of the strengths of Havok Physics 2012 is its flexibility, its ability to accomodate a wide number of scenarios. It offers a wide range of tools and tweaks to optimize both the behavior and performance of Havok Physics 2012 for your specific needs.

This chapter will explore how to profile the simulation and identify bottlenecks, how to tweak the physics configuration to obtain a desired behavior and how to optimize both the performance and memory usage of the simulation.

### 2.4.1 Profiling

Havok Physics 2012 provides a number of information viewers for the Havok Visual Debugger (VDB). Here is a short overview of the viewers provided.

**Collision Information** : `hkpBroadphaseViewer` is used to display the AABB representation of all the bodies in the simulation. `hkpMidphaseViewer` will do the same, but for the child shapes of all list shapes and other complex shape representations in the simulation world.

`hkpConvexRadiusViewer` will simply display the convex radius of every shape that has one, while `hkpShapeDisplayViewer` will simply display all the shapes in the simulation.

**Dynamics** : `hkpConstraintViewer` displays the constraint limits and frames for all constraints. It can be quite useful when debugging ragdoll setups and incorrect reference frame setups. `hkpPhantomDisplayViewer` simply shows all the phantom entities in the simulation.

The `hkpRigidBodyCentreOfMassViewer` and `hkpRigidBodyInertiaViewer` are very useful for identifying the source of odd motion behavior in bodies. More often than not, this can be tracked down to badly positioned centre of mass, or an abnormal inertia.

`hkpSimulationIslandViewer` is very useful for identifying performance problems related to the number of simulation islands. It can show islands that have been merged when they you didn't expect them to, or when too many islands are being used. It also shows which bodies are "active" (being simulated) at any time.

**Welding** : there are two viewers available to view welding information. `hkpWeldingViewer` will display welding that has been performed on any welded shape. Meanwhile, `hkpInconsistentWindingViewer` will display all inconsistent face winding information. This last viewer is very useful for identifying malformed geometry which may benefit from welding (or for fixup in the modeler).

**Contact Points Viewers** : there are three viewers available to display contact points. Active and inactive contact points can be viewed using `hkpActiveContactPointViewer` and `hkpInactiveContactPointViewer`, respectively. `hkpToiContactPointViewer` will display all TOI contacts points in the simulation, while `hkpToiCountViewer` will display a count of TOI events for all bodies involved in TOI collisions.

Full details on the Visual Debugger and the various physics viewers are available in the "Common" Havok manual.

### 2.4.2 Tweaking Physics 2012 Behavior

There are a number of tricks and tips to tweaking the behavior of the physical simulation. Most of the suggestions will affect the feel of the simulation in one way or another (bodies may interpenetrate more or less, objects will feel stiffer or looser, more or less collisions will be generated). Each of these involve a trade-off between performance, memory usage and accuracy.

We present all the different options here to allow you to choose which, if any of these tips can apply to your needs. Please note that it is rare that you will need to make use of all these different features, but being aware of what they offer in terms of advantages and drawbacks will help you make the best decisions for your scenario.

#### 2.4.2.1 Deactivation

Any rigid bodies that come to rest in the simulation are eligible for deactivation. A deactivated body isn't simulated at all, thus saving valuable CPU time. Each body has a deactivator. It tells the system when it is safe to deactivate it (based on the body's state).

The only deactivator type is `hkpRigidBodyDeactivator::DEACTIVATOR_SPATIAL` (apart from `hkpRigidBodyDeactivator::DEACTIVATOR_NEVER`). The initial deactivator settings are controlled by `hkpWorldCinfo::m_deactivationReferenceDistance`, which defines a radius of interest. An object will be deactivated if its position stays within a sphere of the aforementioned radius for about 20 frames (high frequency deactivation), or within a sphere of four times the radius for about 80 frames (low frequency deactivation). When the object exits this deactivation sphere, the center of the sphere is set to the current position of the object and the deactivation counter is reset. Similar rules apply to angular movement.

To get detailed access to the deactivation parameters, refer to `hkpSolverInfo` structure (which is accessed using `hkpWorld::getSolverInfo()`).

A more subtle deactivation can also occur when an object and its island aren't actually deactivated (so they still take simulation time) but the solver doesn't move them. This is known as "Solver Deactivation" and is especially useful when simulating stacked objects. See `hkpRigidBodyCinfo::m_solverDeactivation` for more details about Solver Deactivation.

**Deactivating Simulation Islands**

During the physics simulation separate groups of objects are partitioned into simulation islands (`hkpSimulationIsland`). Objects share an island if there are certain physical effects between them. This includes objects that are potentially colliding according to the collision detection broad phase, those that are joined together with constraints, and (in some cases) those that are operated on by the same

actions.

Simulation islands allow groups of objects to be simulated independently for performance reasons. Simulating small groups of objects separately is good for CPU and memory utilization. In addition, if all the objects in an island are eligible for deactivation (which occurs when an object has come to a definite halt), the island is automatically deactivated and doesn't need to be simulated at all.

Once a single `hkpRigidBody` is re-activated due to a collision or a call to `forceActivate()`, all rigid bodies in that island are reactivated.

Here are a couple examples to illustrate island activation and deactivation:

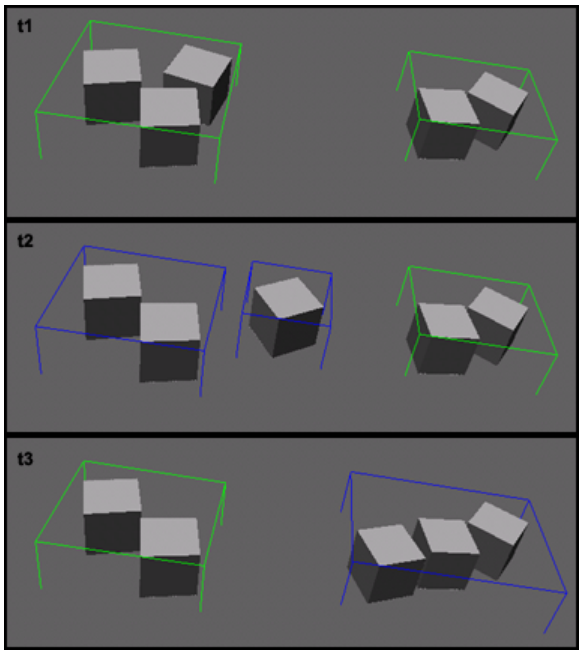



Figure 2.65: An object moving out and back into compound simulation islands

As indicated in the diagram above, there are two inactive (green) simulation islands. Between time t1 and t2, one of the physical objects is moved into the middle of the groups (which activates it, as well as the rest of its initial simulation island). At time t2, there are three simulation islands, two active and one inactive one.

Between time t2 and t3, the object is placed beside the other set of objects, and is automatically added to their simulation island, activating them as a result. The other simulation island is automatically deactivated by the system.

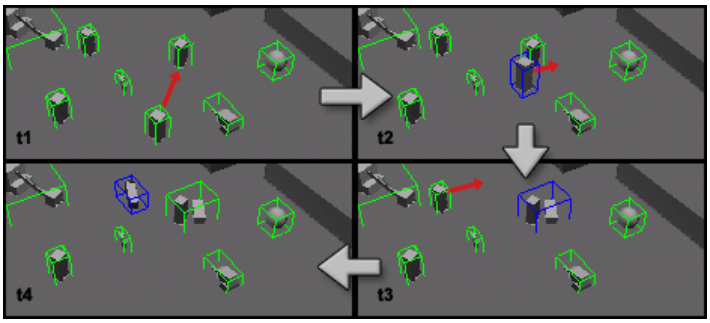



Figure 2.66: Green boxes are inactive AABBs, blue are active ( screens taken from the visual debugger )

In our second example above, a box is moved into the same simulation island as another box between time t1 and t3. At time t4 this simulation island is inactive, and another box in the area is active within its own simulation island .

### Manual Deactivation and Reactivation

Normally rigid bodies are deactivated when their entire island comes to rest (described above). You can speed up this process by marking a body as being ready for immediate deactivation using its `requestDeactivation()` method. You can also force a body to deactivate using its `deactivate()` method, but note that due to the nature of `hkpSimulationIsland`s, calling this method on an object will also deactivate all of the other objects in its `hkpSimulationIsland` (methods exist to find out what other objects are in the `hkpSimulationIsland`).

Deactivated rigid bodies are automatically re-activated when their islands are re-activated. This most often happens during the island-merging process. When an active object's AABB overlaps with the AABB of another object in the deactivated island, it causes the island to activate.

A rigid body can also be manually activated by calling its `activate()` method. This also reactivates the body's entire `hkpSimulationIsland`. Objects are automatically activated when any of the methods that change the `hkpRigidBody`'s position, rotation or velocities are called, and so it is not necessary to call `activate()` when calling one of these methods (e.g. `setLinearVelocity()`, `applyImpulse()`, `setPosition()`). To preset a `hkpRigidbody`'s position, rotation, or velocities without the implicit activation, access the body's `hkpRigidMotion` directly and use its corresponding `setLinearVelocity()`, `applyImpulse()` and other functions.

### Deactivation Issues

Note that actions are *not* applied to objects while they are deactivated. This is because the simulation island which contains the object is also deactivated, and hence will not evaluate any of its actions.

Some `hkpRigidBody` settings can give the appearance that the island deactivation was applied, such as high values for angular or linear damping. Check that these are set to reasonable values when objects deactivate too quickly.

Another issue is when an object is removed from an inactive simulation island: the island isn't automatically reactivated (see Removing a Rigid Body).

### Disabling Deactivation

Deactivation can be disabled for a single `hkpRigidBody` or for the whole simulation. For instance, you can specify that you never want a particular `hkpRigidBody` to be deactivated by setting its deactiva-

tor to DEACTIVATOR_NEVER as in the example below. You might want to do this, for instance, if the hkpRigidBody is the chassis of a Havok vehicle.

```
chassisRB->setDeactivator( hkpRigidBodyCinfo::DEACTIVATOR_NEVER );
```

However, bear in mind that if an hkpSimulationIsland contains an hkpRigidBody that is set to never deactivate, the hkpSimulationIsland will never deactivate itself, which can cause a performance hit.

If you want to ensure that all of the objects in the simulation remain active (e.g. for testing purposes), disable deactivation for the whole simulation world upon its construction:

```
worldCinfo.m_enableDeactivation = false;
```

#### 2.4.2.2 Preventing the "Bullet Through Paper" Problem

Bullet-through-paper (or tunneling) problem occurs when an object has a velocity such that it completely passes through another object within a single time step. The effect may also be witnessed if one object travels over 50% of the way through another object as the collision resolution algorithm will separate the pair by pushing the first object out the backface of the second.

The problem is avoided by using continuous collision detection (the default setting for the Havok Simulation). When an object's collidable quality is set to TOI (hkpCollidable::setCollidableType()), it will make use of continuous collision detection. Refer to the Creating the Simulation section for an overview, and the Continuous Simulation article for an in-depth discussion.

If you continue to experience objects tunneling through each other you can:

- Increase the inertia tensor of the objects involved. This will change the behavior of objects and might not be desirable, but sometimes this is possible, e.g. increasing the inertia of a rocket hardly makes a difference in a game.
- Decrease the mass ratios of the objects involved. This will also result in changed behavior.
- Decrease the allowed penetration depth. This will not change the behavior but will increase your CPU load.

The behavior of continuous simulation can also be tweaked to adjust for different scenarios, as we will see next.

**Limiting the Number of TOI Events**

The Havok continuous simulation solution handles most situations very well, in a finite number of TOI events. However, this number may occasionally become large enough to stall the simulation for too long a time. To allow real-time operation of the engine in complex cases, Havok provides a series of additional tools allowing you to trade collision quality for performance.

**Problem: Too Many TOI Events**

In certain situations, interacting objects generate too many TOI events for the simulation to handle, as the following examples illustrate.

**Example 1: Too Many TOIs Generated Iteratively**

A bouncy object is squeezed in between two heavy blocks and bounces in sequence from the top to the bottom block. As the blocks move closer to each other, the number of bounces (or TOI events to handle) in a given time period increases. This stalls the simulation if the restitution of collision between the bodies is high enough.

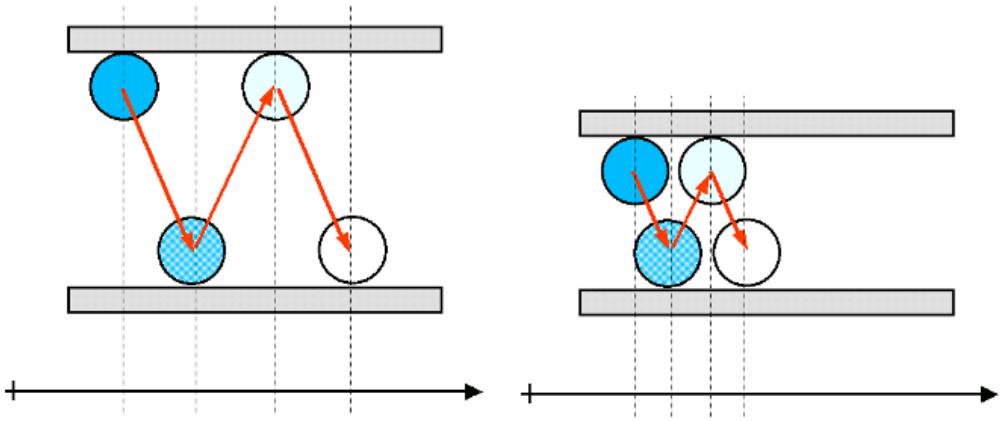


Figure 2.67: TOI events are marked on a time line. Both examples are for a sphere of the same initial velocity. No friction present; unit restitution.

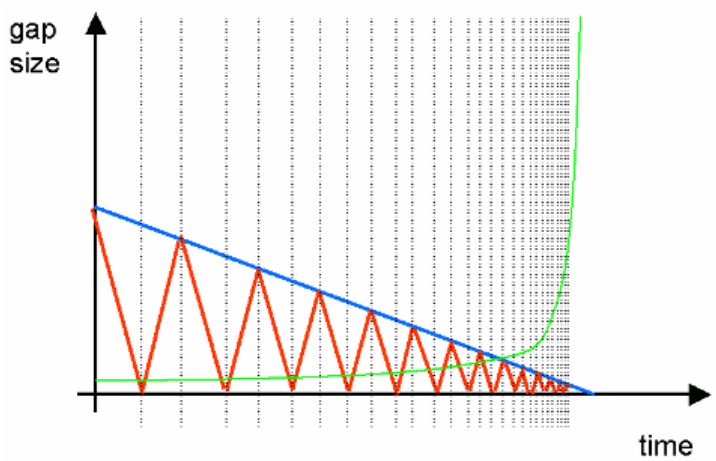
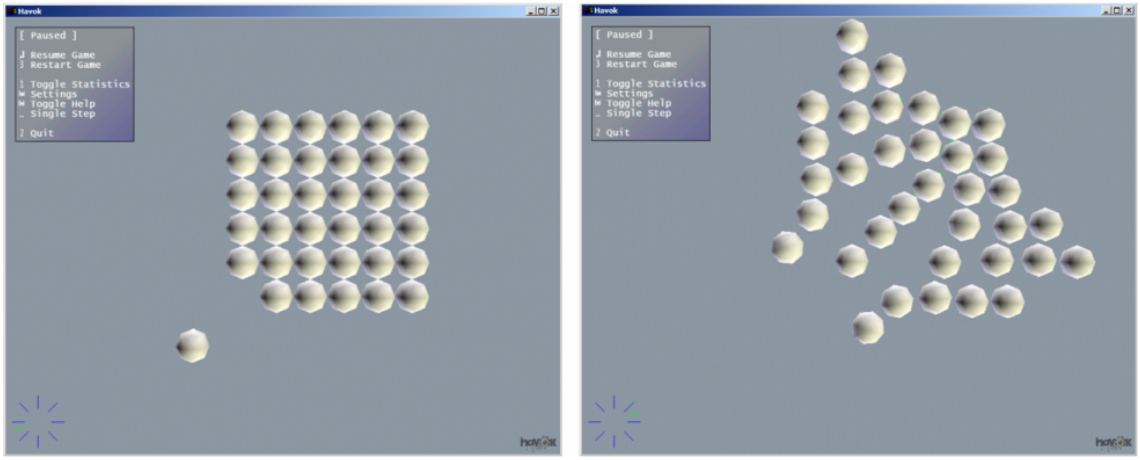



Figure 2.68: This graph shows how TOI events are generated more densely (vertical dotted lines) as the distance between two blocks decreases down to the point when the ball cannot move vertically anymore. The zigzag line shows the trajectory of a reference point on the ball. The green line shows the frequency of generated TOI-events.

**Example 2: Too Many Bodies and/or Contact Points**

A tightly packed array of objects is hit by a high-velocity particle. The fast particle transfers its momentum to a few objects it collides with. Those objects then instantly collide with further objects, which block their movement. As the result of those collisions the bounced-back objects collide again with the initially fast-moving particle. And those collisions trigger recursively resulting in possibly hundreds or

thousands of TOI events.




**Solution: Allow Penetration Depth**

To avoid TOI events accumulating into high numbers at one moment in simulation time Havok has provided the `hkpCollidable::m_allowedPenetrationDepth` parameter, which is assigned to every rigid body. This value is used to allow some penetration for one object. If we increase this parameter, Havok will reduce the number of TOI events generated by allowing bodies to penetrate each other more. This will gradually trade away the solid feeling of the system for a reduced number of TOI events, and therefore better performance.

As a result of allowing a penetration depth, a squeezed bouncing object (see the first example problem above) will gradually penetrate the solid blocks more and more.

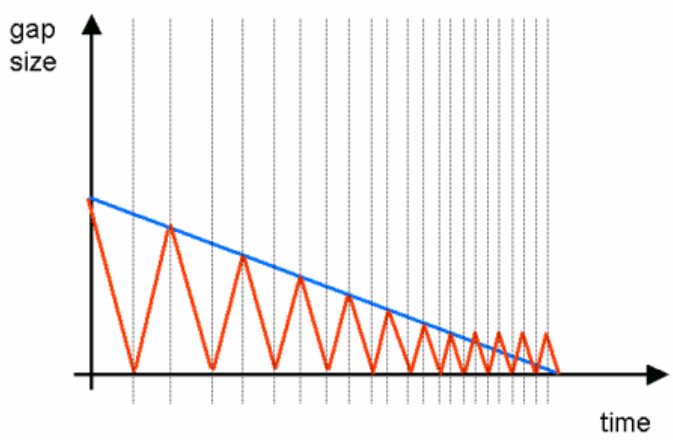
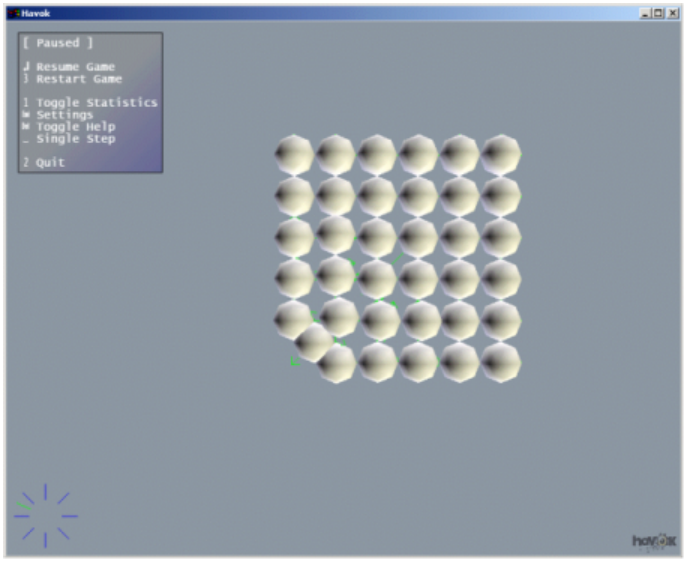



Figure 2.69: Note that the object (the red zig-zag path) penetrates the closing block (the straight blue line). As the block gets closer the system ensures minimum time distance between TOIs causing penetration.

Similarly the fast particle (second example problem above) will initially penetrate the obstacles by a greater distance.




The allowed penetration depth parameter is covered in extensive detail in the Continuous Simulation article.

**Rotate TOIs**

Tweaking the rotate TOI parameter is another way to affect the number of TOI events generated. By default, this value (`hkpWorldCinfo::m_toiCollisionResponseRotateNormal`) is set to 0.2. It defines how much we rotate the normal of a TOI contact point during a TOI event. If we set this parameter's value higher, the colliding surface will behave as if it has become rougher.

This allows sliding objects to have a larger seperating velocity and in turn lowers the number of subsequent TOI events generated. The tradeoff is that the surface will feel somewhat "bouncier". But if this behavior modification is acceptable, this parameter is a good candidate for tweaking to obtain better performance in TOI related situations.

#### 2.4.2.3 How to Represent a Large Scene

When trying to represent a large scene (that is too big to fit completely in memory), we want to split the scene into several single static mesh chunks (one per grid square). Here is a summary of how to approach this problem:

- Put the entire level into the physics system, let it calculate everything at run time and get boxes, crates etc. rolling and bouncing around.

- Decide on the collision representation for the scene: will it be a separate model or will it be the display mesh?

- Decide on the tool chain for the collision representation: most of the Havok related information can be baked in an offline preprocessing step.

- Estimate your memory budget for the physics engine and decide on an appropriate segment sizes, as a first pass dynamically create the geometry and bounding information at run time and check out the memory requirements.

- Later add in the post-processing steps that dynamically stream in the bounding and geometry info.

Let's consider the issues in some more detail. The size of these landscape meshes depend on the memory requirements. We assume that the world is already segmented for display purposes (at least into triangle lists with bounding spheres). Each segment is primarily composed of static meshes but also contains some environmental objects like trashcans, crates etc. The main reason for segmenting the world is to reduce the memory requirement for the physics and hopefully reduce the CPU requirement for collision detection (because the meshes are smaller). When segmenting the world we also want to avoid any issues with physical objects straddling segments and any extra CPU/memory overhead for dynamically instantiating or removing a segment.

The main physics memory drain for the scene will depend on how the static meshes are represented. Havok Physics 2012 will need to be able to access the triangles for the static geometry and some bounding information which speeds up collision detection with large meshes. This bounding information (in the case of MOPP) is extremely efficient and takes between 10 and 12 bytes per triangle. It can be entirely precomputed and is stored in a platform independent binary format (the `hkpMoppCode`). The memory requirement for the actual triangles depends on the internal mesh representation. If the visual and physical representations of the landscapes are the same (using the `hkpExtendedMeshShape`) then Havok will require no extra memory for accessing the triangles.

This has to make sense in the context of each geometry though, as there is no point in actually sending a 2,000 poly representation of a fire hydrant into the physics collision detection system when a cylinder will do just as well. A good rule of thumb for sharing graphical and physical meshes is that the average triangle size in the mesh should be larger than, or roughly the same size as, the dynamic objects it interacts with.

If the game design or character controller requires that a separate collision mesh is used for the scene then obviously Havok will have to be able to access those triangles at run time. Mesh bounding information is stored in memory in a tree structure. It generally has *O(log N)* performance, scaling with the number of triangles stored in it (e.g. a query on 512 triangles only takes 1/8 more CPU time than a query on 256 triangles). Because of this tradeoff it is generally better to store fewer meshes with more triangles than lots of meshes with a small number of triangles each.

With regard to simulation islands, Havok Physics 2012 automatically creates, partitions and merges islands. Dynamic objects belong to the same island if they share a constraint or an action, or are in contact. For example, if three separate piles of objects in the world are all colliding with a single static mesh, Havok will automatically separate these into three islands for simulation. By allowing Havok to handle the islands, you never have to worry about the case where an object straddles two segments, and essentially is in two different islands at the same time.

#### 2.4.2.4 Maintaining Determinism

By default, the Havok Physics 2012 simulation is deterministic (meaning that a simulation will yield exactly identical results if it's run several times from the exact same starting point on the same platorm and configuration). However, through the use of certain features, determinism can be affected. Throughout this documentation, we have highlighted whenever this situation can arise. For reference, this section groups all the possible features that can affect determinism:

- Contact point callbacks of type TYPE_MANIFOLD are not deterministic. To ensure determinism, you should either use the `endOfStepCallbackUtil`, or queue all the callbacks and sort them before processing them.
- Constraint added and removed callbacks are not deterministic.
- Phantoms overlapping collidables are not deterministic.
- Actions simulated on multiple threads can lead to non-determinism, especially if two separate actions work on the same rigid body or simulation island simultaneously. To enforce determinism, force actions to be executed on a single thread.

Havok run in a single thread may not yield the same or even similar results if you:

- Shift the system in space, or offset position of all bodies by a vector. Varying results are caused by finite numerical accuracy.
- Change the starting time of the simulation.
- Add a new body that doesn't even physically interact with the rest of the system. A new body may change the way simulation islands are organized.

Additionally, Havok run in multiple threads will not yield the same results if you:

- Modify the world in actions and most callbacks (when ordering is not enforced). Callbacks' order is nondeterministic when simulation is run concurrently on several processors.
- Change multithreading simulation settings (e.g. minimum size of simulation islands that are solved in multiple threads).

Havok provides two utilities to help check and verify for determinism. The `hkCheckDeterminismUtil` will help you check the determinism of a system. The `hkNetworkedDeterminismUtil` is used to enforce determinism in data passed over the network between a server and clients. Please review the reference manual entry for both utilities for usage details and examples.

The following must be respected in order to run Havok deterministically:

- Re-create the world from the same data.
- If custom actions are performed, these must be done in the same order.
- The internal time of the simulation must be the same.
- Grouping and ordering of islands, entities, constraints, collision entries, etc. must be the same
- Havok must be run in the same simulation mode, i.e. singlethreaded simulation yields different results than multithreaded simulation. However multithreaded simulation will be identical independently of number of threads.
- The simulation must be run on the same platform.

Simulation output is independent of how objects are laid out in memory, i.e. there's no sorting based on memory addresses. Therefore there's no need to re-initialize the `hkBaseSystem`.

#### 2.4.2.5 General Troubleshooting

Here are a few tips to help guide you when dealing with some commonly encountered problems:

| | |
|---|---|
| A brick wall simulation stalls the simulation. | • Increase the allowed penetration depth.<br>• Consider changing body types to `MOVING` or `DEBRIS`. |
| Objects penetrate walls. | • Change the object's type to `CRITICAL`.<br>• Ensure walls are of fixed type.<br>• Decrease the allowed penetration depth. |
| Objects sink through the walls and floor. | • Change the object's type to `CRITICAL`.<br>• Ensure the sinking object is not attached to any `TOI_HIGHER` priority constraints other than its contact constraints with the walls/floor.<br>• Decrease the allowed penetration depth if it has been increased. |
| Systems momentarily fall apart when colliding. | • Make sure you use `TOI` priority constraints.<br>• Sometimes you just can't help it; this feature is not bullet-proof. |
| The engine stalls. | • Check the timings file and check the number of TOI events generated in problematic frames.<br>• Check the quality types of bodies involved and determine the types of agent involved.<br>• Check the constraint priorities involved.<br>• Increase the allowed penetration depth for the bodies involved. |

### 2.4.3 Optimizing Performance

This section will help you identify and address potential bottlenecks in your physics simulation. The section is subdivided into the major areas of concern regarding performance:

- Integration and constraint solving
- Multithreaded scheduling
- Collision detection broad phase
- Collision detection narrow phase
- Simulation queries, e.g. raycasting

#### 2.4.3.1 Optimizing Integration Performance

Every active object in the simulation must be updated by the integration and constraint solving step. The number of active objects in the scene is usually one of the biggest areas that can be addressed to speed up the integration step. Before we explore that topic, we will look at how the partitioning of objects in the world can affect multithreaded performance.

#### Optimizing the Setup of Simulation Islands

The number of simulation islands can significantly affect the performance of the integration step. For a quick overview, please refer to the earlier section on simulation islands. In general, when it comes to the number of simulation islands, it is best to avoid extremes.

Use the VDB's simulation island viewer to examine the number and size of all the islands (both active and inactive) in the world at any time.

The biggest potential bottleneck during the integration step is if all the simulated objects end up in a single large simulation island. This prevents any multithreading of the integration step. At the other end of the spectrum, having too many simulation islands (in relation to the number of simulated objects) can be bad as it will add extra (and unnecessary) processing overhead. Ideally, we only want as many active simulation islands as we have available threads, and usually not many more than that. Unfortunately, tweaking the exact number of islands is not directly exposed to the end user as any overlapping pairs of bodies will automatically merge their respective islands.

We do however have some control on how separate islands are merged together. This is governed by the `hkpWorldCinfo::m_minDesiredIslandSize` parameter. When set to its default (64), it will maintain sparse islands that contain 64 or less bodies. It will only potentially split them if there are more than 64 bodies. If your simulation world contains few dynamic objects, it is a good idea to reduce the value of this parameter. This will allow the creation of more islands where possible, making the best use of multiple threads.

#### Rigid Body Deactivation

Another way to optimize integration performance is to minimize the number of objects simulated in every frame, by ensuring that objects deactivate quickly. The topic of activation and deactivation has been covered in an earlier section. In summary, inactive objects do not partake in integration/solve/collide steps, so we want to make sure that objects are not kept active unnecessarily.

Use the VDB's simulation island viewer to look for any islands that are active (colored green) but where none of its bodies appear to be moving.

Often, bodies are not deactivated because the deactivation settings are too low. Setting the deactivation distance parameter (`hkpWorldCinfo::m_deactivationReferenceDistance`) too low, or simply not enabling deactivation on the body (`hkpRigidBodyCinfo::m_solverDeactivation`) can cause the bodies to remain active and simulated even though they essentially have no effect.

The other common mishap is to call `setPosition` or `setTransform` or `applyHardKeyframe` (and so on) with the same object positions or zero forces every frame. Calling methods that affect a body's transform will always activate that body (or more precisely prevent the body from deactivating), regardless of whether the body's transform is actually changing or not. It is a good practice to wrap all calls to those functions with user versions that exit early if the call won't have any effect.

### 2.4.3.2 Optimizing Broad Phase Performance

The job of the broad phase is to quickly find pairs of AABBs that are intersecting, and thus to identify pairs of objects that require narrow phase collision detection. The `hkpBroadphase` class provides functions for adding, removing, and updating objects in the broad phase. These are used by the Havok world when you add, remove, or move entities in the simulation.

The VDB's broad phase viewer can be used to examine the broad phase AABBs of the world and all of the dynamic bodies within it.

**Configuring the Broadphase for 3-Axis Sweep and Prune Variants**

For all broad phase types except `hkpWorldCinfo::BROADPHASE_TYPE_TREE`, a badly configured collision detection broad phase can have serious performance implications. As part of setting up your Havok world, you need to specify the dimensions of the broad phase. This requirement is due to various speed and memory optimizations performed internally. You can explicitly set broad phase dimensions in `hkpWorldCinfo` or by using the `hkpWorldCinfo::setBroadPhaseWorldSize()` function (default size is 1,000 world units):

```
// Set broad phase size by setting AABB extents
hkpWorldCinfo info;
info.m_broadPhaseWorldMin.set(-50.0f, -50.0f, -50.0f);
info.m_broadPhaseWorldMax.set(50.0f, 50.0f, 50.0f);

// Functionally equivalent:
info.setBroadPhaseWorldSize(50.0f);
```

There are three important points to consider when setting the size of the broad phase:

- Make the broad phase large enough to encompass the whole scene.
- Do not make it excessively larger than it needs to be.
- Use callbacks to be notified when objects leave the scene.

Because the broad phase world has a definite size, it is possible to position objects so that they are *outside* the broad phase. *This must be avoided.* During broad phase evaluation, objects outside the broad phase are projected onto the broad phase AABB (it clips the object AABB to the broad phase AABB, in effect reducing the 3D overlap test to a 2D one, producing spurious overlaps). This incurs a significant performance cost as the relevant pairs are all passed to the narrow phase, greatly increasing the amount of work the narrow phase has to do. It also produces incorrect callbacks from other broad phase listeners, and from broad phase raycasts.

For example, if 100 objects are placed along the X axis *all outside the broad phase, in a line but separated* then all their AABBs will be clipped to have the same min and max X extents. That is, their AABBs each collapse to a rectangle, so the broad phase considers them to be overlapping each other on the X axis. If the objects' AABBs actually overlap on the other two axes then all 100 objects will be considered to overlap with each other. This will cause (100*99)/2 = 4,950 agents to be created, even though most of these will return no collisions when the narrow phase is processed! Collision detection will still work accurately, but it will be *massively inefficient* both in speed and memory consumption.

To avoid this, ensure that your broad phase world size is appropriate for your scene, and remember the broad phase limits when moving your objects. It is also important that you *do not set the broad phase size too large*, as internally the broad phase has to quantize the whole range. Too large a broad phase size leads to reduced accuracy in the overlap tests resulting in more narrow phase tests than needed.

To monitor objects leaving the broad phase during runtime, you can use the `hkpBroadPhaseBorder` class in the `hkpDynamics`. Its `maxPositionExceededCallback()` method is called whenever an object leaves

the broad phase AABB. By default, it removes the entity from the world, but you can implement your own custom handling solution.

This callback is triggered from within the multithreaded sections of the engine. The order of callbacks is by default nondeterministic. However, they can be postponed until the end of the simulation step. In that case they will be sorted and triggered in deterministic order. To use this option enable the `hkpWorldCinfo::mtPostponeAndSortBroadPhaseBorderCallbacks` flag.

### Configuring the Tree/Hybrid Broadphase

The tree broad phase automatically performs incremental optimization on each update. To start with an optimal structure you should call `hkpTreeBroadPhase::fullOptimize()` before the first frame, and after most entities have been registered.

The following functions can be used to further fine tune performance:

- `compact()` will rearrange the memory layout of the tree broad phase to improve cache efficiency. The impact on performance is especially important for PlayStation®3 SPU queries.
- `setJitterMargin(hkReal margin)` sets the width of the margin added around entities when registered in the broad phase. This margin allow for an entity to move slightly (jitter) without triggering an update of the underlying acceleration structure. The default value is 0.01 m (1 centimeter), higher values will improve the broad phase update performance but decrease query speed.
- `setMotionPredictionFactor(hkReal factor)` sets the factor used for motion prediction. Motion prediction will attempt to predict a moving entities position (up to `factor * physicsDeltaTime`). If the entity position is correctly predicted, it will not trigger the update of the underlying acceleration structure. The default value is 2.0, that is, attempt to predict two time steps into the future. Higher values will improve the broad phase update performance but decrease query speed.
- `setUpdateQuality(hkReal factor)` sets the number of entities (as a factor of the total number of entities registered in the broad phase) to update each frame. The default value is 1/60, that is, 100% of the entities will have been optimized after 60 frames. Higher values will decrease the broad phase update performance but increase the query speed. Note that this function can be called with a different value each frame so the quality could, for instance, be made proportional to the current frame rate.
- `setCacheQuality(int passes)` sets the number of cache optimization passes executed at each update, which behaves like an incremental version of the `compact()` method.

### Limiting the Number of Objects

The cost of simulating the broad phase increases with the number of objects in the scene. Specifically, as the number of objects in the broad phase grows, the cost of inserting objects, removing objects, and querying the broad phase (for raycasts etc.) increases **linearly** . However, the cost of colliding objects increases at a slower rate–the simulation is optimized to simulate collision detection as fast as possible.

It is recommended to keep the number of objects in the broad phase as low as possible. Due to the fact that inserting and removing objects increases in costs with the number of objects, maintaining a stable number of objects is also encouraged. In a scene with a large number of objects with few ray casts and little to no object insert/removal, the collision detection will remain very optimal.

A simple way of reducing the number of bodies in the broad phase is by combining multiple objects into one. In particular, the BV Compressed Mesh Shape and the Extended Mesh Shape provide the ability to insert fixed world objects into the landscape shape.

If the simulated scene is large in relation to the area of current player interest, it can be a good strategy to remove objects from the simulation if they are outside of the area of interest.

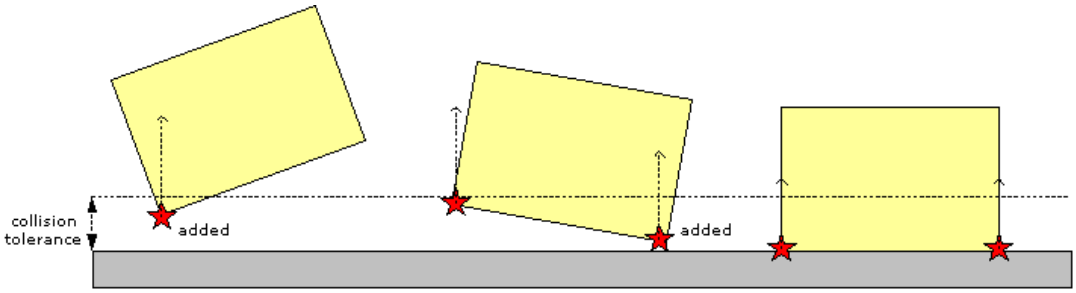
**Collision Tolerance**

The collision tolerance defines when contact points are created for two objects in the broad phase. Contact points are created for two overlapping objects in the broad phase when the shortest distance between the pair becomes less than the defined collision tolerance.

The global collision tolerance for the entire Havok world simulation is defined in `hkpWorldCinfo::m_collisionTolerance` (with a default value of 0.1). This value is specified in world units.

When the two overlapping objects are closer than this distance, one or more contact points are created (depending on the collision agent). These contacts will be processed by the narrow phase solver in each simulation step until the distance between these objects again exceeds the collision tolerance. At this point the contact points are also removed from the simulation.

Consider the following illustration showing a box falling onto a table. The first contact point is added as a corner of the box falls within the collision tolerance. As the box falls further, the first contact point is maintained and a new contact point is added. Finally, the box actually comes to rest on the table, with zero distance between the two objects. All created contact points are maintained by the agent until the box moves outside the collision tolerance of the table again. Alternatively, should the AABBs of the box and table become disjoint (non-overlapping), then the collision agent itself will be deleted, automatically deleting of all associated contact points.




A non-zero collision tolerance greatly helps with two performance-related issues. First, it is useful for the system to pick up potential collisions just before they actually happen. This allows the collision solver to prevent interpenetration as early as possible, even for fast-moving objects.

It is also useful for the system to maintain collision information even when the objects are slightly separated. When an object is sliding across or settling on another object, this allows the system to maintain the same manifold rather than having to create new contact points in each simulation step.

#### 2.4.3.3 Optimizing Narrow Phase Performance

In this section we discuss tips that can improve the performance the simulation when dealing with individual shape collision detection and resolution. These tips focus primarily on how to tweak the properties of shapes and bodies to improve performance.

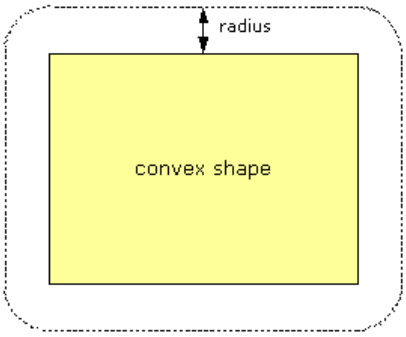
Use the VDB timers (and "stat graph overlay") to examine the time spent in narrow phase collision.

### Convex Shape Radius

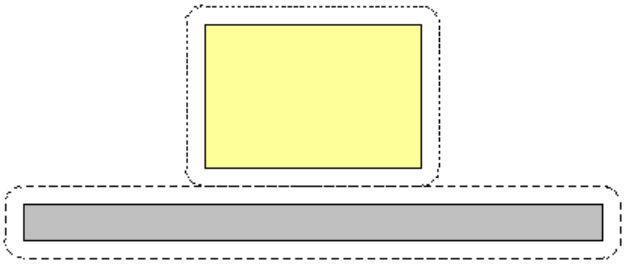
As mentioned in the shapes section, the `hkpConvexShape` class has a radius member that adds an extra "shell" to any convex shape except for the sphere and the capsule. For a sphere and a capsule, the convex shape radius actually defines the radius of the sphere and the capsule, respectively, although it is treated by the collision detector in exactly the same way. The sphere is just a "point", and the extra "shell" is the radius of the sphere.

The convex radius of objects can be viewed at any time using the `hkpConvexRadiusViewer` in the VDB and/or the demo framework.

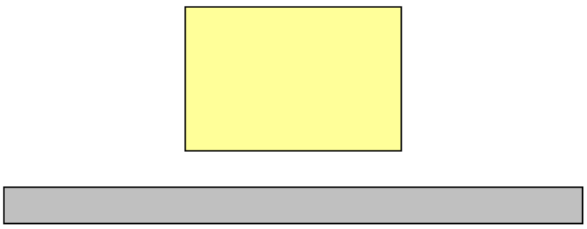
By default, an `hkpConvexShape`'s radius value is 0.05.



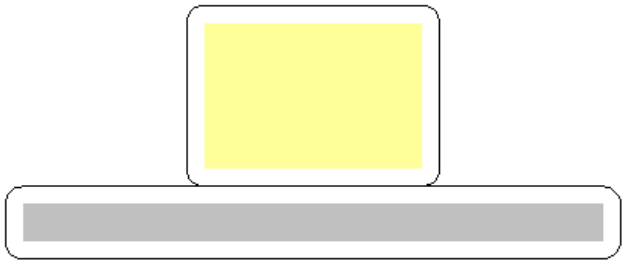

This shell is used as the shape's surface for collision detection. It increases the collision volume of the shape by the specified radius. The collision simulation will treat all objects with this radius as having an extra thickness.



Adding a shell affects where the surface of the object is in the simulation, and where the object will come to rest. For instance, if a box with a shell comes to rest on a table, there will appear to be a small visible gap between the box and the table, as in the next illustration.



However, as far as the collision solver is concerned, the two objects are in direct contact, as shown below.



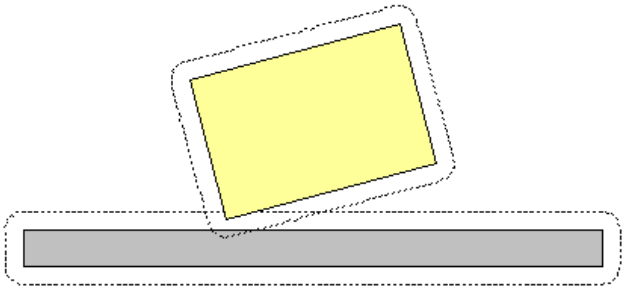
**Why use a radius?**

Adding a radius to a shape can improve performance. The core convex-convex collision detection algorithm is fast when shapes are not interpenetrating, and slower when they are. Adding a radius simply makes it less likely that the shapes themselves will interpenetrate, reducing the likelihood of the slower algorithm being used.

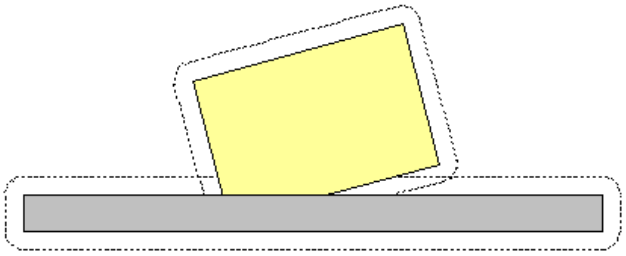
The shell is thus faster in situations where there is a risk of shapes interpenetrating - for instance, when an object is settling or sliding on a surface, or when there is a stack of objects, or when many objects are jostling together.

Raycasting against objects with the radius will also cast against that expanded surface. However, raycasting against triangles does not raycast against the expanded form as it is computationally prohibitive, but as triangle based objects are normally landscapes that usually have zero radius, it is not normally an issue. It is ok for landscapes to have zero radius, because we only need to ensure that any two interacting bodies have some extra radius between them. As long as all dynamic bodies have an extra radius, the extra radius on a stationary landscape is not necessary.

If a small shell interpenetration occurs, as shown in the next diagram, the system will work to restore the gap, the fast algorithm will still be used, and, as shown, the box will still not have appeared to penetrate the table.



Only if the "unshelled" shapes actually interpenetrate is the slower algorithm used, as shown in the following diagram.



**Note:**
If two convex shapes with zero radius collide, it can also lead to numerical instability. The surfaces used for collision detection will be exactly touching when the objects come to rest, making the calculation of the collision normal unstable. This is another reason why all convex objects should have a positive radius value.

### Convex Vertices Shapes and the Convex Shape Radius

When raycasting against a `hkpConvexVerticesShape`, it is assumed that the plane equations given to the shape are already expanded by the radius. Therefore, when creating a `hkpConvexVerticesShape` in code, we must make sure to either expand the plane equations by the convex radius, or use a very small convex radius or use the `hkpShapeShrinker` to shrink the vertices.

When a `hkpConvexVerticesShape`s is shrunk, the vertices are recalculated by using the plane equations. The plane's distances are reduced and the planes' points of intersection are recalculated. These are then used as the new vertices. The new vertices and the original planes are used to create a new shape.

**Note:**
Unpredictable behavior can sometimes result from using `hkpConvexVerticesShape`s with a very small radius value. If this happens, try increasing the radius.

### Visual Implications of Convex Radius

The visual implications of using a convex radius is that the graphical representations of objects will hover above surfaces and bullets will appear to hit invisible barriers. This visual disparity is further emphasized by shadow and lighting solutions. Because setting the object radius to zero is not recommended (due to extra performance costs incurred), an alternative solution must be used.

Objects should be created such that the surface of the (higher resolution) display representation is the same as the physics object when including the radius. This can be easily accomplished in the tool chain by creating the shape in the same dimensions as the display version and then using the automatic shrinking functionality offered in the Create Rigid Bodies filter.

This object processing is not meant to be done at runtime, but in the tool chain and preprocess stages.

### Setting the Convex Radius in the Modeler

When building a collision shape in the modeler the convex radius is automatically computed. The value that can be manually set is actually a relative shrinking value used to shrink the vertices and an absolute maximum displacement. Refer to the modeler section for details.

### Implications of Using Transform Shapes

The transform shape (`hkpTransformShape`) is primarily used in conjunction with list shapes

(hkpListShape) to create compound shapes, allowing you to accurately position the child shapes relative to one another. Transform shapes naturally incur an extra overhead. How significant is this overhead and what parts of the engine are affected by it?

Using transform shapes will incur a minor performance hit in three areas:

- During AABB updates in the broad phase for the body.
- During narrow phase collision detection as an additional collision agent is needed. This extra collision agent will disable some of the memory and performance optimizations.
- There will also be a further overhead due to transformation concatenations (one time cost per frame per object).

However, using hkpConvexTransformShape or hkpConvexTranslateShape can minimize some of these overheads:

- During AABB updates in the broad phase for the body, which is particularly cheap for simple translations
- No extra agent is needed, just the call to getSupportingVertex is modified.

In order to quantify the potential performance loss due to these extra computations we have internally tested sample scenes consisting of many objects using transform shapes. The results described below are for a test performed with approximately 350–400 rigid body interactions. The sample scene consisted of 120 boxes arranged in three stacks of 40 boxes each, which were then allowed to collapse onto one another to create a large pile.

Upon measuring the timings for both the broad phase and narrow phase collision detection we found that the broad phase was 17% slower when using hkpTransformShapes and the narrow phase 23% slower. The overhead for actually creating and using the additional collision agent was only ∼16% of the total overhead incurred. The rest of the time was spent concatenating the transformations. Although this may seem to be quite an increase in simulation time, when all other computations performed by Havok are taken into account (such as integration, contact solving etc.) the total increase in CPU time is only ∼5%. Although this is an extreme case and one unlikely to exist in your game it is worth noting that there is the potential for a noticeable performance hit.

If possible, and performance is critical, this overhead should be removed. As highlighted in the above statistics, almost all of the extra computations are due to concatenating the transformations. If these transforms can be "baked in" (and thus removing the need for the transform) this should be done. If you are using implicit primitives (like a hkpBoxShape) to synchronise the display representations, it is likely that the concatenation will have to be performed at some point in your code. However, if you handle the transformations yourself you may see a performance improvement as you will only have to do so *once* and not *twice* as is the case for Havok (once in the broad phase and again during narrow phase). In general, if you use the hkpTransformShape sparingly throughout your scenes, performance should not suffer too much. To aid with performance tuning Havok will issue the following warning whenever hkpTransformShapes are encountered:

```
Warning : 'Collidable at address 0x01234567 has a
        transform shape as the root shape. This can cause a significant
        performance loss. To avoid getting this message compose the transform
        into the collidable and remove the transform shape.'
```

If you feel that these objects are degrading performance you can quickly track down any occurrences within your code.

Havok also provides utilities to detect and remove unnecessary transform shapes. Check the Reducing the Number of Shapes section for details.

**MOPP Fit Tolerance**

When using a MOPP tree, you can customize the tolerance (called the fit tolerance) that describes how closely the bounding volumes at the leaf nodes (and the internal nodes) approximate the ideal bounding volume. This is done using the `hkpMoppCompilerInput` structure, which you pass to the `hkpMoppUtility::buildCode()` function when building the MOPP.

The `hkpMoppCompilerInput` structure allows you to choose what is most highly optimised: the size of the tree, or how closely the leaf nodes bound the triangles. The effects of changing the fit tolerance involves a tradeoff between performance and memory:

- The smaller the fit tolerance, the less triangles are passed to the narrow phase collision detector and the less processing is needed.
- However, the smaller the fit tolerance of the bounding volumes, the more memory is needed to store the bounding volume information.

For a detailed description of the parameters of the MOPP fit tolerance and background information on the MOPP please consult the `hkpMoppCompilerInput` section in the Reference Manual.

**Contact Point Generation**

During simulation, the collision detector will be called to generate contact points between rigid bodies. Several contact points need to be generated between two rigid bodies in order to resolve the collision correctly.

Consider a box sitting on a flat surface. In this case we generate 4 contact points, one for each corner of the box that is touching the surface. This is called a "contact manifold". In general we do not create more than 4 points between two convex rigid bodies. In certain conditions however (such as when a rigid body is colliding with a highly tessellated triangle mesh) many more contact points can be generated.

This situation is undesirable as a large number of contact points adversely affects performance. Havok has a limit of 250 contact points which may be created between two rigid bodies. If this limit is reached a warning is generated, and any more generated contact points will be automatically rejected. This warning usually indicates something is wrong - your mesh is too tessellated for example. However, if this cannot be remedied, the system will still work properly, as the contact points that are generated will compensate for the ones that are rejected.

In general, **minimize the tesselation level of meshes** to avoid hitting the cap on contact points.

#### 2.4.3.4 Optimizing Multithreaded Performance

There are several issues that can affect multithreaded performance, but they broadly fall into two separate categories: job distribution and concurrent access to data.

### Optimizing Job Distribution

#### Performance problem: threads spend a lot of time idling at the end of a timestep.

After the main part of the simulation step, Havok processes all TOI events. TOI events are generally processed in a singlethread, with the exception of the collision detection which can be processed multithreaded.

The granularity of multithreaded jobs should match the weight of collisions to process. `hkpWorldCinfo::m_maxEntriesPerToiCollideTask` specifies the maximum number of agent entries that are processed by a single task. Tweaking this parameter will help achieve optimal performance for a specific simulation scenario.

- If `hkpWorldCinfo::m_maxEntriesPerToiCollideTask` is low and agent entries only contain simple agents (for example a scene of many small dynamic objects using the sphere-sphere agent), then the engine will spend much more time synchronizing and managing collision jobs on the queue then it will spend doing actual collision detection work. Adjust this value upwards when dealing with many simple collisions.
- If `hkpWorldCinfo::m_maxEntriesPerToiCollideTask` is too high, and the collisions tend to involve complex cases (such as list shapes against complex landscape), then a single thread may be stuck with processing all the collision queries. Adjust this value downwards when dealing mainly with complex collisions.
- If the problem involves tiny collision jobs being multithreaded, you can turn TOI multithreading off entirely by setting `hkpWorldCinfo::m_processToisMultithreaded` to false.

#### Performance problem: Too many small simulation islands.

In Havok, sparse islands will usually get split into multiple smaller islands. In a multithreaded environment, having too many small simulation islands to process can have a significant impact on performance. Increasing the value of the `hkpWorldCinfo::m_minDesiredIslandSize` variable will help limit the number of small islands created. In singlethreaded mode, this parameter is simply ignored.

This value controls the minimum number of rigid bodies and constraints in a simulation island. It has no effect on the actual *spatial* size of the island. It should be noted that in Havok, the spatial size of simulation islands has no impact on the performance cost of processing them. So while the appearance of (visually) large islands in the Visual Debugger may be disconcerting, it can safely be ignored.

### Minimizing Concurrent Access of Data

There are a number of small tips to remember when minimizing the number of concurrent accesses of data:

- In general, batching collision information queries (such as linear casts) will improve concurrency issues by pooling common data together. Batching is also a generally good practice to maximize the performance of any multithreaded platform.
- You should not need access to write-locks when using Havok Physics 2012, but is a generally a good idea to use the `hkpWorld::markForRead()` and `hkpWorld::markForWrite()` methods to verify that you are not accesssing data at the wrong times.
- Minimizing the use of contact callbacks is also strongly advised as they are a great source of concurrency issues.

- The earliest you can discard a potential collision between two objects you should do so. As much as possible, we should try to discard collisions in using the collision filtering system. Unwanted collisions should not be filtered out in contact point callbacks or in collectors (for asynchronous queries), unless you are truly unable to decide whether to discard the collision before it happens.

#### 2.4.3.5 Optimizing Query Performance

There are several factors that can affect the performance of queries, including those mentioned in the previous optimization sections. In general, we recommend the following tips to address any query performance issues:

- **Use a less populated broad phase** : World queries will perform faster when there are fewer objects to test in the broad phase, especially when using the sweep-and-prune broad phase alone.
- **Use phantoms** : Make use of phantoms to limit the breadth of the queries being performed. The smaller the area that has to be tested the faster the broad phase part of the query will perform. It is much cheaper to limit the test of a raycast to a single room in a building then to test the entire broad phase.
- **Use multithreading** : Leverage the multithreading utilities provided by Havok whenever possible, and especially when performing a significant number of queries. Ideally, design your logic such that you can dispatch jobs to other threads at some point in the frame and collect the results of those jobs later when you need them.
- **Batch queries** : Again, if performing a large number of queries, try to batch the queries together to leverage the spatial caching abilities of the Havok broad phase, and reduce the job management overhead if using multithreading.

Also refer to the `OptimizeWorldRaycastDemo`, which illustrates a number of these techniques.

### 2.4.4 Optimizing Memory

Before reviewing the Havok Physics 2012-specific advice on optimizing memory usage, it is **highly recommended** that you first familiarize yourself with the Havok memory management system, which is discussed extensively in the Havok Common book. It provides a number of facilities to track and profile memory allocations and fluctuations over the course of a simulation.

These tools can help you identify bottlenecks in levels and situations that cause the memory to spike (if they exist).

With this knowledge in hand, the following section will address ways the Havok Physics 2012 simulation can be optimized in terms of memory usage.

#### 2.4.4.1 Optimizing Shapes

Shapes are the fundamental building blocks of the physics simulation. They are used everywhere. Their memory impact can be significantly reduced by taking heed of the advice offered in this section.

This section could also belong in the Optimizing Performance section as the following tips will gain significant amount of performance. A large portion of simulation time is dedicated to generating and testing pairs of potentially colliding shapes.

**Reducing the Number of Shapes**

Decreasing the number of shapes used to represent objects will improve the collision detection process, by reducing memory usage (which may lead to performance improvements) as well as reducing the time required to traverse shape hierarchies.

Two utilities are provided in the hkpUtilities library that can be used to reduce the number of shapes and the shape hierarchy depth for one or multiple rigid bodies. As these are CPU intensive utilities, they are recommended for offline asset processing (the *Optimize Shape Hierarchies* filter in the Havok Content Tools uses these two utilities).

- `hkpTransformCollapseUtil` will attempt to remove unnecessary transform shapes by collapsing transformations into the child shape (modifying the child shape's vertices for example). It will also replace `hkpTransformShape` objects with either `hkpConvexTransformShape` or `hkpConvexTranslateShape` objects (which are faster) if the child shape is convex.
- The `hkpShapeSharingUtil` utility class will attempt to detect duplicated shapes in one or multiple rigid bodies and replace them by references to the same shape.

**Shapes and Sharing**

Reduce the runtime memory footprint required to run a simulation by sharing shapes whenever possible. Each shape and all associated information is stored in the shape's local space. This means you can have a single instance of a shape and share it among several rigid bodies or even the same shape hierarchies.

**Case 1: 500 crates but only 1 shape.** The simplest example of this is when we have a game object that appears many times in a level e.g. an ammo clip. We create one shape to represent the surface of the ammo clip and then reuse this shape every time it appears in the level by creating an `hkpRigidBody` for each instance of the clip and setting the shape to our single surface representation.

**Case 2: Instanced fixed geometry.** As with the example above we can create a single instance of a piece of fixed geometry, e.g. a telegraph pole, and reuse this in several places in our level. These objects can be integrated with the landscape using either the BV Compressed Mesh Shape or the Extended Mesh Shape. The bounding volume collision detection information need only be created and stored once, and the instance of the shape can be inserted multiple times into the mesh (each instance wrapped in a transform shape so it can correctly be positioned). If we didn't reuse the single instance, a lot of memory would be wasted storing the same geometry multiple times in the Mesh.

**Case 3: Compound shapes.** When we have shapes that represent basic building blocks or components we will often want to put these together to build a final, more complex shape e.g. consider building a house with shapes for a door, a window, a roof and a wall. We can put this together using a list shape and use transform shapes to offset multiple instances of the same component shape in the final construction. The final building with 2 doors, 4 walls, a roof and 10 windows might contain 17 transform shapes. Each of these transform shapes would refer to one of the original 4 components (door, roof, window, wall) together with a transform to position and orient the component. It is worth noting that the `hkpTransformShape`s do have a performance cost associated with them due to the extra transformation incurred, so where possible bake the transform into the shape data.

Havok provides utilities to detect and share duplicated shapes. Check the Reducing the Number of

Shapes section above for details.

#### 2.4.4.2 Memory Used in Havok Physics 2012

**Heap-Memory**

Havok allocates and frees memory (for objects, caches and internal messages) constantly. Changing Havok to handle allocation requests returning `HK_NULL` would increase the complexity of the physics engine considerably. So we face the following challenges:

- No allocation in Havok should ever fail.
- The amount of memory allocated within each physics step is not predictable.
- Fragmentation in the memory allocator could cause memory allocations to fail, even if there is memory available.

The goal is to keep the memory consumption well within the total amount of memory available. The user can normally do this simply by removing rigid bodies from the Havok simulation and deleting them.

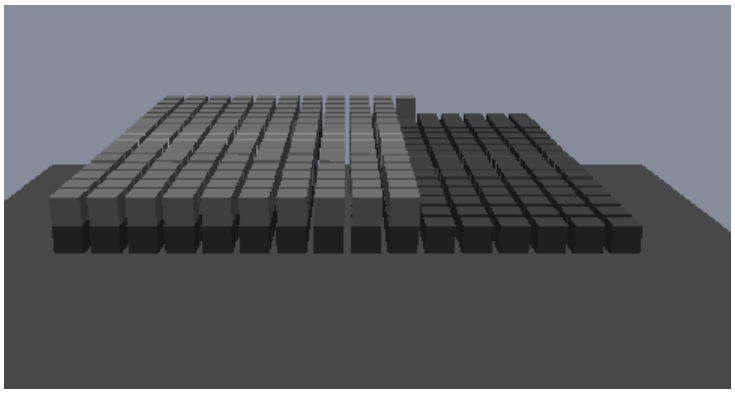


Figure 2.70: A screenshot from the LimitingMemory2 demo. Blocks are removed from the world to recover memory when memory usage gets too high.

As it can be very difficult to predict how much memory Havok will allocate within the `hkpWorld::step()` functions, you can:

- Set `hkFreeListMemory::m_softLimit`.
- If `hkFreeListAllocator` detects that the current memory consumption exceeds `hkFreeListMemory::m_softLimit`, it will call `hkSetOutOfMemoryState(MEMORY_STATE_OUT_OF_MEMORY)`, which is a global flag to signal the physics that we are really low on memory.
- If Havok physics detects `hkGetOutOfMemoryState() == MEMORY_STATE_OUT_OF_MEMORY`, it will immediately abort all physics operations, leaving the physics world in an inconsistent state. This is done to try to avoid further allocations. If this happens, the following operations are performed:
  - If set, the `hkpWorld::m_memoryWatchDog` will be called and is expected to free memory. It will also re-run the physics step to bring it into a consistent state.

- If the watch dog is not set the simulation will return with a failure case and you will have to manually free memory and call `hkSetOutOfMemoryState(MEMORY_STATE_OK)`. The next time you call `hkpWorld::stepDeltaTime()`, the physics will re-run the previous frame to return to a consistent state. See the `hkpWorldMemoryUtil` as a guideline for your own implementation.

You can always check the current memory consumption by calling:

```
hkMemoryRouter::getInstance().heap().memUsage( memUsageOut );
```

**Solver Memory**

The constraint solver uses a temporary buffer to store all solver matrices. This buffer is created using the `hkMemoryRouter:getInstance().solver()` allocator. You should use the `hkSolverAllocator` for this type of memory. The `hkSolverAllocator` manages a single input block of memory, and (as required by the Havok simulation) breaks this block into smaller blocks whenever necessary.

We must ensure that this block of memory is always big enough (for example, a complex scene using 4,000 active rigid bodies can require a solver buffer size of up to 4 MB).

Because Havok's simulation loop looks like this:

- Solve using the contact points from the last frame.
- Collide.
- Solve TOIs.

the solver buffer size can easily be calculated before calling `hkpWorld::stepXXX()`:

```
hkpWorld::MemUsageInfo memInfo;
m_world->getMemUsageForIntegration(memInfo);
```

So when running out of solver space you can:

- Increase the solver buffer temporarily.
- Remove rigid bodies from the biggest island (recommended).
- Decrease the number of contact points by filtering out collisions (not recommended, because it is slow and not reliable).

There are 2 ways to implement the above functionality:

- Before calling `hkpWorld::stepST()` or `hkpWorld::stepDeltaTime()`, call `hkpWorld::calcBufferSize()` to see if your solver buffer requirements are over budget. If they are, take appropriate action. Refer to `hkpWorldMemoryUtil` as an example implementation which deals with this situation. Note that this function calls

hkpWorldOperationUtil::cleanupDirtyIslands() at the beginning for accurate results and also takes into account the solver memory needed by hkpToiResourceMgr (128 KB in the default case). When implementing your own custom TOI resource manager make sure to return in hkpToiResourceMgr::getScratchpadCapacity() the maximum amount of solver memory needed.

- Implement a memory watchdog (recommended), which gets called from within hkpWorld::stepXXX() if the solver buffer is too small. You can even use the default implementation of hkpDefaultWorldMemoryWatchDog, which simply removes entities from the largest island.

There are actually several different types of out of memory conditions:

- The largest island needs a solver buffer which is bigger than the supplied one. In this case either the solver buffer has to be increased or rigid bodies have to be removed from the simulation island.
- The largest island requires a buffer bigger than the supplied buffer, but another simulation island has already allocated a big portion of the full buffer. The physics scheduler will reschedule this big island to the end of the simulation step when all the other islands have freed their solver buffer. In this case, you will experience a small extra CPU hit.
- In the rare situation where you are stepping two Havok simulation worlds simultaneously, one world may allocate a buffer so big that the other world cannot continue simulating. In this case the stalled world will spin on the job queue until the big buffer becomes available. In this case, you could experience a fairly significant CPU spike.

### Memory Watchdogs

An implementation of hkWorldMemoryAvailableWatchDog should ensure that the amount of allocated memory stays within a certain boundary. The interface looks like:

```
class hkWorldMemoryAvailableWatchDog
{
      /// Free memory until: allocatedMemory < (hkpFreeListMemory::m_softLimit - m_freeHeapMemoryRequested
        )
      /// This function is called from many places in the engine to check the memory consumption.
      /// It is also called when the memory limit is exceeded. A typical implementation should remove
      /// unimportant rigid bodies from the world or apply more restrictive collision filtering
      /// in this function to reduce the memory consumption of the engine.
  virtual void freeHeapMemoryTillRequestedAmountIsAvailable( hkpWorld* world );

      /// This is called from checkMemoryForIntegration. If there's not enough memory to perform solving
        and integration
      /// for a given island, the user is asked to lower their number e.g. by removing constraints or
        bodies.
  virtual void reduceConstraintsInIsland(hkpSimulationIsland* island, int solverBufferSize);
};
```

Once you have created an instance of this class, you can activate it by calling hkpWorld::setMemoryWatchDog().

An example implementation is provided in the LimitingMemoryDemo.

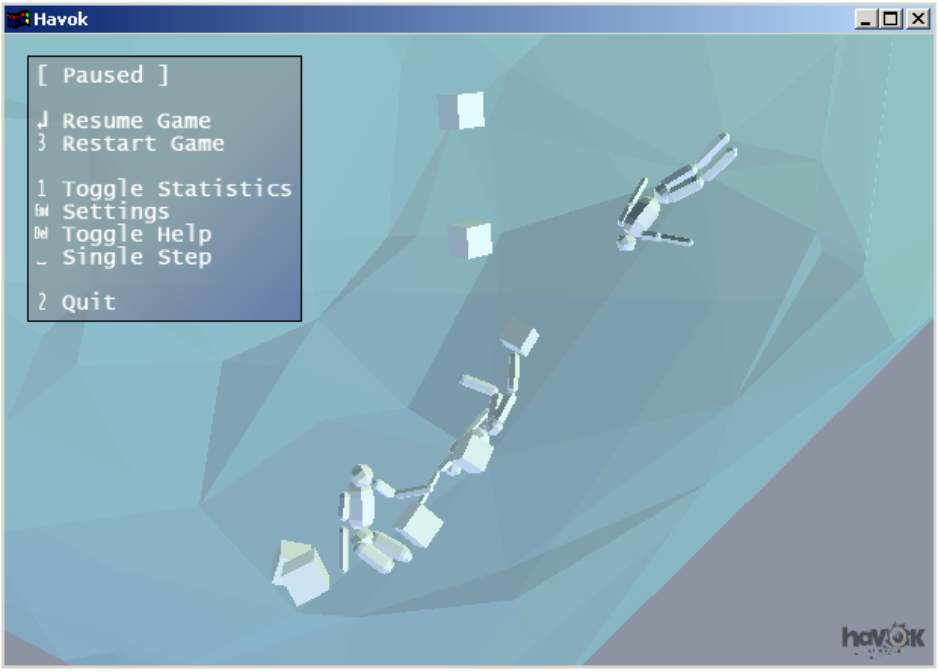



Figure 2.71: The LimitingMemoryDemo, in which debris gets removed before ragdoll components.

#### 2.4.4.3 Reducing Code Size

Havok Physics 2012 contains a number of optional features that can be stripped out if code size is an issue.

- Check your use of collision agents and shapes. You can save considerable code if you remove references to unused collision agents and shapes.
- Do not link with the visual debugger in your final build. To make sure of this, you could do the following steps:
  - Include the hkpUtilities project file into your project directly, and link with it instead of the prebuilt library.
  - Remove the "visualdebugger" folder and all its sub-folders.
  - You could also remove any other code from this project file you do not expect to be linking with, to make sure it does not get included.
- Do not link with unnecessary libraries, such as the vehicle library if you don't use the Havok Vehicle utility.
- If you are using serialization versioning patches, consider updating all your assets to the latest Havok version to avoid having to include extra versioning code.

**Reducing the Use of Agents and Shapes**

By default, Havok registers all the available shape types and collision agents (algorithms that handle collision detection and resolution on a per-collision-pair basis). This is done for simplicity, but if you wish to reduce code size, you can manually register only the shapes and agents you use.

In this situation you can avoid registering less commonly used agents (such as does involving heightfields, or special types of meshes). Please refer to the source code of `hkpAgentRegisterUtil::registerAllAgents()` to create your custom registration code. Do note that the order of registration is important and should be preserved as it in `hkpAgentRegisterUtil::registerAllAgents()`.

#### 2.4.4.4 Localized Solving and the hkpToiResourceMgr

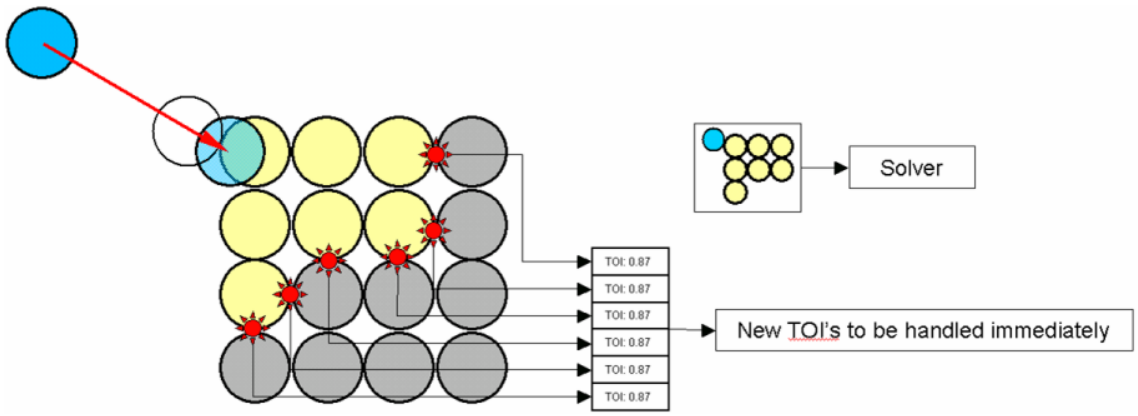
The `hkpToiResourceMgr` class is an interface that grants fine grained control of the `hkpContinuousSimulation` and how it performs localized solving. It allows the user to:

- Filter TOI events.
- Set parameters and assign memory resources for each TOI event being handled.

Before handling a TOI event, the engine calls `hkpToiResourceMgr::beginToiAndSetupResources(..)`. This callback is responsible for allocating memory to be used by the solver and for setting hard limits on the maximum number of bodies and constraints to be passed into the solver. The callback may also return `HK_FAILURE` to make the simulation skip this event. If the event is not skipped, the allocated memory should be freed in the `hkpToiResourceMgr::endToiAndFreeResources()` callback fired at the end of TOI-handling function. `hkpToiResourceMgr::getScratchpadCapacity()`returns the amount of solver memory needed by the TOI resource manager (for the default implementation this is 128 KB).

When the group of activated bodies exceeds the limits set or when the system runs out of memory for the solver, a `hkpToiResourceMgr::resourcesDepleted()` callback is called to let you decide how to handle this case.

The TOI handling routine also assures that `TOI_HIGHER`-priority constraints are not violated. If it fails the user receives a `hkpToiResourceMgr::cannotSolve()` callback.




The example above illustrates a fast particle hitting an array of bodies. In order to calculate a perfect non-penetrating solution all of the bodies have to be passed into the solver. Assuming we have limited memory only a part of the array is activated, while the gray objects will be ignored. After handling this TOI event and performing collision detection new TOI events may be created (marked on the picture with the red spots). Possibly though the particle's momentum is redistributed well enough and the interpenetrations

(at the red spots) are small. In that case pre-setting `hkpRigidBodyCinfo::m_allowedPenetrationDepth` to a large-enough value would suppress new TOI events being generated.

To recap, the `hkpContinuousSimulation` queries the following `hkpToiResourceMgr` methods:

- `beginToiAndSetupResources(hkToiSolveInfo, hkpToiResources)`: to accept or reject a TOI event, and to set parameters and resources for the event.
- `endToiAndFreeResources(hkToiSolveInfo, hkpToiResources)`: called at the end of the TOI event to free up resources.
- `cannotSolve()`: to inform you that some of the `TOI_HIGHER` priority constraints have been violated.
- `resourcesDepleted()`: to inform you that a TOI needs more resources in order to fully expand its local group of bodies being collided and reintegrated.

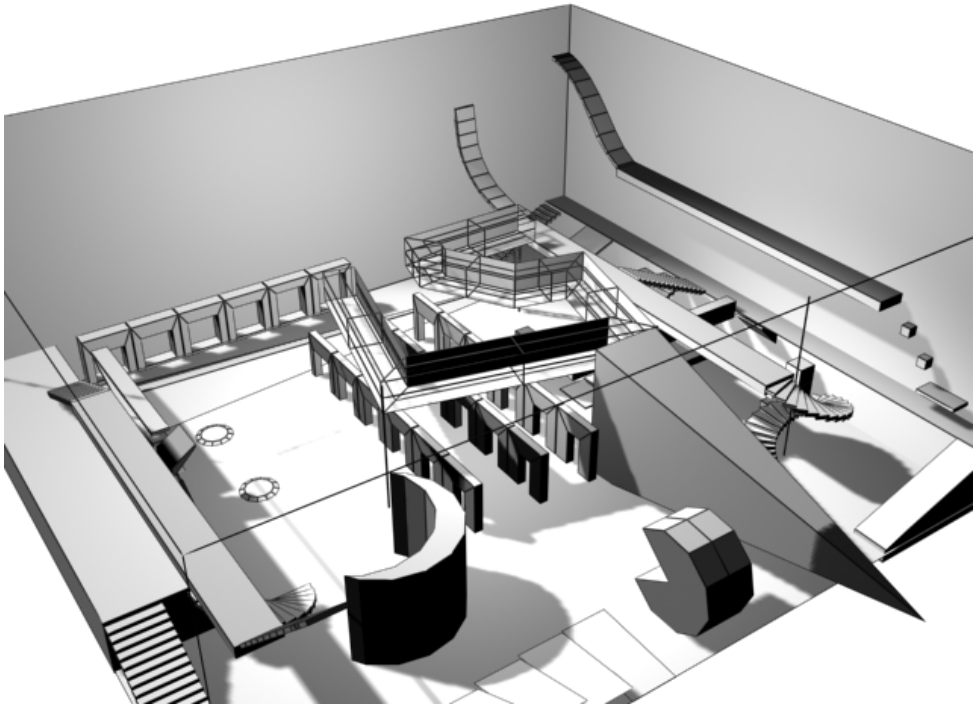
The default implementation used by Havok is `hkpDefaultToiResourceMgr`. To override this implementation, create an instance of your custom manager in the constructor of `hkpContinuousCollision`.

## 2.5 Extensions

This section explores the use of several useful Havok utilities and extensions. These include:

- The Character Controllers.
- The Vehicle Physics SDK.
- The Contact Points Saving Utility.

## 2.5.1 Character Control



This chapter provides an overview of character control strategies in Havok Physics 2012. The chapter is designed as an overview of the core classes and methods that facilitate integrating our character controllers, or your own custom solution.

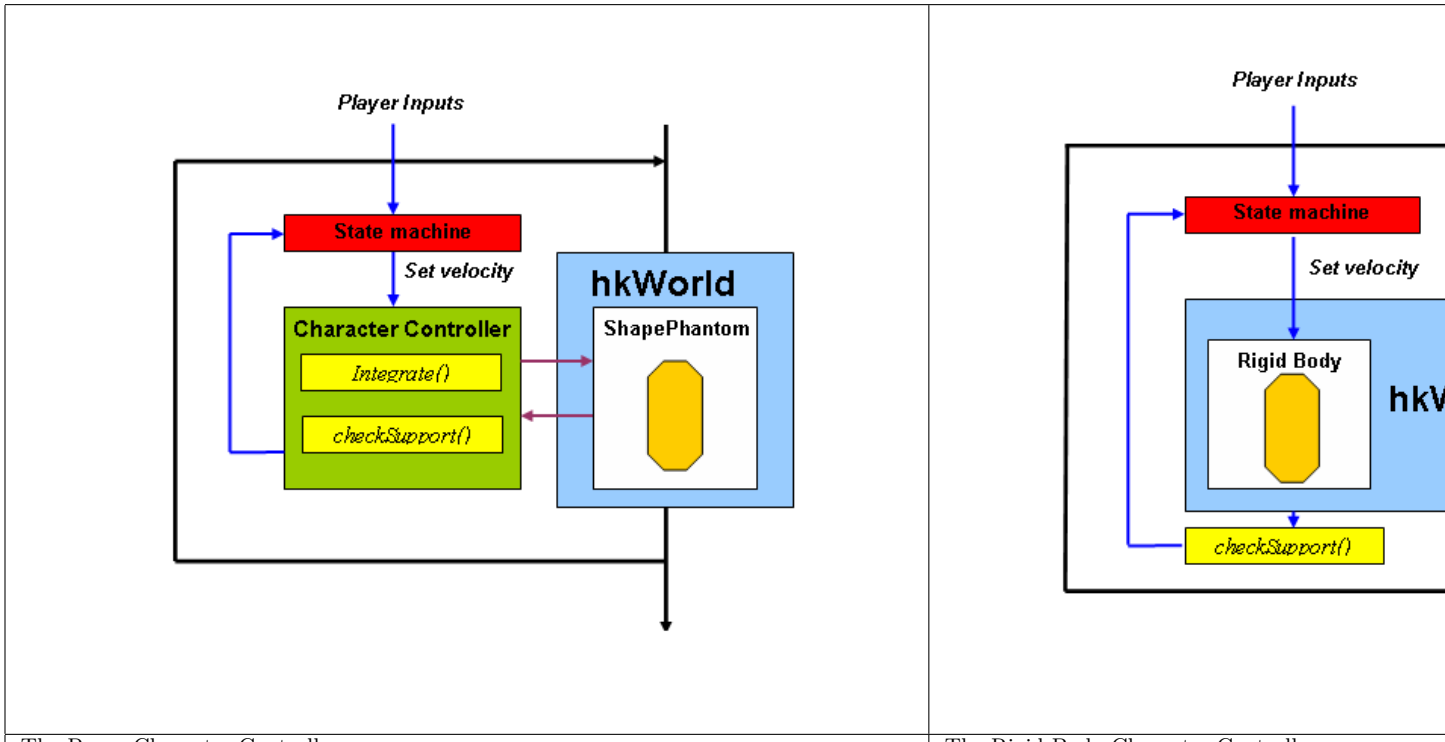
### 2.5.1.1 Overview

Often each character controller will have a very distinctive feel to entire gaming. Havok provides two different versions of character controllers:

- The Rigid Body Character Controller
- The Proxy Character Controller

The main differences in the integration philosophies between the rigid body and proxy character controller are shown in the figure below. The rigid body character sits inside the normal simulation loop and it is an integral part of the dynamic world (`hkpWorld`). The rigid body is controlled by directly setting its velocities (linear, and in specific cases also angular). The proxy character controller sits outside the normal simulation loop and it is represented only with the shape phantom (`hkpShapePhantom`) in the world. It is linked with the physical simulation in two ways. The first way takes the collision information from the simulation via shape phantom and the second way puts information into the simulation in the form of impulses applied to rigid bodies. Both controllers use the same state machine (`hkpCharacterContext`) to control the character state transition and calculate the required control velocity according to player inputs. Each controller has its own implementations of the `checkSupport()` method which provides informations about the surface under the character. More details about controllers can be found below in separated sections.

In the comparison section below, we list the advantages and disadvantages of each character controller. To summarise, we recommend using the rigid body character controller over using the proxy controller. However there are a number of caveats to be aware of in this choice. The behavior of the rigid body character controller may not be as smooth as the proxy controller, particularly for stair climbing. Also the character controller currently has less in game testing than the proxy controller.




| The Proxy Character Controller | The Rigid Body Character Controller |
|---|---|

Table 2.8: Different integration philosophies for controllers

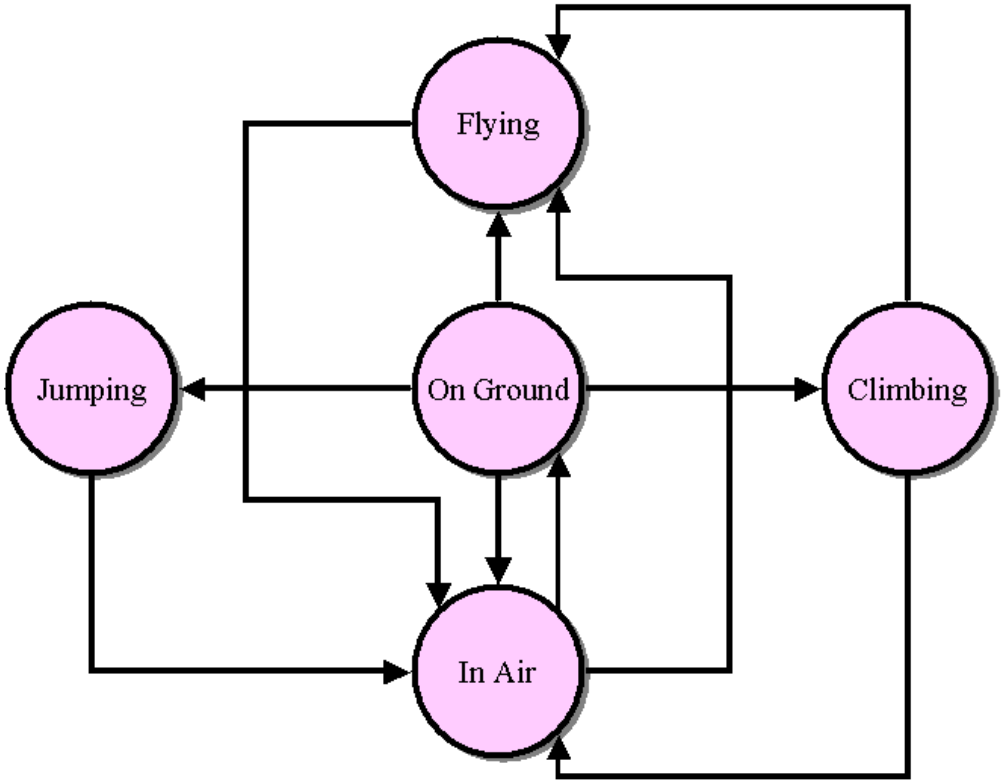
The structure we provide is designed to be extended and replaced as you see fit. The controllers consist of the following major distinct components

- The State Machine
- The Character Rigid Body or The Character Proxy
- The checkSupport Method
- The Simplex Solver
- The Interaction Listener

These components can be coupled together or used in isolation to produce the required behavior. The state machine and the simplex solver are the same for both controllers. The simplex solver is a major part of the character proxy movement calculation (`hkpCharacterProxy::integrate()`) and it is also used in both implementations of the `checkSupport()` method. The character rigid body and the character proxy are completely different as described above.

#### 2.5.1.2 The State Machine

A reusable state machine controls the transition between discrete character states. The currently implemented state and their transitions are shown below. Full source code for the state machine can be found in `Physics2012\Utilities\CharacterControl\StateMachine`.




The state machine is constructed through a state manager. The state manager allows the user to construct and register their own specific states at run time. Each state machine is designed to be shared. Usually one instance will exist for each type of character in your game e.g. one for Humans and another for Orcs. To construct a state machine you must first register each of your desired states with the state manager, for example

```
hkpCharacterState* state;
hkpCharacterStateManager* manager = new hkpCharacterStateManager();

state = new hkpCharacterStateOnGround();
manager->registerState( state, HK_CHARACTER_ON_GROUND);
state->removeReference();

state = new hkpCharacterStateInAir();
manager->registerState( state, HK_CHARACTER_IN_AIR);
state->removeReference();

state = new hkpCharacterStateJumping();
manager->registerState( state, HK_CHARACTER_JUMPING);
state->removeReference();

state = new hkpCharacterStateClimbing();
manager->registerState( state, HK_CHARACTER_CLIMBING);
state->removeReference();
```

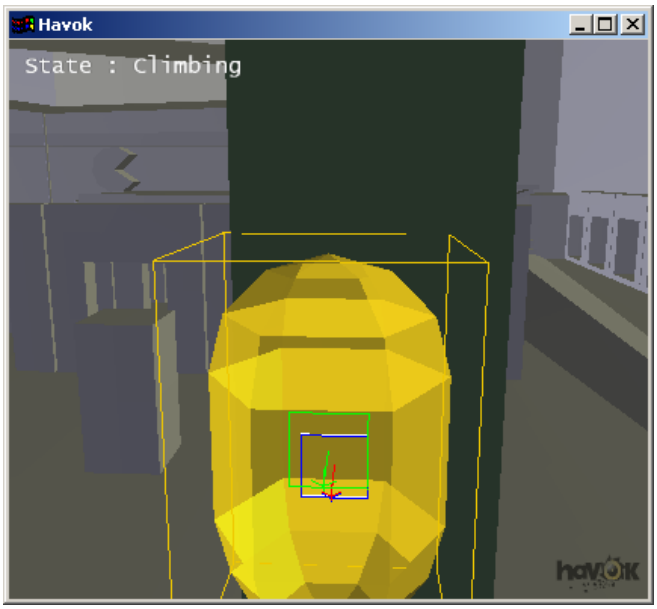
As you can see references to new states are managed appropriately by the state manager.

Once you have registered your state you create a context for each character you wish to control. All state transitions and updates are done through the character context. It contains the current state for the character. To create a context you pass in a manager and an initial state. The context will then use that manager to transition between states (when a transition is requested)

```
m_characterContext = new hkpCharacterContext(manager, HK_CHARACTER_ON_GROUND);
manager->removeReference();
```

Once you have successfully created a character context you can fill out an input structure and pass it to the context's update method. This will cause the current state to update, potentially causing transitions in the state machine. The output from the update is a new desired output velocity for the character. The user chooses exactly how each of the fields in the input structure should be filled out. This decoupling allows the state machine to be run in isolation. Its output can be used with either the character proxy we describe below or with a normal rigid body.

If you run the character controller demo, the current state appears in the left side of the screen.




The same state machine is used for both types of character controllers. Most of the code is identical, but for example the state logic transition or the output velocity filtration can differ for the particular controller. To set the concrete character type use `hkpCharacterContex::setCharacterType()` helper method. The default is the character proxy (`hkpCharacterContext::HK_CHARACTER_PROXY`).

```
m_characterContext->setCharacterType(hkpCharacterContext::HK_CHARACTER_RIGIDBODY);
```

#### 2.5.1.3 The Character Controller Behavior

The character in the demos is set up to behave like a typical first person shooter character controller. It is worth taking a moment to understand what affects this behavior, as it will help you if you wish to customize or modify the behavior for your own character. The best demos to try are the CharacterControllerDemos, which shows the character controllers in a static landscape which has been specially set up to create situations that are typically difficult for character controllers to deal with: slopes of varying angles, gaps which the character can just fit through / just not fit through, deep valleys, undulating terrain, etc.

**Default Dynamic Behaviour**

We define the speeds at which the character moves in the default state machine. The default walk speed is 10 metres per second. This is about 20 mph, which is 5 times faster than a human really walks, however it is a standard FPS walk speed. This speed is used in the "on ground", "in air" and "climbing" states. The jump height for the character by default is 1.5 meters. Gravity is passed into the character state machine, and in the demo it is set to -16 m/s/s. This is because a real world gravity of -9.8 m/s/s actually looks quite low, given the character's faster than life walk speed, however this higher value does mean that the character will fall faster than physical objects.

Because the state machine output velocity is used to control objects, which are part of dynamic system, it is a good practice to filter this velocity to ensure smooth and stable control signal. The filtration is

done in two levels. The first level is performed inside the state machine in particular states, namely the "on ground" and "in air" states. The maximal acceleration is limited to 200 m/s/s for on ground state and to 50 m/s/s for in air state. Smoothness and delay of the signal is controlled by pre-multiply gains (0.95 for on ground state and 0.05 for in air state). The second level of filtration is performed on output velocity at the end of the update method. The limits are 625 m/s/s for maximal acceleration and 20 m/s for maximal linear velocity. Limitation of maximal velocity is applied first. We recommend that you decrease these limits with hkpCharacterContext::setFilterParameters() for highly populated scenes with a lot of interactions with abnormal objects (light bodies, hight inertia, etc.) or for complicated asymmetric shapes. If you prefer to have a 'stronger' or 'more dynamic' character, don't forget to increase maximal acceleration limits. If you want to model a character sitting on fast motion platform, don't forget to change maximal velocity limits.

When the character is running up a slope, (in the on ground state) we try to maintain its forward velocity as if it were on level ground, i.e. it still runs at 10 m/s up the slope. We also wish to make sure that the character runs in the direction it is pointing; so it does not get affected by the slope. This actually means we compute a velocity in the frame of the slope, and then re-project that velocity to the horizontal plane, which creates a new desired velocity that points slightly into the slope, and is slightly larger than 10 m/s. Intuitively this means that to maintain a desired velocity running up a slope at a glancing angle, we actually try to run "into" the slope at a faster speed. The code to do this can be seen in the "on ground" state machine. Note that the on ground state takes the surface velocity into account; it tries to maintain 10 m/s relative to the surface on which the character is standing.

There is a further inconsistency between the default on ground state and the default jump state. When the character runs up a slope of 60 degrees at 10 m/s, it has an upwards velocity of 8.66 m/s. With a gravity of -16 m/s/s, this is higher than the upwards velocity needed to jump to a height of 1.5 meters (6.29 m/s). The default jump state takes the current upwards velocity into account to give a consistent jumping behavior. This means that when the character runs up a steep enough slope, jumping does nothing (as the character has a velocity of greater than 6.29 m/s upwards, relative to the ground velocity). This code can be seen in the hkpCharacterStateJumping::update implementation. Because the character can run so fast, it can launch itself quite high into the air (in our default case, higher than it can jump) if it runs off the top of a steep ramp. To prevent this from happening, the on ground state has a flag m_killVelocityOnLaunch, which is set to true by default. When set, the character sets the upwards velocity to zero when the it leaves the ground (unless it is jumping). This also has the effect of keeping the character closer to the ground when it is moving on uneven terrain and frequently going briefly to the in air state. Note that this flag only works correctly when the character is on a static landscape.

When the character is in air, by default it behaves as though it is on the ground, in that the character can still move at 10 m/s in any direction specified. This is also standard FPS behavior. There is no sliding state in the state machine currently. This could be implemented to give better behavior on slopes that are too steep for the character to climb. Currently the character stays in the in air state on these slopes, and so the player can try to move the character in the direction of the slope, and so slow down or stop the character falling down the slope. To implement custom sliding behavior a sliding state could be added.

**The Check Support Method**

Both characters have a function called `checkSupport()`. This function is used in all the character demos, as an input to the state machine, to determine whether the character is standing on the ground, and if so, what the resultant surface normal and surface velocity is. The implementations of these methods is very similar for both controllers, i.e. both methods use the simplex solver. Creating the input to the simplex solver is done in different ways depending on the character representation, otherwise the code is identical.

The check support function returns one of three states: SUPPORTED, UNSUPPORTED, or SLIDING.

The SLIDING state is reported if the character is standing on a surface which has a slope greater than max slope. The SUPPORTED state is reported if the character is standing on a surface which has a slope less than the max slope. The check support function will return SUPPORTED even if the surface is moving away from the character. It is effectively a static query of the geometry around the character in a given direction. For example when the character jumps, at the moment of jump the character will have a velocity moving away from the surface. However as it is still touching the surface, check support will return SUPPORTED.

Sometimes it is necessary to register collisions with trigger objects in a scene, but not have those collisions affect their character controller. In addition to the normal checkSupport( ) method, `hkpCharacterProxy` provides a checkSupportWithCollector( ) method. This method takes the same arguments as checkSupport( ), along with a third argument for a point collector. Point collectors are essential for rejecting unwanted collisions and modifying character states. The following is an example of a custom point collector that ignores collisions with "jump boxes" so characters can pass through them while jumping:

```
class MyJumpCollector: public hkAllCdPointCollector
{
public:
    HK_DECLARE_NONVIRTUAL_CLASS_ALLOCATOR( HK_MEMORY_CLASS_DEMO, MyJumpCollector );

    // Constructor
    MyJumpCollector()
        : hkAllCdPointCollector()
    {
        m_jump = false;
    }

    // Accessor Method
    hkBool shouldJump() { return m_jump; }

    // Mutator Method
    void setShouldJump( hkBool jump ) { m_jump = jump; }

    virtual void addCdPoint( const hkCdPoint& event )
    {
        const hkpCollidable* rootCollidable = event.m_cdBodyB.getRootCollidable();
        hkpWorldObject* wo = hkpGetWorldObject( rootCollidable );

        if (wo->hasProperty(HK_OBJECT_IS_JUMP_TRIGGER))
            m_jump = true;
        else
            hkAllCdPointCollector::addCdPoint( event );
    }

protected:
    hkBool m_jump;
};
```

### The Controller Shape and Stair Climbing

The character shape is usually a capsule shape (`hkpCapsuleShape`) which is 2 metres high. It has a radius of 60 cm. This radius actually affects two important things: how wide the character is (in our case, 1.2 metres) and how the character climbs stairs. Currently stair climbing is not explicitly handled by the demos or the state machine. In the examples provided the character climbs steps according to the geometry of the character. Some approaches to handle stair climbing include:

- Using a capsule allows the rounded bottom of the capsule to force the character upward.

- Use a traditional approach: Query the environment for a surface which the character cannot climb (you can use `hkpCharacterProxy::checkSupport()` for an indication of what's in front of the character) and then use a linear cast to move up to a maximum step height, retry the blocked move and then linear cast downward to contact the ground.
- Use your own `hkpConvexVerticesShape` to represent your own character controller with a sloped footprint to allow for smooth transitions over surfaces of a fixed height if you find a capsule does not give you desired behavior. Note however that a capsule will be faster on triangular meshes, as Havok has special a capsule triangle collision detection.

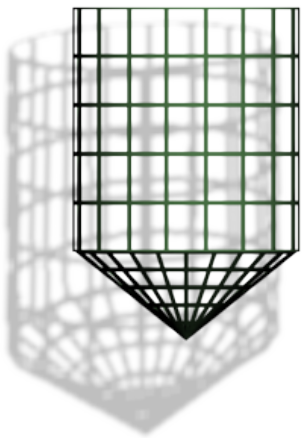
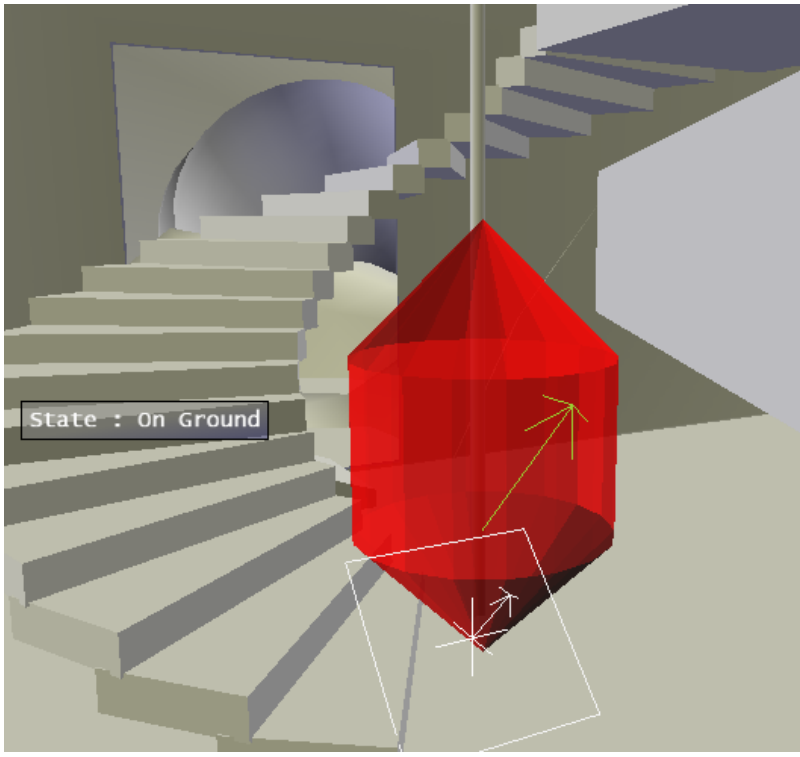
| | |
|---|---|
|  |   |
| The special hkpConvexVerticesShape shape | The character on stairs (on ground state) |

Table 2.9: The special hkpConvexVerticesShape character controller shape

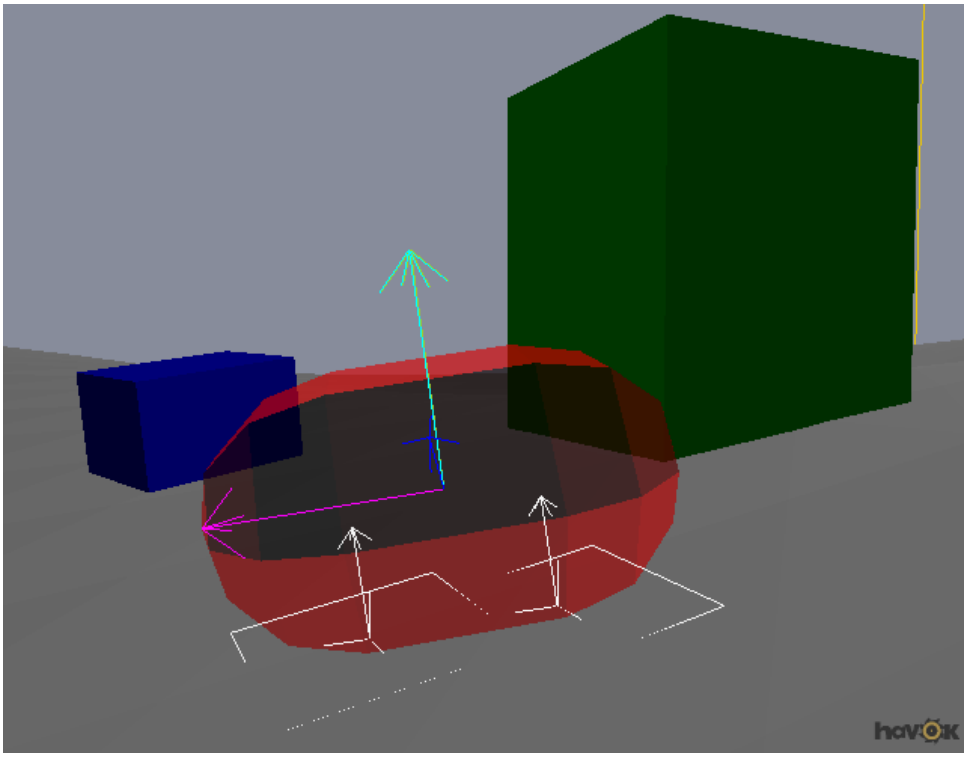
### The Rotating Asymmetric Shape

Currently the character moves by sweeping the current shape along its current velocity and probes the environment for new surfaces. Obviously this linear sweep does not take rotational effects into account. If you want to use a shape that is not rotationally invariant about the up axis then you have to calculate desired rotation according to character current orientation and surrounding environment. Each character type has its own rotation shape calculation strategy.

- The Character Rigid Body: Because the character rigid body is a part of the dynamic world, you can safely set only its angular velocity. The angular velocity can be calculated as an angle difference between desired and actual orientation of character rigid body divided by timestep. The ideal approach is to apply a simple forward controller with pre-multiplication gain and with limitation of maximal angular acceleration to ensure smooth rotation.

- The Character Proxy: You should rotate the shape manually outside the proxy integration step. You may choose to check for penetration as crouching does in the section below, or you may allow the controller to recover from penetration as it normally does.

The corresponding asymmetric character demos show how this might be done for particular controller type.



**Restricted Movement**

Both character controllers have built-in support for restricting the movement of the character on steep slopes. The character controller construction info structure has a max slope value which defines the steepest slope it can climb. Setting this to PI / 2 (i.e. vertical) disables this feature (i.e. allows the character to climb any slope). Implementation of this feature significantly differs between the two controllers.

- The Character Rigid Body: Internally all normals of contact points are checked and if any of them exceed maximal slope an additional contact point with normal perpendicular to the rigid body up direction is added to the contact manager which prevents movement.
- The Character Proxy: Internally all surface planes are checked and if any of them exceed maximal slope an additional plane is added to the manifold which prevents movement. The additional plane is placed at the same position as the original and oriented vertically, preventing movement up the slope. In the image below you can see that one of the original green planes in the manifold has been classified as too steep to climb and an additional orange constraint has been dynamically added to the manifold.

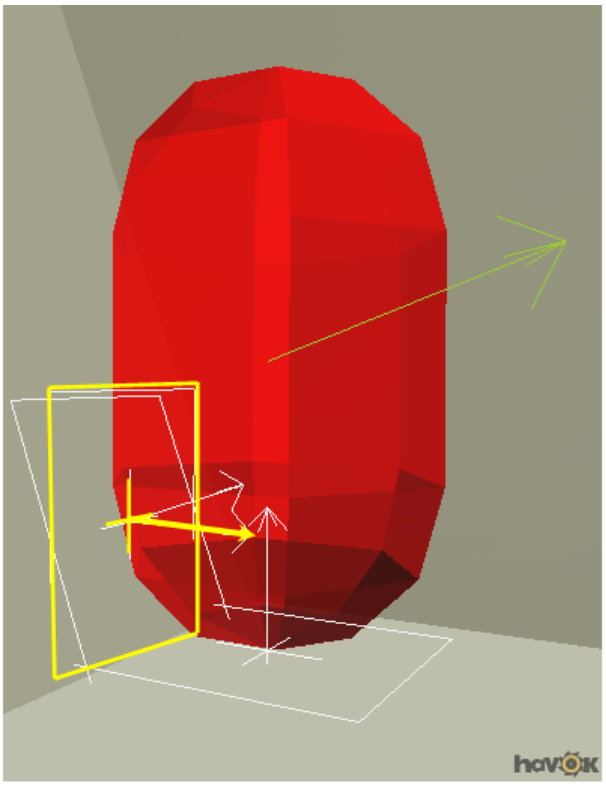
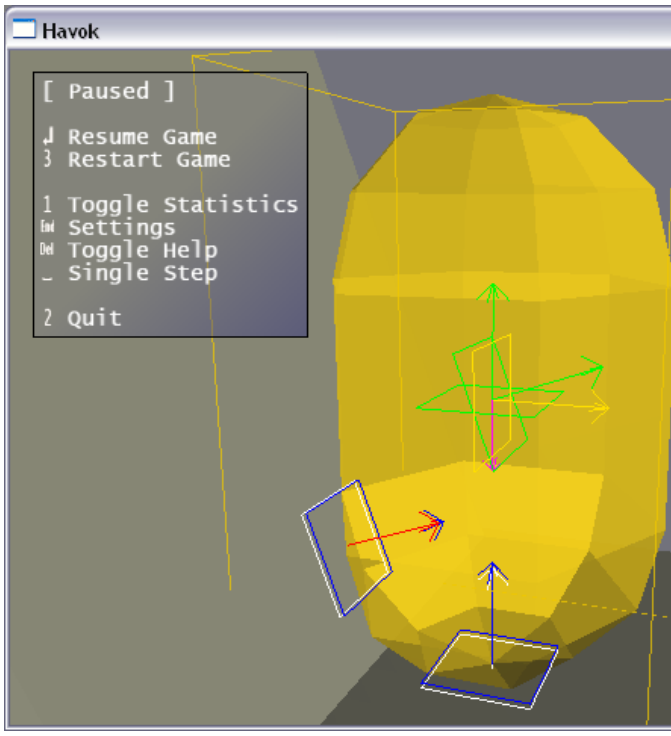
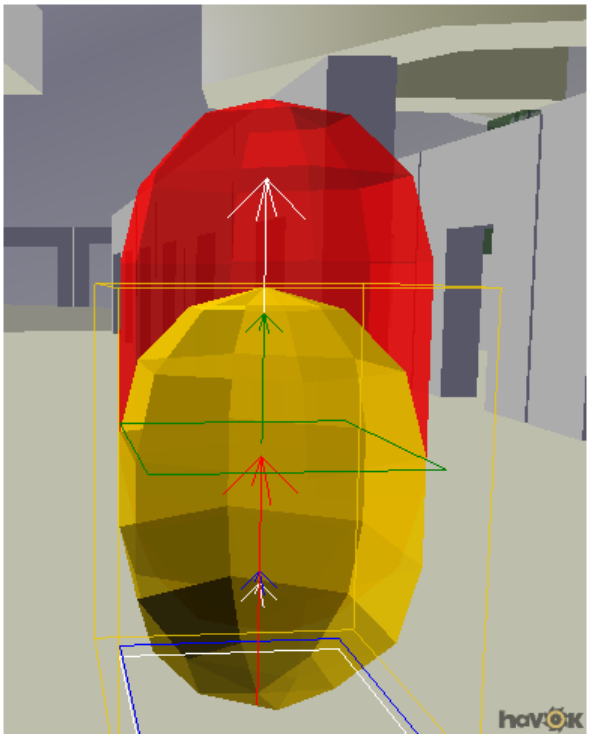
| | |
|---|---|
|  |   |
| The Character Rigid Body: Contact point added to the same position as original, but with new contact normal (yellow) | The Character Proxy: Orange constraint plane added |

Table 2.10: Maximal slope implementations

### Crouching

The character controller implements crouching by swapping the shape used to represent the character at runtime from a standing version to a crouched version. In examples we use two capsules to represent these states. The shape swapping is simple in the case of the rigid body character controller, we simply call `hkpRigidBody::setShape(newShape)` which will safely perform the update. This situation is more complicated for the proxy controller. See details in The Character Proxy chapter below.



#### 2.5.1.4 The Character Rigid Body

The character rigid body `hkpCharacterRigidbody` is used to represent a rigid body that can move dynamically around the scene. It is a rigid body analogy to the `hkpCharacterProxy` controller where, instead of using a phantom, an actual rigid body is used. This class creates a `hkpRigidBody` which you add to the world. There are several differences between the rigid body created by this class and a normal rigid body used for simulation:

- The character rigid body has an infinite inertia tensor (i.e. it cannot rotate)
- Gravity is not applied to the character rigid body (it uses a special motion called MOTION_CHARACTER which internally prevents gravity being applied).
- It uses a custom collidable quality type, HK_COLLIDABLE_QUALITY_CHARACTER, which is the same as HK_COLLIDABLE_QUALITY_CRITICAL except that it disables some internal optimizations which produce artifacts when used with rigid bodies that cannot rotate.
- Its movement is controlled by directly setting its linear and angular velocities.

All source code for the character rigid body is in `Physics2012\Utilities\CharacterControl\CharacterRigidBody`.

**The Character Rigid Body Listener**

Certain aspects of the character rigid body's behavior are managed by an attached `hkpCharacterRigidBodyListener`. The listener's callback is invoked at the end of every simulation step and its responsibilities include:

- Limiting the character's ability to climb steep slopes.

This is determined by the character's `m_maxSlope` member and is achieved by adding new contact points to block motion up steep slopes.

- Limiting the force the character can exert on other entities.

  This is determined by the character's `m_maxForce` member and is achieved by artificially reducing the mass of the character when it collides with other entities.

- Modifying contact point normals (in the case that the character's shape is a capsule)

  This is determined by the character's `m_unweldingHeightOffsetFactor`.

The listener contains no state, so it can be shared between character rigid bodies. Moreover, most of its methods are virtual, so that it can be easily extended. The listener is set by calling the `setListener` method of `hkpCharacterRigidBody` (see the section on construction below).

Adding a listener is optional. However, the `getSupportInfo` method also uses `m_maxSlope`, so when no listener is used, this should be set to ÏĂ/2 to avoid inconsistencies.

**Support**

Like the character proxy, the character rigid body can be queried for its **support** , i.e. whether it is standing on a surface and, if so, what kind of surface it is standing on. This information is needed by the state machine to control character motion.

In the character rigid body, support is obtained in a two phase process:

1. The `getSupportInfo` method determines which of the character's contact points, if any, correspond to a supporting surface. It returns the supported state of the character (`SUPPORTED`, `UNSUPPORTED` or `SLIDING`). If the character is supported or sliding, then it provides an array of support information containing the relevant contact points and corresponding rigid bodies.
2. The `getGround` method takes the array of support information provided by the `getSupportInfo` method and abstracts it into a single "supporting surface" by averaging the appropriate properties of the contact points. (A subtlety requires the caller to specify whether the velocities of dynamic bodies are used when computing the surface velocity. See the `MovingSupportRbDemo` demo for details.)

The `checkSupport` method provides the default functionality for support queries: it performs `getSupportInfo` and then `getGround` if appropriate. Its implementation is simply:

```
void hkpCharacterRigidBody::checkSupport( const hkStepInfo& stepInfo, hkpSurfaceInfo& ground ) const
{
    HK_TIMER_BEGIN("checkSupport", HK_NULL);

    hkArray<SupportInfo> supportInfo;
    ground.m_supportedState = getSupportInfo( stepInfo, supportInfo );
    if ( ground.m_supportedState != hkpSurfaceInfo::UNSUPPORTED )
    {
        // By default, dynamic body velocities are not used.
        getGround( supportInfo, false, ground );
    }

    HK_TIMER_END();
}
```

### Support parameters

Two members of the character rigid body influence how the character is supported.

- The `m_supportDistance` member is the maximum distance from the character a contact point can be for it to be judged as supporting. This should be less than or equal to the world's collision tolerance.
- The `m_hardSupportDistance` member is used to control how close to its supporting surface a character should be. (Note this applies only to non-dynamic surfaces.) This should be less than or equal to the supportDistance.

When the character travels across uneven terrain, bumps and troughs in the ground can cause the distance between it and the surface to change. These parameters particularly affect how smoothly the character can travel over such a surface.

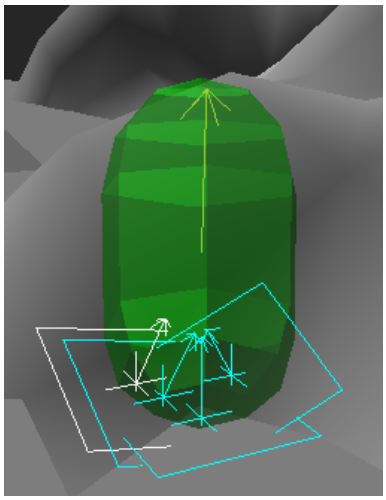
The supportDistance allows the character to be considered supported even if it is a small distance from the ground. It defines a vertical range within which the character can move without losing support.

On the other hand, hardSupportDistance tries to keep the character a little above the ground. This can be useful as the contact planes corresponding to seams in the ground will be a little less steep when first encountered.

### Debug display

The two check support methods display information useful for debugging:

- The `getSupportInfo` method displays all the character's contact points. Those colored cyan are judged to be providing support. Those colored red are the contact points added to stop the character climbing steep slopes. Other contact points are colored white.
- The `getGround` method displays the normal of the supporting surface by a yellowish arrow drawn at the character's position.



### Constructing the Character Rigid Body

To create a character rigid body you must fill out a `hkpCharacterRigidBodyCinfo` structure. The structure variables can be divided into the following three groups:

- Standard parameters used to setup a rigid body such as `m_mass` and `m_shape`.
- Character controller specific values, for example `m_up` and `m_maxSlope`.
- Helper parameters used by the `checkSupport` method, for example `m_maxSpeedForSimplexSolver`.

The character rigid body listener should be set by calling the `setListener` method of `hkpCharacterRigidBody`.

The character's rigid body should be added to the world.

The following code fragment shows how a character rigid body might be constructed and added to the world:

```
hkpCharacterRigidBody* characterRigidBody;
{
    hkpShape* characterShape;
    {
        hkVector4 bottom( 0.0f, 0.0f, 0.0f );
        hkVector4 top( 0.0f, 0.8f, 0.0f );
        hkReal radius = 0.6f;
        characterShape = new hkpCapsuleShape( bottom, top, radius );
    }

    hkpCharacterRigidBodyCinfo info;
    {
        info.m_shape = characterShape;
        info.m_mass = 100.0f;
        info.m_up.set( 0.0f, 1.0f, 0.0f );
        info.m_position.set( 0.0f, 0.65f, 0.0f );
        info.m_maxSlope = HK_REAL_PI * 0.33f;
        info.m_supportDistance = 0.1f;
        info.m_hardSupportDistance = 0.01f;
    }

    characterRigidBody = new hkpCharacterRigidBody( info );

    hkpCharacterRigidBodyListener* listener = new hkpCharacterRigidBodyListener();
    characterRigidBody->setListener( listener );
    listener->removeReference();

    m_world->addEntity( characterRigidBody->getRigidBody() );

    characterShape->removeReference();
}
```

### Application example: first person camera character

The `hkpFirstPersonCharacter` is an application example for the `hkpCharacterRigidBody`. The idea is to update the camera position to the simulated rigid body position, thus allowing for physical behavior of the camera. You can retrieve the orientation and position through its interface methods `getForwardDir()` and `getRigidBody()->getPosition()`. See the `CharacterUtils::stepCharacter()` method in the Havok Demos for an example.

You can configure the first person behavior through the `hkpFirstPersonCharacterCinfo` passed in upon construction:

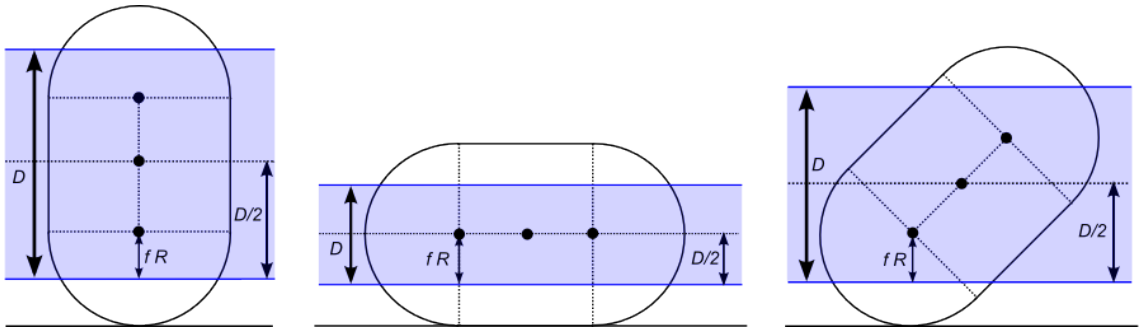
```
m_world->markForWrite();
{
    hkpFirstPersonCharacterCinfo info;
    if (characterInfo)
    {
        info = *characterInfo;
    }
    else
    {
        info.m_position.setZero4();
        info.m_direction.set(-1, 0, 0);

        // set human proportions (8 head model)
        info.m_capsuleHeight = 1.8f;   // 1.8m size
        info.m_capsuleRadius = 0.25f;  // 50cm waist
        info.m_eyeHeight = 0.85f;      // eyes at 1.65m
        info.m_gravityStrength = 1.0f; // 75kg
        info.m_maxUpDownAngle = HK_REAL_PI * 0.4f;
    }

    info.m_world = m_world;
    info.m_up.setNeg3(m_world->getGravity());
    info.m_up.normalize3();
    info.m_flags = (hkpFirstPersonCharacter::ControlFlags)(info.m_flags | hkpFirstPersonCharacter::
      DISABLE_JUMP);

    m_char = new hkpFirstPersonCharacter( info );
    m_char->m_characterRb->getRigidBody()->setMass(75.0f);
    m_char->m_characterRb->getRigidBody()->addProperty(HK_PROPERTY_DISPLAY_PTR, 0); // hide in demo
      display, still show in VDB
    m_char->m_characterRb->m_unweldingHeightOffsetFactor = 2.0f;
    m_char->m_characterRb->m_maxSlopeCosine = hkMath::cos( 80.0f * HK_REAL_DEG_TO_RAD );
    m_world->addEntity( m_char->m_characterRb->getRigidBody() );
    {
        hkpCharacterRigidBodyListener* listener = new hkpCharacterRigidBodyListener();
        m_char->m_characterRb->setListener( listener );
        listener->removeReference();
    }
}
m_world->unmarkForWrite();
```

### Unwelding

Character rigid bodies are given very high inertia to stop them rotating due to collisions. A side-effect is that they respond badly to incorrectly orientated contact point normals. Contact points with inappropriate normals sometimes arise as a result of welding. In the case that a character's shape is a capsule, the `hkpCharacterRigidBodyListener` can reorient contact points to ensure that they point in an acceptable direction. We call this process *unwelding*.

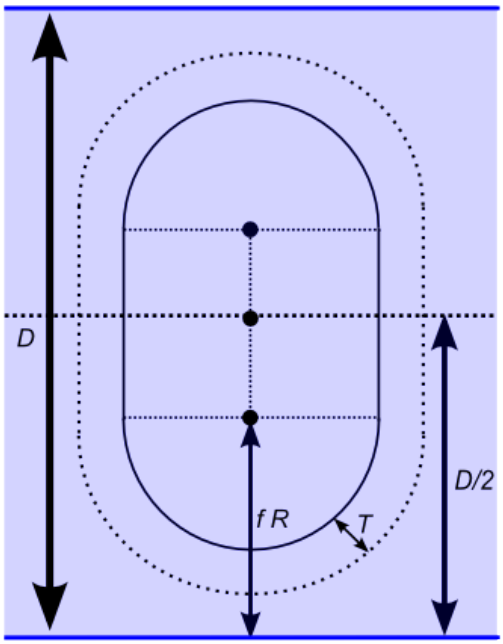
Since welding is nevertheless useful in smoothing a character's movement over tessellated terrain, unwelding can be tuned so that it applies only to a vertical region around the capsule's center point. `hkpCharacterRigidBody` has a parameter `m_unweldingHeightOffsetFactor` which is used to determine the height (with respect to the character's `m_up`) of the region.

In the diagrams below, the height or the unwelding region is denoted by D. All contact points in the blue regions will be reoriented.




In order to provide a sensible scale-independent default, `m_unweldingHeightOffsetFactor` is expressed as factor of the capsule's radius, R. This factor is denoted by f in the diagrams and may be set using the `hkpCharacterRigidBodyCinfo` structure. D is then calculated by adding fR to the vertical distance between a capsule vertex and its center. The default for `m_unweldingHeightOffsetFactor` is 0.5.

If it is intended that all contact points be unwelded, then `m_unweldingHeightOffsetFactor` should be set to a large positive value. This ensures that the region extends beyond the collision tolerance, denoted by T in the diagram below.

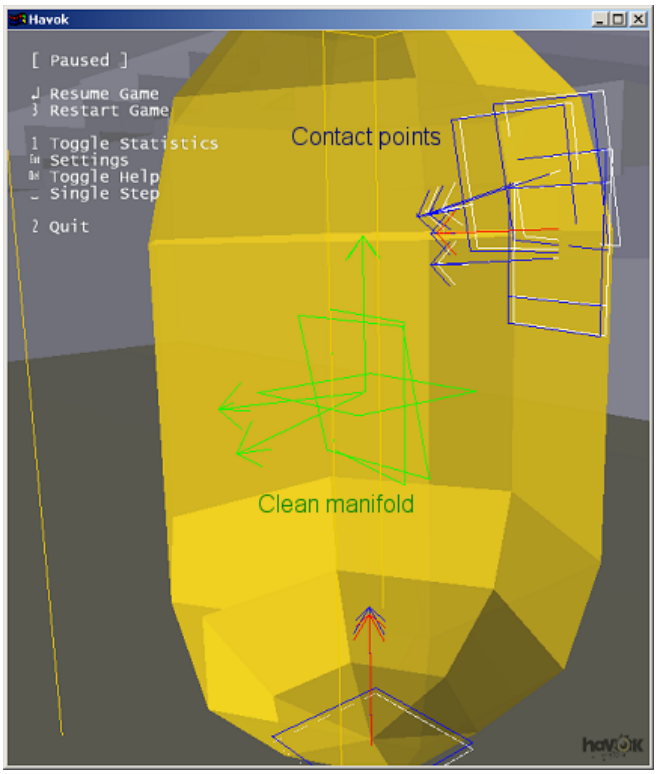



Conversely, if it is intended that no contact points be unwelded, then m_unweldingHeightOffsetFactor should be set to a large negative value, so the region has zero height.

#### 2.5.1.5 The Character Proxy

The character proxy represents the surface boundary for the character. The proxy encapsulates a shape with some spatial and temporal information and builds up a representation of the surrounding environment. For the user you should consider this simply as a shaped particle. It has a surface, a position and a velocity. To interact with it you simply set one or more of the attributes and call `integrate` explicitly.

The character proxy will take care of the rest. It will maintain a clean manifold of plane equations that represents the surface contacts it currently has. These plane equations are fed to the next layer which produces a new position and velocity for the character. All source code for the character proxy is held in `Physics2012\Utilities\CharacterControl\CharacterProxy\`.




If you are using the debug versions of the Havok libraries then the Visual Debugger will display the contact planes shown in the image above. You can see how the character proxy uses linear casting to build up a manifold of planes for the simplex solver . The original hit position of each point in the start point collector is shown as a white plane. We also display this same plane displaced by the distance returned by the hit, in blue if the distance is positive or red if the distance is negative. Planes returned by the cast collector are shown in magenta. The character proxy uses these planes to verify and update its manifold. This maintenance converts the relatively noisy signal returned from the linear caster into a smoother cleaner signal that is passed to the simplex solver. The clean planes passed to the simplex solver are shown in the middle of the character in green. When looking at this debug output you should verify that the green planes will appropriately restrict the characters movement.

### Maintaining the Manifold

When filtering the signal from the linear caster the character proxy uses both the start collector and cast collector results. The start collector contains all the contacts that surround the character in its current position at the start of the movement step or cast. The cast collector holds the distance to the first point of contact for each surface along the character's current path. These results are checked and a manifold of valid planes is constructed and maintained according to the following rules.

- All planes currently in the manifold must verify that they are still valid. This is done by checking that they are present in the start point collector.

- The closest (or most penetrating) plane picked up by the start point collector is added to the manifold, if it is not there already. This will ensure that an object which penetrates the character (say by moving in from the side) will be picked up by the manifold.
- All penetrating planes have their distance set to zero when passed to the simplex solver, but are artificially given velocity so the character is pushed away from them and out of the penetrating state.
- The first cast collector plane is added. This represents a surface immediately blocking the character and must be taken into consideration.

These rules produce a clean and minimal manifold that can be passed to the simplex solver. The simplex solver ultimately uses these to restrict the movement of the character. These rules are also set out specifically to ensure that the character does not hit triangle edges as it moves over a tessellated landscape.

**Constructing the Character Phantom**

To create a character phantom you must fill out a `hkpCharacterProxyCinfo` structure. When filling out this structure you will be asked to specify a `hkpShapePhantom`. The shape phantom is used by the proxy to probe the environment for surface details. When constructing this shape phantom you specify the shape you wish to use for your character.

**Crouching**

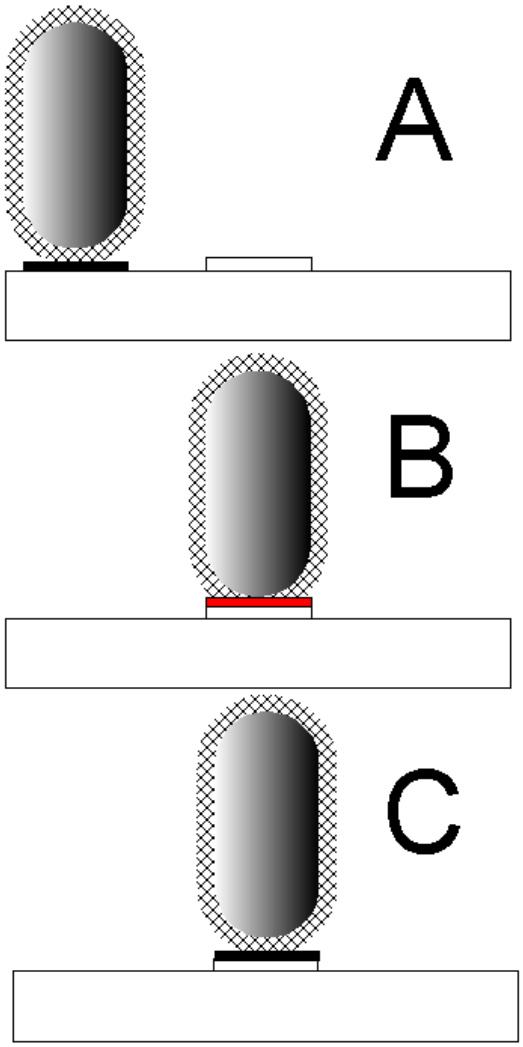
When switching between the shape we first ensure that the transition will not cause penetration. This prevents the character from standing in areas where the ceiling is too low. The code to perform this safe swapping is shown here

```
void CharacterDemo::swapPhantomShape( hkpShape* newShape )
{
    // Remember the current shape
    hkpShape* currentShape = const_cast<hkpShape*>(m_phantom->getCollidable()->getShape());

    // Swap to the new shape.
    // N.B. To be safe, we always remove the phantom from the world first, then change the shape,
    // then re-add, in order to refresh the cached agents in any hkCachingShapePhantoms which
    // may also be present in the world.
    // This also forces the display to be rebuilt, which is necessary for us to see the new shape!
    {
        // Note we do not have to add a reference before removing becasue we hold a hkpCharacterProxy
        // which has a reference to this phantom - hence removal from the world cannot cause this phantom
          to
        // be accidentally deleted.
        m_world->removePhantom( m_phantom );
        m_phantom->setShape(newShape);
        m_world->addPhantom( m_phantom );
    }

    //
    // We use getClosestPoints to check for penetration
    //
    hkpClosestCdPointCollector collector;
    m_phantom->getClosestPoints( collector );

    // Allow a very slight tolerance (approx 1cm)

    if (collector.hasHit() && collector.getHit().m_contact.getDistance() < .01f)
    {
        // Switch the phantom back to our current shape.
        // N.B. To be safe, we always remove the phantom from the world first, then change the shape,
        // then re-add, in order to refresh the cached agents in any hkCachingShapePhantoms which
        // may also be present in the world.
        // This also forces the display to be rebuilt, which is necessary for us to see the new shape!
        {
            // Note we do not have to add a reference before removing becasue we hold a hkpCharacterProxy
            // which has a reference to this phantom - hence removal from the world cannot cause this
              phantom to
            // be accidentally deleted.
            m_world->removePhantom( m_phantom );
            m_phantom->setShape( currentShape );
            m_world->addPhantom( m_phantom );
        }
    }
}
```

### Keep Distance

The character proxy has an internal parameter called `keepDistance` which provides a soft penalty shell around the character shape allowing it to smooth the signal coming from the collision detector.

The controller gathers plane equations using both a cast collector and the start point collector. The start point collector is used to verify existing contacts in the manifold. It represent the environment around the character in its current position. When making this query the tolerance used is the `keepDistance`. The cast collector is used to check for objects the character is about to run into. These contacts are translated into plane equations and passed to the simplex solver. The `keepDistance` is also used to expand the planes returned by the collectors. The following image shows what happens when a character with a large `keepDistance` travels onto a small bump.




The diagram shows the shaded capsule which represents the characters shape and a hatched area showing the `keepDistance`. At point A the character is stationary. The plane is black was initially added when the character hit the ground. It was added by a cast collector and verified on successive frames because it remained present in the start point collector. Note that it has been expanded by the `keepDistance` so the character floats above the ground. At point B the character has cast along the ground. The cast collector does not use the keepDistance as a tolerance so the cast missed the small step. The step does appear in the start point collector and since it is inside the `keepDistance` it is marked as penetrating. Penetrating planes are handled by adding velocity to them, which slowly pushes the character away; their distances are zeroed before they are passed to the simplex solver so they do not produce a sharp discontinuous movement. If the character stops on the step (shown at C) it will rise up slowly. If the character passed quickly over the step then the plane verified by the start point collector will drop back to the original height again and the penalty recovery will cease.

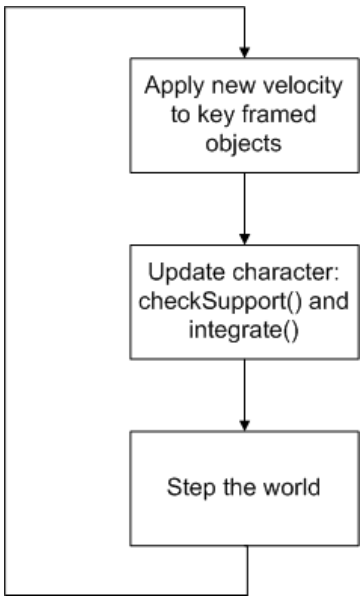
In summary: The shape plus radius defines a hard shell around the character which the cast collector and simplex solver enforce. The `keepDistance` provides an additional soft shell which is enforced using a simple penalty recover method. This soft penalty recovery filters high frequency discontinuities from the collision detector in the range of the soft shell.

**Moving Platforms and Low Frequency Characters**

While Havok is a continuous physics engine, the character controller is effectively discrete in its simulation approach. Although this is fast, and means that characters can be stepped independently of the physics engine, and at lower frequencies, this can also lead to some accuracy problems. The first of these is that of the moving platform. When a character is in the air above a platform that is rapidly moving upwards, it may not see the moving platform until it is too late. When `hkpWorld::step()` is called, the platform will be integrated upwards some amount, depending on its velocity. At a certain point the AABB of the platform and the character controller will overlap. When this happens, at the next integration call for the character controller, the character will see the moving platform and if it is close enough, add an appropriate plane into its simplex solver. However it is possible that at this stage, the platform will already be penetrating the character proxy. The character proxy will be able to recover gracefully from this (see the section on maintaining the manifold, above) however you may see some frames where the character and the platform are visibly penetrating.

The situation is also apparent when a character is on a moving landscape. During the character controller `integrate()` call, all calls to the collision detection are with the landscape in the same position (i.e. the landscape does not move as time advances in the character controller). Because of this, additional artifacts such as occasional hitting of triangle edges may be apparent in fast moving landscapes; where possible for best simulation of the character controller, it is advisable to keep landscapes fixed.

Note that if you are changing the velocity of a platform with which the character is interacting, it is best to set the velocity of the platform (for example using the keyframed utility) prior to calling the character controller integrate call. The best order of calls is as follows:




If you apply velocities to your keyframed objects before calling the character controller, then the character controller will react correctly to the keyframed object. Otherwise, if for example you have a character on an accelerating elevator, the character will always see the elevator's velocity from the last step, and will sink slightly into the elevator as it accelerates (although penetration recovery should prevent it sinking too far).

As all character controllers are stepped independently, it is possible to step them at low frequency. For example you could step the physics simulation at 60 Hz, but step a character controller at 20Hz. Obviously this will give you a performance gain (if you make sure that you step different characters out of sync, and so spread the CPU load between frames), however it makes the situation described above worse.

The quality of interactions between the character and dynamic objects, keyframed objects and other characters will be degraded at lower frequencies. You are more likely to see penetrations such as those described above. In general, if you can afford it, it is best to step the character controller at the same frequency as your simulation. If you have many AI characters, it may be worth reducing the frequency for them, however you should experiment to make sure that any resultant penetrations are acceptable.

### The Character Proxy Manifold

The character proxy caches an array of `hkpRootCdPoint` structures, called the manifold, which you can access through the call `getManifold()`. This is the data that is used to build planes for the simplex solver which are used for both `integrate()` and `checkSupport()` calls, and it is the character's view of its immediate world. However, if you are using this data yourself, for example in the state machine, it is worth noting that the data can easily be out of date. In fact, at the end of the `integrate()` call, the character will have moved from the position at which the manifold was built (to the end of its step) and so the contact points in the manifold will not be exactly correct. The character proxy call `refreshManifold()` can be used to bring these points up to date. This call re-evaluates all collision detection at the current position of the character controller, and updates the manifold. Note that moving any other character controller, or calling the hkpWorld step functions may also make the manifold out of date. At the start of the `checkSupport()` call, `refreshManifold()` is called automatically, so after `checkSupport()` the manifold will be correctly up to date.

### Characters and Heightfields

Because of the logic used to maintain the character's manifold, you should be careful if you wish to use characters with heightfields (i.e. classes deriving from `hkpSampledHeightFieldShape`). Heightfield collision detection simply collides the vertices of the moving object against the heightfield; see the heightfield section of the collision detection chapter for more details. In the case where a capsule is used as a character controller's shape, only one point is ever generated against the heightfield, so only one point will ever be in the character's manifold. Unless the heightfield is virtually flat, this is likely to cause problems. If a character is in a narrow valley, the manifold should contain points on either side of the valley, but if the valley is represented by a heightfield, only one point will be generated, and jitter is likely to result. As a workaround for this, you should wrap the heightfield with `hkpTriSampledHeightFieldCollection` and `hkpTriSampledHeightFieldBvTreeShape`, which will give triangle accurate collision results.

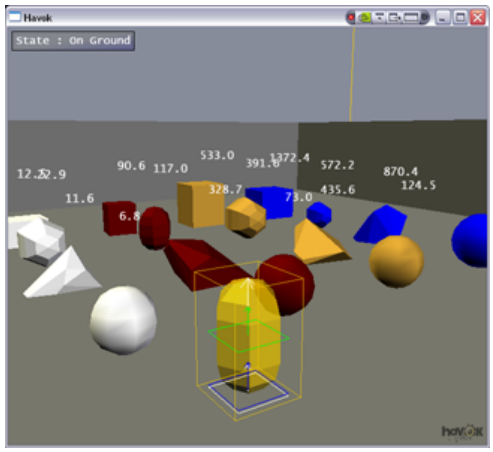
### Physical Interaction and the Interaction Listener

The character proxy listener provides 2 methods specially designed to control character interaction. The first deals with character object interaction.

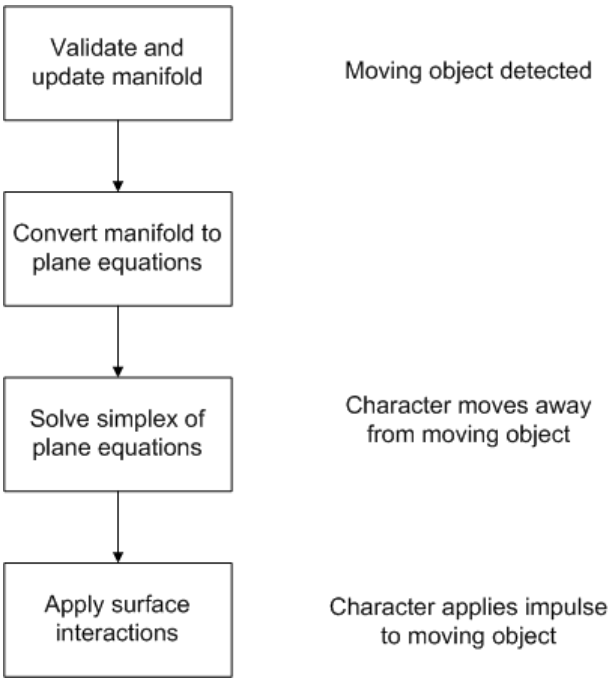
```
// Called when the character interacts with
      another object virtual void objectInteractionCallback(hkpCharacterProxy*
      proxy, const hkpCharacterObjectInteractionEvent& input,
      hkpCharacterObjectInteractionResult& output )
      {}
```

When the simplex has been passed a set of plane equations it solves these as described below. For each of the dynamic surfaces (not owned by a fixed or keyframed body) touched while finding a solution it calls back the listener with an object interaction callback. Before calling back the listener, it calculates an appropriate impulse to apply to the object and fills in the details of this calculation in the input structure. This information can be particularly useful when trying to play sounds or estimate damage to apply to the character. The output structure contains the impulse to apply to both the object and the character. When the callback returns, these results are applied immediately. You are free

to completely override the suggested values passed in the input structure. The character interaction demo, shown below, demonstrates the default character interaction behaviour. It contains a scene with random objects with varying masses. The character's strength contained in the `hkpCharacterProxyCinfo` structure determines how easily the character will push the various objects around.




Interaction between the character controller and rigid bodies is not perfect, due to the fact that the character controller interacts by applying impulses to rigid bodies outside the constraint solver. While in general this gives good results, there are some drawbacks to this approach. A simplified version of the character integrate() loop is as follows:




It can be seen that the interaction between a character controller and a rigid body is a two phase process. Firstly the character sees and avoids the interaction plane generated from collision with the rigid body. After this the character applies an impulse to the rigid body. With the loop ordered this way, when an object hits the character, it will always be able to move the character slightly before the character gets a

chance to apply an impulse to it. This can be unwanted, if you want the character to be infinitely strong. This can be seen in the character interaction demo, where you can fire boxes at the character from the side. It is possible to fix this problem by using the `objectInteractionCallback` described above to zero the velocities of the planes in the simplex solver. See the demo code from the character interaction demo for an example of this.

It might be tempting to rewrite the above loop, so that the character applies impulses before solving the plane equations, to avoid this problem. However this leads to a worse problem: consider a character pushing a box against a wall. The box will not be able to move, so from frame to frame, we should see a static plane in the character manifold resulting from this. If we were to apply an impulse to the rigid body before solving the simplex, however, the plane would be seen incorrectly to be moving away from the character. This leads to considerable unwanted jitter.

Another problem can arise when a character controller gets squeezed between two rigid bodies. Because the controller applies impulses to each rigid body without reference to the other, if the bodies are sufficiently heavy, the system will not be solved correctly (as would, for example a similar system where the middle body was not a character controller, but a rigid body). In this case the rigid bodies will penetrate the character controller, and may eventually hit each other. While this is not ideal, it is an unavoidable consequence of simulating the character controller as a phantom "outside" the simulation.

The character imparts a downwards force on rigid bodies it is standing upon. This is controlled by the `m_characterMass` value in the character proxy. By default this value is 0, which means this feature is disabled. Setting the value to say 100 means the character will apply a force equivalent to that of a 100 kg rigid body. Note that the gravity parameter passed into the character integrate function is used with the `m_characterMass` vale to calculate the downward force. It must be set to the gravity of the `hkpWorld` to get the correct behavior. This value does not affect how fast the character falls when in air; that is controlled outside the integrate call by the state machine.

The second interaction method deals with character-character interaction. When two characters collide the proxy calls the following listener callback

```
// Called when the character interacts with another
      character virtual void characterInteractionCallback(hkpCharacterProxy*
      proxy, hkpCharacterProxy* otherProxy, const hkContactPoint& contact)
      {}
```

This will be called once per character if all characters are running at the same frequency, or once per interaction if characters are running at different frequencies. This method is passed handles to both controllers involved in the collision. You are free to decide exactly how the characters should respond to the interaction by changing either linear velocity, position or even potentially the shape of the characters. The default behavior leaves both characters unaffected.

### Multithreading the hkpCharacterProxy

As of version 2010.1, support for multithreading the simulation of `hkpCharacterProxy`s on both CPU and SPU is available. To multithread the processing of characters, you create `hkpCharacterProxyIntegrateJobs`, containing a set of `hkpCharacterProxyIntegrateCommands`, which are put on the job queue.

To simplify this process, we provide a `hkpCharacterProxyJobUtil` utility. You must create a `hkpCharacterProxyJobUtil::JobData` structure describing the work to be done. You must give the number of jobs to create, the job queue onto which to put them and an array of `hkpCharacterProxy`

pointers. Since the jobs are given a fixed buffer into which to write their results, you must also specify the maximum number of interactions which should be handled for each character.

If synchronous simulation is satisfactory, you can use the static method `simulateCharactersSynchronously()` to do all the work. This will allocate a buffer, create the jobs, run them using all available threads (those in the `hkpThreadPool` argument and the calling thread), handles the results and deallocates the buffer before returning.

On the other hand, if you want to simulate the characters asynchronously, then you can use the method `buildAndAddJobs()` to create the jobs and the method `handleResults()` to handle the results after you've run them. You must provide your own buffer for this, which should be at least `getBufferSize()` bytes in size.

**Notes for Multithreading on SPU**

The character proxy builds its manifold using world linear casts on SPU. You must provide a broadphase or a KD-tree in the `hkpCharacterProxyJobUtil::JobData`. (One subtlety worth noting is that the `hkpKdTreeWorldManager` can change its `m_kdTree` pointer when `updateFromWorld()` is called.)

If a `hkpCharacterProxy` has a `hkpCharacterProxyListener`, the listener's callbacks will not be fired if integration runs on SPU. There are a few different ways to manage this.

- Firstly, you can force the job onto CPU by setting its `m_jobSpuType` to `HK_JOB_SPU_TYPE_DISABLED`.
- Hooks have been provided for callback functions on SPU. Call the `hkpSpuCharacterProxyUtil::registerCustomFunctions()` method in the `hkpSpuCharacterProxy` to register your functions.
- You can write your own function which handles the results of the job and processes the object interactions results in whatever way you see fit.

**Interactions with Phantoms and Trigger Volumes**

Since the `hkpCharacterProxy` is responsible for its own collision handling, special treatment is needed for certain interactions with other `hkpWorldObject`s.

**Interaction with Phantoms**

If unmodified, the `hkpCharacterProxy` will not penetrate other phantoms (`hkpPhantom`). This is appropriate for phantoms which are other characters, but probably undesirable for non-character phantoms.

The easiest and cheapest way to allow characters to penetrate non-character phantoms is to use a collision filter to disable collisions between them. The collision system will entirely ignore their collisions, and no information about the collision will be available after the fact.

An alternative is to provide a special collector which ignores collisions with phantoms which are not also `hkpCharacterProxys`.

Depending on how the character is being integrated, there are different ways of providing the collector:

- When running the character singlethreaded, you can provide your own collectors by calling `integrateWithCollectors()` directly.

- When multithreading on CPU, you can provide a factory function in the `m_createCdPointCollectorOnCpuFunc` member of the `hkpCharacterProxyJobUtil::JobData`.
- On SPU, you should call `hkpFixedBufferCdPointCollector::registerCustomAddCdPointFunction()` in place of `hkpFixedBufferCdPointCollector::registerDefaultAddCdPointFunction()` in `hkpSpursCharacterMain.cpp`. We provide a function which can be used: `hkpSpuCharacterProxyUtil::customAddCdPointImplementation()`.

**Interaction with the Trigger Volume**

The character's integrate function does not fire the callbacks which would allow the Trigger Volume (`hkpTriggerVolume`) to detect its interactions (see the hkpTriggerVolume section). Instead, we have hard-coded support for the `hkpTriggerVolume` into the character.

Contact points with a `hkpTriggerVolume` do not end up in the manifold. Rather, a pointer to the `hkpTriggerVolume` is recorded in a separate array.

When multithreading the character, you must specify a limit for how many collisions with `hkpTriggerVolumes` should be recorded during the step.

#### 2.5.1.6 Comparison

This section summarises the difference between the two character control solutions, and help you make an informed choice about which approach is most suitable for your needs. On the whole, we recommend using the rigid body character controller over using the proxy controller

**The hkpCharacterRigidBody**

The `hkpCharacterRigidBody` is the newer of the solutions for character control. It benefits from higher performance and improved simulation quality at the cost of some loss of user control.

**Advantages**

Some advantages of the `hkpCharacterRigidBody` are:

- The `hkpCharacterRigidBody` is physically present in the world, so most of its processing (including collision detection and solving) is done within the physics step*. The work to be done is efficiently multithreaded with the rest of physics and, consequently, the `hkpCharacterRigidBody` performs better than the `hkpCharacterProxy`.
- The character's rigid body is integrated and solved using the same algorithms that apply to other entities. Thus, simultaneous collisions involving several characters and other bodies are handled correctly.
- The character's collisions trigger callbacks on the standard `hkpContactListener` interface (this applies to any listener attached to the character, the colliding body or the world).

* Some processing, e.g. maxSlope handling, is done outside the physics step.

**Disadvantages**

Some disadvantages of the `hkpCharacterRigidBody` are:

- The `hkpCharacterRigidBody` is necessarily updated at the physics simulation frequency. This excludes the possibility of simulating characters at a different frequency, e.g. in sync with game logic/user input/AI/network synchronization/animation frequency.
- The user has less control over its behaviour. Physics 2012 will try to make the character behave (largely) like a typical `hkpRigidBody`*, but such behaviour is not always appropriate for a character controller. For example, the `hkpCharacterRigidBody` is more likely to lose contact with the ground when moving up stairs than the `hkpCharacterProxy`, and it is more difficult to correct those problems for the `hkpCharacterRigidBody`.
- Unless welding is enabled for terrain, it will likely collide and bounce off triangle edges.
- The `hkpCharacterRigidBody` has infinite inertia in order to stop it being rotated during collisions with other bodies. However, this can cause problems if it encounters an edge with bad welding information.

* Actually, it is not quite typical, because it has a motion type of `MOTION_CHARACTER` and a collision quality of `COLLISION_QUALITY_CHARACTER`.

**The hkpCharacterProxy**

The `hkpCharacterProxy` is the older solution for character control. The user has more direct control, but it is not as high performance, nor does is benefit from using Havok's standard collision detection and solving.

**Advantages**

Some advantages of the `hkpCharacterProxy` are:

- Since the character is not a rigid body, the user has more direct control over its position and velocity.
- Character update can happen at a different frequency than physics, e.g. in sync with game logic/player input/AI/animation.

**Disadvantages**

Some disadvantages of the `hkpCharacterProxy` are:

- Almost all processing for the `hkpCharacterProxy` is done outside the physics step* so its collision detection is not automatically multithreaded. Typically, character proxies are simulated in a singlethreaded phase outside the physics step. (Support for multithreading a collection of `hkpCharacterProxy`s on CPU threads has been added in 7.0 and multithreading on SPU threads has been added in 2010.1. See above.)
- The algorithms which calculate the collision detection for the `hkpCharacterProxy` take account of each collision independently. It does not handle multiple simultaneous collisions correctly.
- High speed `hkpRigidBody`s can pass through the `hkpCharacterProxy` (since the character's collidable is not in the world, it will not generate a TOI).

- The `hkpCharacterProxyListener` interface is unique to the `hkpCharacterProxy`. You cannot attach a hkpContactListener to the `hkpCharacterProxy` and callbacks on the `hkpContactListeners` of colliding bodies are not fired automatically.

* The characters' phantoms are in the world, so some broadphase work is done during the physics step.

#### 2.5.1.7 The Simplex Solver

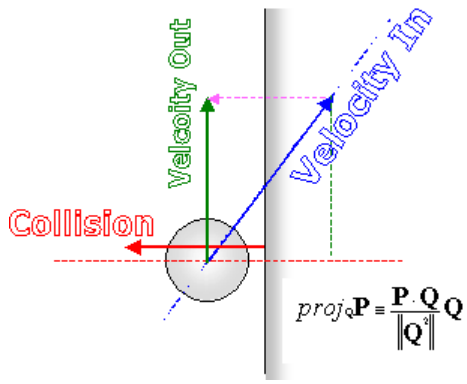
The simplex solver is employed in both character controllers. It has a very similar usage in either implementations of `checkSupport()` method. However for the character proxy controller the simplex solver is a core of movement calculation as described above. The planes gathered from the character proxy represent a set of simultaneous constraints that the proxy may not violate. Each of these constraints is detailed and includes surface, velocity, friction and priority information.

```
/// This represents a single constraint fed to the simplex solver
struct hkpSurfaceConstraintInfo
{
    /// The plane direction (normally identical to the contact normal). Note: .w component is distance
    hkVector4 m_plane;

    /// The velocity of the plane
    hkVector4 m_velocity;

    /// The static friction (behaves pretty much as expected from real friction)
    hkReal m_staticFriction;

    /// An extra friction in the up direction, if this function is used (value != 0), than
    /// the overall friction will be split into two components:
    ///  - one in the most up direction
    ///  - one which is perpendicular to the up direction
    /// Then each component is solved separately (box friction instead of friction circle)
    hkReal m_extraUpStaticFriction;
    hkReal m_extraDownStaticFriction;

    /// The dynamic friction:
    ///  - if set to 1.0f, then the simplex will simply project the velocity onto each plane
    ///  - if set to 0.0f, then the algorithm will try to maintain the input velocity
    hkReal    m_dynamicFriction;

    // The relative priority of the surface
    int       m_priority;
};
```

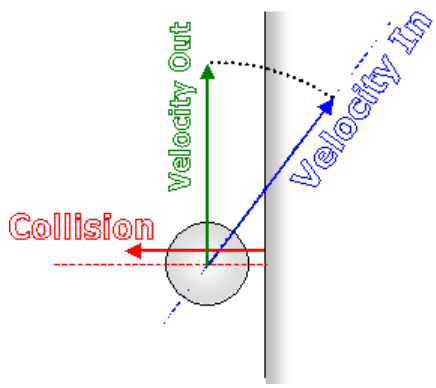
A collection of these constraints together with a velocity, position and time information is sent to the simplex solver. The solver then attempts to maximize the distance the character can move in the given direction over the given time step. It produces a new position and velocity. Many traditional approaches attempt to solve these surface constraints one after the other rather than in parallel. This can lead to unwanted jittering in the final controller. The user has direct control over the input structure passed to the solver. By registering a `hkpCharacterProxyListener` and implementing a `processConstraintsCallback` you will receive a callback containing the complete simplex input structure before it is passed to the solver. You may override any of the variables and dynamically add planes on the fly.

**Friction**

The solver handles both dynamic and static friction. Dynamic friction controls how much velocity or momentum is lost when a character hits a surface. When dynamic friction is set to 1 all velocity in the direction of the surface is clipped. This is the common behavior for first person shooters. The static friction gives you control over when the proxy will stick to a surface. If the ratio between the surface tangent velocity and the velocity projected in the surface normal direction is less than the static friction coefficient then the velocity of the controller in the tangent plane is clipped to the velocity of the surface at the point of contact. Note: If you are simply trying to restrict movement e.g. prevent the character from moving up a slope it is better to add additional planes to the manifold than directly change this value.




When dynamic friction is set to 0, kinetic energy is conserved.




#### 2.5.1.8 The Character Controller Examples and Use Cases

**Character Proxy Demos**

These demos introduce the basic concepts and behaviour of the character proxy controller. They can be found under "Character Proxy" in the demo "Character Control" menu (`\Physics2012\UseCase\CharacterControl\CharacterProxy\`).

- Asymmetric Character: Shows how to create a character with rotationally asymmetric shape and how to ensure its safe rotations. Use cases are swimming characters, horses, dogs, etc.

- Character and Triggers: Shows an interactions with rigid bodies and phantoms and how to implement phantom trigger objects using the custom collectors.
- Character Controller: Shows the behaviour of character in test static scene with many problematic objects from real game levels such as stairs, ladders, gaps, etc.
- Character Interaction: Shows interaction with objects of different sizes and weights.
- Character Phantom Interaction: Shows another technique how to interact with phantoms and how to implement phantom trigger objects.
- Character Priority: Shows how the character controller uses priority to handle insolvable situations.
- Low Frequency Characters: Shows how the character controller can be used efficiently for multiple characters at heightfield and how to simulate these characters at different frequencies.
- Multiple Characters: The demo used to test the performance of multiple characters.
- Character Weight: A demo which puts a character and a box on one side of a pan scales and two boxes of equal mass on the other. Some mouse plucking may be required if the demo takes a while to stabilize.
- Stair Climbing: A tweakable demo which tests the character's ability to climb stairs of various sizes.

**Character Rigid Body Demos**

These demos introduce the basic concepts and behaviour of the character rigid body controller. Most of them are analogies to the character proxy demos presented above. The demos can be found under "Character Rigid Body" in the demo "Character Control" menu (`\Physics2012\UseCase\CharacterControl\CharacterRigidBody\`).

- Asymmetric Character Rb: Shows how to create a character with rotationally asymmetric shape, how to control it by setting angular velocity to ensure safe rotations. Use cases are swimming characters, horses, dogs, etc.
- Triggers and Phantoms Character Rb: Shows an interactions with rigid bodies and phantoms and how to implement phantom custom listeners.
- Character Controller Rb: Shows the behaviour of character in test static scene with many problematic objects from real game levels such as stairs, ladders, gaps, etc.
- Platforms Character Rbs: Shows an interaction of the character rigid body with vertical and horizontal moving keyframed platform.
- Pushed Character Rbs: Shows an interaction of the character with another "free" and "AI" controlled characters.
- Interaction Character Rb: Shows interaction with objects of different sizes and weights.
- Multiple Character Rbs: The demo used to test the performance of multiple characters.
- Character Weight Rb: A demo which puts a character and a box on one side of a pan scales and two boxes of equal mass on the other.
- Moving Support Rb: A demo which illustrates how to get character rigid bodies to stand on rigid bodies which are themselves riding on moving platforms.
- Stair Climbing Rb: A tweakable demo which tests the character's ability to climb stairs of various sizes.

**Character Proxy Vs Rigid Body Demo**

There demos compare the two kinds of character controller. They can be found under "Character Proxy Vs Rigid Body" in the demo "Character Control" menu (`\Physics2012\UseCase\CharacterControl\CharacterProxyVsRigidBody\`).

- Character Control Vs: This demo compares the behaviour of the character proxy and the character rigid body in a test static scene with many problematic objects from real game levels such as stairs, ladders, gaps, etc.
- Uneven Terrain Vs: This demo compares how well the characters travel over bumps in the characters. The demo can be tweaked to alter parameters common to both characters, or parameters just affecting the Character Rigid Body.

### 2.5.2 Vehicle Physics

The *Havok Vehicle Kit* is a subset of the Havok Physics 2012 SDK to help you develop simple, physically-based, ground vehicles.

Figure 2.72: Examples of vehicle games using Havok Physics 2012

The Vehicle Kit includes classes that define and implement the set of behaviors and algorithms associated with car simulation and driving. It also provides a game-oriented implementation of a vehicle and its components, such as its wheels and transmission. While the default implementation is functional and tweakable, quite often a significant amount of effort is required to tune it for a specific use case in a specific game.

This chapter is divided into:

- An introduction to the *Vehicle Kit*, which also explains how the use of "unreal" physics can produce better gameplay, describes the output of the Vehicle Kit, and offers tips on customizing it for your own vehicle game.
- A *programming guide* that provides detailed information about the Kit, its classes, and the interaction between them. You can use the programming guide as a reference when using the tutorial or when working with the Kit in your own games.

Alternatively, due to the object-oriented nature of the Vehicle Kit, you can override the default behaviors and extend the provided functionality to suit your game needs (for example, implementing a custom wheel friction model based on existing tire data).

**Note:**

Havok's vehicle implementation is intended to be used for ground vehicles and works best for simple, four-wheeled cars. While other uses are possible, the further one gets from this simple application, the more custom tuning and programming will be required. In general, Havok's vehicle kit is not suited for aircraft or watercraft, and does not work well for caterpillar tread vehicles such as tanks or half-tracks, or vehicles such as motorcycles, or other transport methods such as bicycles or animal mounts.

Most modern driving-focused games that use Havok Physics 2012, use their own (existing) vehicle simulation models while still leveraging Havok Physics 2012 collision detection and rigid body dynamics.

Before using Havok's vehicle, carefully consider how closely your use case matches what Havok provides out-of-the-box. If there is a significant difference, it may be less work to create your own implementation from scratch, than completely re-working Havok's model piece-by-piece. The Havok Vehicle Kit's use of Havok Physics 2012 rigid body dynamics and collision detection may still be useful as a reference.

A crucial detail when making a decision on any of the above approaches is how (e.g. using what tools? in what environment?), and for how long (e.g. hours or weeks per vehicle?), Game Design will tune and test vehicle behavior.

#### 2.5.2.1 The Havok Vehicle Kit

**Vehicle Physics**

**Introduction**

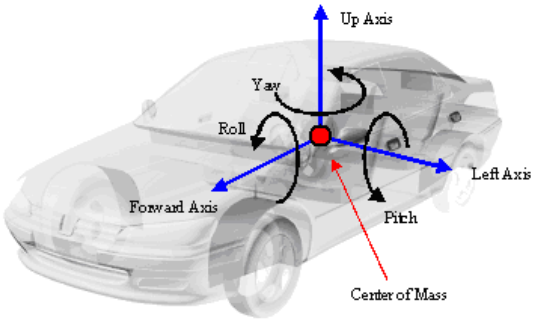
The primary focus of this chapter is to present the basics of the physics used within the Havok Vehicle Kit. It covers physical principles, Vehicle Kit parameters and some good parameter settings for various game scenarios.

Perfect vehicle simulations result in very realistic driving behavior, but in some situations you will want to employ various tricks within the simulation to make the driving experience more fun.

This document focuses on the default implementation provided with the Vehicle Kit. Keep in mind that you can always replace the default implementation of the vehicle or any vehicle components with your own custom implementation if you want to add a feature that doesn't already exist within the Kit. You can find more detailed information about the Vehicle Kit classes in the Vehicle Kit programming guide.

**Coordinate System**

A vehicle's coordinate system defines all positions and directions relative to the vehicle itself. The Kit uses the coordinate system shown in the diagram below, which defines its coordinates using an origin at the vehicle's center of mass and three axes:



You need to set the directions of these local forward, right and up axes when constructing a vehicle chassis using the Kit. By default, these are set as follows:

- **Forward axis** : local X axis.
- **Up axis** : local Y axis.
- **Right axis** : local Z axis.

**Suspension**

A vehicle's suspension is a mechanical system that attaches one or more wheels to the chassis of the vehicle. In real life there are many different types of suspension, each of which has its advantages and disadvantages. In principle, all suspension systems allow the wheel to:

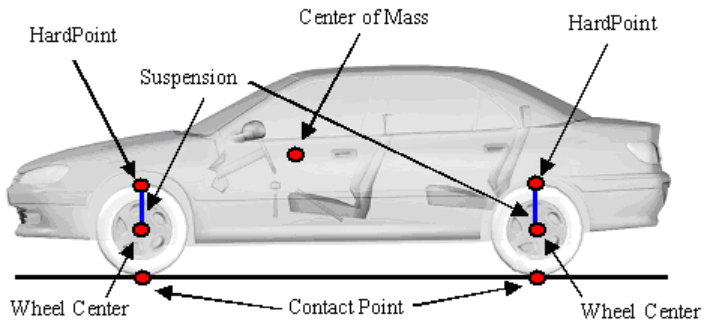
- Rotate around the wheel axis
- Move upwards and downwards within a given range. Normally this movement is performed in a straight line (due to mechanical restrictions in some suspensions, the wheel might move along a curve instead).

A spring, an anti-roll bar and a shock-absorber (sometimes known as a dashpot in Havok Physics 2012) are used to control the forces that act when the wheel moves up or down. If the spring becomes compressed, it tries to expand itself and raises the car chassis at its hard point ( see the diagram below ).

Combining a spring and the chassis of the vehicle will result a spring mass system, which tends to oscillate. In order to quickly dampen this oscillation, dashpots are added to create an opposition force to the wheels' relative movement. Note that dashpots create different forces depending on the direction of the wheel. If the ground pushes the wheel upward, a much lower force is created compared to when the wheel moves downwards. The ratio is roughly 1:3.

The Havok Vehicle Kit uses a simplified suspension model:

- It assumes that the movement of the wheel is limited to the straight line of the suspension direction.
- The suspension strength is independent of vehicle mass.
- The suspension spring connects to the chassis at a point in chassis space called the hard point and extends down through the wheel to the center of the tire (half the wheel width). This is illustrated in the following diagram.



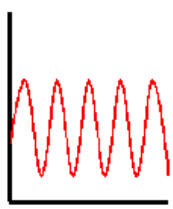
The spring and shock absorber force F depend on:

- The distance d between the wheel's center and the hard point.
- The resting length l of the suspension spring.
- The spring constant c of the suspension spring.
- The relative velocity v with which the wheel's center approaches the hard point.
- Two damping coefficients e, one if v is negative and one if v is positive.

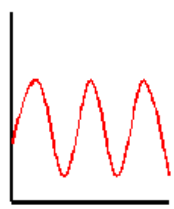
$F = (d-l) * c + v * e$

**The Suspension Spring:**

When a spring is compressed, it creates a force. The higher the strength of the spring, the stiffer the spring will be. In the Vehicle Kit, a suspension spring and the chassis form a spring mass system. The oscillation frequency of this spring mass system depends on the spring strength and the mass of the car.



High spring strength results in high frequency



Low spring strength results in low frequency

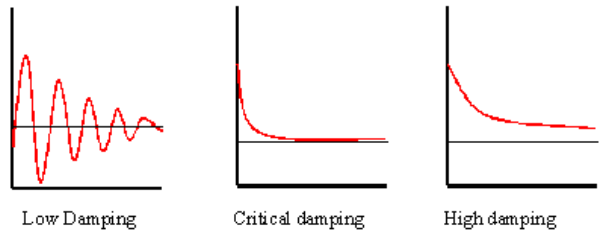
The Havok Vehicle Kit defines the spring constant c as a user-defined constant multiplied by the mass of the vehicle (c = cuser * mvehicle). Hence, as the mass of the vehicle is altered, the spring strength is altered proportionally, keeping the chassis in the same position. This means that the mass of the car can be altered without affecting the suspension length. Good values for the user spring constant, cuser, range from 5 to 50, depending on the type of vehicle you want to simulate. Some examples are given below:

- Sports car: 50
- Off road cars: 30
- Limousines: 15

Note that if the spring constant is too high, the physical simulation becomes unstable.

**The Suspension Dashpot**

Without any damping, the suspension spring and chassis mass would oscillate forever. Since damping is necessary, shock absorbers (i.e. dashpots) can be added that damp any movement in the suspension direction. The higher the damping coefficient, the quicker the dynamic energy is removed:

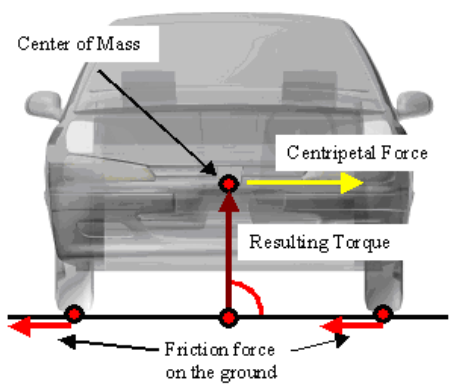



The "critical damping" value refers to the situation where the dashpot completely removes the suspension movement in less than one full oscillation cycle. To keep the chassis forces low, real cars actually have suspension damping lower than the critical damping.

The critical damping value is dependent on the suspension strength. Therefore you should try varying the suspension strength before determining appropriate damping values for your vehicles. Good values for damping range from *10%* to *50%* of the suspension strength. If the damping values are very high, the physical simulation becomes unstable.

**Friction**

When taking a corner, a car tends to maintain its current direction of travel. However, the friction between the tires and the road forces the car to follow the curve:




As the friction force is applied at the tires and not at the center of mass, the resulting torque rolls the car. There is a limit to the friction force *maxF*, which depends on:

- The type of road and tire resulting in a friction function *f*.
- The relative velocity *v* between the tire surface and the ground. This is zero if the car is not sliding.
- The angular velocity of the wheel *a*.
- The force *F* acting on the tire, pressing it onto the surface.

Hence, we get $maxF = f(v, a) * F$

For simple vehicle simulation, the vehicle Kit assumes a constant function for *f*

*f (v, a) = constant (Friction Coefficient)*

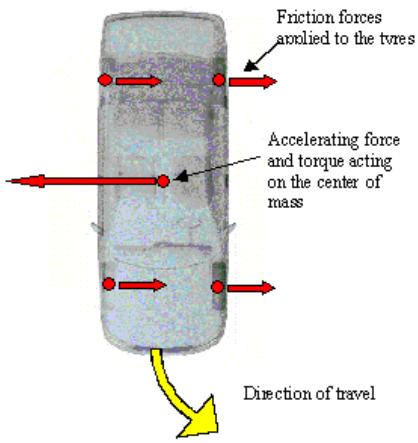
As a result, if the mass of the car changes, the resulting friction behavior of the car does not change. In real life, friction values around 0.8 are realistic.

**The Relationship Between Front and Back Friction**

If you double the friction values for all tires, the car can take corners faster without sliding, however the overall driving behavior will still be quite similar provided the car does not flip.

However if you increase the front tire friction coefficient by only 20 percent without changing the friction values for the other tires, the difference is dramatic. Let's look at an example.

Imagine a car taking a left turn:




As shown in the above diagram, the friction forces applied to the tires accelerate the car to the right. The steering angle of the front tires defines the turning radius of the car. If this turning radius results in friction forces exceeding the maximum allowed friction forces, the car loses its grip on the road and slides.

If the front tires lose grip first, the car "understeers", sliding on its front tires. It will not follow the curve defined by the steering angle. This usually means that the front friction coefficient is less than that of the back tires.

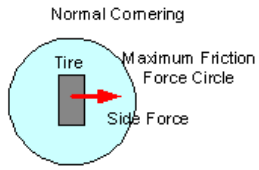
If the back tires slide first, the car "oversteers", sliding on its back tires. This means that the car may spin uncontrollably in curves.

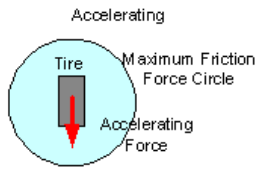
If all four tires slide at the same time, the car tends to keep its current direction while sliding, making the car more difficult to control but fun to drive.

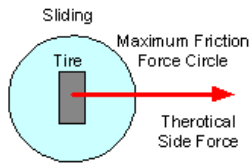
As you can see, the relationship between the front and back friction coefficients is mainly responsible for these effects. In a physical simulation, the brake and acceleration balance also influences this behavior. You will find out more about this later on in this manual.

**The Friction Circle**

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 



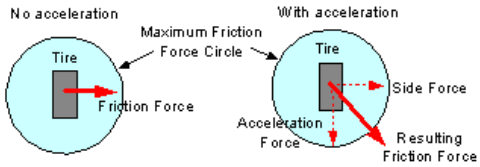




In reality, a tire's friction circle is not 100% circular, but slightly deformed. However, in most cases it is sufficient to assume a perfect circle.

What does this circle mean?

Imagine a car taking a sharp turn without braking or accelerating. Assume the car is not sliding yet. If the driver accelerates, an extra force component adds to the current force, such that the resulting force exceeds the maximum friction force. The car starts to slide:




As a result, any forward or backward torque on the wheel effectively reduces the maximum side force, making it easier for the wheel to slide. A good example of this is a tire burnout or braking with blocked wheels.

Knowing this principle, the distribution of the brake torque and acceleration torque to the individual wheels is very important. This is also known as the brake or acceleration balance.

If you create a car with a very strong engine and use normal friction values for the wheels, you will experience:

- A normal acceleration
- A very slippery car

The reason for this is that the wheels can simply continue spinning without gaining grip.

**The Brake and Acceleration Torque Distribution**

The brake and acceleration torque distribution describes how the car's brake and acceleration torque is distributed to the 4 wheels, and has a significant effect on driving behavior.

As you apply more forward torque to a wheel, its maximum allowed side force decreases. In other words, applying torque to a tire effectively reduces the friction coefficient. The wheel thus becomes more prone to slip.

In practice, this means that rear-wheel drive cars tend to oversteer in curves, whereas front wheel driven cars tend to understeer.

**The Influence of the Center of Mass and the Rotation Inertia**

In vehicle dynamics, the mass of the chassis plays a minor role. The mass tends to have more of a scaling effect than an actual influence on the entire vehicle's behavior. However, the position of the center of mass inertia tensor has big effects.

The inertia tensor is a 3x3 matrix defining how difficult it is to rotate a body around the principal axis. The inertia tensor is dependent on the mass itself and on the mass distribution. For example, it is relatively easy to spin a one-kilogram iron ball compared to a one-kilogram bicycle wheel.

The inertia is roughly equal to the mass of the object multiplied by the average distance of all points in the object's volume to the center of mass. Hence, the inertia of a car with dimensions 4m X 2m X 4m for the principle axis is:

- Up axis: about 2 meters * 1500 kg
- Forward axis: about 1 meters * 1500 kg
- Right axis: about 2 meters * 1500 kg

A higher center of mass means that the car rolls more heavily in curves. It also means an increase in a braking car's rotation around the pitch axis.

Higher rotation inertia makes it more difficult to change the angular velocity of the car. It also reduces the frequency of rotational oscillations about the center of mass. A car oscillating around the roll axis at 2 Hertz only oscillates at a rate of 1 Hertz if its inertia is doubled. Hence, the inertia can be used to reduce the likelihood of a car spinning out of control when colliding or sliding.

**Steering**

Steering translates the direction of the steering wheel into a rotation about the yaw axis. In real life there is a simple linear relationship between the steering wheel of the car and the steering angle of the tires. However if you tried to steer a real formula-1 car at 200 mph with a game pad, you would not stay on the road for long!

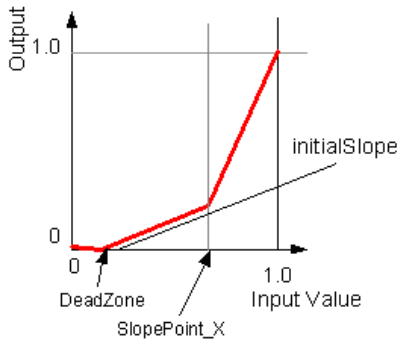
Hence, in a game environment you have to preprocess the raw input data of the input components using several techniques. The Havok Vehicle Kit does this preprocessing in two steps: first, a *DriverInputComponent* is used, followed by a *SteeringComponent*. You can find more detailed information about these components and their default implementations in the Vehicle Kit programming guide.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

### Dual Slope (DriverInputComponent)

A dual slope input controller uses a very simple relationship between input value and output. For input values less than the controller's "dead zone", it returns zero. For input values x greater than the dead zone *dz* and less than *SlopePointX*, it returns *(x – dz) * initialSlope*

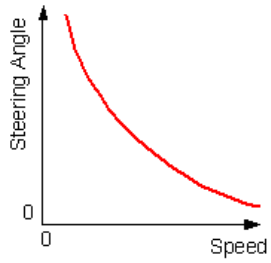
For input values greater than *SlopePointX*, the value is interpolated in a way such that it reaches a value of 1 for an input value of 1.




### Speed-dependent Steering Angle (SteeringComponent)

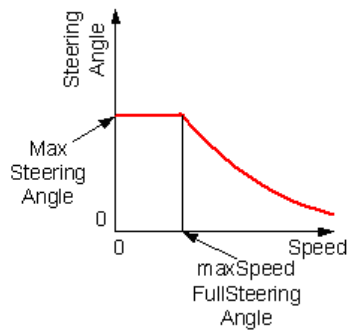
As the accuracy of a joystick is much smaller than that of a real steering wheel, the Vehicle Kit can change the accuracy of the input depending on the current velocity.

The basic idea is that the higher the speed of the car, the less sensitive the steering should be. Given a certain friction coefficient and vehicle speed, a theoretical steering angle can be determined where the car is on the verge of sliding:




The steering angle as a function of vehicle speed, where the vehicle is on the verge of sliding.

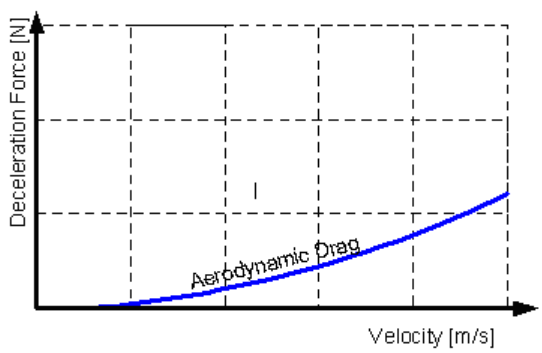
For practical reasons the maximum steering angle is clipped:




To tune the car, decide a maximum steering angle for slow speeds and then play with the *maxSpeedFull-SteeringAngle* parameter to get the desired high-speed behavior.

**Aerodynamics**

As air has a certain density, it influences a moving car. These influences are called aerodynamic drag and lift. Aerodynamic drag creates a force opposite to the car's direction of travel. Aerodynamic lift creates an upward (or when negative a downward) force. Drag is the main effect that limits the top speed of the car.



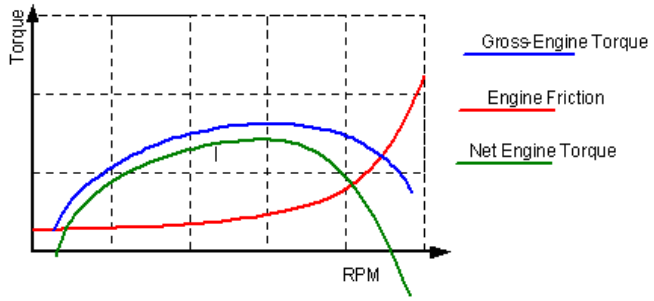

The formula used in the default vehicle implementation is:

$F_{drag}$ = Â¡ * airDensity * dragCoefficient * frontalArea * velocity$^2$

$F_{lift}$ = Â¡ * airDensity * liftCoefficient * frontalArea * forward_velocity$^2$

**Engine**

The engine generates all the energy, or torque, that drives the car. The most common engines currently used for vehicles are combustion engines. The basic principle of this type of engine is that it burns petrol in a cylinder, thereby creating a gross-engine torque. Part of this gross-torque is actually eliminated by the engine friction. The engine torque and friction are dependent on the engine's revolutions per minute. This relationship is illustrated in the following diagram.




The engine's netTorque can be calculated as (grossTorque – friction).

As a result, the performance of an engine is only optimal for a very small rpm range.

**Havok Vehicle Kit**

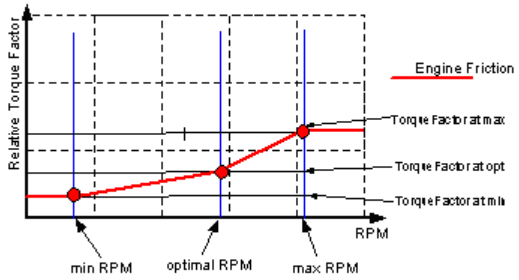
Havok's default implementation uses simplified curves. You only have to specify three points on the *RPM/Torque* diagram.

- *Min RPM/Torque:* If the engine runs slower than minRPM, the Kit assumes that the clutch kicks in such that the engine's RPM never falls below minRPM.
- *Optimum RPM/Torque:* The RPM value where the engine produces its maximum net torque.
- *Max RPM/Torque:* Defines the maximum RPM for the engine. If the engine exceeds this value, the engine shuts off.

All of these values are just a factor to the engine's net torque at optimum RPM. If the engine produces 700 [NM] net torque at 4000 RPM and the engine friction at *optRPM* is 0.5, the engine's net torque is 350.

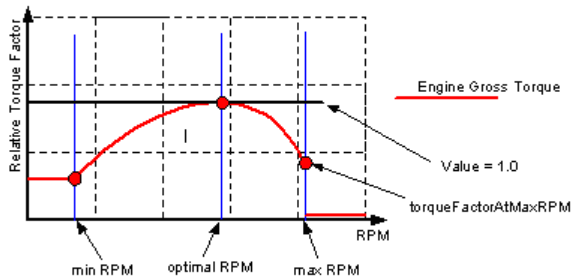
**Engine Resistance**

The engine resistance is linearly interpolated using two line segments, as shown in the diagram below.

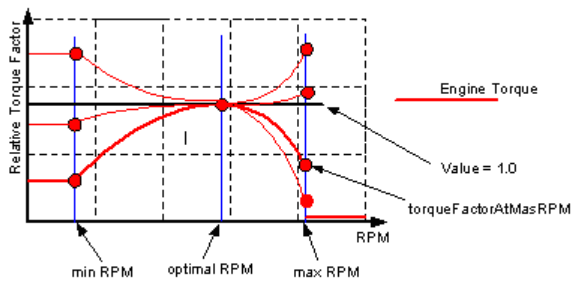



**Engine Net Torque**

The engine net torque is interpolated using two parabolic curves. The engine net torque at the optimal RPM is defined as the *torque* factor. The values at the minimum and maximum RPM are just relative factors, which the Havok default engine multiplies by the *torque* parameter during simulation.




The default engine uses piecewise parabolic curves to interpolate between three points. So possible engine curves may look like the following diagram:

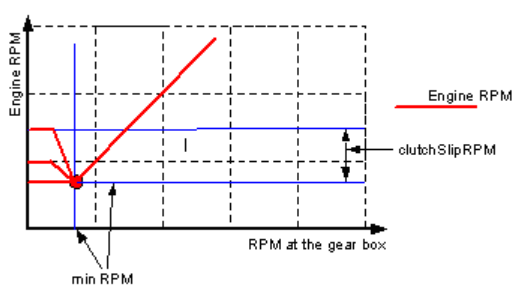



**Engine Clutch**

A vehicle's engine cannot work properly below a certain RPM. To get it working above this lower limit, the driver of the car must press the clutch to allow for some slip. As a result the engine RPM is actually higher than the RPM at the transmission shaft. To simulate this behavior, Havok introduces a clutch slip for RPMs less than the minimum RPM.

For gearbox RPM less than *(0.5 * minRPM)*, engine RPM is calculated using the following formula:

*EngineRPM = minRPM + clutchSlipRPM * throttleInput*




*ThrottleInput* is a value between 0 and 1 describing how strong the driver pushes the accelerator pedal.

**Transmission**

To keep the engine working within optimal RPM ranges for different vehicle speeds, you use a variable transmission. The transmission is a device that uses gearing or torque conversion to change the ratio

between engine revolutions and wheel revolutions. If the engine revolutions are increased but the wheel revolutions stay the same (making the ratio bigger), the car produces more torque. On the other hand, if the engine revolutions are decreased and the wheel revolutions stay the same (making the ratio smaller), the car achieves higher speeds.

A typical transmission includes a clutch, a gearbox and a differential. The clutch disconnects the engine from the gearbox while you change gear. The gearbox has several gears to translate the torque from the engine to the drive shaft. The differential translates and distributes the torque from the drive shaft to the left and right wheel.

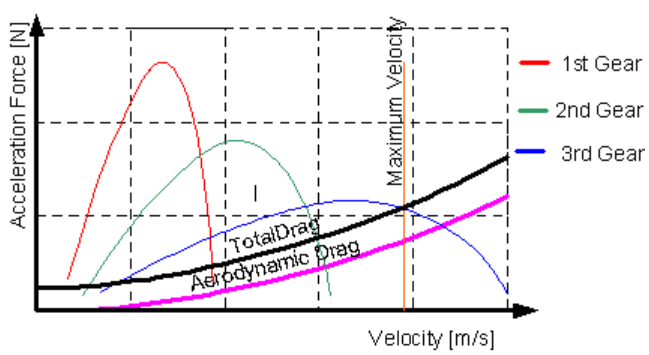
The following formulas describe the relationship between wheels and engine:

*EngineRPM = WheelRPM / (primaryTransmissionRatio * GearRatio[currentGear])*

*WheelTorque = EngineTorque * primaryTransmissionRatio * GearRatio[currentGear]*

- *EngineRPM* is the Revolutions Per Minute of the engine.
- *WheelRPM* is the average Revolutions Per Minute of the wheels.
- *PrimaryTransmissionRatio* is the transmission ratio at the differential.

When comparing the final wheel torque (which results in an acceleration force) to the total drag of the car, you can calculate the maximum velocity of the car.




For practical reasons the ratio between two consecutive gears should be constant:

$$\frac{gear0_Ratio}{gear1_Ratio} = \frac{gear1_Ratio}{gear2_Ratio} = \frac{gear2_Ratio}{gear3_Ratio}$$

The default Havok Vehicle Kit implementation provides an automatic transmission, which includes the following parameters:

- The gear ratio of all gears.
- The primary transmission ratio.

- The min and max engine RPM when gear shifting should be performed.
- The time it takes to shift a gear (ClutchDelayTime).
- WheelsTorqueRatio, which defines how the engine torque should be distributed to the wheels. For example, if you want to create a front-wheel driven car, the two front wheels should have a value of 0.5 and the rear wheels a value of 0.0. The sum of all wheel torque ratios should be 1.0.

**Tuning tips**

A good way to start tuning the gear ratios is to set the gear ratio for the top gear to 1.0 and the primaryTransmissionRatio using the following formulas:

*TopSpeedCar[m/s] = TopSpeedOfCar[mph] * 1.605 / 3.6*

*MaxWheelAngularVel = TopSpeedCar[m/s] / wheelRadius*

*MaxWheelRPM = MaxWheelAngularVel * 60 / 2 * PI*

*PrimirayTransmissionRatio = maxEngineRPM / MaxWheelRPM*

The rest of the gears can be set using a gear-to-gear ratio of 1.5, e.g.

$$\frac{gear0_Ratio}{gear1_Ratio} = \frac{gear1_Ratio}{gear2_Ratio} = \frac{gear2_Ratio}{gear3_Ratio} = 1.5$$

**Brakes**

Brakes slow down a vehicle using a mechanism that converts kinetic energy into heat energy through friction. The most common example of this is found in the wheels of cars, where the brake shoes or disc pads press against the brake drum or brake disc, thereby creating a braking torque on the wheels.

The Havok Vehicle Kit multiplies the input value from the brake pedal by the maximum brake torque. The resulting torque is then distributed to the wheels.

Applying brake torque to a wheel actually means reducing the maximum side force the wheel can resist during cornering. The driver of the car can often make use of this behavior to get the car into a sliding mode. The easiest way to achieve this is to use the handbrake, which completely blocks the wheels.

**At the Limits or How to Cheat Physics**

**Introduction**

Physically accurate driving behavior is challenging and can be fun, especially if your game is focused on realism. However, if the focus of the game is to be fun while easy to learn, you probably need to use several tricks to help or cheat the driving simulation. This chapter covers several tricks and algorithms used in the Havok Vehicle Kit and demos.

**Subjective Time and Real-Time**

In driving game, players often do not expect the game to run in real-time - they expect it to run faster

than that! They want to drive around at 200 miles per hour but also to be able to take sharp turns around city corners. They expect the car to react faster to input commands than a real car would.

However, in certain situations, a real car behaves too fast for the player to react. In these situations the player wants a slow motion behavior, especially if the car is sliding or crashing.

This creates a dilemma, because you need to simulate the car faster and slower than real time, at the same time. Fortunately, the Havok Vehicle Kit allows you to "cheat" its physics simulation in order to deal with this. You can achieve desired behaviors relatively easily using a few parameters.

The following cheats allow you to increase the car's responsiveness:

- Increase gravity or add an extra gravity to the car
- Increase the friction coefficient of the tires up to 3.0
- Make the suspension harder
- Add extra yaw torque and extra yaw angular impulse

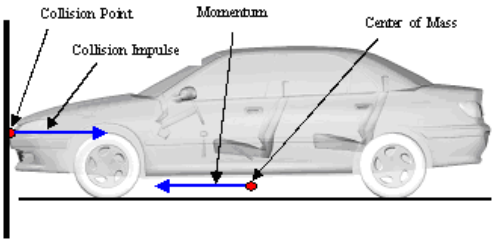
These cheats allow you to decrease the car's responsiveness:

- Lower the center of mass or virtually lower the center of mass by reducing all friction torque factors.
- Increase inertia
- Damp angular velocities
- Apply extra forces and torques (e.g. motorcycle controller)

**Center of Mass**

In general, a low center of mass results in improved car handling. Lowering the center of mass results in reduced roll and pitch torques during cornering, braking and accelerating. The car "sticks" to the ground better. It also reduces the likelihood of a car flipping over.

However, there are some limits to how low you can shift the center of mass and still have a realistic simulation. If the car crashes, a low center of mass may make the front of the car jump upward. The reason for this is that the point of collision is above the center of mass, thereby creating an angular impulse. If the center of mass is moved below the ground, the roll and pitch torques are reversed, resulting in a very unrealistic behaving simulation.



There are tricks that allow you to lower the center of mass without getting these negative effects. If the angular impulse at a collision is a problem, increasing the inertia by a factor of 2 for the pitch axis reduces this effect. Alternatively, rather than lowering the center of mass of the rigid body, you can effectively lower the center of mass by setting the *torqueRollFactor* and *torquePitchFactor* to values less than 1.0.

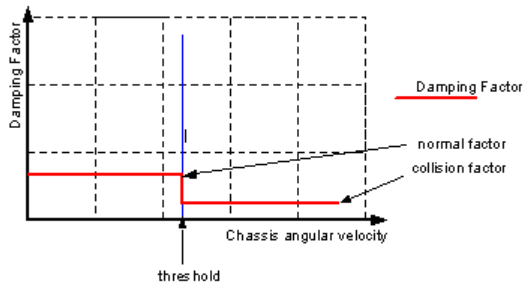
**Inertia**

If a very fast car hits a solid object, it might start spinning very violently, which would be frustrating for a player. However, there are a few tricks that you can use to decrease the likelihood of high angular velocities. One way is to simply increase the inertia. As there are three values, one for each axis, you can tune the different axes independently. Changing the inertia a little (20%) changes the entire car behavior. Do not try to change the inertia dramatically.

**Angular Velocity Damping**

If a car takes a big jump, its angular velocity does not change while it is in the air. This may result in the car flipping over when it hits the ground. This accurate physical behavior might be frustrating for the player. Hence, a mechanism to damp the angular velocity is provided.

The Havok Vehicle Kit provides a controller that damps angular velocity. However, you might not want to damp the angular velocity for everything, since you want to see a spinning car after certain collisions. Havok measures the current angular velocity and if it is below a certain threshold it damps it using the normal angular velocity damping. If it is above the threshold, Havok uses the collision angular velocity damping:




The range for the normalSpinDamping should be between 0 and 3.0, and values between 0.5 and 1.5 should give reasonable results. A spin threshold of ĬĂ means that if the car spins at less than 180 degrees per second, Havok uses *normalSpinDamping*.

If your car uses realistic collisions, then *collisionSpinDamping* values between 0 and 0.5 are reasonable. However if your gameplay results in lots of collisions, then use high values between 3.0 and 10.0. This should remove any high angular velocities. This technique can help prevent the player losing orientation.

**Friction Equalizer**

The tires of a stopped car on a flat surface press against the ground with a certain force, called static load distribution. However, if the car is moving, these forces change due to various effects. This is called dynamic load distribution. The *frictionEqualizer* parameter allows you to change dynamic load distribution to be more equal to the static load distribution.

The effect is that the driving behavior of the car does not change when the car accelerates, brakes, goes over hilly terrain, etc.

Setting this parameter to values between 0 and 1.0 virtually averages all the suspension forces. For example, let's say the value is set to 0.8, and the back tires are not touching the ground while the front tires are. The front tires are pressed onto the ground at 10000N of force. In this case, the vehicle Kit redistributes this 10000N to the individual tires using the following equations:

Front tires: *5000N* (1-0.8) + 5000N*2/4*0.8 = 3000N*

Back tires: *5000N*2/4*0.8 = 2000N*

Hence, if you set *frictionEqualizer* to 0.0 you get physically correct behavior. If you set this parameter to higher values (up to 1.0) you improve the driving behavior on rough and off-road terrain.

**Extra Angular Torque**

When a car starts to slide, it becomes very difficult to control. Even professional racing drivers sometimes have problems stopping their car from spinning. Applying some non-physical impulses and torques around the yaw axis can help to control the car in sliding conditions.

The *extraTorqueFactor* uses the current steering angle to create a torque around the yaw axis e.g. steering to the left creates a small constant torque to the left. Good values range from 0 to 2.

Due to coordinate system differences, this factor may need to be multiplied by –1.0. Check the behavior. If it is wrong, simply negate the value.

**Extra Gravity**

One of the most common and best tricks for improving the overall game driving experience is to add an extra gravity to the car. This has the basic effect of accelerating the subjective time. Good values are up to three times the normal gravity.

**Extra Constraints**

In many game scenarios, you need some extra forces to get the desired effects. A typical example is a motorbike game where a small helper function keeps the bike upright. There are endless ways to change the physical behavior of the car, just use your imagination.

**Vehicle Kit Simulation Output**

**Introduction**

When using a simulated vehicle in your game, you will obviously need to display the vehicle on-screen – you may also want to make the player's driving experience even more realistic by adding force feedback, sound, and tyremarks. The Vehicle Kit simulation provides you with the information you need to do this. You can find more detailed information about how to use the relevant classes in the Vehicle Kit programming guide.

**Graphics**

To draw the vehicle, you just need the local to world transforms of the chassis and the wheels. As the chassis is already a rigid body in the physics simulation, you can simply get the transform from this rigid body. Because the current Vehicle Kit does not require the wheels to be simulated as rigid bodies, you can get the wheel transforms from the *WheelInfo* struct in `hkpVehicleInstance`.

### Visual Debugger

The Havok Visual Debugger can be used to draw some debug information for any objects of type `hkpVehicleInstance` or any classes derived from `hkpVehicleInstance` in the simulation.

For each vehicle the following components are displayed:

- Each **wheel** of the vehicle is drawn as a white (or brown, if it is not colliding) circle.
- A red line shows the **ray** that was **cast** for each wheel.
- A yellow-green line is drawn for **each axle** of the vehicle, connecting the center of all wheels attached to that axle. If there are more than 2 wheels connected to an axle, this line will zag-zag between all of these wheels.
- The **suspension hardpoints** are drawn as a yellow cross at the top of each wheel.
- The **contact point** between the wheel ray and the ground is shown as a yellow cross at the bottom of each wheel. If the wheel is not in contact with the ground, then it will be drawn in brown, to keep indicating the current length of the suspension.

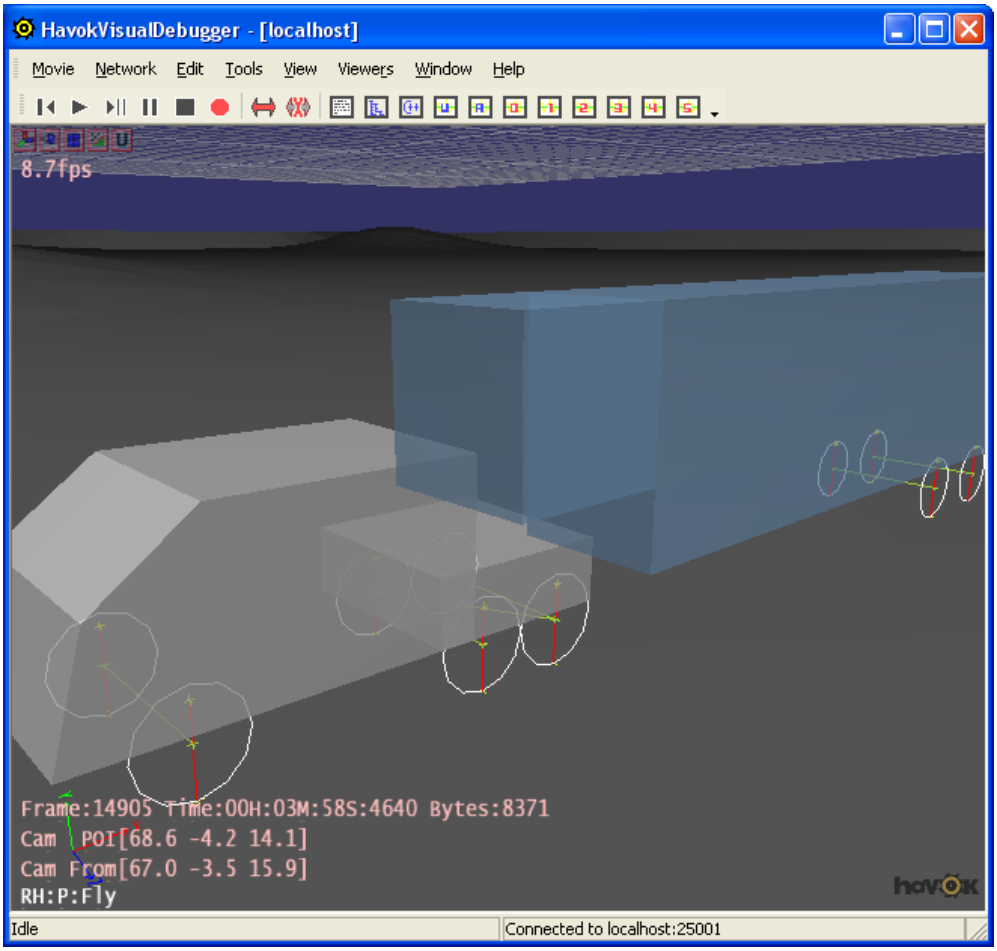



Figure 2.73: Visual Debugger Vehicle Viewer

The position of the information displayed by the vehicle viewer (wheels, raycasts) lags behind the position of the chassis by one frame. This is because the vehicle simulation is performed as an action, which is evaluated before the Havok world is simulated, and the resulting forces are applied to the chassis rigid body as input for the world step. The Visual Debugger displays the state of the world after each Havok step. The vehicle viewer has been designed to show the input for the vehicle that caused it to reach the current location, to aid debugging.

**Force Feedback**

The best approach when using force feedback is to take the values from the simulation and pass it to the steering wheel. Note that in reality the force feedback strength is a combination of different factors:

- *Engine rumble* The rpm of the engine can be translated in a sine wave that can influence the steering force. This value is stored in hkpVehicleInstance::m_rpm.
- *Wheel rotation rumble* Instead of using the engine, you can use the angular velocity of the wheels to define the frequency of the steering force (retrieve from the `WheelInfo::m_spinVelocity` member.)
- The *centripetal (side) force* that acts on the steered wheels: This can be retrieved using `WheelInfo::m_sideForce`.
- Collisions may generate an impulse on the steering wheel.

Different force feedback devices have very different characteristics and you should test all major steering wheels and joysticks. You may need different settings for different kinds of steering wheels.

**Tyremarks**

If a car tire slides, the wheel produces rubber marks on the road. Drawing these tyremarks is important for the player's immersion in the driving game. The darkness of tyremarks is directly proportional to the energy converted to heat divided by the area of the tyremark.

You can retrieve this parameter from the simulation from the `WheelInfo::m_skidEnergyDensity` member. The resulting value should be scaled to achieve the desired effect.

**Sound**

Sound is an important part of the feedback the user gets from driving a vehicle. A typical vehicle sound is a simple combination of the different vehicle sound sources:

- Engine sound based on engine RPM
- Skid sound based on `WheelInfo::m_skidEnergyDensity`.
- Wheel sound based on terrain and the angular velocity of the wheels
- Air sound based on the velocity of the chassis

Sound can be important for tuning. It often makes sense to implement the sound model before you start tuning.

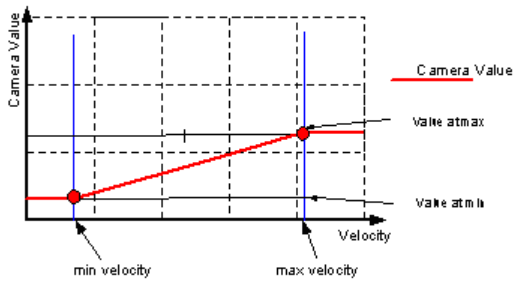
**Camera**

The player sees the whole scenery through the eyes of the camera. One goal for a car game camera is to be unobtrusive. The camera should just feel natural. A bad camera can easily destroy the gameplay without it being evident that the camera is the main problem.

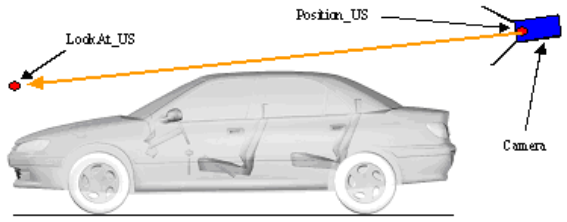
In principle there are several ways to implement a camera. A damped mass-spring system technique is used in many past Havok demos. However, it is not good enough for game cars because:

- It results in a very shaky camera
- It is difficult to control
- The distance between the car and the camera changes.

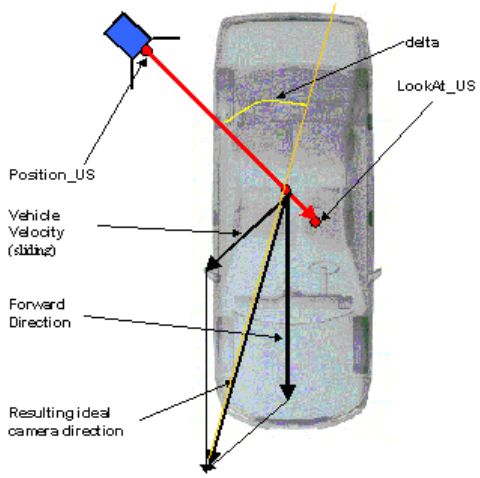
The default Havok Vehicle Kit camera, hkp1dAngularFollowCam, uses a different approach. The damped mass spring technique only applies to calculating the direction of the camera. All other values are fixed. The camera allows for changing its behavior for different car speeds. In order to get the value of one parameter, the camera implementation interpolates between two velocity values:




The default vehicle camera is positioned at a constant distance from the center of mass of the chassis rigid body, and looks at a fixed point, also relative to this center of mass. The camera always looks from its position to the lookAt position. Hence you need to specify these two positions relative to the center of mass of the chassis rigid body, using the simulation's world co-ordinate system (world space):




The ideal camera direction is a combination of the direction of the vehicle and the direction of its velocity. The *m_speedInfluenceOnCameraDirection* value weighs the influence of the vehicle's velocity. A value of 0 means only the vehicle direction counts, a value of 1 results in roughly equal weighting. Higher values base the ideal direction wholly on the vehicle velocity.




The *current camera yaw angle* tries to match the ideal camera direction. The *m_angularRelaxation* parameter defines how quickly the camera follows this direction. Values between 2 and 6 are reasonable. The higher the value, the easier it is to keep the car in view.

Based on the current camera yaw angle, the two camera positions, the position and the lookAt position, rotate around the world up-axis to get the final camera position and camera lookAt position.

**Pre Tuning Checklist**

Before tuning a vehicle, it is highly recommended to check the following items.

- The size of the geometry should reflect real world sizes. If you use meters, Newtons and kilograms as your base units, it should make the job easier.
- Get a very simple car moving. Use your final game graphics and tracks if possible.
- Use the gamepad, joystick or steering wheel to tune the car. Do not use the keyboard for initial tuning.
- Make sure you get all possible feedback from your car:
  - Tyremarks
  - Motor sound and skid sounds
  - Shadows
  - Optional: Force feedback
- Set up a decent camera close to the car. Use popular driving games as a guide.
- Set the center of mass first as it influences all other parameters. Changing the center of mass is more difficult late in development than at the beginning.
  - The center of mass should be inside the chassis, never below the ground.
  - Collisions should feel realistic, banking and diving can be tuned without changing the center of mass.

  One interesting change to try is to lower the center of mass of the chassis rigid body. This has a big affect on the drivability of the car, as it is less likely to roll over on sharp corners with a lower center

of mass. You can do this easily by calling the hkpRigidBody::setCenterOfMassLocal() method. Be careful, however, since the suspension hardpoints are specified relative to the center of mass of the chassis rigid body, you will have to move these in the opposite direction to your center of mass shift to put them back into the correct position after the shift has been applied.

**Tuning**

Tuning vehicles can be time consuming. In principle it means repeatedly playing the game thoroughly and using all the parameters, thereby learning and improving their influence.

This tuning cycle should include:

- Normal driving situations
- Collisions
- Sliding
- Jumps

#### 2.5.2.2 Vehicle Programming Guide

This section provides an overview of the core classes that define and implement the set of behaviors and algorithms associated with the Havok Vehicle Kit.

**Overview**

The Vehicle Kit separates data that changes at runtime from data that is the unchanging specification of a vehicle. This makes it possible to share vehicle definitions between multiple vehicle instances.

A vehicle can be considered to be composed of several components. Default implementations of all of these components are provided by Havok that can be overridden to satisfy your needs.

- **hkpVehicleInstance** : The class that holds the vehicle together. This contains pointers to all other components. It provides access to all output of the vehicle, as described in Vehicle Kit Simulation Output section. This class stores all cached and runtime data for the vehicle, and so cannot be shared between vehicles.
- **hkpVehicleData** : A container for vehicle specs such as:
  - The number of wheels
  - Radius, mass, friction, etc., for each wheel
  - Chassis orientation and the roll factors that affect the chassis
  - Description of the vehicle for friction solving
  - Any unchanging data that is not particular to any other component
- **hkpVehicleWheelCollide** : Handles the collision detection between the wheels and the ground. This component cannot be shared between vehicle instances. This component allows you to override the friction calculation in order to implement friction maps based on collision information input (shape keys etc.). See `hkpVehicleWheelCollide::wheelCollideCallback()`.

Havok provides two wheel collide implementations. The default implementation, `hkpVehicleRayCastWheelCollide`, casts a ray against collidables near the wheel. (See Wheel collision detection using ray casting.) The other implementation, `hkpVehicleLinearCastWheelCollide`, provides more accurate collision data by casting wheel shapes against the collidables near the wheels. (See Wheel collision detection using linear casting).

- The components `hkpVehicleAerodynamics`, `hkpVehicleBrake`, `hkpVehicleDriverInput`, `hkpVehicleEngine`, `hkpVehicleSteering`, `hkpVehicleSuspension`, `hkpVehicleTransmission` and `hkpVehicleVelocityDamper` all have a similar structure. They contain only data that is set during vehicle setup. Each has a method that is called every step to calculate the effect the component will have on the vehicle. This calc function has well defined inputs and outputs – the output structure is given in the base class (e.g. `AerodynamicsDragOutput` in `hkpVehicleAerodynamics`). All these components can be shared.
- Additionally, we provide an `hkpTyremarksInfo` class to demonstrate how to handle tyremarks left by the vehicle, and an `hkp1dAngularFollowCam` as a sample vehicle camera.

Figure 2.74: Components of a hkVehicle

Note that the shareable components contain only public data members (that should be set by the user to specify the vehicle), while the non-shareable components, in addition to such data, also contain state information and data specific to each vehicle (such as the chassis rigid body).

**Integrating Havok Vehicles into your game**

### Setting up a vehicle

You can use vehicles in their serialized form to load a vehicle into your world. Have a look at the `SerializedVehicleDemo` or `SerializedManagerDemo`.

The steps for setting up a vehicle are as follows:

1. Create the chassis (an `hkpRigidBody`).
2. Create the vehicle instance (an `hkpVehicleInstance`).
3. Set the vehicle's components (e.g. assign an instance of `hkpVehicleEngine` to the vehicle's `m_engine` member).
4. Call `hkpVehicleInstance::init()`.
5. Add the vehicle's chassis and phantoms to the world using `hkpVehicleInstance::addToWorld()`.

You must also ensure the vehicle is simulated during the game step by doing one of the following:

- Adding the vehicle instance to the world using `hkpWorld::addAction()` (see Simulating a vehicle using an action);
- Explcitly calling `hkpVehicleInstance::stepVehicle()`; or
- Adding the vehicle instance to a vehicle manager using `hkpVehicleManager::addVehicle()` and calling a method which causes the manager to simulate its vehicles. (See Simulating a vehicle using a vehicle manager).

Naturally, it will also be necessary to perform other vehicle related tasks during the game step, such as getting driver input and updating tire marks.

### Simulating a vehicle using an action

The simplest way to simulate a vehicle is to take advantage of the fact that the `hkpVehicleInstance` class is a subclass of `hkpUnaryAction`. By adding the vehicle to the world using `hkpWorld::addAction()`, the vehicle will be automatically updated and simulated at the beginning of each physics step.

Explicitly calling `hkpVehicleInstance::stepVehicle()` during your game update, is essentially the same as using an action, but may offer more control.

### Simulating vehicles using a vehicle manager

One of the most computationally expensive parts of vehicle simulation is performing collision detection for the wheels. Fortunately, this operation does not modify the world, so multiple threads can safely perform wheel collision detection in parallel. We offer an approach to simulating vehicles which takes advantage of this.

The benefit of multithreading wheel collision detection is greatest when it can be done for many wheels. Thus, our approach works on sets of vehicles so the collision detection for all their wheels can performed together.

An `hkpVehicleManager` manages a set of vehicles. The base `hkpVehicleManager` class offers a simple interface to add and remove vehicles, and to simulate them sequentially. In practise, you use a subclass of vehicle manager specialized for the kind of wheel collide component used by your vehicles:

- `hkpVehicleRayCastBatchingManager`s manage vehicles whose wheel collide components are instances of `hkpVehicleRayCastWheelCollide`.
- `hkpVehicleLinearCastBatchingManager`s manage vehicles whose wheel collide components are instances of `hkpVehicleLinearCastWheelCollide`.

A method or methods must be explicitly called by the game update to simulate vehicles with a vehicle manager. There are several options:

- Calling `hkpVehicleManager::stepVehicles()` will simulate the vehicles singlethreaded.
- Calling `hkpVehicleCastBatchingManager::stepVehiclesSynchronously()` will multithread the collision detection of the wheels. The calling thread will try to do some of the work and will return when the collision detection is finished.
- Using the interface for asynchronous multithreading offered by `hkpVehicleCastBatchingManager`, the calling thread can do useful work while other threads do the wheel collision detection.

Which of the methods is best is situation dependent. As implied above, multithreading the wheel collision detection confers a benefit when there are many wheels. There will be a threshold below which it is faster to use the singlethread implementation. Even then, there may be a benefit to using the asynchronous interface. Havok's Visual Debugger may be useful when profiling performance for these different options.

**Additional vehicle behaviour**

There are other times you may want to modify the behaviour of the vehicle. For example:

- damp the movement of a vehicle in certain directions to increase stability
- ensure that an overturned off road player-vehicle doesn't stay upside down for too long

The Havok Physics 2012 SDK is flexible enough to allow you to do this and some of the behaviours are integrated in the Vehicle Kit. For example, angular velocity damping is provided by the `hkpVehicleDefaultVelocityDamper` and there is a `hkpReorientAction`.

**Wheel collision detection**

**Why not use rigid bodies?**

Two different approaches to wheel collision detection are provided. Neither of them use rigid bodies to simulate the wheels. Some of the reasons for this are as follows:

- In order to keep the wheels attached to the chassis, complex forces and constraints need to be applied between them during simulation, with the consequent expense of CPU. Rigid bodies in a simulation have a high amount of degrees of freedom, while wheels in a vehicle have not.

- Real vehicle wheels are not actually rigid bodies. They deform during driving, which is the reason why cars stick to the ground. Simulating wheels as rigid bodies doesn't provide good traction due to the fact that the interaction between a road surface and a tire is very different to that between two rigid bodies.
- Wheels spend most of their time on the ground (except when the vehicle is crashing or jumping). Thus, collision detection and resolution between wheels and ground can be greatly simplified, again reducing the CPU cost.

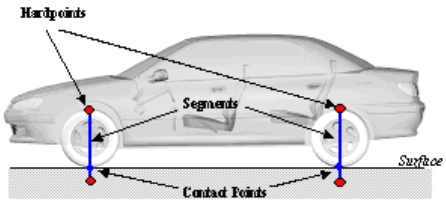
The approach taken is to assume that the wheels will always be attached to the chassis, and that their position can be calculated at any time as a function of the current position of the chassis and the ground. This approach frees the vehicle simulation from the task of keeping the wheels attached to the chassis, with consequent improvements in speed and stability.

**Wheel collision detection using ray casting**

This approach calculates the position of a wheel by casting a ray from the chassis at the point where that wheel's suspension is connected – also known as the *hardpoint* - in the direction of the suspension. The length of this ray is the maximum distance the wheels can be from the chassis.

If the ray hits a collidable, the wheel is considered to be in contact with the ground. The position of the wheel can be calculated from the intersection point (by subtracting the radius).

If the ray doesn't hit, the wheel is considered not to be in contact, and its position is set to the end point of the ray (again, by subtracting the radius).




To limit the number of collidables against which the rays must be cast, an `hkpAabbPhantom` is used to cache the set of collidables near the wheels. The phantom is kept just big enough to contain the rays for all the wheels.

To create a vehicle which uses ray casting for wheel collision detection:

- During vehicle set up, prior to calling `hkpVehicleInstance::init()`, the `m_wheelCollide` member should be set to a new instance of `hkpVehicleRayCastWheelCollide` (new because wheel collide components cannot be shared).

**Wheel collision detection using linear casting**

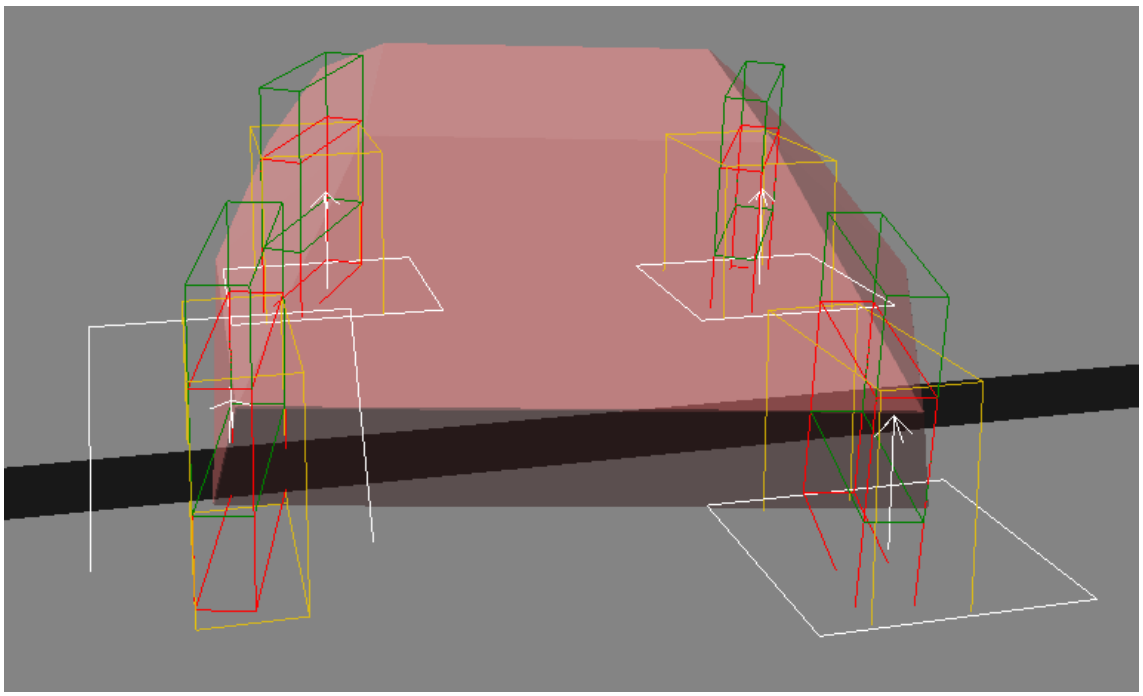
This approach calculates the position of a wheel by casting an `hkpShape` (by default, an `hkpCylinderShape` of appropriate size and orientation) along the suspension. The first hit defines the position of the wheel.

Using linear casting is more expensive than ray casting, but gives more accurate results. For example, cars mount curbs more realistically when their wheel collision detection is done by linear casting.

To limit the number of collidables against which the shape must be cast, for each wheel an `hkpAabbPhantom` is used to cache the set of nearby collidables. Each phantom is defined to be just big enough to contain the wheel shape along its cast path (excluding the upper half of the wheel in its start position).

Using a phantom for each wheel has little benefit in the average case, when the wheels are only colliding with the landscape. The intention is to reduce the worst-case cost when the vehicle drives over lots of collidables.

The image below, which shows a car mounting a curb, illustrates various aspects of the linear cast wheel collide object. The green boxes represent the position from which the wheel shapes are cast. The red boxes represent the target positions of the cast. The yellow boxes are the `hkpAabbPhantom`s at the wheels. The white squares and arrows represent the contact points of the wheels.



To create a vehicle which uses linear casting for wheel collision detection:

- During vehicle set up, prior to calling `hkpVehicleInstance::init()`, the `m_wheelCollide` member should be set to a new instance of `hkpVehicleLinearCastWheelCollide` (new because wheel collide components cannot be shared).

By default, the `hkpVehicleLinearCastWheelCollide` component will create `hkpCylinderShape`s for each wheel. This is somewhat wasteful. Instead, you can provide pointers to shared wheel shapes using `hkpVehicleLinearCastWheelCollide::setWheelShapes()` before `hkpVehicleInstance::init()`. Shapes provided in this way are not required to be cylinders. Assertions will check user provided cylinders (only) for consistency with the vehicle's wheel parameters.

### Tuning vehicle chassis collision

Gameplay often requires that a (player-driven) vehicle chassis be affected differently by collision than other rigid bodies in the simulation. Options to tweak the interaction results are listed by use case below but can be mixed and matched in most circumstances.

**Note:**
The current Havok Physics 2012 SDK documentation and demos should be consulted in all cases as some of the following features have potential performance impact - the Havok Visual Debugger can be useful in analyzing performance. The Havok Vehicle Kit includes logic hooks for some of this out-of-the-box but most suggestions are relevant to any vehicle with a Havok Physics 2012 rigid body chassis.

First confirm that the over-aggressive collision response is due to chassis collision and not other systems such as suspension, rollover-correction etc. - the Havok Visual Debugger with contact point viewers is an essential tool for this diagnosis.

### Reducing torque

Torque is often the least desirable response. There are several ways to reduce torque effects including the following:

- `hkpCenterOfMassChangerUtil` allows the center of mass to be (virtually) moved per contact point.
- Increase vehicle chassis inertia. This can help but may affect general vehicle handling. See `hkpRigidBodyCinfo::m_inertiaTensor`, `hkpInertiaTensorComputer`, `hkpVehicleData::m_chassisUnitInertiaRoll`, `hkpVehicleData::m_chassisUnitInertiaPitch`.
- Decrease angular velocity limit. See `hkpRigidBodyCinfo::m_maxAngularVelocity`.
- Increase angular damping. See `hkpRigidBodyCinfo::m_angularDamping`, `hkpVehicleDefaultVelocityDamper`.

### Collision with dynamic objects and/or other vehicle chassis

A common requirement is that (player-driven) vehicle momentum should not be (significantly) affected by collisions with dynamic objects, or that player-vehicles should have greater impact on NPC-vehicles than vice versa.

- `hkpCollisionMassChangerUtil` allows the relative mass to be changed per contact point.

### Collision with breakable, (initially) static objects

Similar to above but the objects that should not affect momentum are initially static and/or part of another static shape.

- `hkpBreakOffPartsUtil` and `hkpBreakOffPartsListener`

### Collision with static objects

In this case, the goal is to reduce effects due to collision with walls, "bottoming-out" after jumps, "bottoming-out" when climbing ramps etc.

- Reduce restitution between both bodies (`hkpRigidBodyCinfo::m_restitution` or `hkpContactPointProperties::setRestitution()`).
- Reduce `hkpWorldCinfo::m_toiCollisionResponseRotateNormal` if the vehicle chassis uses `hkpCollidableQualityType::HK_COLLIDABLE_QUALITY_MOVING` or higher. Lower values can reduce the amount of bounce the vehicle exhibits during these types of collisions.

**Collision with static objects - curbs (and kerbs)**

Another common use case is driving into or over high curbs with a low-riding vehicle chassis, without affecting momentum or control. In this case, since the requirement is to mitigate both torque and linear force, the only solution is to rotate the contact point normal using a `hkpContactListener` callback.

### 2.5.3 Saving Contact Points

The `hkpSaveContactPointsUtil` allows you to save contact information for chosen colliding pairs of bodies or for an entire scene at once.

For each collision entry (i.e. a pair of bodies in contact) we store `hkContactPoints`, their `hkpContactPointProperties` and the entire collision agent hierarchy for the case when compound shapes are involved. The utility does not support all existing collision agents. Therefore when unsupported agents are encountered in an agent hierarchy, the saving of a collision entry is aborted.

There is also some limited serialization support for the contact point information, which allows you to store & restore full contact information, but puts restrictions on the runtime's configuration (see the Limitations section).

The Physics 2012 / Api / Dynamics / World / World Snapshot with Contact Points demo is a simple example of how to save a `hkpHavokSnapshot` with contact points and how to load it back from a disk.

#### 2.5.3.1 Methods and Parameters for Saving

There are three `hkpSaveContactPointsUtil::saveContactPoints()` function variations which let you save collision entries for the entire `hkpWorld`, for a list of `hkpEntities`, or an explicit list of collision entries (represented by `hkpAgentNnEntry` structures). (The variation with specified list of `hkpEntities` attempts to save all collision entries, where at least one of the involved `hkpEntities` is on the input list.)

If a single collision entry cannot be saved (i.e. when unsupported agent types are encountered), it's just ignored, and the utility continues to process the remaining collision entries.

Collision entries are saved into an array of `hkSerializedAgentNnEntries`, enclosed in a `hkpPhysicsSystemWithContacts` instance. (Be cautious with saving collision entries one-by-one as there is no checks implemented to check for duplications, i.e. multiple `hkSerializedAgentNnEntries` stored for one pair of bodies.)

The serialized collision entries have to be linked back to their respective `hkpEntities` later. To do that you may chose to either store `hkpEntity` pointers directly, or use custom unique ids. The former solution is useful if you're saving an entire `hkpHavokSnapshot` with `hkpRigidBodies` included in it. The latter solution is helpful if you want to remove involved `hkpEntities` from the `hkpWorld` after saving, destroy them and then recreate later before you load the points again.

Use the `SavePointsInput` structure to choose between saving `hkpEntity` pointers or custom ids. If using ids you'll also need to provide a callback function that assigns an `hkUlong` id for a given `hkpEntity`. You'll need to provide a symmetric function later when loading collision entries.

#### 2.5.3.2 Methods and Parameters for Loading

There are two functions for loading contact points. One attempts to restore collision entries for the entire `hkpWorld`, the other function only attempts to restore collision entries involving `hkpEntities` from a list.

When loading points, the utility does not expect to match all saved collision entries. For each saved collision entry it attempts to find a matching pair of `hkpEntities`. If one does not exist that saved collision entry is ignored.

The `LoadPointsInput` structure allows you to customize the way that saved contact points are loaded.

- You can choose to destroy a serialized collision entry after it is loaded (that's useful, if you're loading contact points gradually one by one).
- You can zero `m_userData` stored in `hkpContactPointProperties` (might be useful, if you're using user data to store pointers to your own structures).
- You may choose to fire or skip collision callbacks. Namely `contactPointAdded` and `contactPointConfirmed` callbacks. Note however that using those callbacks usually requires special handling in your callback code, different to handling of normal callbacks.

  The `contactPointAdded` callbacks don't have the proper `hkpCdBody` pointers – you may choose to pass the top level `hkpCollidable` pointer or `HK_NULL`. (That's because normally those callbacks are fired from within the collision pipeline, when they can reference temporary `hkpCdBody` objects.)

  Also note that `conactPointConfirmed` callbacks are not called for all points. They are only called for points newly created in the previous simulation step, as it was in the orignal simulation, from which the contact points were saved.
- Finally you should provide a callback function for mapping your custom unite entity ids to `hkpEntity` pointers. This function will only be used by `hkSerializedAgentNnEntries` that are marked to use those custom ids instead of direct `hkpEntity` pointers.

#### 2.5.3.3 Serialization

You can serialize a `hkpSerializedAgentNnEntry` or a collection of such entries stored in a `hkPhysicsSystemWithContactPoints`. When storing contacts separately from `hkpEntities`, remember to map the `hkpEntity` pointers to unique ids (see the 'Methods and parameters for saving' section above).

You can also use the `hkpHavokSnapshot::save(hkpWorld*, âĂę, bool saveContactPoints = false)` function to quickly create a physics snapshot with contact points.

There are platform restrictions for serialization of saved contact points. See the section below.

#### 2.5.3.4 Limitations

**Types of Collision Agents Supported**

The `hkpSaveContactPointsUtil` only supports collision agents working in the streaming-agents technology, i.e. all agents whose class name ends with the -3 suffix. Other agents are not supported, and saving is aborted when they're found.

**Restrictions on hkpCollisionDispatcher's Configuration**

It is advised to have the same set and order of `hkCollisionAgents` registered both when loading and saving contacts.

That's because when saving a collision entry, the utility saves raw data from an agent stream. That data also includes the agents' types, which in turn are simply indices of registration records in a table in the hkpCollisionDispatcher.

**Cross-Platform and Layout Issues**

Saved contact entries contain raw data from agent streams, have no reflection data, and cannot be versioned. Therefore, in the case of a change in the data structures of core collision agents in the future releases, the user will have to recreate the contact point data.

For similar reasons we cannot use our serialization framework to port the data between platforms with different data layout rules, except if the only difference between the platforms is their endianness. There is the specialized `hkpSaveContactPointsEndianUtil` that converts the endianness of the stored data.

## 2.6 Articles

This section contains general issues relevant to using Havok Physics 2012 in your games. The following topics are covered:

- A discussion on mathematical conventions and representations entitled "Rotations, Handedness and all that".
- An in-depth exploration of Havok's Continuous Simulation.
- A guide on integrating particle effects with Havok Physics 2012.
- A guide on best practices for setting up and using ragdolls.

### 2.6.1 Rotations, Handedness, and all that.

#### 2.6.1.1 Note On Notation

*Note: In this document we interchangeably refer to rotations/orientations, and also translations/positions. Strictly speaking rotations and translations are best thought of functions, and orientations and positions*

*are the things they act on. However since there's an obvious mapping between e.g.. "the translation which maps the origin to (1,3,6)", and "the position (1,3,6)", we don't draw a distinction.*

*Notation: In this document, the expression "R(Z,90)" is used to mean "the rotation of ninety degrees around the Z axis".*

#### 2.6.1.2 Introduction

This document explains common representations and conventions for rotation and transformation. It also explains how to identify and convert between different representations.

At Havok we're in quite a unique position. We're exposed to more integration issues than most other middleware companies, mainly because most other middleware is either entirely adopted (together with a toolchain, math conventions etc. like most of the rendering middleware companies) or the flow of information is one way e.g. from toolchain to renderer, or from game engine to sound middleware. Information in Havok flows both ways, the scene is initially set up in a toolchain and this information makes its way to construct Havok objects and initial scene positions and orientations, and then when the simulation takes place this orientation information must flow back from Havok to the renderer. Generally most of this information is transformation information i.e. positions (translations) and orientations (rotations). At each integration interface you potentially have to convert from one set of conventions to the other.

- Part 1 of this document describes common sets of conventions found in the game industry used to represent transformations. It explains the reasons and sources of confusion, particularly with respect to rotation and handedness conventions. It's very useful if you are beginning integration with any 3D tool or if you're designing an engine from scratch.
- Part 2 describes how to convert between the different sets of rotation conventions. It gives a common practical example for converting between Right and Left handed systems and then generalises this to an arbitrary "change of basis" conversion. Finally we present an a working example going from a common modeller (3DS Max) to constructing a Havok scene and feeding the results straight into a DirectX rendering example.

#### 2.6.1.3 Representations

Havok does not have a "handedness".

Havok is not "right-handed".

Havok stores rotations in two objects, a matrix format (hkRotation), and a quaternion format (hkQuaternion).* The way in which the rotation is stored in each case has been chosen to adhere to certain mathematical conventions, however it is a choice and other people will (may) have similar looking objects which represent (to them) the same rotation, but the data may be different.

Or, looking at it the other way, you may interpret one of our objects like one of yours and your interpretation of the resulting rotation may be different to ours.

Why is this?

Well, the thing about rotations is, they're awkward to represent. To justify this, let us first consider "positions".

**Note:**

*Havok also stores rotations "in" two other formats, namely wherever "torque" or "angular velocity" is used. This will be dealt with later. The rotations inside hkTransforms are stored as hkRotations

### Representing Positions

The thing about positions is, "you know where you are with positions".

For a start (and we're considering 3D positions here), there's a pretty universal convention for describing them:

Implicitly there are three canonical bases:

(1,0,0), called X

(0,1,0), called Y

(0,0,1), called Z

No-one calls e.g. (1,0,0) 'Y', but even if you called the bases Bob, Phil and Geoff it wouldn't matter because there is an implicit order here, given by the fact that we write them down right-to-left, rather than, for example as a set:

which would be not much use to anybody. How should one find the vector between:

So to avoid any confusion we normally write these as (for example):

(4, 2, 1) and (4, -2, 3)

That's great, because we can then go and subtract them, dot them, cross them etc.

[There's also this nice symmetry between the elements of a "position" : X,Y,Z are all "equal" (you can go into the mathematics of this to justify it, but basically the X-direction isn't any longer, or whatever than the other directions, and anything you can do to the x component of a position, like multiply it, or subtract it or zero it, you can do to the others in just the same way)].

So if I say to you "consider the position (1,2,3)", you'll have a pretty good idea what that "looks" like. Now, you'll probably imagine some orthonormal system where you've picked X to be "right", Y to be "up", Z to be "in", or some permutation of these. It doesn't matter. It doesn't matter if Z is "up", or if X is "left", that's all contained in the embedding of this abstract space into the real world. Mathematically no one of them is "up" or "right". The important thing is that if I then say to you "now consider the position (1,5,7)", and ask you how far away it is you'll say "3 Y units and 4 Z units" or somesuch, no matter how you're thinking about it. Also, if I ask you anything about dot products, cross products, etc., you'll get the same answer as anybody else.

How does this apply to Havok?

Well, Havok is abstract too. It deals with triples of real numbers (X,Y,Z). There is no "up", there is no "right". The default value for gravity is (0, -9.8, 0), but this is not hardcoded.

If you do the Havok equivalent of (1,5,7) – (1,2,3) which is something like.

```
hkVector4 a(1,2,3);
hkVector4 b(1,5,7);
hkVector4 c;
c.setSub4(b,a);
```

then you'll get the same result anybody else will with any other engine, i.e., (0,3,4) because mathematically this is the correct answer, and there's no reason they would have written a math library which claimed to implement "Vectors" which didn't adhere to mathematical definitions. This applies also to the interface which gets/sets the X,Y and Z elements. They don't have to store them contiguously in memory as X,Y,Z, you can store them how you like, but it is highly probable that the storage in memory will be in the same order as X,Y and Z are alphabetically

So "positions/translations" are pretty much unambiguous / easy.

**"Representing" Rotations**

"Rotations" are difficult. For a start, how do you represent them? Here's one representation:

Here's another:

[0.0916, (0.183, 0.274, 0.939)]

And another:

(0.7, 0.267, 0.534, 0.801)

They all represent the same rotation in Matrix, Quaternion and Axis-Angle format respectively. [They all look pretty different, and they each have both "good" and "bad" properties in terms of ease of use, efficiency etc. This is why all three are common in the games industry and are hence worth mentioning, though we shall not go into any comparative details here].

And they're all *ambiguous*.

The easiest way to demonstrate this is to take the matrix case because it's the one people are most familiar with (even though people tend to use Angle-Axis in conversation and examples, because it's easier to say (less info), and fits on one line).

We all know you can multiply matrices by vectors in two ways: pre-multiplying and post multiplying, and in general these give completely different results. For example

But

which brings us to the core of understanding rotation representations:

**"Defining" Rotations**

Rotations are less defined by what they are, but rather what they do.

So we can uniquely define a rotation for example by specifying the action it has when applied to all possible vectors. Fortunately, due to the nature of vectors, it's possible to simplify this to just specifying the action it has on each of the three basis vectors X, Y and Z.

So think of rotations as functions or operators. You haven't fully specified them until you say how they apply to vectors.

So:

with the operation of left-multiplication on a position interpreted as a column vector, i.e. $v$ -> $Mv$

is unambiguous (because left-matrix multiplication is).

"[0.0916, (0.183, 0.274, 0.939)] as a quaternion [a,b] = q with the operation of

$p$ -> $qpq^{-1}$, where $p = (0,v)$, and the result is the imaginary component of the resulting quaternion."

is unambiguous (because quaternion multiplication is).

"(0.7, 0.267, 0.534, 0.801) as a angle,axis pair [a,b] with the operation of

p -> pcos(a) + sin(a)(b x p) + (1-cos(a))(b $\cdot p)b''$

is unambiguous (because the cross product is).

**Removing Ambiguity**

The reason why this can cause problems is because people frequently don't even realize there is this ambiguity, hence don't supply the extra information necessary to resolve it. So long as you are "consistent" in your use of rotations, there *is* no ambiguity (as far as you know). It involves a bit of "stepping back" from the problem to even realize it exists:

For example, consider the "rotation":

"Ninety degrees around Z"

This certainly means a rotation which leaves (multiples of) the Z-axis invariant, and the angle between v and v' (v after being rotated) is 90 degrees for any v perpendicular to the Z-axis. But there's two

possibilities here - either (for example) X goes to Y, or X goes to –Y. They are different rotations, in the same way that “clockwise” and “anticlockwise” are different. And you could associate one of them with being “clockwise” and one with being “anticlockwise”, but in the absence of any convention there’s no way to say which should be called which.

Once you have decided on one of the two choices, you have effectively determined which way a “positive” angle rotates, i.e. you have defined “positiveness” for an angle-axis representation.

As the diagrams below show, the same rotation can be thought of as either “clockwise” or “anticlockwise” depending on the “handedness” of your system.

Figure 2.75: Left And Right-Handed Coordinate Systems

**The ‘Right-hand Rule’ (and why it almost helps)**

Now there is also a convention for defining “positiveness” of rotation, without specifying the “handedness” of your coordinate system, known as the “right-hand rule”. This says that a positive angle is measured *anticlockwise* around the axis of rotation. It is a convention that in such interpretations of clockwise/anticlockwise that the axis is pointing “out” of the clock towards you (thank goodness).

The right hand rule is frequently illustrated by something like this:

Figure 2.76: The "Right Hand Rule"

This *seems* to remove any ambiguity because it is clear in any diagrammatic interpretation, such as lining up your right hand with a screen image, or either of the coordinate systems above, which way is a positive rotation. *But* as you can easily see, applying this rule to the “right-handed” coordinate system would define positive as rotating X to Y, whereas applying it to the other would mean we’d get X rotating to -Y. These are two different rotations, so saying “we use the ‘right hand rule”’ isn’t enough.

Figure 2.77: The Right-Hand Rule In Both Coordinate Systems

**Ambiguity Resolved.**

We know for certain that different people use coordinate systems of different handedness, it’s too late to change that now (even if we wanted to). One possible way of resolving ambiguity would be to get everyone to adopt the “right-hand rule”, but this would mean (as illustrated above) that the actual rotation(as an operator) described by people, for example “Ninety degrees around Z”, would depend on their coordinate handedness.

Fortunately, there is a corresponding “opposite” rule for rotations called the “left-hand rule” which defines positive rotations in the opposite direction, i.e. clockwise, and it happens that:

*(Almost) everyone who uses a right-handed coordinate system adopts the “Right-hand” rule, and those who use a left-handed coordinate system adopt the “Left-hand” rule.*

This is true of (amongst others): DirectX (Left), Renderman (Left), Unreal (Right), Renderware(Right), 3DSMax( Right), Maya (Right). It is mostly true of OpenGL which uses the right hand rule, but can use left or right-handed coordinates. It can be thought of as being true of Havok too, which (since it doesn’t have a renderer as part of the core engine) is neither right not left handed, but defines positive rotations

in the same way (their operation is the same) as all the others.

The net result is that (almost) everybody agrees that a rotation of+90 degrees around Z maps X to Y, or in other words, we all agree on the definition for a positive rotation (as an operation).

**Warning:**
We say "almost" because occasionally there are anomalies, and since the choice of handedness/rule is somewhat arbitrary, it's perfectly OK to have a system which rotates "the other way".

Example: 3DS Max uses a "right handed coordinate system". Its user-interface adopts the "right-hand rule", which means that all rotations are counterclockwise around the axis. However, its internal SDK adopts the "left-hand rule", which means that all rotations are clockwise around the axis.

Similarly, Gamebryo/NetImmerse uses a right-handed coordinate system, but the left-hand rule.

**Representation Conventions**

Now that we have dealt with the two possible "mathematical" conventions which may be adopted, it's time to look at the possible "representation" conventions which any implementation in code might adopt.

The following are 4 "rotation representations":

- Axis, Angle
- Euler Triple
- Quaternion
- Matrix

In this document we will not address Euler angle representation because direct conversion to/from Euler angles is normally done via a matrix format (which we will address).

We will also assume that the basic real-number representation chosen will be a *float*.

For *Axis,Angle* the obvious data representation is always chosen:

A single float for the angle, and a float triple for a *normalized* axis. The only storage choice to be made is whether to put the angle or the axis first in memory. With proper accessors (such as the hkQuaternion.getAxis() method), conversion of Axis,Angle to/from hkQuaternion, hkRotation is trivial, and dealt with in the next section on "Conversions".

For *Quaternion* the obvious data representation is also always chosen:

A single float for the real component, and a float triple for the imaginary component, where the real component is cosine(angle/2) and the imaginary component is sine(angle/2)**normalized* axis. Again, the only storage choice to be made is whether to put the angle or the axis first in memory.

For *Matrix,* there are two conventions (unfortunately). Since you can multiply a matrix on either side of a vector (in one instance the vector is considered a row, in the other it is considered a column), you have two choices of how to "represent" a rotation in matrix format.

In the following examples we assume a 4x4 affine matrix. Frequently some components are unused

(ignored) or even not stored at all (effectively only 12 values are stored). We assume that the reader is familiar with such simplifications, and we do not deal with them here.

For example, R(Z,90) (rotation of 90 degrees around Z), which maps X to Y can either be represented as

0 -1 0 0

1 0 0 0

0 0 1 0

0 0 0 1

if the matrix applied on the left to column vector – "pre" multiplies. Mx = x'

*or*

0 1 0 0

-1 0 0 0

0 0 1 0

0 0 0 1

if the matrix applied on the right to row vector – "post" multiplies. xM = x'

An interface to these matrices which allowed us to, for example, get the first column would return

0

1

0

0

in the first case, but

0

-1

0

0

in the second case, so clearly were we to try and copy over data from one to the other via such an interface we would be accidentally changing the rotation (in fact, inverting it because for rotations R-inverse = R-transpose). Worse, if the matrix had a translational component, we might attempt to copy over

0 -1 0 7

1 0 0 4

0 0 1 3

0 0 0 1

directly into a matrix which is supposed to be applied on the right, and clearly the translational data would be put in the wrong place. The resulting matrix if applied on the right to the origin (0,0,0,(1)) would give us (0,0,0,(1)), which clearly has not been translated, hence the conversion cannot be correct.

Fortunately, it happens that a by-product of the conceptual way in which the rotations/transforms are applied (either to "column" or "row" vectors) means that *usually* the data will actually be *identical when laid out in memory*, no matter which "representation" is chosen.

To see why this is, consider the choice of using matrices which are to be "pre" multiplied (applied on the left). Because the mathematical object which these matrices apply to must be a "column" vector, it is *likely* that the matrix itself is also stored "by columns", or in column-major format. Column-major format of a matrix stores the elements linearly in memory, column by column. That is to say the elements in a single column are adjacent in memory and their address increases as the row index increases. For example:

0 -1 0 7

1 0 0 4

0 0 1 3

0 0 0 1

is stored as:

0 1 0 0 -1 0 0 0 0 0 1 0 7 4 3 1

Now consider the choice of using matrices which are to be "post" multiplied (applied on the right). Because the mathematical object which these matrices apply to must be a "row" vector (opposite to the case above!), it is *likely* that the matrix itself is also stored "by row", or in row-major format. Row-major format of a matrix stores the elements linearly in memory, row by row. That is to say the elements in a single row are adjacent in memory and their address increase as the column index increases. For example, the matrix *which represents the same transformation as above when applied on the right*, is

0 1 0 0

-1 0 0 0

0 0 1 0

7 4 3 1

and when this is stored row-major, it is stored as

0 1 0 0 -1 0 0 0 0 0 1 0 7 4 3 1

just like the other one.

Note that in both cases the *translational* components of the transform are "at the end" of the memory block.

So, it is *very likely* that any two matrix storage formats (either 4x4 or, if ignoring the last column/row 4x3 or 3x4) will be identical in memory. Havok, Renderware, DirectX, Renderman and Unreal all have the same storage in memory for their matrix representation, even though Havok is column-major whereas Renderware, DirectX, and Unreal are row-major.

**Note:**

Unless a system explicitly ties "columns" or "rows" to the block of data which stores the matrix as a whole, the storage cannot be said to be either "row major" or "column major". OpenGL for example specifies only that the matrix is stored in 16 consecutive floats – there is no reference to "columns" or "rows" except in the GL documentation, which does so for ease of explanation, but actually rather confuses things by implying an ambiguity – namely whether the matrices are pre- or post- multiplied. In fact the actual definition simply requires that (like the "common" matrix storage format above), the translational components are the $13^{th}$ $14^{th}$ and $15^{th}$ floats

See http://www.opengl.org/documentation/specs/version1.4/glspec14.pdf p34 and p36 and http://research.microsoft.com/∼hollasch/cgindex/math/matrix/column-vec.html

Another way to determine which parts of the matrix data are rows, and which are columns is to examine the result of a matrix multiplication where the order is specified, since you know that the resultant matrix must be formed from dot products of rows from the first, and columns from the second (mathematical convention).

Havok has accessors for elements by (row,column) index pair, and it also has explicit getRow() and getColumn() methods and in fact its internal storage for an hkTransform is as an hkRotation (3 hkVector4s, m_cols) followed by an hkVector4 (m_translation). This means that Havok definitely stores its transforms in column-major format, and as described above, since it also uses the "common" storage format, one can deduce that Havok applies its matrices on the left (pre-multiplies, conceptually).

As a contrast, DirectX's matrix is defined as:

```
typedef struct _D3DMATRIX {
  union {
      struct {
          float _11, _12, _13, _14;
          float _21, _22, _23, _24;
          float _31, _32, _33, _34;
          float _41, _42, _43, _44;
      };
      float m[4][4];
  };
} D3DMATRIX;
```

which means that row elements are clearly contiguous in memory, hence it is row-major and applies its matrices on the right (post-multiplies, conceptually).

**Note:**

Once again, although this is adopted in most of the major packages and standards, this is not a universally accepted convention, so it is (at least in theory) possible for a system to store what is the "same" rotation (the operation is the same) differently in memory to the way Havok (for example) stores it.

#### 2.6.1.4 Conversions

**"Positive Rotation Rule" conversions**

The left-hand/right-hand rules concern the mathematical conventions which determine which way a "positive" angle will rotate, and hence whether conversion is required for any of the structures which have an interface allowing you to get/set an angle (or axis), or which implicitly store data related to this representation. In Havok these are:

- hkQuaternion
- hkRotation
- (hkTransform, which contains an hkRotation)
- "angular velocity" as an hkVector4
- "torque" as an hkVector4

As described in the section on Removing Ambiguity, the convention for defining positive rotations is that you should apply the same "hand" rule as the handedness of your coordinate system. If, for some reason, you do not use this rule, but rather define positive rotations in the opposite direction so that you have R(Z,+90) maps X to –Y, then a conversion must be made if you wish calculations made in Havok to be the same as those made in your engine.

In this case the conversion is easy, and it simply involves changing the sign of the angle used. Since this is mathematically equivalent to inverting the rotation (the inverse of a rotation of theta degrees around an axis v is clearly a rotation of –theta degrees around v), this is what we must do for each representation used.

**hkQuaternions**

For an hkQuaternion, on creation from an axis,angle pair, just negate the angle:

```
hkVector4 axis(myAxisX, myAxisY, myAxisZ);
q.setAxisAngle( axis, -myAngle );
```

If the hkQuaternion has already been created for some reason, call

```
q.setInverse(q).
```

Alternatively, we can directly alter the internal parts of q:

An hkQuaternion stores a rotation (theta,v) (say) as:

hkReal cosine(theta/2), and an hkVector4 sine(theta/2) * v.

Clearly then the rotation (-theta,v) is stored as:

cosine(-theta/2), sine(-theta/2) * v

which is equal to

cosine(theta/2), -sine(theta/2) * v.

So all we have to do is negate the imaginary (hkVector4) part of the hkQuaternion.

Alternatively, since it is a (nonobvious) property of quaternions that both q and –q (both real and imaginary parts of q negated) represent the same rotation, we can just negate the real part instead.

In Havok, both parts are munged into the single hkVector4, the real part being stored in the (otherwise unused) $4^{th}$ component, so the simplest thing to do would be to negate this $4^{th}$ component like this:

```
q.setReal(-q.getReal());
```

Similarly if converting from your own quaternion format then you will need to copy over the imaginary part directly, but negate the real part.

To convert back to axis,angle, call getAngle() and (if the angle is non-zero*) getAxis() and then negate the angle.

**Warning:**
*Remember - the axis of rotation is undefined if the angle is zero!

### hkRotations

For an hkRotation, if constructing using setAxisAngle(), then negate the angle.

WARNING: Now here's where the first problem may arise – if you are directly copying over matrix data, then you must know whether you adhere to the "common storage" convention or not. Havok uses the common storage convention, so if you use it too then everything's fine, there's no extra work to be done, and the next couple of paragraphs will tell you how to fix your hkRotations to allow for the difference in "positive rotation rule" convention. *However*, if you store your matrix data differently, then it is almost certainly the "transpose" matrix representation, and hence your rotation will already be the inverse of what Havok uses, *so there is no need to do the extra step below.*

If an hkRotation has already been created for some reason, call

```
r.transpose()
```

which transposes the hkRotation in-place. Since a rotation matrix is always orthonormal, this will invert it.

If you are converting from your own data (and have read the warning above), then either transpose as you copy over, or copy directly, and then call r.transpose().

### Angular Velocities/Torque

In Havok, angular velocities and torques are stored in hkVector4s, and they also follow the same "positive rotation rule" as used for axis,angle representations. That is to say, for example, a body at the origin, aligned with the world axes, with an angular velocity of PI/2 radians/second around the Z axis (0,0,1) will have its local X axis rotated to be coincident with the Y axis after 1 second. In other world, positive angular velocities rotate anticlockwise when displayed in a right handed system, and clockwise when displayed in a left handed system.

Torques have the same property.

Since angular velocities are stored as vectors v, where length(v) = rotational speed in radians/second, and v (normalized) is the axis of rotation, converting these involves the same logic as converting axis,angle data – we simply negate the angle. The result is that we negate the entire velocity, and the same is true of torques e.g.

```
angVel.setNeg4(angVel);

torque.setNeg4(torque);
```

**"Handedness" conversions**

It may be the case that you wish to take data that has already been "modelled" in a system which has a different coordinate frame "handedness" to that which you wish to render it in.

For example, you may have used the Havok Exporter, or your own toolchain to output "right-handed" data from 3DSMax, which you then wish to alter so that it appears identical (visually) when put through your renderer which may be "left-handed".

Clearly all positional data must be modified (flipped), but it is less obvious how to change, for example, rotational data, such as a hkQuaternion or hkRotation, so that it correctly reflects the change in handedness. Because the matrix format is more common, and easier to explain, we deal with this format first, then we deal with quaternions.

To illustrate what must be done we shall first take the simplest (mathematical) example and derive the necessary operations "from first principles". This section may be ignored if you wish to move on to the more general solution, which we shall explain with less rigor, but more completeness.

**Example. Matrix "mirror" conversion.**

We consider the "–X flip" to be the canonical illustration of "conversion" from "left" to "right". The first Figure in Section 3.4 illustrates this. Imagine that your modeller was right-handed (the coordinate system on the right), but that your renderer was left handed (coordinate system on the left). Clearly were you to export your data directly, and render it, the resulting "views" of the scene would be mirror images of each other. This is why you must do some form of "left-handed to right-handed" conversion.

To be precise in order that (with the camera in the correct place) both the LH and RH views *look* identical, we must mutate the data we export in some way.

Let us assume we start off with some mesh data and an associated transform. For simplicity, we also assume that the transform here is a "rigid" transform, that is it contains only pure rotation (no reflection, no scaling), and translation components.

What we want to get out is another set of data, and another "rigid" transform, which, when plugged into the renderer, will produce the same view. We do this by "mirroring" all our data and transforms, i.e. "flipping X".

If you flip X, then clearly:

- The vertex data must be"mirror" flipped. "Proof": consider an asymmetric object with identity transform. Clearly, since the final transform must also be the identity, the data must be flipped for the object to be displayed correctly.
- The translation of the transform must be flipped. "Proof": consider any object (e.g. a point) displaced from the origin e.g. to the right. Clearly, the X component of the translation must be flipped in order to displace the object to the left in the other handedness.
- The rotation of the transform must be "mutated". This is the tricky bit, so we'll go into a little more detail below.

To work out what the new rotation must be, given the old rotation (as a matrix), let's remind ourselves of what the rotation matrix is:

It's the image of each of the canonical axes, stored as columns (or rows, see Representation Conventions Section). So in order to mutate our rotation matrix we just have to work out what the image of the "new" canonical axes would be under the same rotation, expressed in terms of these "new" canonical axes. We're preserving Y and Z, and flipping X, so the "new" axes are –X,Y and Z.

Figure 2.78: X, Y, Z and -X

We can "show" the rotation in our right-handed system by showing the vectors R(X), R(Y) and R(Z), representing the image of X,Y and Z under the rotation R:

Figure 2.79: Our rotation in the right-handed system.

In the Left-Handed system, the same rotation would act on –X,Y and Z. What we want now is to know what R does to –X, but clearly R(-X) = -R(X):

Figure 2.80: Our rotation in the left-handed system

We can now write down the rotation in the left-handed space, with canonical axes X', Y' and Z' say. In the left-handed space, all x components are flipped, i.e. Y' is Y and Z' is Z but X' is –X. To keep things clear, we're going to specify which space every vector is written in by appending either RH or LH to the vector to specify which vectors are being used as the basis e.g.

(1,2,3)RH

which is equal to (-1,2,3)LH

because

(1,2,3)RH = 1*X + 2*Y + 3*Z

1*(1,0,0) + 2*(0,1,0) + 3*(0,0,1)

which is equal to

-1*(-1,0,0) + 2*(0,1,0) + 3*(0,0,1)

= -1*X' + 2*Y' + 3*Z'

= (-1,2,3)LH

So if:

R =

R00 R01 R02

R10 R11 R12

R20 R21 R22

(written as a pre-mult matrix, i.e. X under this rotation maps to (R00, R10, R20))

then we want R' (a rotation acting on the left-handed space generated by X', Y', Z') which maps:

X' to R'(X')

Y' to R'(Y')

Z' to R'(Z')

Now we know where each of X', Y' and Z' will go because, for example Y' = Y, so

R'(Y') = R(Y)RH = (R01, R11, R21)RH = (-R01, R11, R21)LH.

So the second column of R' is (-R01, R11, R21)

Similarly

R'(Z') = R(Z)RH = (R02, R12, R22)RH = (-R02, R12, R22)LH.

And finally

R'(X') = -R(X)RH (note the minus sign here!)

= -(R00, R10, R20)RH = -(-R00, R10, R20)LH = (R00, -R10, -R20)

so

R' =

R00 -R01 -R02

-R10 R11 R12

-R20 R21 R22

So this is how we observationally work out the matrix R' (acting on vectors in the LH space), which has the same effect as R (acting on the corresponding vector in the =RH space. Hence, by the "rotations are defined by what they do" understanding, R' and R are the same rotation, so we have successfully converted R from right handed to left handed.

**General Solution (Change of Basis Conversions)**

Alternatively, we could use the following intuition:

If we wanted to work out how a vector v = (*a*,*b*,*c*) in the LH space was affected by R, we could always:

- Flip *a* (i.e. the x-component)
- Apply Rotation to generate e.g. (*a'*, *b'*,*c'*)
- Flip *a*' again

What we're doing here is representing v in the space we have the rotation in, then applying it, then reinterpreting our result back into the LH space. This obviously must give the correct result. In matrix form, the "flipping" can be expressed as:

F =

-1 0 0

0 1 0

0 0 1

since F(a,b,c) = (-a,b,c)

and we are applying FRF to the vector v. If you work out FRF you will indeed get the same result for R' as we got before.

*In general*, we can say if T is a transform with zero translational component, and we wanted to work out how a vector (a,b,c) in the transformed space was affected by a rotation R, we could always:

- Apply T-inverse (maps point back to original space)
- Apply Rotation
- Apply T (maps T back to transformed space)

i.e. apply TRT-inv (expressed as pre-mult, obviously (T-inv)RT if post-mult).

It simplified a bit in the "flipping" case because F-inv = F, but in general this is what you'll have to do to convert your rotation. This will work not only for the "flip" case, but also for example like "swapping

Y and Z", or "rotating so that Y points up/down". If you represent the change you are doing by the correct transform, you can apply the above rule to convert your rotations as well as your translations.

It may happen that the actual operation of applying T and T-inv will simplify, like in the "flip" case where the final matrix could be computed by just flipping the sign of four elements. In this case, it may be useful to hardcode the conversion process, but be warned that where more complicated conversions occur it is easy to obfuscate what is going on, so unless there are performance reasons (and certainly when first writing this code!) it is better to use the full form: TRT-inv.

For example, here is a Havok function which will convert an hkRotation given a rotation matrix which may have reflection in it (using the changeBasis function of hkRotation, which does exactly this TRT-inv multiplication):

```
void convertRotationToNewSpace(const hkRotation& rotToNewSpace, hkRotation &r)
{
r.changeBasis(rotToNewSpace);
}
```

**Quaternions, and Axis/Angle**

Now the above technique of map, rotate, map back will also work fine for any other rotation representation except that it is less general. Quaternions, in particular, cannot store reflections, such as those required to convert left-handed to right-handed. This means that you cannot pop the rotation part of your matrix T into a quaternion and compute qrq-inv as you might expect (where q is T as a quaternion, r is your original rotation as a quaternion, and we define the product using quaternion multiplication). In some cases you may be able to strip out the reflection component of the matrix which causes the problem, but we won't deal with that here, and we'll use a more direct approach.

Because quaternions are harder to "read" because they store the rotation in such a compact and computationally efficient format, we will clarify the conversion process by first examining axis/angle representations. See the picture below:

Figure 2.81: Axis/Angle rotation in right-handed system.

Intuitively, to covert the rotation in axis/angle format all we should have to do is change the axis about which the rotation takes place (it must take place around some axis!), and keep the angle the same (the amount of rotation surely does not change), with a minor caveat in that perhaps we will need to change the sign of the angle.

Our intuition is right here, the conversion of the axis is trivial, you just flip the x-component. As for the angle, the gotcha here is that now that we're in a left-handed system, we will (almost certainly, according to the "Positive Rotation Rule) consider positive rotations to be rotating the other way. This means that we should flip the sign of the angle too:

Figure 2.82: Axis/Angle rotation in left-handed system.

*In general*, we can say if T is a transform with zero translational component you simply transform the axis by T:

converted axis = T*axis,

and then flip the sign of the angle *if T contains a reflection component*, such as when we change handedness by flipping X. Otherwise, for example in the case of a pure rotation where we are not changing handedness, we do not need to flip the sign of the angle.

Of course, we'd like to avoid, in the case of a quaternion, actually extracting the angle, and the axis, which can be expensive.

Since a quaternion stores a rotation (theta,v) (say) as:

[cosine(theta/2), sine(theta/2) * v], then we wish to compute:

[cosine(theta/2), sine(theta/2) * Tv]

(We'll deal with the angle flip shortly)

But since sine(theta/2) is a constant, we can move this into the computation sine(theta/2) * Tv and rewrite this as T * sine(theta/2)v. Thus we need only apply T to the imaginary component of the quaternion.

newImag = T * oldImag,

and the changing the sign of the angle, as discussed in the hkQuaternions subsection of the "Positive Rotation Rule Conversion" section only changes the real-component (i.e. the cosine(theta/2)). So if we need also to flip the sign of the angle, we do this by flipping the sign of the real component.

One way of determining whether there is a reflection component in the matrix if you do not already know (but you should), is to examine the determinant. If it is -1, then there is a reflection, and you will need to flip the angle. For example, here is a Havok function which will convert an hkQuaternion given a rotation matrix which may have reflection in it:

```
void convertQuaternionToNewSpace(const hkRotation& rotToNewSpace, hkQuaternion &q)
{
hkVector4 imag;
imag = q.getImag();
imag.setRotatedDir(rotToNewSpace, imag);

q.setImag(imag);

// Need to know whether there's a reflection here or not, so calculate determinant.
{
hkVector4 r0; r0.setCross( rotToNewSpace.getColumn(1), rotToNewSpace.getColumn(2) );

hkSimdReal determinant = rotToNewSpace.getColumn(0).dot3(r0);

HK_ASSERT2(hkMath::fabs(hkMath::fabs(determinant) - 1) < 1e-2f, "Matrix has scaling in it!");
 // If close enough to -1.
if(hkMath::fabs(determinant + 1) < 1e-2f)
{
 q.setReal(-q.getReal());
}
}

}
```

**When To Convert**

There are in effect two ways to sync your renderer/game engine with Havok, assuming that you need to do some data munging to swap between Left/Right handed, or even just to rotate you data so that your preferred canonical axis represents "up". You can either:

- Method 1.Export all your Havok data straight from the modeller in "modeller space", and ensure that at run time whenever you get or set Havok data, you apply the relevant conversion as specified above.
- Method 2. Apply the conversion as you export (just like you do for other data), in which case no run-time conversion is needed because Havok's data is already in the same space.

**Note:**

Note: Both these examples assume you are not changing the "Positive Rotation Rule" convention on export. If you are also changing this rule, you will have to apply the conversion process described in the section "Positive Rotation Rule Conversions".

It is far, far preferable to use method 2 because that way the export path is separated from your game runtime. It could be very confusing trying to view values in a debugger which were in two different "spaces", but which represented the same point. This will happen if you use method 1, so it is to be avoided.

Unless you have an additional "munge" transform which you concatenate into e.g. your camera matrix, using method 2 will ensure that Havok data and your game data are both interchangeable. For example, after setup, if using Havok to simulate the world, the transforms of all rigid bodies can be extracted directly for display (assuming you use the "common" matrix format – if you use another format you will have to do some additional conversion).

### Putting It All together: A Concrete Example

To put all this theory to the test, let's attempt a concrete example which requires us to convert data from a right-handed system (in this case 3DSMax), to a left-handed system (in this case DirectX), including full integration with Havok.

There is an example app called directxexample (located in the demo/simpleapps directory) which implements the conversions discussed here. It is a DirectX application, integrated with the Havok SDK, and the Havok Visual Debugger. You will need DirectX installed to run the app. You will also need to get the Visual Debugger Client running to view the Havok "world" being simulated. It is necessary to get this example up and running so that you may follow the description below.

To build the DirectX app, open the .dsw/ .sln provided, and compile.

To get the visual debugger up and running, see the Advanced Profile Analysis section.

The example we shall use has 3DS Max as its modeller. There are two MAX files, hand.max and rotatedhand.max, which illustrate the mesh, and "scene" we shall use respectively. If you have 3DS Max you can open them to view the data, but if not the screenshots below illustrate the setup completely.

### The Problem

The problem setup is as follows:

We have a "right-handed modeller" (in this case 3DS Max). If you open up the file hand.max, or examine the screenshot below you can confirm this. You can see that it is right handed because the World

Coordinate Axes in the figure below are displayed in the bottom right of the four view panes. They use the standard colouring convention:

R,G,B = X,Y,Z

So X-axis is Red, Y-axis is Green, Z-axis is Blue. They are also labelled. You can clearly see that the axes form a "right-handed" coordinate system, with X and Y in the plane of the hand, and Z pointing up (like the thumb). Note that the hand mesh itself is not aligned to illustrate this point, what is important is the coloured coordinate triple should be obviously "right-handed".

Figure 2.83: The "Hand" Mesh in Max (hand.max)

So, we have established the handedness of the modeller in question. Let's now set up a scene we wish to export using some obviously asymmetric data. The "hand" mesh below will be the object we place somewhere in the scene. The four panes show it from top/front/left and user camera views. Its pivot (centre) is currently over the origin of the World (i.e. the mesh object currently has the identity transform). It's clearly a "right-hand", so later when we compare other views of this mesh in both the DirectX window, and the Visual Debugger window, we can quickly sanity check things by confirming that what we see is still a "right" hand, not a "left" one!

If we take this mesh and both translate and rotate it in the world, we arrive at the scene stored in rotatedhand.max, and illustrated by the image below.

Figure 2.84: Initial Scene In Max (rotatedhand.max)

The important thing to note here is the bottom right-hand pane, labelled Camera01. This is the view of the world we shall attempt to export and match up in our DirectX renderer. We have set up our camera to point down the Z-axis, with the camera's "up" direction being positive Y.

Now we are going to export this data so that we may read it back in and display it using DirectX. DirectX is a "left-handed" system. Hence we will have to do some data conversion. For example, were we simply to export the mesh data directly, it would appear "backwards" in our DirectX display window, even if we positioned and rotated this correctly. This would be immediately obvious to us in this case because the mesh is the mesh of a person's right hand. Displaying this data in a left handed system would mean the mesh would look like a left hand!

**"Mirror" Conversion**

There are several ways to convert left handed data to right handed data. The example in the "Conversions" section above uses a "Mirror X" method. Another way to do it might be to "Swap" Y and Z values, which we shall call the SwapYZ method. These are but two examples, your own toolchain may have a different method, but in any case the "Conversions" section details how you can express this "toolchain transformation" in a Matrix format, which allows you to convert both positions and orientations. The DirectX application example illustrates both "MirrorX" and "SwapYZ" methods/toolchains, as is documented in the .cpp. It defaults to the "MirrorX" toolchain, and requires a recompile to use the "SwapYZ" toolchain. We describe the "MirrorX" toolchain example first.

As described in the section 4.2.1, three pieces of data must be mutated to effect the conversion: the mesh data, the translation, and the rotation. We deal with the mesh data first. The conversion we are performing merely flips the sign of the X component. This can be easily done before export of the mesh to a DirectX .X file using the Mirror tool. The resulting file, called handMirrored.X is located in the

application folder.

The rest of the conversions will be done on initialization of the DirectX app, and will all use the "toolchain conversion transform", stored as g_toolchainConversionTransformCM in the code. This is the key piece of data in the conversion process. In the case of the "MirrorX" conversion, the transform is:

F =

-1 0 0

0 1 0

0 0 1

which is stored in column major format as

```
float g_toolchainConversionTransformCM[16] =
{
-1, 0, 0, 0,
 0, 1, 0, 0,
 0, 0, 1, 0,
 0, 0, 0, 1
};
```

and hence can be read directly by (or written directly into) a Havok hkTransform or a DirectX D3DXMATRIXA16 (which inherits from the D3DXMATRIX structure). We already derived this transform in Sections 4.2.1 and 4.2.2. The mesh data output into handMirrored.X has (implictly) already had this transform applied to it, i.e. all the vertices have been multiplied by this transform. It remains to apply this transform to all other data, namely the position and orientation of the object, as well as the position, direction and up vector of the camera, and (for simulation purposes), the direction of gravity. There is no actual toolchain here (we use no "exporter"), so we read each piece of data directly from examination of the MAX scene as follows:

We extract the modeller position/orientation of the object as a transform using the MAXScript Listener, as detailed in the code comments:

```
// Generate Object transform in Max from ‘print selection[1].transform’
// (matrix3 [0.833395,-0.374604,0.406355] [0,0.735247,0.677799] [-0.552678,-0.564874,0.612751] [1,0.5,-0.5
  ])
// This is the transform given by the object in our toolchain, in column major data
// so, for example, the translation is given by the 13th, 14th and 15th floats.
// This is the "common" matrix format, referred to in the docs.
float g_objectInModellerCM[16] =
{ 0.833395f, -0.374604f,  0.406355f,  0.0f,
0.0f,       0.735247f,  0.677799f,  0.0f,
-0.552678f, -0.564874f,  0.612751f,  0.0f,
1.0f,       0.5f,      -0.5f,       1.0f
}
```

We also extract the camera:

```
// This is the specification of the "camera", which we have extracted from Max by examining the
// Camera using
// print \$Camera01.Transform.position
// print \$Camera01.Target.Transform.position
// and assuming Y is up (which is also the direction "gravity" is expected to be defined in the scene).
float g_CameraEyeInModeller[3] = {0.0f, 0.0f, 5.0f};
float g_CameraLookAtModeller[3] = {0.0f, 0.0f, 0.0f};
float g_CameraUpModeller[3] = {0.0f, 1.0f, 0.0f};
```

And we "know" the gravity direction we want to be the same as the camera "Up" direction (but of opposite sign!), namely in the -Y direction:

```
// This is the specification of the "gravity" direction, which we have extracted from Max by examining the

// camera "Up" direction - usually your camera "Up" will be the same as the direction of gravity for
// obvious reasons.
float g_GravityModeller[3] = {0.0f, 1.0f, 0.0f};
```

If you examine the code you will see that everywhere a Havok object is set up, the "toolchain conversion transform" is applied first, exactly as described in Section 4.3 "When To Convert", i.e. using Method 2.

The net result is that when the app window comes up you should see the following:

Figure 2.85: DirectX Window - "Mirror X" Conversion

Here the window looks identical (except for aspect ratio, and lighting) to the bottom right pane of the Max screenshot image "Initial Scene In Max (rotatedhand.max)", which demonstrates that we have correctly converted our "right-handed" data to be displayed in a "left-handed" system. I say "identical" but of course there is one small difference because the canonical axes are displayed in both cases, and you can quite clearly see that in Max the X-axis (in red) points to the right, whereas in the DirectX window it points to the left, confirming that "MirrorX" conversion has been performed.

Now the app has been set up to continuously update the transform of the hand mesh from an associated rigid body, so we can be sure that not only was Havok "set up" correctly, but also it really is running "in the same space" as the renderer. To verify that Havok is indeed correctly "synced", we can bring up the Visual Debugger Client, and take a peek at the physics simulation happening in the app. I presume that the user had read the Advanced Profile Analysis section.

First we run the VDB, and connect to the machine running the DirectX app. We should see something like this:

Figure 2.86: Visual Debugger Client Window - Initial View

This does not look like the DirectX window because we have not yet set and of the rendering options for the VDB (which is highly generic, and can be customized to render in many different ways). So now we set up the visual debugger to reflect the renderer settings of the app. There are three things to do here:

- Set the renderer to be left-handed. From the View menu at the top select
  Axis - Use Left Handed System

- Set the camera to be the same (user) camera set up by the app. As well as setting up the DirectX camera using the "converted" camera settings in the method SetupMatrices(), the app also sets up a VDB camera with identical parameters in the method StepVisualDebugger(). We can select this camera to be used by the VDB by going to the View menu and selecting

  User Cameras - DirectX Example

- (For tidiness only) Set the "Up" direction (used to set the "world plane grid") to be Y (gives an idea of what direction really is "Up", for example we expect gravity to act at right angles to this plane). Set this by going to the View menu and selecting

  Axis - Y Up

Having done these three things, the VDB should now look like this:

Figure 2.87: Visual Debugger Client Window - "Mirror X" Conversion

Again this is exactly the same (except for aspect ratio, and lighting) as the DirectX example, and if you use the mouse picking to grab and move the hand in the VDB you should see the hand in the DirectX example move in exactly the same way, proving that you have correctly set up all the Havok data to match the DirectX view. In fact you'll notice an grey block below the hand which was not visible in the DirectX window. This is a separate Havok rigid body which was added after the hand was, but no visual representation was added to the DirectX display, and the reason for doing this was so that you can turn on gravity in the app (set GRAVITY to a non-zero negative value and recompile), and see the hand fall and collide with the box, thus demonstrating that the physics is also fully working!

**"SwappedYZ" Conversion**

At this stage I hope you are convinced that the theory can be put into practice and produce the expected results. As a final example we can set the app to run using the alternative "SwappedYZ" toolchain, by recompiling the app with EXAMPLE #defined to be YZ_SWAPPED_EXAMPLE. In this case we convert from right to left-handed by swapping the Y and Z values. This transform can be expressed in matrix form as

SXY =

1 0 0

0 0 1

0 1 0

which is stored in column major format as

```
float g_toolchainConversionTransformCM[16] =
{
 1, 0, 0, 0,
 0, 0, 1, 0,
 0, 1, 0, 0,
 0, 0, 0, 1
};
```

The mesh file is already exported as handSwappedYZ.X.

If we run this example, the DirectX window looks like:

Figure 2.88: DirectX Window - "Swap Y and Z" Conversion

and we can see that again the hand is in exactly the right place, i.e. it looks identical (except for aspect ratio, and lighting) to the bottom right pane of the Max screenshot image "Initial Scene In Max (rotatedhand.max)". Note that here the X-axis (red) still points to the right as it has not been changed by the transform, but that Y (green) and Z (blue) have been exchanged (Y now points directly out of the screen, so is unfortunately not very visible!)

Once again we double-check that Havok has been correctly initialized by hooking up the VDB. You must re-select the user camera if you have not closed down the VDB to ensure it picks up the new camera values. Note that this time our renderer has Z as "Up", so we reflect this in the VDB by right-clicking and selecting

Axis - Z Up

Figure 2.89: Visual Debugger Client Window - "Swapped YZ" Conversion

You can see that this correctly reflects the DirectX window view as expected. Again you may wish to turn gravity on/off, and /on grab the hand in the VDB to see that it updates correctly win the DirectX window.

These examples together with the detailed explanations of the transform derivations in previous sections should enable you to correctly adapt Havok to any toolchain conversion process.

#### 2.6.1.5 Appendix A : Some references, and relevant quotations.

DirectX:

Left-handed coordinates:

http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dx8_vb/directx_vb/graphics_intro_8w8j.asp

“Microsoft Direct3D uses a left-handed coordinate system”

Left-hand rule (clockwise):

http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dx8_vb/directx_vb/graphics_intro_8wqb.asp

“Angles are measured clockwise when looking along the rotation axis toward the origin”

OpenGL:

No handedness.

http://www.opengl.org/documentation/specs/version1.4/glspec14.pdf

p 252. Appendix B, #15 “OpenGL does not force left- or right-handedness on any of its coordinates

systems."

Right-hand rule:

http://www.opengl.org/documentation/specs/version1.4/glspec14.pdf

p 35, Section 2.10 Coordinate Transforms. "The computed matrix is a counter-clockwise rotation about the line through the origin with the specified axis when that axis is pointing up (i.e. the right-hand rule determines the sense of the rotation angle)."

## 2.6.2 Continuous Physics

### 2.6.2.1 Introduction

Continuous physics is Havok's concept of high quality rigid body simulation. Continuous physics means that we do not ignore what is happening between two physical time steps, but handle all collisions at the exact time of their impact if necessary.

Most existing physics engines perform computations based on discrete states. That type of simulation will be referred to hereafter as discrete physics.

Discrete simulation breaks physical motion into discrete physical time steps. It performs collision detection at those discrete times and employs simple integration methods to advance the state of bodies in time. So all physical operations are handled at one physical synchronous instance in time (later referred to as PSI). Collision detection is only based on the relative position of bodies at each step in isolation. For small or fast objects it is hard to determine coherency between frames, therefore the simulation can easily break in these cases.

The continuous physics core is based on motion state representation, which contains information from two consecutive frames. With that information we can interpolate the state of simulated bodies in-between frames and perform continuous collision detection.

With such precise collision information we employ our solver to chronologically handle each occurring collision at its exact time of impact (TOI). This ultimately brings our simulation to a physically correct state in the following frame.

Using continuous physics is very simple. Even if you are new to Havok Physics 2012 you need only read the next section on basic API parameters to get started. Then you should be ready to start experimenting with the simple demos. Subsequent sections explain in more detail the concepts behind continuous physics, and are meant for more advanced users of the engine.

### 2.6.2.2 Fundamental Concepts

The following sections present an in-depth description of the concepts behind continuous physics. It is not necessary to understand this part in order to use the engine at a basic level.

Havok Physics 2012 has implemented new simulation loops and collision detection algorithms. This section progresses from the basics of motion data representation to the new functionality of the collision detector, and explains our implementation of the continuous physics simulation loop.

### Transform and Swept Transform

hkTransform is a representation of a body's state i.e. its position and orientation at a given moment in time. In discrete simlulation that position and orientation are constant for the duration of a frame and are updated at its end.

In continuous physics we need to know the body's precise position and orientation at any moment in-between discrete 'frame ticks'. So we represent the rigid body's state by an `hkSweptTransform` structure which basically stores the position and orientation at two chosen moments in time.

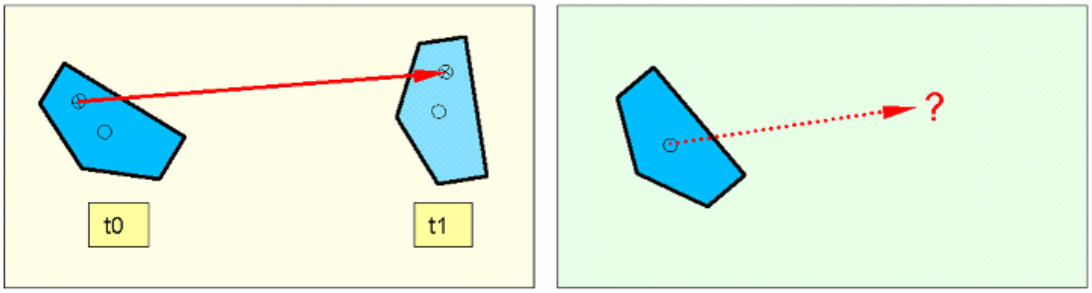



Figure 2.90: Illustration of a hkSweptTransform, which stores position and orientation information at two moments in time. Information about mass centre allows for proper interpolation of position in time. In contrast, discrete physics' representation uses position and orientation information from one time moment. Motion State values from consecutive frames are therefore not explicitly linked in any way.

`hkSweptTransform` also stores location of the body's mass center allowing for proper interpolation of position in time. In contrast, discrete physics' representation uses position and orientation information from one time moment. A body's state information from consecutive frames is therefore not explicitly linked in any way.

A `hkSweptTransform` stores:

- Time `t0` and `inverseDeltaTime`, such that `t1 == t0 + 1/inverseDeltaTime`.
- The body's mass centre position and the body's orientation at `t0` and `t1`.
- The center of mass in the body's local reference frame.

The position of the mass center is required to properly interpolate the body's transform between the two time endpoints.

To get body's transform at arbitrary time `tx` (where `t0 <= tx <= t1`) we interpolate the position of body's mass center and interpolate its orientation quaternions. Here's a simplified code snippet:

```
// Performs linear interpolation of positions
post = pos0 * (1 - t) + pos1 * t
    // Performs linear interpolation orientations
qt = lerp(q0, q1, t)
    // Calculates position of the body's reference point
    // using the mass center stored in local space mcPos
post = post - qt * mcPos
```

**Note:**

One may ask why we need the center of the mass information and why we can't just associate the shape's origin with the mass center of a rigid body. The necessity for that information becomes clear when we consider `hkpShapeCollections`, where a number of shapes is associated with one rigid body. Collision detection is performed for each of those shapes in isolation from others and each of those shapes may be positioned far from the body's mass center. In such case the trajectory travelled by the geometrical center of the shape will not be linear and we need explicit information about the mass center to interpolate the shapes' position properly.

### State Integration

To integrate a body's state we use simple Euler integration. At each 'frame tick' (also referred to as a PSI) assume that the body has reached its final position at `t1`. This becomes the initial position of the updated `hkSweptTransform`. After applying all gravity, constraint forces, reaction forces and impulses we get the body's final velocity that we can use to calculate its end position at `t1`. This new end position is perceived only as predicted or potential because it may be modified later when in-between-frames collisions are processed (we refer to such collisions as TOI or time-of-impact collisions).

### Collision Detector

The collision detector performs two tasks – it verifies the motion of all simulated bodies throughout the frame step and it generates contact points between those bodies for the specified end time of the frame step.

### Verifying Motion and Generating Time-of-Impact Events.

The collision detector is given two `hkSweptTransform`s, which overlap in time. It interpolates the transform of each of the associated bodies along time and checks for collisions.

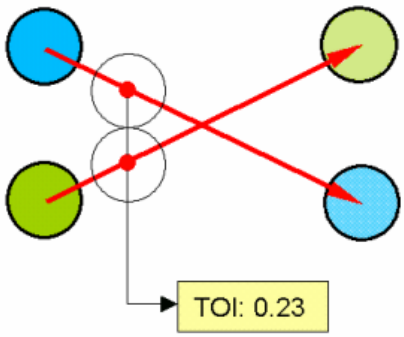



Figure 2.91: As the positions of the two bodies are interpolated they start moving towards each other. The moment when they touch each other is the time-of-impact and it is reported in a TOI event. If we interpolated their positions further the objects would interpenetrate significantly and eventually pass through completely.

When interpenetration of bodies is detected the collision detector generates a collision event. Such events are called time-of-impact (TOI) events. They contain the time of reported collision/impact and references to the two colliding bodies. Collision detection is performed for all pairs of object and ultimately yields a list of collision events.

### Calculating Contact Points

Contact points generated by the collision detector are always calculated for one given moment in time. This operation is done independently of TOI events. By default, the collision detector calculates contact

characteristics at the end of the current frame.

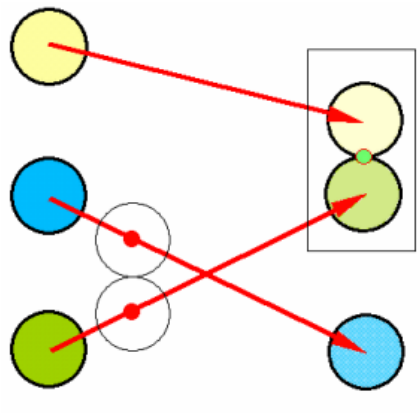


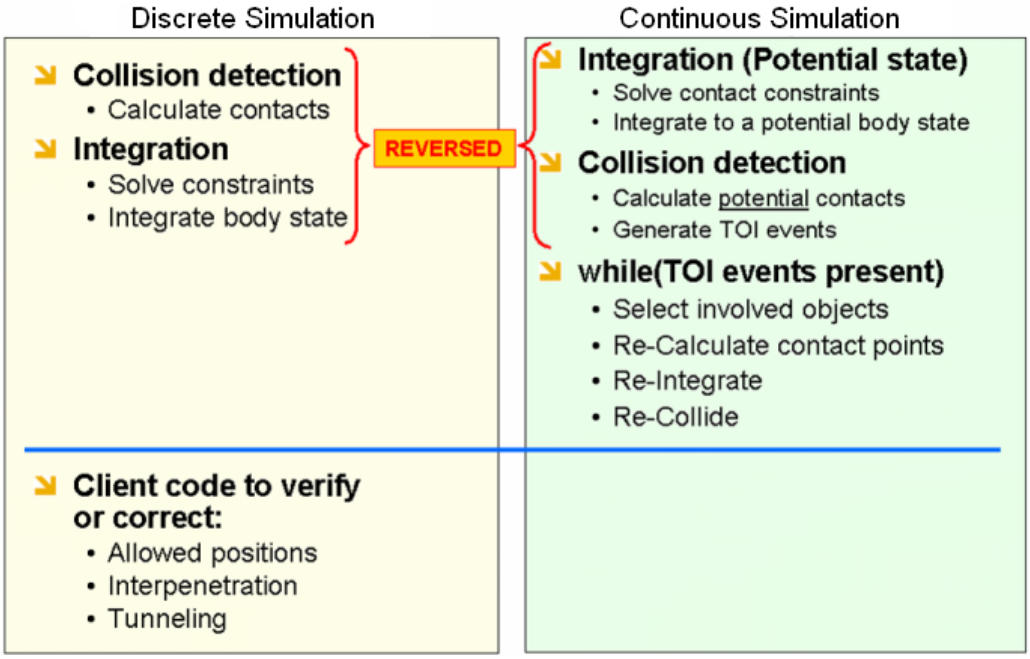
Figure 2.92: Pairs of objects found in proximity or contact at the end of the time step (objects selected within the box) generate contact points (small dot).

At that time the collision detector does not know whether or how any occurring TOI events are going to be handled. It therefore generates contact information, which may be used by constraint solver in the next integration step (or may be overridden with newer information).

This part of collision detector functionality corresponds to the whole of the discrete collision detection in Havok. Note that thanks to TOI events there is no longer a need to have a large additional collision tolerance in hkpWorld. For discrete collision detection, the tolerance layer was used to generate contact points early to avoid object penetration or tunneling in the following frame. However, it is still a good idea to have a big (∼10cm) collision layer as it can avoid CPU intensive TOI events. However a large collision tolerance can have the side-effect that objects bounce off each other before actually hitting each other. To avoid this side-effect, the collision tolerance can be reduced to a small value (around 1 cm), however more CPU time will be used to handle the additional TOIs.

**Continuous Simulation**

The picture below highlights differences between discrete and continuous collision detection in Havok. Collision detection and Integration are common to both solutions. They are referred to as the Physical Synchronous Instance Step. The final while loop in the continuous collision detection is referred to as the TOI-handling loop. It handles occurring time-of-impact events chronologically, and is intended to replace some of the user's motion verification code usually created outside of the engine.




**Physical Synchronous Instance Step**

Physical Synchronous Instance (PSI) Step corresponds to the discrete simulation pass – it is responsible for performing a synchronized collision detection and integration for all bodies.

The discrete simulation approach is based on processing information from one physics frame only. Given the positions of bodies it performs collision detection and finds any colliding or overlapping object pairs that need to be corrected. The constraints solver uses that information to apply proper forces to the bodies. Finally the correct new state is integrated into the system.

In contrast, the continuous simulation approach assumes that it already has proper contact information for the current state at the beginning of each pass of the simulation loop. (Even if it doesn't, any unacceptable penetrations will be reported via TOI-events before the next frame is reached.) It runs the constraint solver and integrates the bodies' states first. This yields their predicted or potential final states.

Collision detection is responsible for verifying the bodies' movement from their initial to their final state. If there are no interrupting TOI events generated by the collision detector then the predicted state is validated and remains unchanged. Otherwise it may change as TOI events are handled (as is described in the following paragraphs).

**Time-of-Impact Event Handling Loop**

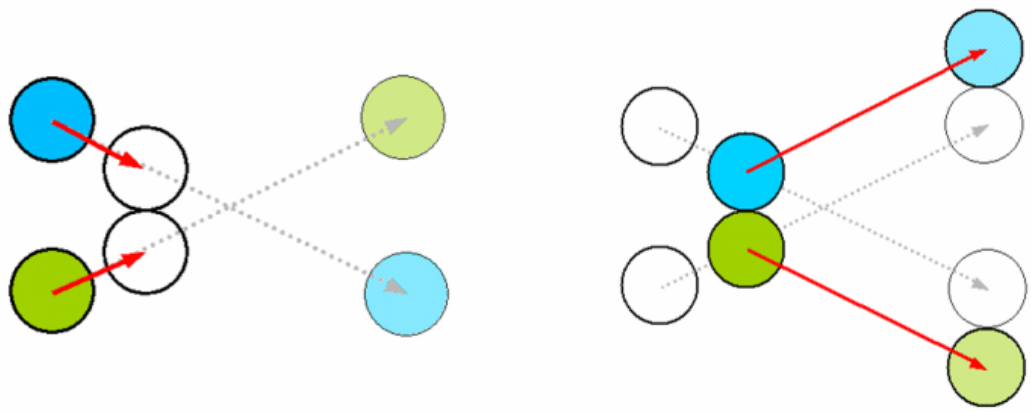
All TOI events for the entire world are kept on one list. The TOI-handling loop processes the events chronologically according to their time-of-impact.

In each TOI-event the motion of some bodies is altered. For those bodies, collision detection is performed again which may result in removal, modification, or addition of some TOI-events.

**Solving Collision Impact Response**

For each event the two colliding bodies are identified and a simple collision response algorithm is used to process the impact in isolation from the rest of the world. (It will be shown in the Optimizations section below that we may also process a greater number of bodies in one TOI event.)

Once the proper reaction is calculated, the motion states of the bodies must be recalculated. We set time `t0` to the moment of impact and use the corresponding interpolated transform as the new initial transform at `t0`. Then we integrate the state to `t1`. Note that this time we integrate only by a fraction of the original PSI step.



**Note:**

As the result of TOI-response, `hkSweptTransform`s of all bodies end at the very same moment in time – the time of the next PSI. However, their `t0` values may differ.
Since each processed TOI event has a later or equal time-of-impact, each processed event always happens in between the beginning and the end of the `hkSweptTransform` time of every simulated body.

### Partial Collision Detection

After altering the motion of bodies according to the calculated collision response we run collision detection again to verify the newly calculated trajectories of the bodies. We must therefore collide each of the colliding bodies with each other and with the rest of the world's objects.

This generates a list of new TOI events, which are inserted to the current world's list of TOI events to be handled.

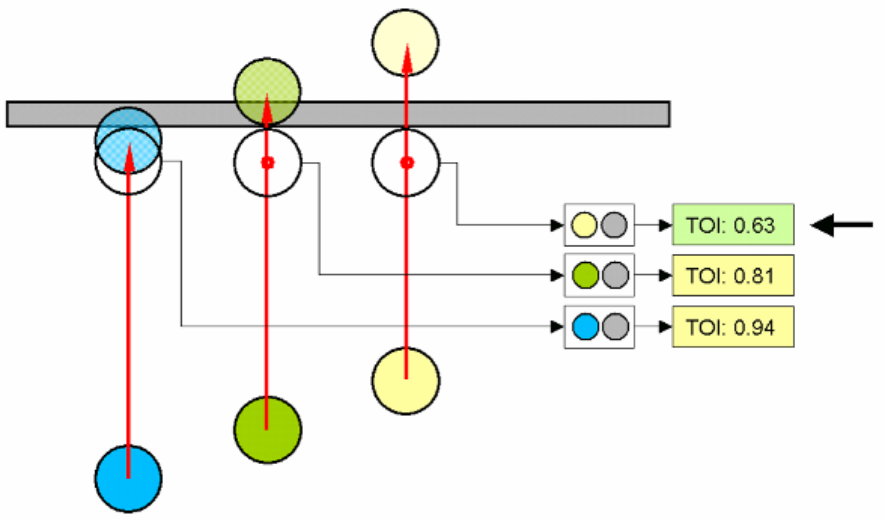
Also notice that previously calculated events, which referred to the reintegrated bodies, are now invalid. Such events are either removed or removed-and-replaced by the newly calculated events with new time-of-impact values.

### Graphical Examples

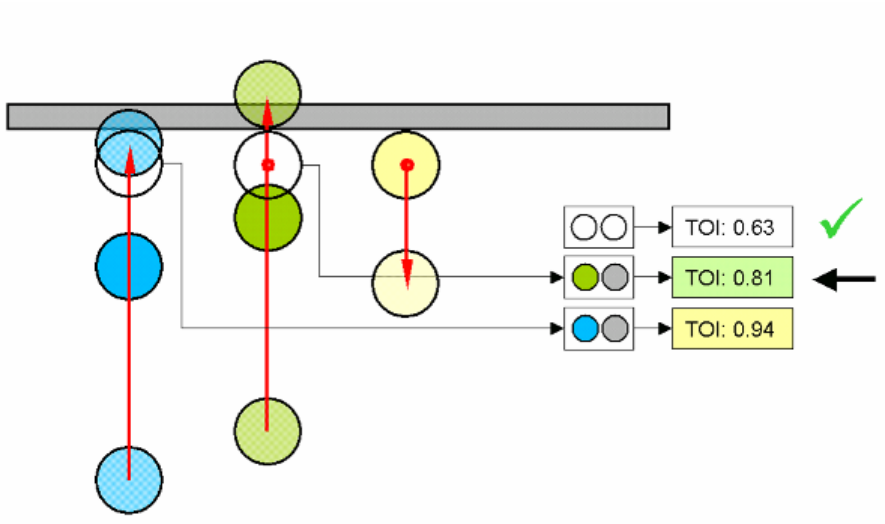
The following examples illustrate the operation of the collision detector in various situations.

### Independent Collisions

Three spheres are moving in parallel, and are stationary relative to one another. In the picture below , the spheres at the bottom represent position at `t0`, and the uppermost spheres represent position at `t1`. They are about to hit a fixed wall (gray block).




The collision detector determines potential interpenetration and generates a TOI event for each of the spheres (represented by a black ring on the movement path of each of the bodies). Times for each of the TOI events are displayed in the table on the right. The system handles the earliest TOI event (for the rightmost sphere) first, marked by an arrow on the right side.




Once the first TOI collision is handled, the rightmost sphere's motion is reintegrated and the collision detection is re-run. It does not return any new TOI events in this example. The processed TOI event is taken off the TOI event list. The next earliest TOI is processed next.

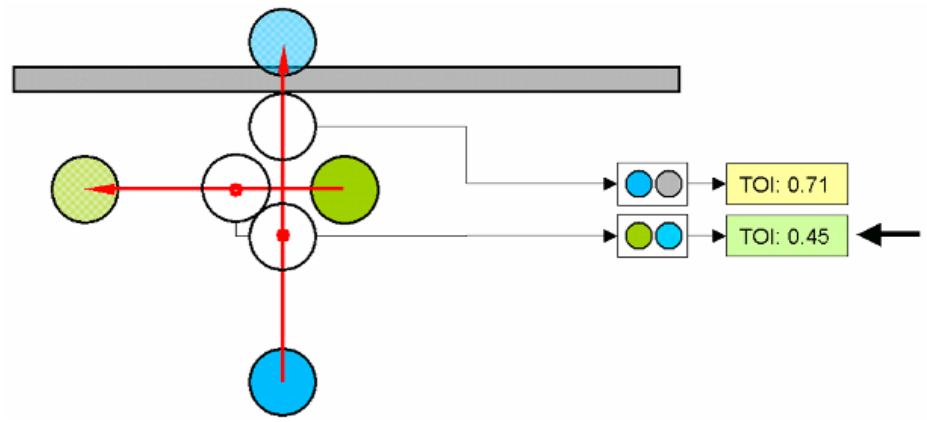
**Multi-body Collisions**

The symbols used here are analogous to those in the example above.

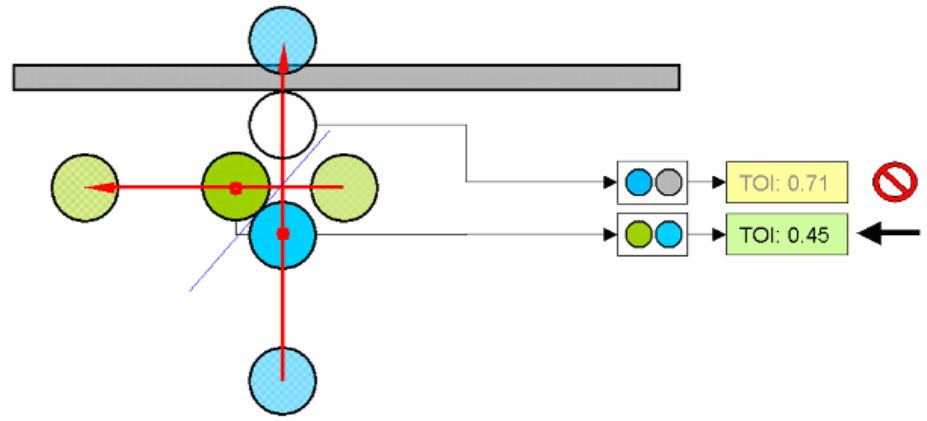
This example presents two spheres about to collide. Moreover, if that collision weren't handled, the blue sphere (the one with vertical movement) would hit the fixed wall.

**Note:**

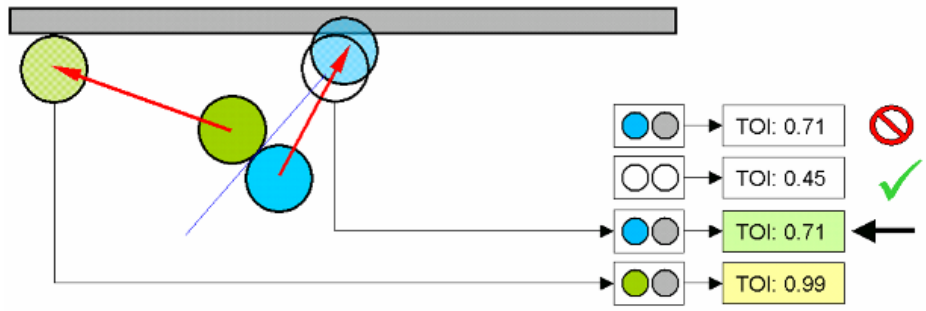
Every body may have a number of `hkpCollisionAgents` with other bodies. Each of them may generate a TOI event. Therefore one body (like the blue sphere in the example) may end up with a list of a few TOI's.




The sphere-sphere collision is handled first. Collision characteristics are calculated, and both bodies change in motion. Therefore the TOI-event between the blue sphere and the fixed wall will be removed from TOI list.




Collision detection is run with the newly calculated `hkSweptTransforms`. Each sphere generates a new TOI collision with the fixed wall. One event replaces the previously removed one; the other event emerges as a result of the handled collision.




After both TOI events are handled, we receive the correct world state for the next PSI.

**Simplified Continuous Simulation**

Havok features simplified TOI handling dedicated to collision of non-critical objects against landscape. That simplified TOI handling is an inexpensive way to keep fast moving objects in the simulation world. Instead of properly solving the collision, simplified TOI only moves the dynamic body to the time-of-impact and freezes its motion there till the end of the frame.

The simplified TOIs are based on normal continuous collision detection performed each frame. Their handling however is only limited to applying response impulses and freezing the motion. This way processing time is saved by skipping localized TOI solving (see "Localized Solving" further below), reintegration of motion, and also broadphase and narrowphase collision detection.

Full TOI handling can be performed multiple times during a frame for a single body pair. The simplified TOI handling is meant to eliminate the cost of additional collision detection calls. Since the motion of the dynamic body gets frozen at the first simplified TOI, no more TOI will occur for that pair during that one simulation frame.

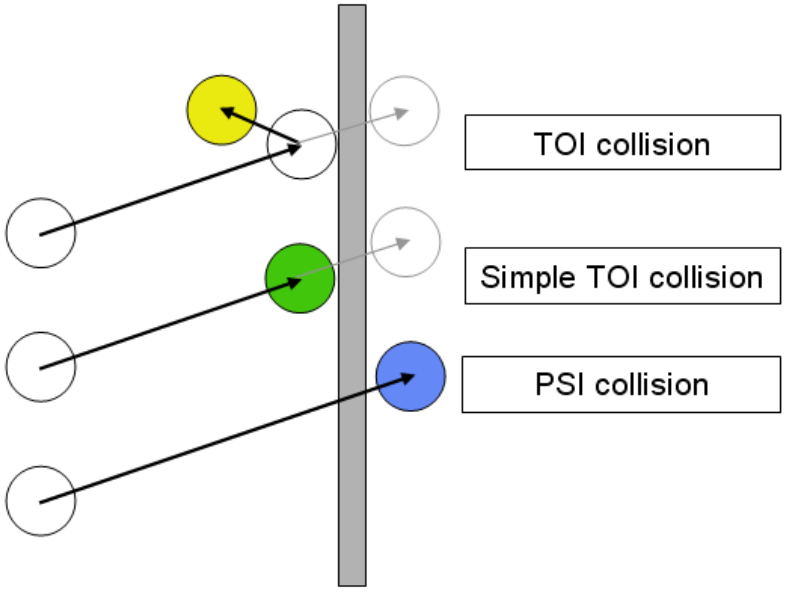



Figure 2.93: Results of similar collision being handled in different ways. PSI collision does not stop the object form tunneling through. Simple TOI stops the object, but also looses travel-distance. Full TOI collision bounces the object in mid-simulation-frame.

Simplified TOI handling is approximately 10 times cheaper then full TOI handling. It is e.g. a good way to handle debris objects that are not critical but are important to stay in the simulation world to add visual complexity.

**Landscape collisions only**

Pictures below explain why we only use simplified TOI handling for collisions of dynamic object against static landscape. If used for collisions between two dynamic objects it wouldn't do its work.

Here continuous collision detection is performed between all bodies. The green and yellow bodies are dynamic, and the gray box is a fixed part of the landscape. Only one simplified TOI event is generated.

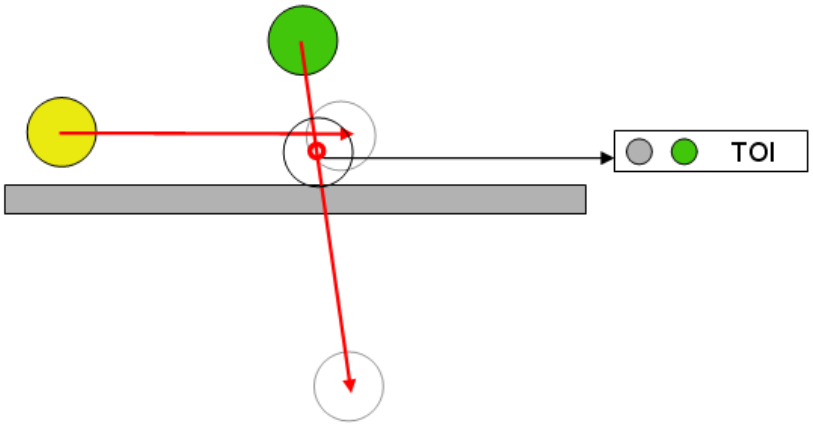



Figure 2.94: Collision detection and TOI's between two dynamic bodies and fixed landscape.

As the result of the event handling, the green body is backstepped and frozen. However, since collision detection is not recalculated, the yellow body just tunnels through without warning.

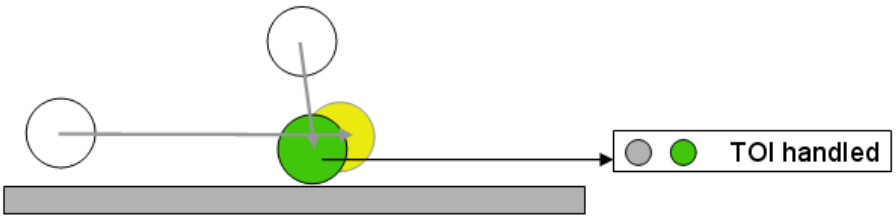



Figure 2.95: With simple-TOI interaction bodies could tunnel through.

The solution is to use normal TOI handling between the dynamic bodies. And as you will see further below, the collidable quality type `DEBRIS_WITH_SIMPLIFIED_TOI` is characterized by simplified TOI collisions with landscape, and full TOI collisions with fast-moving bullet-type objects.

This solution guarantees that TOI events will be generated and handled properly for the dynamic objects. Still however, this reaction may result with new penetration with the landscape. This time however, those penetrations will not be prevented, as we only support one simplified TOI with the landscape per frame.

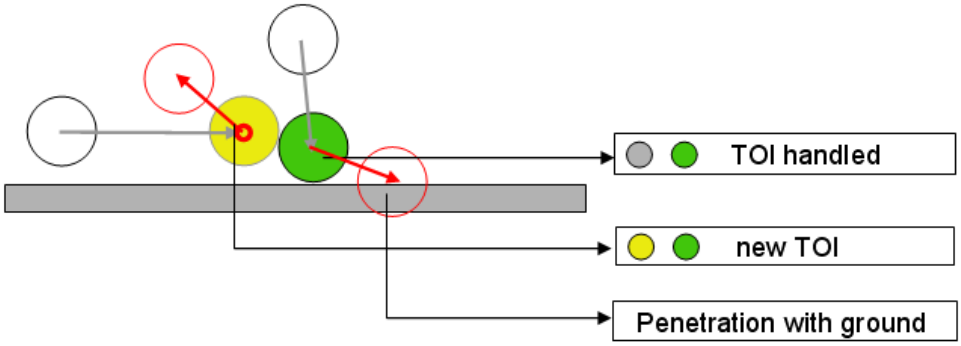



Figure 2.96: With full TOI interaction bodies bounce off each other. Note that subsequent penetration with the ground may occur.

**Violating constraints and energy added**

In simplified TOI handling bodies are backstepped, i.e. they "loose" their time or travelled distance. Such a backstepped body may be a part of a constraint system, e.g. a ragdoll. When some bodies are backstepped and some are not, their linking constraints may get significantly violated. On the subsequent simulation frame this may result in a large energy impulse added to the system.

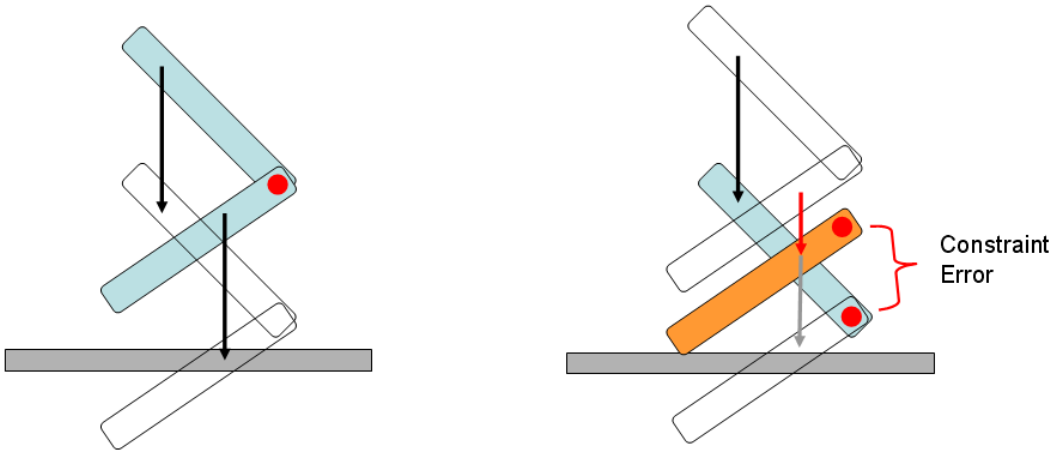



Figure 2.97: One body is backstepped and is marked orange. This results in a significant constraint error.

**Constraint Priorities**

Simplified TOI's are meant to limit the amount of collision processing for a given pair of bodies. For that reason we don't want the simplified-TOI interactions to take part in any other full TOI handling. For that reason any constraints linking to a `DEBRIS_WITH_SIMPLIFIED_TOI` body must be of the lowest PSI priority. The engine will assert when adding or handling constraints of higher priorities.

Note that using higher priority constraints is likely to result in `DEBRIS_WITH_SIMPLIFIED_TOI`-type bodies penetrating through landscape. That's because they can get activated when solving other TOI's, but their interaction with the ground is not processed at that time.

For more information on constraint priorities see "Constraint Priorities" further below.

#### 2.6.2.3 Priorities and Quality Types

Constraint priorities and the quality of objects control the effort Havok Physics 2012 puts into proper simulation of systems of bodies.

When simulating a massively populated world, we can often distinguish between a few critically important (gameplay-relevant) objects and numerous other non-gameplay objects that simply add complexity to the world to make it more realistic.

**Constraint Priorities**

Constraint priorities decide whether and how constraints are handled. These are the available priority settings provided by `hkpConstraintInstance::ConstraintPriority` (listed in order of lowest to highest):

- `PSI` - Collisions are handled only in PSI steps. TOI events are **not** generated. Objects are allowed to interpenetrate.
- `SIMPLIFIED_TOI` - Collisions generate TOI events. This collision type is only used for debris colliding with fixed landscape. It is resolved by backstepping and freezing the debris object at the first

moment of impact with landscape. This type of continuous collision response is very cheap, and still quite efficient in preventing debris from leaving the world. See Simplified Continuous Simulation for more details.

- `TOI` - Collisions generate TOI events.
- `TOI_HIGHER` - Collisions generate TOI events. Contact constraints have higher priority in the constraint solver. This prevents interpenetration even when other objects 'push' bodies into each other. An example would be a very heavy car pushing a hand of a ragdoll into a wall. In this case we want the hand to penetrate the car but not the wall. Therefore wall-to-hand priorities should be set to `TOI_HIGHER` and car-to-hand priorities should be set to `TOI`. See the Collision Quality Level dispatch table in section Object Quality Types for details.
- `TOI_FORCED` - Same as `TOI_HIGHER`. Additional checks are performed when this collision is touched in a TOI event. If the constraint is violated then a `hkpToiResourceMgr::cannotSolve()` callback is fired.
- `CHARACTER` - Same as `TOI_FORCED`. Uses additional tolerances to enable more accurate generation of contact points.
- `n/a` - Collisions ignored.

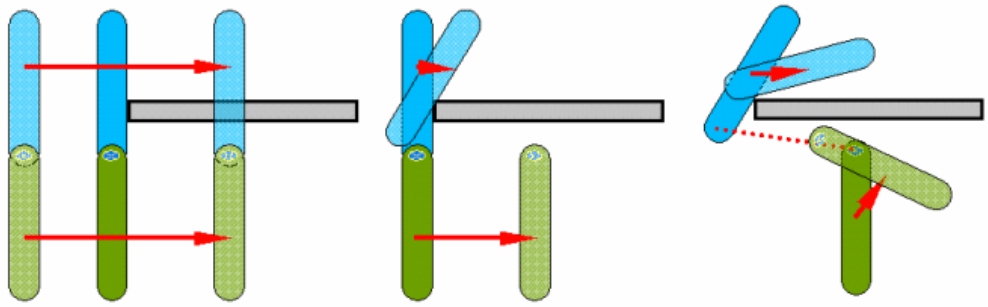
**Note:**

`TOI_HIGHER` and `TOI_FORCED` types should be used only between a moving and a fixed object – never between two moving objects (see this example on why this is the case). Both collision levels generate higher-priority contact constraints. Such constraints are used to prevent game play-critical objects from leaving the game world. For this reason it is wise to guarantee no conflicts between higher-priority constraints, and the easiest way to do it is to allow them only between pairs of fixed and non-fixed objects.

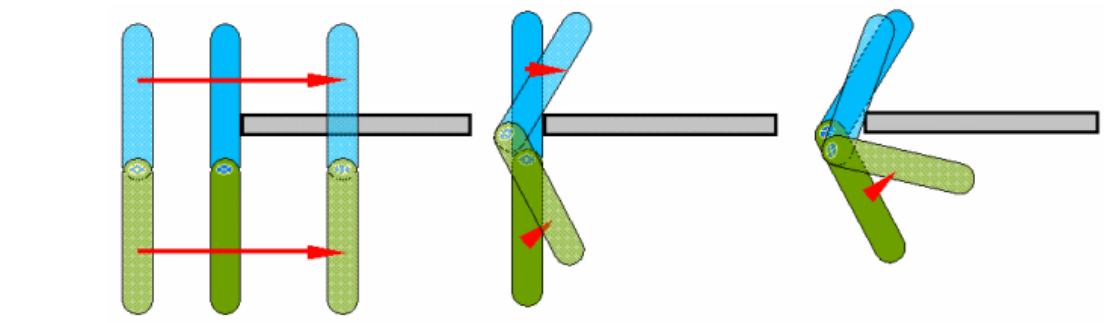
### PSI and TOI Priorities

TOI constraints add extra quality to continuous simulation. When a body collides in a TOI event, its attached TOI-priority constraints are processed and may affect movement of other linked objects. If the lower quality PSI constraints are used, parts of constrained systems, e.g. ragdolls, may drift apart during TOI events.

The schematic below presents a hinge hitting a fixed obstacle with one of its attached bodies. On the left side we see the hinge at physical frame 0, and on the right side we see the hinge at the physical frame 1. The hinge hits the fixed obstacles in between these two physical frames. If the hinge uses a PSI constraint, the hinge is not touched in the TOI event and as a result the two bodies drift apart.



In case of a TOI constraint, the hinge is processed in the TOI event and as a result the error is much less noticeable.



The following screenshots present PSI- and TOI-constraint-based ragdolls hitting a fixed obstacle.

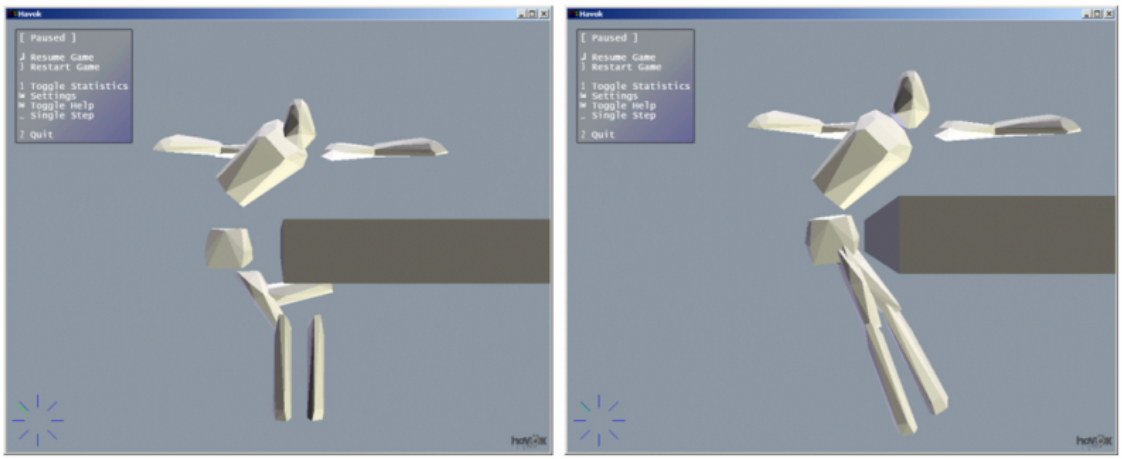


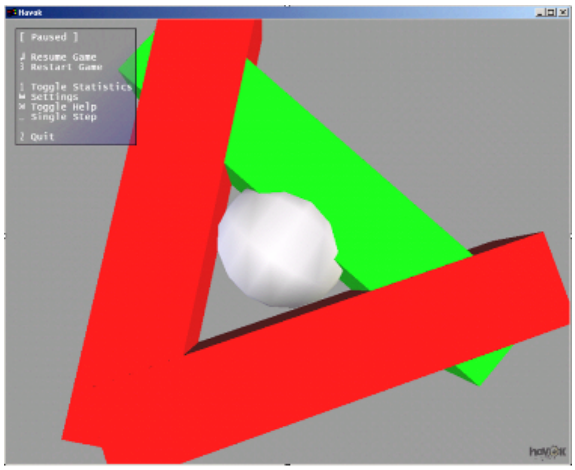

Figure 2.98: Left: PSI-ragdoll; Right: TOI-ragdoll. Note the clearly visible drift of the ragdoll's lower legs on the left picture.

**The TOI_HIGHER / TOI_FORCED Priority**

Constraints are assigned either a normal (PSI / TOI) or a higher priority level. In unsolvable or conflicting cases normal-priority constraints are violated before the high-priority ones. If two constraints are assigned the same priority it is ambiguous which will yield first.

The example below shows an 'impossible' scenario, where three boxes are being keyframed into a dynamically controlled sphere. Here, the sphere has higher-priority contact constraints with the red boxes and a normal-priority contact constraint with the green box. Note that only the normal-priority constraint is violated as the green box slides toward the ball and into a conflicting position.

It is also important to note that in such a situation penetration can still occur between objects using higher-priority constraints (in this case between the sphere and the red boxes). This is due to the fact that TOIs are solved in a sequential order and do not know about any other TOIs that might get solved later. In a situation where two TOIs with similar internal settings fight each other, the result might still be ambiguous. Please see the section on the allowed penetration depth for details on these inner workings and on how to tweak them to achieve the desired results.




**Object Quality Types**

In a game you may want to use different quality settings for different kinds of interaction.

As it might be inconvenient to set a quality level using callbacks for every occurring interaction, Havok categorizes objects into groups and uses the `hkpCollisionDispatcher::CollisionQualityLevel` table to determine the quality level of each collision.

For example, you may expect critical game play objects (e.g. a key), to have solid high-priority collisions with the world so that they never accidentally sink into ground where they cannot be found. On the other hand you may want to allow penetrations between the world and 'debris' objects, which only serve the purpose of enriching the visual side of the game, e.g. small car parts flying off from a car crash. Moreover, you might still expect those debris objects to interact with bullets you shoot.

Objects are therefore categorized as follows:

- `FIXED` - For static world elements. Especially objects represented by a triangle mesh.
- `KEYFRAMED` - Animated object with infinite mass.
- `KEYFRAME_REPORTING` - Use this for moving objects with infinite mass which should report contact points and TOI collisions against all other bodies.
- `DEBRIS` - Low importance objects adding visual quality.
- `DEBRIS_SIMPLE_TOI` - Low importance objects without gameplay significance. These are the same as `DEBRIS` with the exception of using simplified TOI's for collisions against landscape to prevent them from penetrating.
- `MOVING` - Regular objects populating the world. Furniture, etc.
- `CRITICAL` - Gameplay-critical objects that are never allowed to penetrate or leave the game world.
- `BULLET` - Fast-moving projectiles.
- `CHARACTER` - For use with Rigid Body Character Controllers.

Please see above for a detailed overview on the available types of collisions/interactions:

| | FIXED | KEYFRAMED | KEYF.REPORT | DEBRIS | DEBRIS+TOI | MOVING | CRITICAL | BULLET | CHARACTER |
|---|---|---|---|---|---|---|---|---|---|
| CHARACTER | CHARACTER | TOI | TOI | PSI | PSI | TOI | TOI | TOI | TOI |
| BULLET | TOI_HIGHER | TOI | TOI | TOI | TOI | TOI | TOI | TOI | TOI |
| CRITICAL | TOI_FORCED | TOI | TOI | PSI | PSI | TOI | TOI | TOI | TOI |
| MOVING | TOI_HIGHER | TOI | TOI | PSI | PSI | PSI | TOI | TOI | TOI |
| DEBRIS+TOI | SIMPLE_TOI | PSI | PSI | PSI | PSI | PSI | PSI | TOI | PSI |
| DEBRIS | PSI | PSI | PSI | PSI | PSI | PSI | PSI | TOI | PSI |
| KEYF.REPORT | TOI | TOI | TOI | PSI | PSI | TOI | TOI | TOI | TOI |
| KEYFRAMED | n/a | n/a | TOI | PSI | PSI | TOI | TOI | TOI | TOI |
| FIXED | n/a | n/a | TOI | PSI | SIMPLE_TOI | TOI_HIGHER | TOI_FORCED | TOI_HIGHER | CHARACTER |

Figure 2.99: Collision Quality Level dispatch table.

**Notes:**

- No fixed and keyframed objects interact with each other.
- Debris can interpenetrate with any object but responds to bullet hits properly.
- Critical objects are forced to not penetrate with fixed objects, and do not have higher-priority contact constraints with any other objects.
- Moving objects generate TOI events with fixed, keyframed, critical and bullet types but are allowed to penetrate slightly with other moving objects or debris.

#### 2.6.2.4 Performance Optimizations

Havok implements a number of algorithms to maintain high performance and limits spikes in the number of TOI events generated. Further optimizations to the continuous simulation that can be done by users are discussed in the Optimizations chapter.

**TOI Solving Causes Extra Performance Hit for Discrete Collisions**

When two bodies of quality type MOVING interact, one might assume that their collision detection is only run once per frame. This is not the case if one of the bodies is involved in a TOI collision.

All bodies that take part in a TOI collision may have their motion affected and their collision detection is performed again.

The said two MOVING bodies may re-collide if one of them is involved e.g. in a TOI collison with fixed landscape.

Note that when there are many MOVING-quality objects in proximity, then a single TOI may cause a lot of collision detection work. Additionally, since TOI's are handled in a single thread, this collision detection work will not be spread across several CPU's causing even a greater performance hit.

#### 2.6.2.5 Asynchronous Simulation

A discussion of stepping the world synchronously and asychronously is presented in the section Stepping the Simulation Forward. In continuous simulation the advanceTime() call (referenced in that section) will advance the time of the simulation forward and solve any TOIs that arise between the current

time (from the last time advanceTime was called) and the time it tries to advance to. This is visualized below for a ball travelling from one PSI to the next, with a frame time marker set between the frames. In discrete simulation advance time simply advances the world current time to "Frame 2" below:

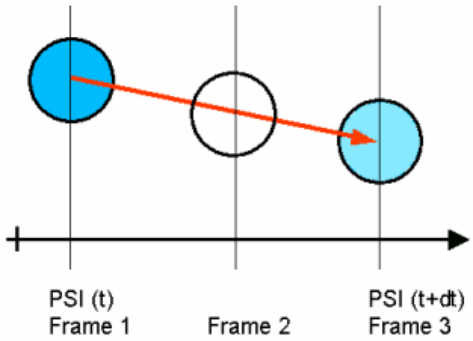



Figure 2.100: To display an object at an asynchronous point between two psis, you can set the frame marker, then call approxCurrentTransform to interpolate between the 2 psis.

In continuous simulation, the advanceTime() function will additionally process all TOIs up to the time it is advancing to. This is shown in the diagram below. The first time advanceTime() is called it will process TOI1 to advance to the "Frame 2" marker.

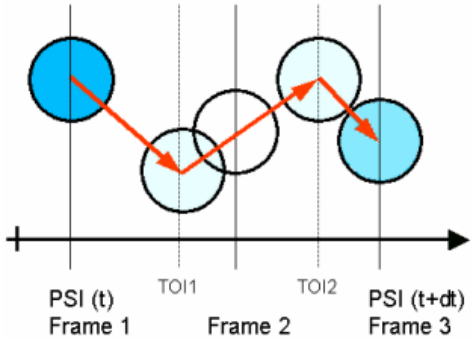



Figure 2.101: In continuous simulation, TOI events are handled when advanceTime() is called. When Frame 2 is rendered only TOI1 is handled, and TOI2 is simply kept on the TOI list until the next execution of advanceTime().

#### 2.6.2.6 Advantages of Continuous Simulation

**Small Objects**

With continuous simulation it is safe to use small objects (<10cm in size). That's because a TOI event is generated whenever the objects attempt to penetrate scene borders. With discrete simulation a small object could easily penetrate a triangle-mesh object even when moving with small velocities. Additionally, varied constraint priorities can be used to keep critical objects from being pushed through floors or walls.

You may attempt to stack small objects. However accuracy decreases with object size giving poor results for objects below 10cm in size. To improve the quality of solver results for smaller bodies you may increase their rotational inertia. This causes noticeably unrealistic movement in larger objects, but helps to minimize jitter in smaller ones. The issue of small objects is well illustrated in the Pyramid demo.

**Low Frequency Simulation of Ragdolls**

Ragdoll simulation frequency can be set at low frequencies when using continuous simulation. Simulation at 15 Hz gives a fairly good visual quality, although limb separation is slightly noticeable when the rag doll collides with environment. The trick of increasing rotational inertia of body parts is also applicable to ragdolls and gives significant improvement in quality for high-velocity ragdoll impacts.

At even lower frequencies (e.g. <10 Hz) it is already noticeable that gravity only is applied to rag doll body at discrete moments resulting in instantaneous velocity change. This feature is well illustrated by the Ragdoll Vs MOPP example.

#### 2.6.2.7 Limitations of Continuous Simulation

**Angular Errors**

When processing contact constraints, Havok Physics 2012 does not check angular movement as such but only looks at the resulting relative linear velocity of contact points on colliding bodies. Contact constraints prevent each of the bodies from penetrating each other along one direction, or through one contact plane.

This solution works efficiently with objects flying with high angular velocities as long as the solution uses a limited number of TOIs. That's the case for all convex bodies.

Consider however a hollow object which acts as a centrifuge. The example contains a sphere enclosed in a rotating hollow box. As the box's angular velocity rises the sphere is pressed against its walls.

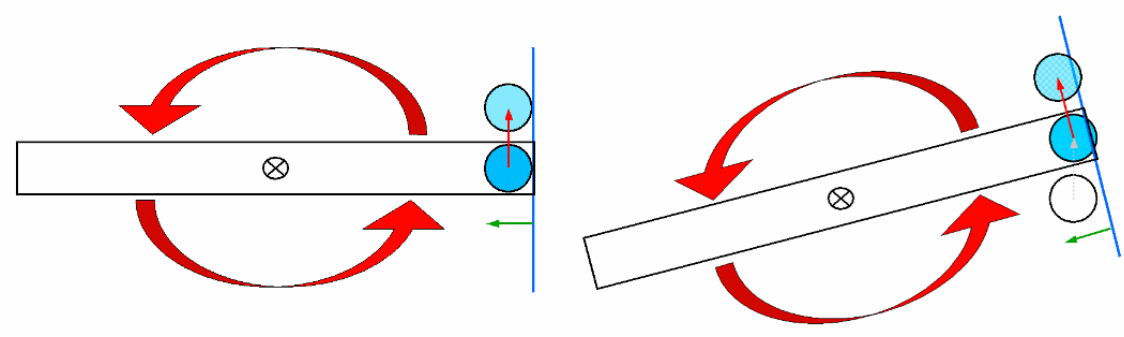


Figure 2.102: Note the penetration between the sphere and the centrifuge after the first step (right picture). This and consecutive incremental error will accumulate causing the sphere to tunnel through the centrifuge's wall.

In the first frame (left picture) the contact constraint only prevents the sphere from going further to the right through the blue contact plane. However after the next integration step the wall of the centrifuge moves to the left due to the centrifuge's angular motion. This error accumulates over consecutive simulation frames. It is corrected partially by penalty methods, but when the centrifuge's angular velocity reaches a certain value, the sphere will eventually tunnel through.

Consider continuous simulation now. As the hollow box accelerates the contact constraints keeping the sphere inside must be updated more and more often, resulting in more and more TOI events to handle. However since we have specified a minimum time distance between two consecutive TOI events handled, the sphere will start to penetrate the box's walls at a certain angular velocity. Eventually the result is the same as in discrete simulation.

The effect of angular errors is well visualized by the Centrifuge demo in the Havok demo framework.

**Angular Velocity Clipping**

The Havok Physics 2012 engine has a hard limit for the angular velocity of bodies. This is necessary for performing interpolation of body orientation between two frames. In every simulation frame bodies can rotate by no more than ∼170 degrees. At simulation frequency of 60 Hz this corresponds to the angular velocity of the wheels of a car traveling at ∼190 km/h (assuming the wheels to be 60cm in diameter).

Additionally the angular velocities of individual bodies may be limited further with a `hkpRigidBody::setMaxAngularVelocity()` call.

**Linear Velocity Clipping**

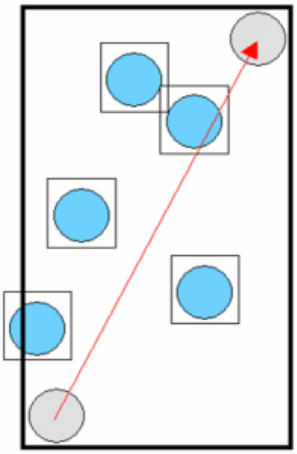
In order to prevent the engine from stalling some internal safety values are used. You can set the maximum linear velocity of a body using `hkpRigidBody::setMaxLinearVelocity()`. Make sure that the tolerances in the `hkpCollisionDispatcher` are tuned correctly by setting the `hkpWorldCinfo::m_expectedMaxLinearVelocity` to the highest velocity you allow in your game.

**Less Efficient Use of Broadphase Collision Detection**

Two positions are stored in the motion state of a rigid body - the position of the rigid body in the current physics frame and its expected position in the next frame. The `hkAabb` representation of a body in the broadphase encloses the entire volume the body passes through between the physics frames.

**Linear Motion**

Fast projectiles have their `hkAabb`s stretched along their linear velocity vector. Note that depending on the direction of travel a `hkAabb`'s volume may be relatively small and stretched (if the projectile moves along one of the global reference frame's axes), or it may be large and stretched along 3 axes.

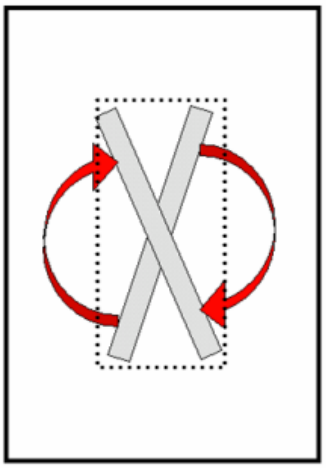


The above picture illustrates an emerging problem. Collision agents are created and queried as the `hkAabb` representation of the particle overlaps with other objects. On the next simulation step all agents must be deallocated.

Fast projectiles may instantly merge numerous separate islands containing stationary objects. Such

islands may need to be first activated by the engine, which causes additional overhead.

**Angular Motion**



Long, thin, rotating objects must be appropriately expanded, as the space they move through cannot be simply based on their initial and final `hkAabb`s. Havok Physics 2012 uses a rough approximation to expand the `hkAabb`s in all directions, taking the angular velocity of the body into account.

**Collision Agents**

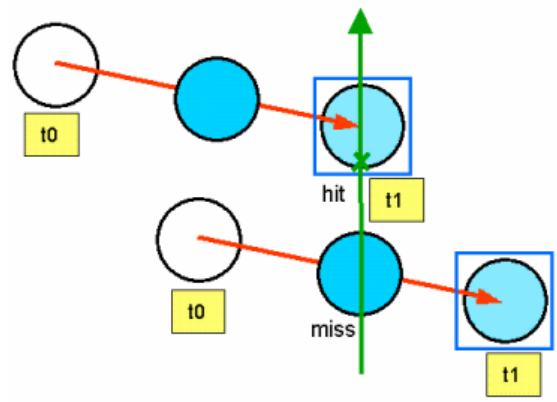
The `hkpPredGskfAgent` (predictive convex-convex) agent is the only agent that supports continuous collision detection. The agent has been improved in terms of CPU efficiency and memory usage as compared to its predecessor `hkpGskConvexConvexAgent`. However it suffers from not being able to generate the full collision manifold instantaneously, as the box-box agent can. As a result some additional small jitter might be noticeable when stacking large numbers of small boxes.

**Asynchronous User Calls**

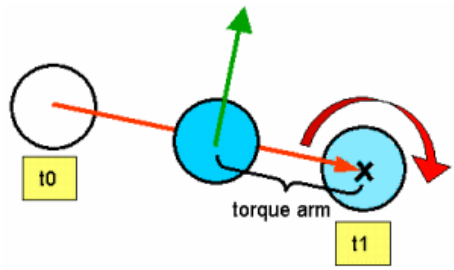
The Havok Physics 2012 engine is optimized to perform all 'out-of-main-simulation-loop' asynchronous queries and calls at the time of PSIs only. It does not store proper mid-frame data. The `hkpRigidBody` transforms used by collision detection (accessed via `hkpRigidBody::getTransform()`) are always set to the end of the PSI step (`hkSweptTransform`'s position/orientation at time `t1`).

As a result raycasts fired mid-frame may incorrectly miss or hit objects.

The following diagram shows a ray-cast (marked by the green vertical arrow) executed at frame time in-between PSIs. The objects move from the last PSI position `t0` to the next PSI position `t1`. We assume that we render a frame just between `t0` and `t1`. So if we fire our ray when we render the frame, we expect the ray to hit the upper object and miss the lower one. However the raycast uses the transforms stored in the object for time `t1`. That means our ray will hit the lower one and miss the upper object. This is only a problem if objects are moving at high velocities.




Similarly any forces applied to rigid bodies at specific points may use incorrect position of center of mass information and cause unexpected rotational motion.




In this example force is applied at the point of expected mass centre of a body. The system uses end-of-PSI-step transforms what causes unexpected rotational movement.

When running asynchronous simulation, it is a good practice to always use the `hkTransform` stored in the `hkpRigidBody` in the simulation engine, and not taking one from e.g. the graphics engine.

#### 2.6.2.8 Parameters Overview

Below is a summary of the key parameters used to perform continuous simulation.

**hkpWorldCinfo**

- `m_simulationType` must be set to `SIMULATION_TYPE_CONTINUOUS` or `SIMULATION_TYPE_MULTITHREADED` to enable continuous simulation, or `SIMULATION_TYPE_DISCRETE` for discrete simulation.
- A number of other parameters to continuous simulation are covered in the Creating the World section.

**hkpRigidBodyCinfo**

- `m_qualityType` (also accessed in `hkpCollidable`) determines how the engine handles the body's interactions with other entities. Available settings are: `HK_COLLIDABLE_QUALITY_FIXED`, `_KEYFRAMED`, `_DEBRIS`, `_MOVING`, `_CRITICAL`, `_BULLET` or `_KEYFRAME_REPORTING`. For more information refer to the Object Quality Types section above. Note that the quality cannot be changed once the rigid body has been added to the world.
- `m_allowedPenetrationDepth` (also accessed in `hkpCollidable`) defines how deeply an object is allowed to penetrate in normal situations. However, the engine is allowed to breach this value to solve critical situations. The value should be small for small objects (about 5% to 10% of the object's diameter) and can be larger if quality is not a primary issue. If you reduce this value, you will get a higher quality at the expense of some extra CPU. The default value is 0.05 (metres) and works well for objects as small as 20cm.

**hkpConstraintInstance**

- `m_priority` determines whether a constraint should be considered in processed TOI events (PSI and TOI priorities). Higher-priority (`TOI_HIGHER`) constraints may be used to override normal-priority (PSI or TOI) when the two are in conflict.

#### 2.6.2.9 Demo Walkthroughs

**Continuous Basics**

This demo introduces the basic concepts of continuous physics. Basically a simple ball is shoot against another ball. Once the balls collide a line is drawn (and remembered) in which direction the balls travel after collision. We then continuously iterate the game. As Havok is deterministic, repeating the demo would give the same results over and over again. So to make things more realistic, we vary the starting position of the ball at each iteration. As a result the balls will hit each other at different times. This is similar to what happens in a game, where it is not defined when exactly a player's car will hit the road block.

**Slow ball hitting other ball**

This scenario is simulated using discrete physics. As we simulate the demo at 60Hz, there are enough timesteps available when the two balls collide. So this demo works as expected.

**Faster ball using discrete physics**

Here we simply increase the velocity of the striking ball. As discrete physics only has discrete time steps and the balls are colliding at varying times, it is kind of random how the balls are positioned when the physics time step is simulated. This gets more obvious when running the demo in slow motion. As a result, the outcome of the collision vary each time dramatically.

**Fast ball in slow motion**

This shows the previous example frame-by-frame. Notice that at the moment when collision response is applied the balls may either just touch or deeply interpenetrate. Such varying input information causes the collision results to be hard to predict. As we only perform collision detection at discrete moments in time we rarely happen to apply collision response at the exact time of impact of the two bodies. We are only able to apply it before the collision (using extra collision tolerance boundary which adds a thick layer around the bodies' geometries) or after the actual collision – in that case bodies already penetrate

because their state was integrated from the previous frame by a fixed timestep without consideration of this collision.

### Fast ball in slow motion interpolated

This demo introduces the concept of continuous motion. Havok represents the state of bodies with the hkSweptTransform structure which holds information about a body position and orientation at two specified moments in time. The transparent shapes visualize those end positions of the hkSweptTransforms of the moving bodies. Notice how hkSweptTransforms are only updated at the time of simulation step and how the current position of bodies is simply interpolated from them. If you use this interpolation technique for extraction the transforms of the physics for your graphics engine, it allows us to run the physics at a different frequency than the graphics (like in this demo) (e.g. by using hkAsynchronousSimulation).

### With physics clock

This demo introduces the clock commonly used in the continuous physics demos. The clock is located in the lower left corner of the screen. The blue lines correspond to moments when physical simulation is run (called physical synchronuous instance = PSI). The green line indicates the current time of visualization. Whenever the green line passes over a blue line, a PSI is executed. Then, until the green line reaches the next blue line, the physics engine is suspended. To preserve the illusion of continuous motion of the bodies, the rendering engine calculates their positions from their hkSweptTransforms at every frame. The hkSweptTransform stays constant in-between the physical steps.

### Very fast ball

This example accelerates the striking ball to even higher velocity. Many of the collisions are simply 'missed' (or not recorded) by the physics engine. However even when they are recorded, the collision response outcome is highly unpredictable.

### Very fast ball with continuous physics

`hkpContinuousSimulation` calculates the moment of impact of the colliding bodies. It then uses `hkSweptTransform` to determine their position at that time and applies precise collision response.

**Note:**

The majority of the iterations result in exactly the same outcome. However with time you may observe a slight variation in the final velocity of the objects. This inconsistency is related to the fact that somtimes the balls hit between two PSIs, sometimes the balls hit exactly at a PSI (within a certain tolerance). In both cases different code paths are used to process the collisions:

- If the collision is handled in between the main physical simulation steps, it is handled at its exact time of impact via a special time-of-impact event handler.
- When the time of impact is however close enough to the moment of a main simulation step then the collision is simply handled in the main simulation step

### Discrete vs Continuous

A scenario where very-fast-moving cubes are collided against a landscape modelled with mesh geometry. Such a scenario is likely to fail with discrete physics as fast objects can easily tunnel through the walls of zero thickness. Havok allows you to mark objects with the HK_COLLIDABLE_QUALITY_BULLET property which ensures that continuous collision detection is performed between bullet-objects and fixed landscape objects. Notice how the fast objects are shot into the corners of the simple landscape and how they may cause more than one impact collisions per frame. After their first collision the bodies are reintegrated and again re-collided with the environment. This yields the information necessary to process all subsequent collisions occurring before the next main simulation step.

**High speed collision**

Presents a grid of spheres, placed one next to another, that is hit by a fast particle. This demo presents a good use case for localized solving algorithm and for the hkpRigidBodyCinfo::m_allowedPenetrationDepth parameter. If localized solving was not used this example might cause a huge number of TOI events, as every application of simple collision response would be followed by more collisions with surrounding bodies in contact. Localized Solving brings the number of TOI events down by an order of magnitude. Also notice that the spheres are assigned a relatively high allowed penetration depth parameter, which allows for a major interpenetration before a TOI event is triggered. This is the reason for which the particle initially sinks-in into the grid of spheres and only then properly disperses its kinetic energy.

**Squeezed ball**

This demo illustrated the results of usage of incremental allowed penetration. As the small ball is being squeezed it bounces more and more often between the two slabs, finally it is allowed to penetrate into the moving slab (as it has a lower surface quality than the fixed slab) in order to keep the number of TOI events reasonable. See section 5.2 for more details.

**Hinge hitting table**

This demo shows the importance of using TOI_PRIORITY constraints in systems composed of bodies that use continuous collision detection. The first three variants of the demo show a hinge composed of two rigid bodies and a hinge constraint: 1. using discrete simulation, 2. using continuous simulation but with a non-TOI_PRIORITY constraint, and 3. using continuous simulation and a TOI_PRIORITY constraint. When the hinge hits a fixed obstacle, the first two variants fail to enforce the hinge constraint. The third example is simulated properly, as the hinge constraint is considered in the localized solving algorithm at each TOI recorded between the hinge bodies and the fixed obstacle. The next three examples show a ragdoll with analogical setting of discrete/continuous simulation and priority settings of the ragdoll's constraints. The last two examples introduce the idea of incrementing the rotational inertia tensor for ragdoll limbs. This trick is an efficient way of improving the stability of the system, as it reduced the angular components of motion in the solver.

**Three way squeeze**

This demo illustrates use of normal- and higher-priority constraints.

**Centrifuge**

This demo illustrates the problem of high angular velocities.

### 2.6.3 Integrating Particles with Havok Physics

#### 2.6.3.1 Introduction

The Havok Physics 2012 SDK does not include direct support for particle systems. We consider them best left to game developers to handle since they are relatively straightforward to implement and are usually strongly tied to graphics and game effects.

We do however provide an example implementation that can serve as a reference for integrating particles with Havok. This implementation is located in `Demos\DemoCommon\Particles` and a demo

that simulates simple particle-based smoke, dust, spark, debris, and explosion effects can be found in `Demos\Physics2012\UseCase\ParticlesTest`. All of these effects are implemented with minimal physics to achieve good performance. As such, they all use Euler integration, generally with constant acceleration. No interparticle or spatially-variant forces are computed.

The most important class is `ParticleSystem`, an abstract base class that defines the interface for emitting, updating and drawing particles. It has the following features:

- It handles all memory management for the particles, which are maintained in a packed state. Packing allows faster iteration through particles and is achieved by copying existing particles from the end of the list to fill holes left by deleted particles.
- The base class knows nothing about the properties of a particle. It is up to the derived classes to define this by implementing two pure virtual functions, one to initialize a list of particles in a contiguous block of memory and another to update them over a time step. This is a lot faster than calling virtual functions to initialize and update particles one by one.
- The only information the base class has about a particle is the number of bytes it occupies. This value can be set by derived classes which makes it easy to inherit from a particle system class while adding extra properties and behavior to the particles. For example, `SmokeParticleSystem` defines a particle as having position, velocity, size, age and life time attributes, while `HkgSmokeParticleSystem` extends this with graphics specific attributes such as the index of the sprite to use for rendering.

#### 2.6.3.2 Event Handling using Listeners

Listeners can be used to trigger particle emissions in response to Havok events. Some examples are as follows:

- Using `hkpCollisionListener` to emit particles when rigid bodies collide or grind against one another.
- Using `hkpConstraintListener` to emit particles when constraints break.
- Using `hkpBreakOffPartsListener` to emit particles when pieces break off a compound body.

In the example collision listeners we provide, an array of contact objects is maintained and updated whenever a collision event is triggered. This array records information such as contact points, contact normals and collision speeds. After the physics world has been stepped forward in time, each contact from the array can then be processed to guide particle emissions, after which the contact array should be cleared. An alternative approach would be to emit particles inside the listener callback function itself. However, this function must be thread-safe in multithreaded environments, so it is preferable to execute as little code as possible here and instead just use a critical section to protect access to the contacts array.

In most applications, a mechanism will also be required to identify the type of rigid bodies involved in an event so that, for instance, different types of particles can be emitted depending on the type of rigid bodies involved in a collision. Property keys are one of many ways to achieve this. In our example collision listener, a property key is associated with the listener at construction time and then, any time a collision occurs between two rigid bodies, the contact is only recorded if either of those bodies have that property key.

#### 2.6.3.3 Collision Processing

Example code is provided for processing collisions between particles and rigid bodies using two different approaches.

**Detecting Collisions using Phantoms**

The function `ParticleCollisionUtil::moveParticles()` advances particles over a time step while detecting and resolving collisions. Collision detection is performed using a single `hkpSimpleShapePhantom` object, where the shape is a `hkpSphereShape` representing a spherical particle with a very small radius. For every particle, the phantom is repositioned at the location of the particle and `hkpSimpleShapePhantom::getClosestPoints()` is called to see if the particle penetrated another rigid body. If so, the particle is moved to the contact point and the particle velocity is modified using a simple formula taking friction, restitution and the relative velocity at the contact point into account.

Note that in some cases it would be faster to use a separate phantom for every particle to take advantage of broadphase acceleration. However, this approach will require a lot of overhead and will scale poorly when the number of particles is very large. A good compromise would be to only process collisions for a small subset of particles and maintain a separate phantom for each of these. In any case, it is usually also a good idea to only process collisions for a subset of particles each frame.

**Detecting Collisions using Raycasting**

The function `ParticleCollisionUtil::moveParticlesWithRaycastCollisions()` advances particles over a time step while detecting collisions by performing a raycast into the world for each particle. This approach is significantly faster than using phantoms, especially when KD-trees are enabled to improve raycasting performance. The slight disadvantage is that there are cases where collisions with *moving* rigid bodies will go undetected. This is due to the fact that rays in Havok that start inside a rigid body will not register hits.

### 2.6.4 Ragdoll Best Practices

#### 2.6.4.1 Introduction

This document contains advice for tuning and setting up a ragdoll in the modeler and in the runtime to achieve a good visual effect and optimal runtime performance.

#### 2.6.4.2 Setting up the Ragdoll

There are many factors in what makes a ragdoll look convincing in a game. Some of those factors come from the way the ragdoll is setup and some come from how the game interacts with the ragdoll.

First, let's take a look out how we can set up a ragdoll in the tools to get a stable, robust, and physically believable baseline that can have additional effects added to it in the runtime.

**The Rag Doll Toolbox**

To help with the construction of ragdolls, Havok gives users the Ragdoll Toolbox. Its "Create Proxy RB" feature gives a good starting rigid body bone for each animation bone. It also allows users to add custom

shapes to the ragdoll and easily drive a number of animation bones with a single ragdoll bone.

Setting up constraints is much easier using the toolbox as it has presets for common human joint constraints and it has the ability to create template constraints that can be reused. For more information on the ragdoll toolbox, please read the Tutorial > Rag Doll Toolbox section of the Content Tools Manual that is appropriate for the modeler being used. The Rag Doll Toolbox works with 3ds Max, Maya, and XSI.

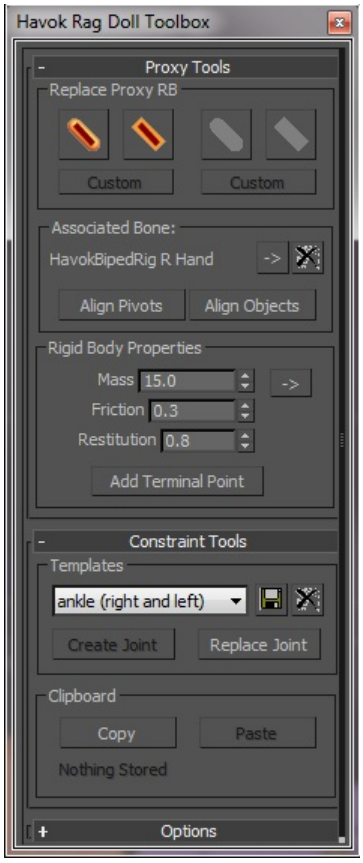



Figure 2.103: Rag Doll Toolbox

The Rag Doll Toolbox will be referenced throughout this document so make sure you are familiar with it before moving forward.

### Creating the Ragdoll Bones

For this section, let's assume that the character is already rigged, skinned and ready for us to setup its ragdoll. The first step in creating a believable ragdoll is creating rigid body bones that will map to the animation bones of a character. There are several considerations that need to be taken into account as we get started.

### Performance

Key points:

- Keep the number of rigid body ragdoll bones to a minimum.

- Use lightweight shapes whenever possible.

The first consideration and one of the most highly asked questions is how to improve performance around ragdolls. The performance cost of the ragdoll should be thought about throughout the construction of the ragdoll. Two big factors in the cost of the ragdoll are the number of rigid bodies and constraints in the ragdoll and the shape types of the ragdoll bones. Before creating any rigid bodies, try to think about which animation bones really need rigid bodies in order to act physically correct in the game.

- Does the ragdoll need clavicle bones?
- Does every spine bone need to be represented separately or can some of them be represented by a single body (easily done in the Rag Doll Toolbox)?
- Is the neck necessary?
- Can the fingers be included in the hands? (Almost always "Yes!")
- Can the hands be included in the lower arm?

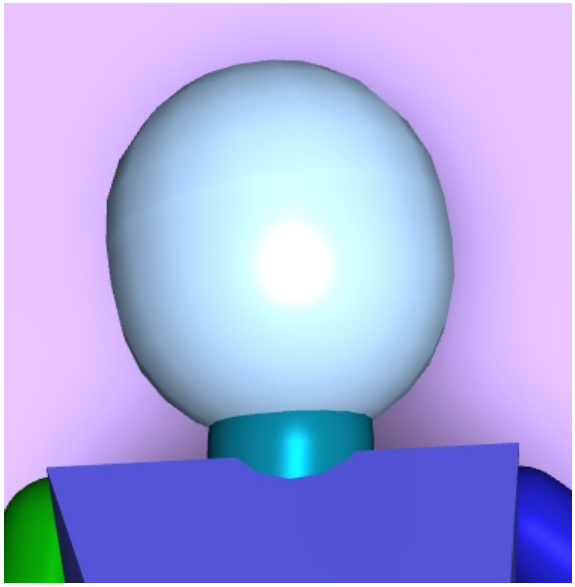


Figure 2.104: Ragdoll with an unneeded neck bone.

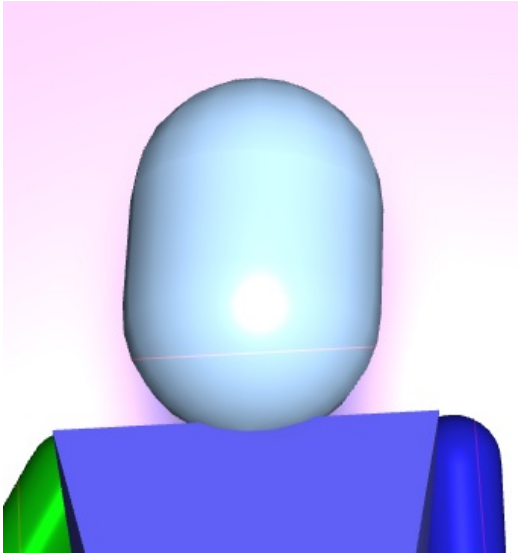


Figure 2.105: Ragdoll without a neck bone. This one and the one above cover pretty much the same volume, but the neck version will be more expensive.

Each rigid body will have an associated collision detection and integration cost. Each constraint will also have an associated constraint solving cost. The number of constraints is directly tied to the number of rigid bodies, so decreasing the number of rigid bodies will also have the benefit of decreasing the number of constraints. The fewer rigid bodies and constraints there are, the better the performance.

The next part of the performance problem is the cost of the collision detection itself. If you have not done so already, please read the Havok Content Tools > Artist Best Practice for details on collision detection costs. This is a good primer for choosing which shape types should be used for the ragdoll. Capsules tend to have a closer fit to human anatomy, but capsules and boxes are both cheap shape choices. The torso is typically the only place where a capsule or box would not be used. In order to match the skin properly, a convex vertices shape is often needed. In this case, try to keep the number of vertices to a minimum.

**Clipping**

Key points:

- Create the ragdoll bones to match the skin as closely as possible.
- Overlap ragdoll bones.

The next considerations revolve around how the ragdoll will look in the game and the way in which the character's skin matches up with the ragdoll and the environment.

A problem that is very noticeable to the players and easily preventable is when the ragdoll penetrates itself or the environment. The first step is to make sure that the ragdoll skeleton closely matches the skin of its character. The Rag Doll Toolbox makes it easy to create rigid bodies for the animation bones; however, these bones might need to be tweaked in order to better fit the skin of the character. Only ragdoll bones will collide with the Physics 2012 world. The skin of the character will not. If the ragdoll bones don't closely fit the skin then visible penetrations of the world are possible.

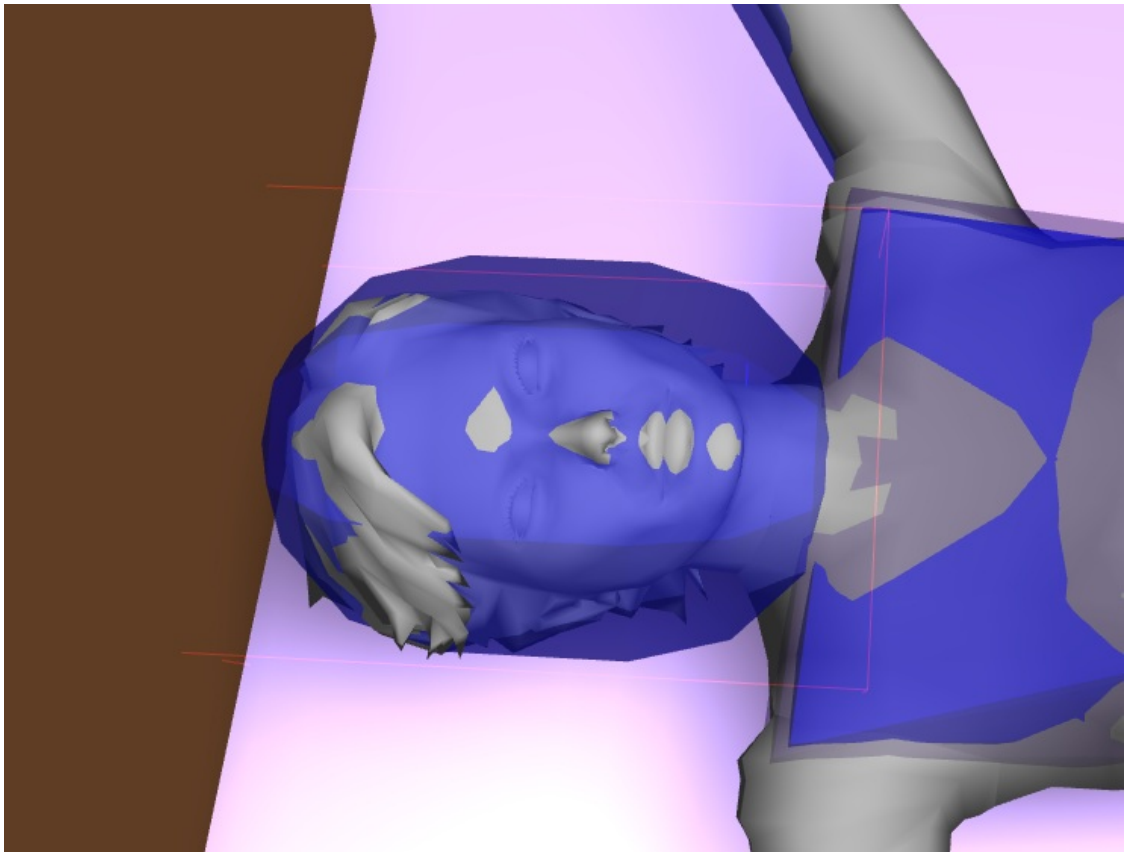


Figure 2.106: Ragdoll with a closely fitting head rigid body that completely covers mesh head.

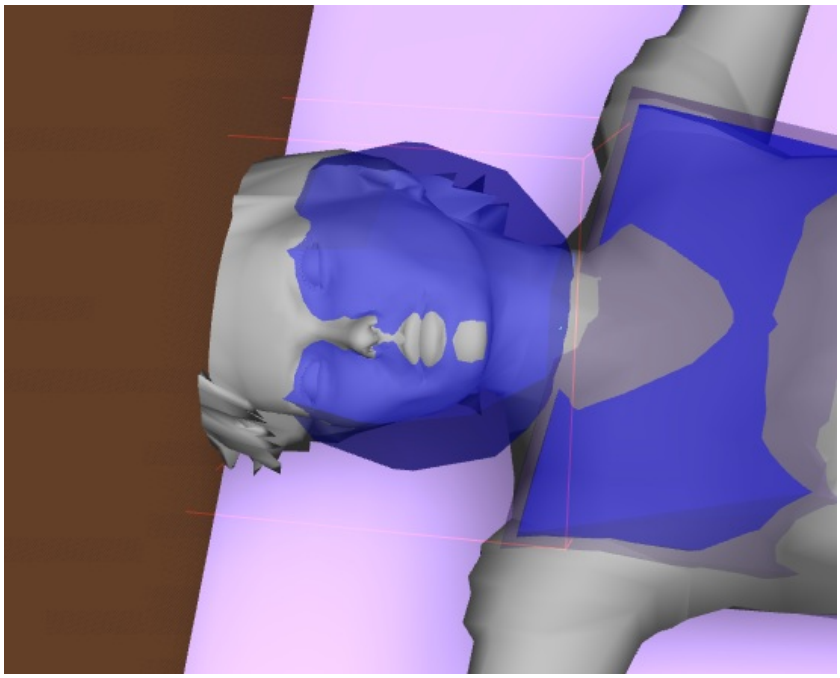


Figure 2.107: Ragdoll with a head rigid body that doesn't completely cover the skin allowing the character's skin to clip into the wall.

This is especially true as the ragdoll moves around. What is good in the reference pose might not be good as the ragdoll moves within its constraints. When creating the bones, make sure they stay aligned with the animation bone (the Havok Content Tools Create Mapping Filter will warn if they are not). The Rag Doll Toolbox's Align Pivots and Align Objects options make this easy. If you manually move a rigid body in the modeler, be sure to re-align the rigid body to the animation bone using these features.

Another common case is where rigid bodies in the environment penetrate in-between rigid body bones as seen in the picture below.

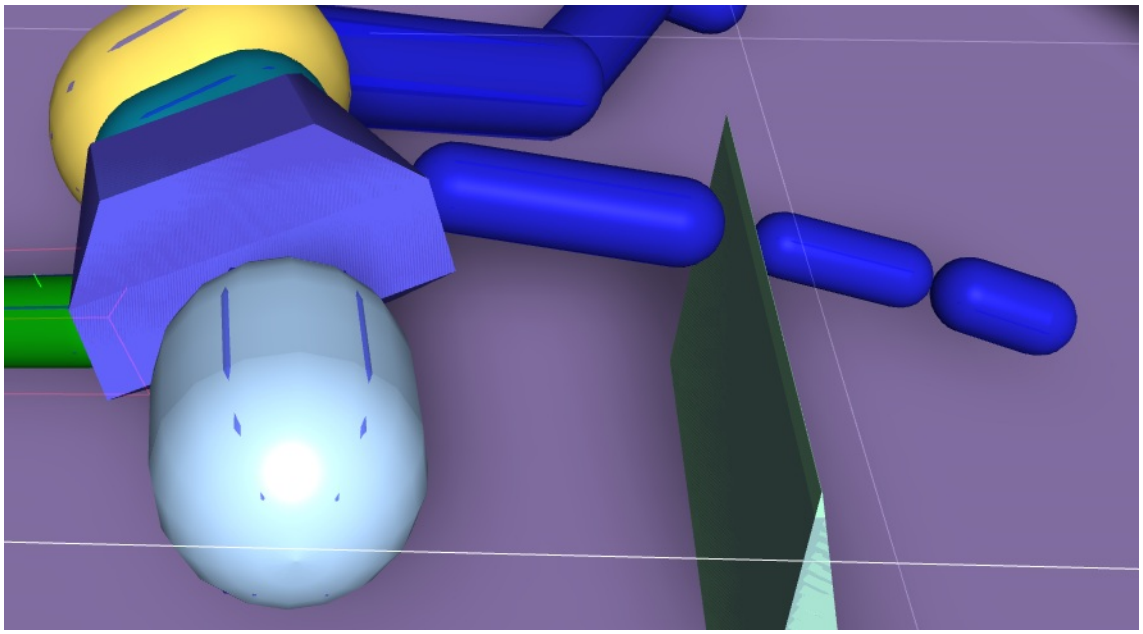


Figure 2.108: A gap is left in-between the upper and lower arm allowing a rigid body to get lodged

If an outside rigid body gets stuck in between the ragdoll bones' joints it can look very unnatural, cause jitter in the bones (as the contact constraints are fighting against the ragdoll's constraints), or cause dynamic rigid bodies to get shot out with extra energy. Overlapping the rigid body ragdoll bones greatly mitigates situations when this can happen.

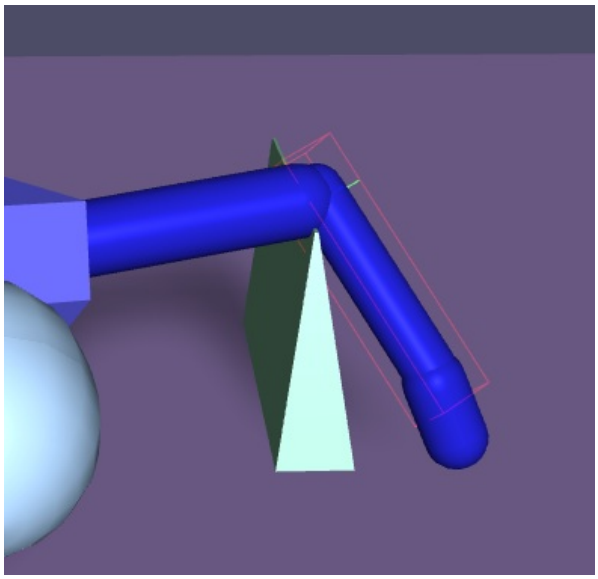


Figure 2.109: Properly overlapped ragdoll bones

**Creating Ragdoll Constraints**

Key points:

- Use ragdoll constraints where possible.
- Make sure parent and child spaces are correct.
- Create constraints so they are correctly oriented with the animated joints.
- Maintain constraint limits that make sense when the character is in the ragdoll state

The next step in setting up the ragdoll is getting the constraints in place to connect the rigid bodies together. Again, the Ragdoll Toolbox can make this easy with its preset human joint constraint templates. Most of the provided templates are for "Ragdoll" constraints. As the name implies, these constraints work well for ragdolls and should be used as much as possible. If it looks like the constraint should be a hinge constraint, still consider using the ragdoll constraint with a narrowed plane min and max. We will go into more detail on why this is important in the "Correcting 'Bad' Initial Situations" section of this document.

When creating constraints it is important to take note of the parent and child space arrows. The parent space is in green and the child space is in red. The child space starting position/orientation will be forced inside the constraint limits when the ragdoll activates. In the figure below, you can see that the ragdoll constraint's twist limit child space is outside of its parent constraint's limits. When the ragdoll activates the arm will spin around, deforming the skin unnaturally.

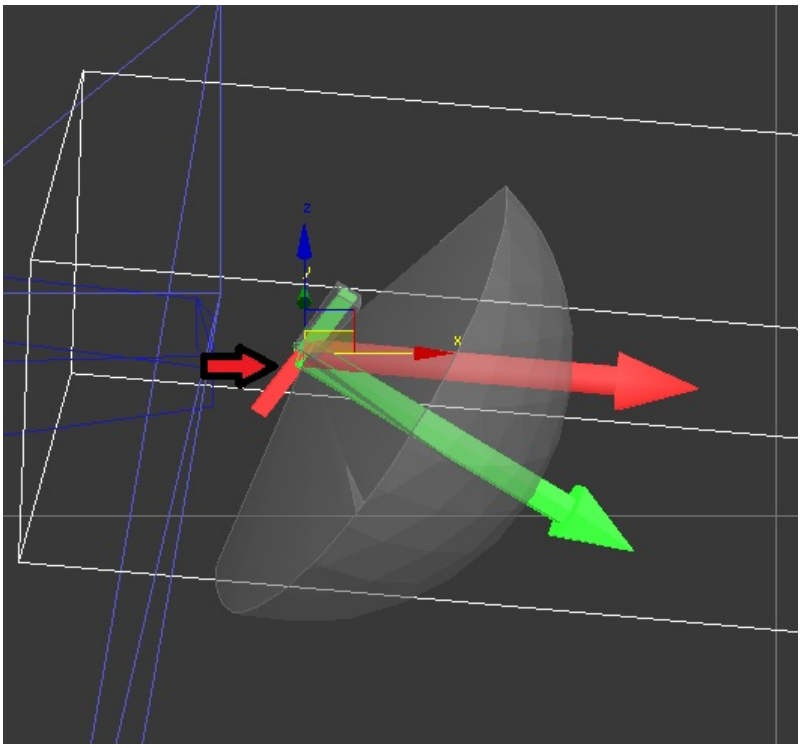



Figure 2.110: Easy to miss: The child twist starting location (red bar being pointed to by the arrow) is outside its constraint limits. The child space starting locations should always be inside the constraint limits (grey region). Arm will be twisted when ragdoll settles.

As a rule of thumb, always make sure the red arrow (child space arrow) is always inside of the grey region (constraint limits) of the constraint.

The Ragdoll Toolbox will also automatically position the constraint so that it lines up with the animation joint. If these are misaligned then the ragdoll constraint will pivot differently than the animation joint, producing unwanted results.

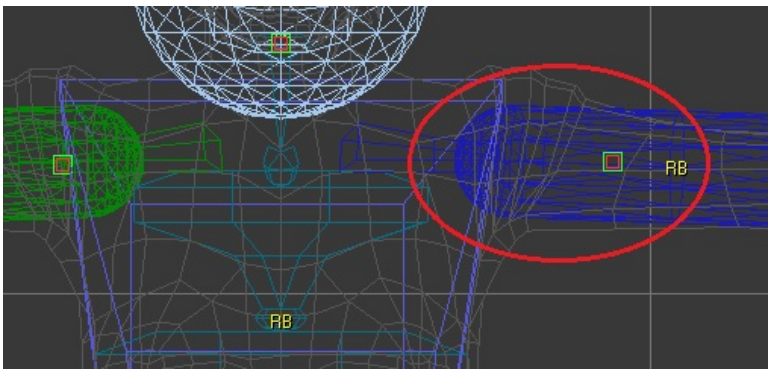



Figure 2.111: Constraint on right arm is aligned with the animation joint.

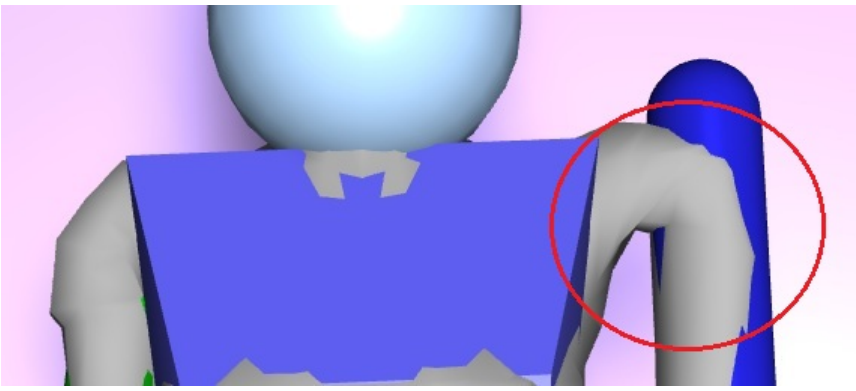


Figure 2.112: Constraint on left arm is not aligned properly and the skin deforms unnaturally.

Lastly, we want to make sure that the ragdoll cannot get into "ridiculous" poses. While a particular pose might be physically possible for a human, in the game it might look wrong for a character in its ragdoll state.

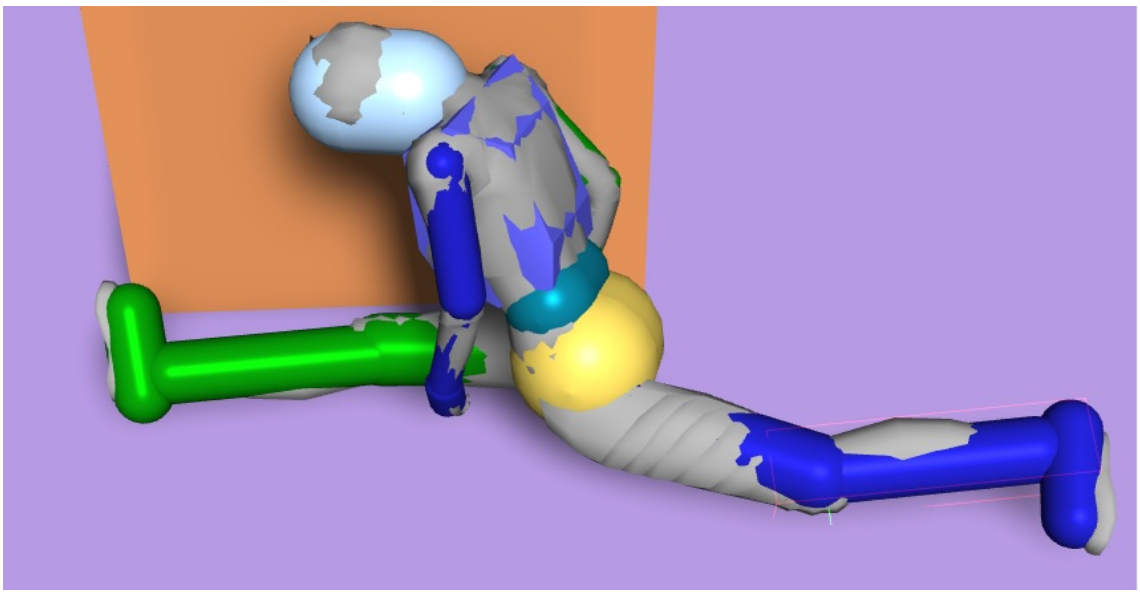


Figure 2.113: Constraint limits on hips are possible for (some) humans, but would not look good for a character ragdoll in a game (unless that character was a gymnast)

Don't worry about allowing the ragdoll constraint limits to reach all the poses in the character's animations. Instead, think about the limits you want to impose on its ragdoll poses. Havok constraints have no concept of a "return spot," a place where ligaments/tendons/muscles might drive a limb to when a human is unconscious. Because of this, it is best to impose tighter restrictions and use constraint motors on the ragdoll to force it into more natural poses. This can lead to the problem where a ragdoll's starting pose, given by an animation, is outside of its limits, but we will deal with this case in the "Correcting 'Bad' Initial Situations" section of this document.

**Setting up Rigid Body and Constraint Properties**

Key points:

- Distribute masses somewhat evenly throughout the bones.

- Use joint friction on constraints.
- Increase rigid body inertia tensors.

Now that the rigid bodies and constraints are set up, we can tune the settings on those objects in order to get a better looking ragdoll.

The first thing we should look at is mass. It might be tempting to give each section a mass about equivalent to what it would be in a normal person, where most of the mass in the torso and there is not very much mass in the extremities. This is not ideal for the constraints and can potentially lead to instabilities. At the start, giving each bone relatively close masses is better. Reducing mass as it goes out from the torso by 10–20% for each bone usually works well. The masses can be tuned later on.

If you have previewed the ragdoll at this point, you might have noticed that the constraints feel very loose. It doesn't look like the character has resistance at the joints which would be expected due to muscles and ligaments. To give this effect, try increasing the Max Friction Torque on the constraints. This will give a frictional force on the constraints and slow down the movement. Note that this is a force value and not a setting from 0–1. A rule of thumb is to start out at 20% of the mass and increase or decrease as necessary.

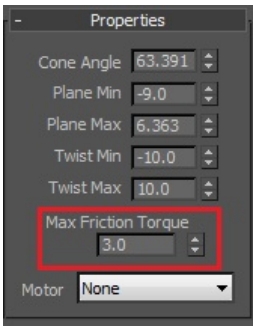



Figure 2.114: Max Friction Torque Parameter (display from 3ds Max)

Having this set to a reasonable value will also increase the stability of the ragdoll as it will reduce constraint violation.

Finally, if the ragdoll still feels "loose," scaling the inertia tensor of the rigid body bones can help. Increasing the inertia tensors will mean that rigid bodies resist their rotational forces more, translating into slower rotations and less stress on the constraints. Increasing it too much will cause the ragdoll to be very stiff. While this is possible to set in the modeler on a per-rigid-body basis, it is much easier to do across the whole ragdoll in code using Havok-provided utility functions. This is covered further in the "Constraint Separation at Runtime" section of the docs.

**Creating the Mapping**

Key points:

- Parent animated twist bones to their sibling ragdoll bones.
- Lock translations on as many bones as possible.

The ragdoll is now ready to be exported and as part of the export process an animation to ragdoll skeleton mapper will need to be set up. This will convert animation poses to ragdoll poses and vice versa so that

proper ragdoll driving can occur. The “Create Mapping Filter” (available to Havok Animation clients) in the Havok filter manager will set up this mapper.

A common problem when creating the ragdoll mapper is skin deformities from sibling bones. This often happens with forearm twist bones in the animation skeleton. These animation bones are often not mapped to any ragdoll bone.

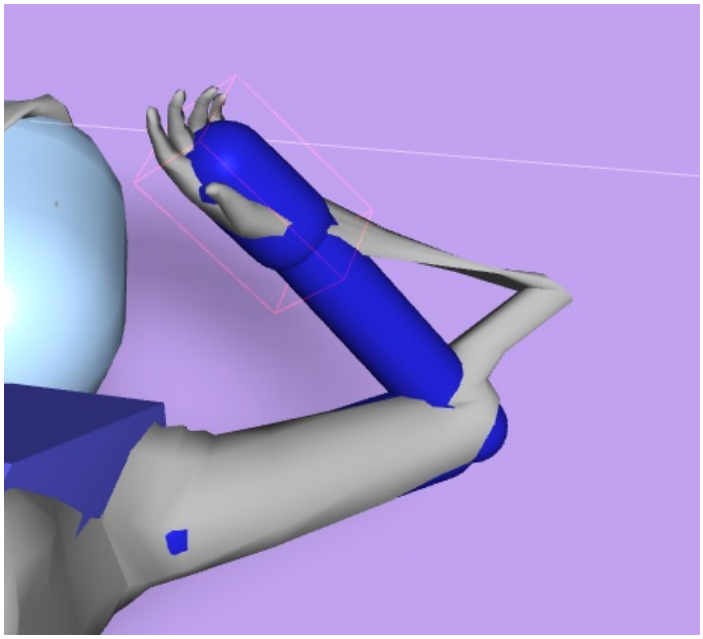


Figure 2.115: Unmapped Twist Bones

Please note that while this is very common for twist bones, it can occur for any bones which are siblings to mapped bones (clavicles and certain pelvis setups are other examples). The sibling setup needs to be reflected in the ragdoll mapping. In the “Create Mapping Filter” where the B->A mappings (ragdoll to animated skeleton) are specified, add additional mappings for the forearm ragdoll bone to the animated twist bones.

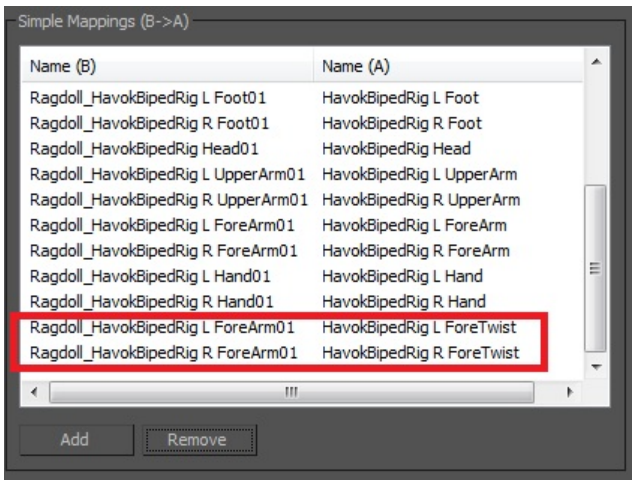



Figure 2.116: Mapping Twist Bones

The ragdoll mapper can also solve the problem where separating joints in the ragdoll cause the skin to

stretch. The mapper has the ability to only apply rotations to the animated bones. This is called "locking translations" and can be setup in the "Create Mapping Filter". This will only work with rotation-only joints (human joints are of this type). Most of the time, the only bone you want to have unlocked is the root bone or pelvis (so it can translate through the world).

**Tuning Tips**

Key points:

- Iterate on the ragdoll in the Previewer.
- Test with the Root Bone FIXED.
- Create simple test cases in the modeler.
- Expose ragdoll parameters in the game.

The Havok Previewer gives very good indication of how the ragdoll will perform in the game. However, you need to make sure that the previewer settings match the games settings. Make sure to adjust the Physics step size to match the size used in your game (usually 30 Hz or 60 Hz). Ragdolls at 60 Hz will perform differently than ragdolls at 30 Hz, so make sure to tune accordingly.

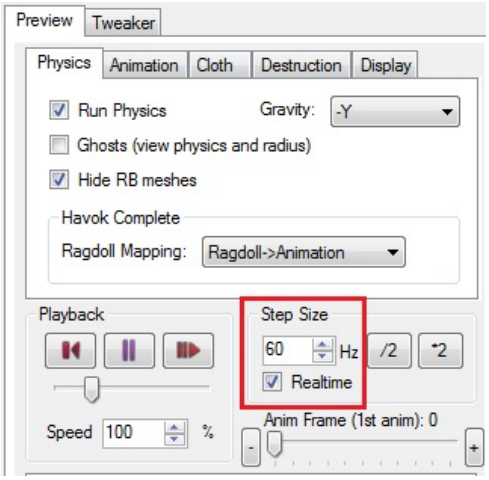



Figure 2.117: Previewer timestep

When testing constraints, it can be beneficial to fix the root of the ragdoll so that it doesn't move all over the place. Set the root bone's mass to 0 and preview. Hovering over a bone and holding 'Space' will allow you to pull on a rigid body to test how it reacts under stress.

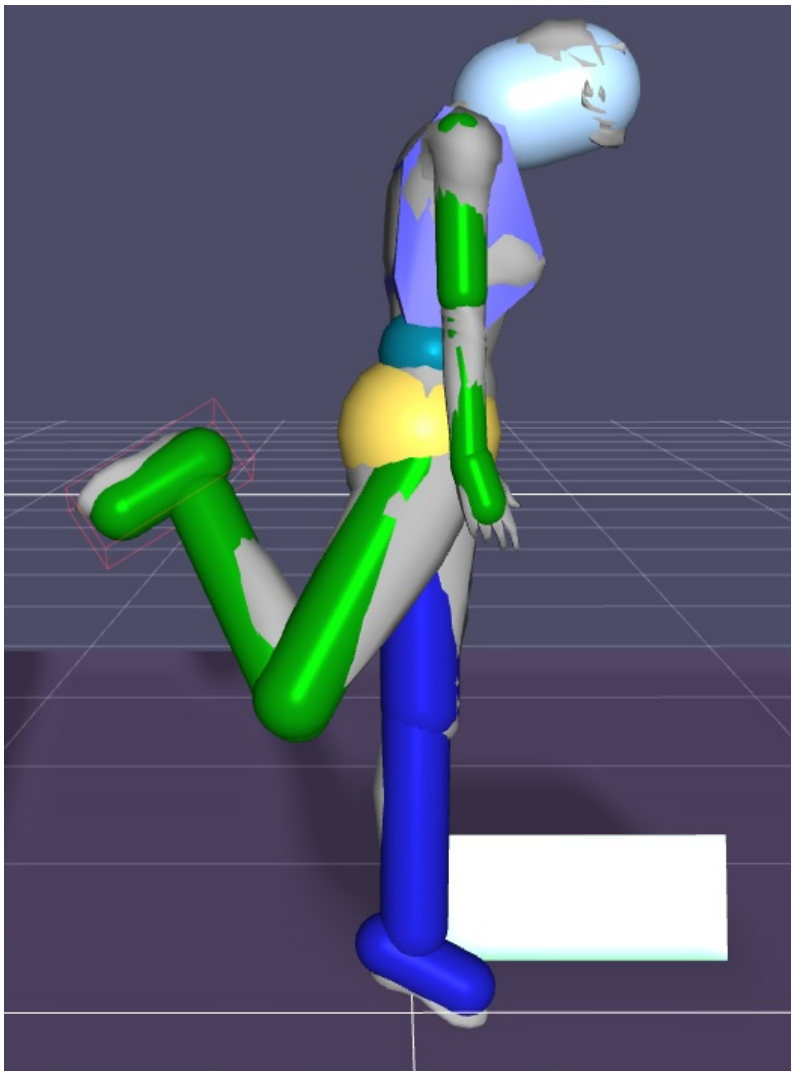


Figure 2.118: Testing knee constraint limits while the root is fixed

The previewer can also be used to test the ragdoll against test environments. A floor is usually a good start, but other fixed and dynamic objects can be added to the scene.

The previewer is very helpful, but to see exactly how the ragdoll performs in the game, it needs to be in the game. It is a good idea to expose some of the tuning parameters in the game, so artists can tune the ragdoll in real game situations.

#### 2.6.4.3 Ragdolls in the Runtime

Once there is a tuned ragdoll ready for the runtime, there are still some concerns and use cases that can be problematic.

**General Tips**

Key points:

- Use a fixed timestep.
- Don't use constraint chains.
- Make sure to use newer, "stabilized" constraints.

In general, it is highly recommended that a fixed timestep be used for the Havok Physics 2012 step. The Havok Physics 2012 Constraint Solver accumulates unsolved error from one frame and will try to solve it in subsequent frames. When the timestep is variable then error can be over- or under-solved. This manifests itself as ragdolls with stretchy or oscillating joints. Fixing the timestep should alleviate these kinds of problems.

There are situations where using Chain Constraints might be necessary and Havok offers them for those limited use cases. However, it is generally not recommended to use Chain Constraints with ragdolls. They are significantly more expensive. This is especially true on the PlayStation®3 where the ragdoll's entire simulation island is forced to the PPU. Chain constraints cannot be solved on the SPUs.

It should be noted that since Havok 2010.2 Havok has offered a more stable form of constraint solving for ball-and-socket, hinge, limited hinge and ragdoll constraints. This is the default when using the Create Constraints Filter and can be turned on in code using the `hkpConstraintStabilizationUtil`. A setup using this solving method is more stable than constraint chains in many cases and runs on the SPUs.

**Correcting "Bad" Initial Situations**

Key points:

- Use the `hkaDetectRagdollPenetration` utility.
- Use the `hkpEaseConstraintsAction`.

It sometimes happens that a ragdoll might start off in a "bad" situation. Havok offers several tools for resolving these situations nicely.

A common situation is a ragdoll that starts with some of its bodies penetrating another Physics 2012 body, i.e. a ragdoll arm sticking through a wall. This can be resolved by using our first tool, the `hkaDetectRagdollPenetration` utility (available with Havok Animation).

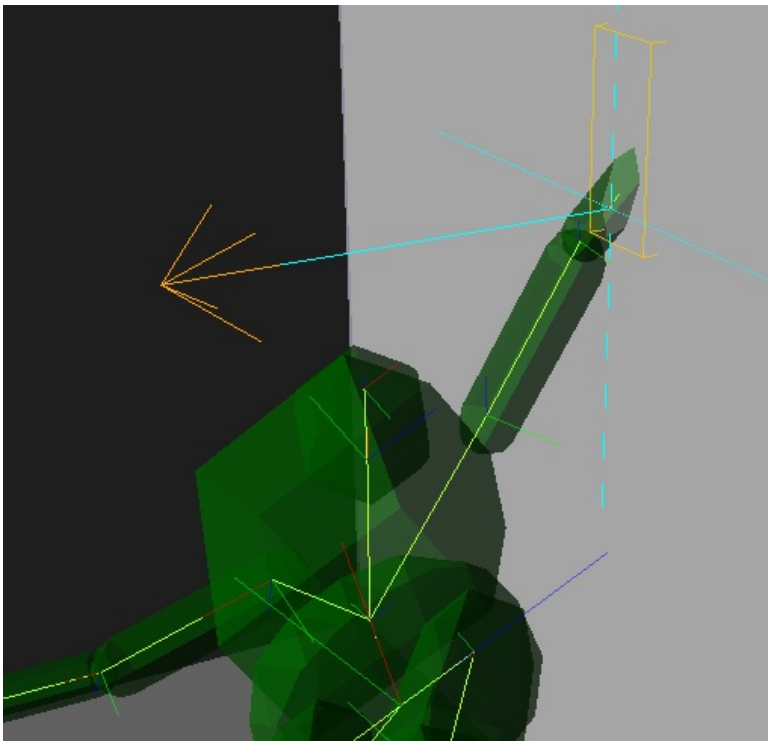


Figure 2.119: Ragdoll penetration is detected and corrected

The utility works by raycasting from one bone to the next across all the bone's constraints. It does one of these raycasts for each bone in the ragdoll. If the raycast encounters a physics shape that is not part of the ragdoll then it reports the body back to the user. From there, the user can disable contacts and push the ragdoll out of the bad situation. An example of doing this is found in the Penetration Detection Demo (`Demo\Demos\Animation\Api\Ragdoll\PenetrationDetection`).

The second case, which we mentioned earlier in this document, is where the animation skeleton's pose is in a state that violates the ragdolls constraints when it is spawned.

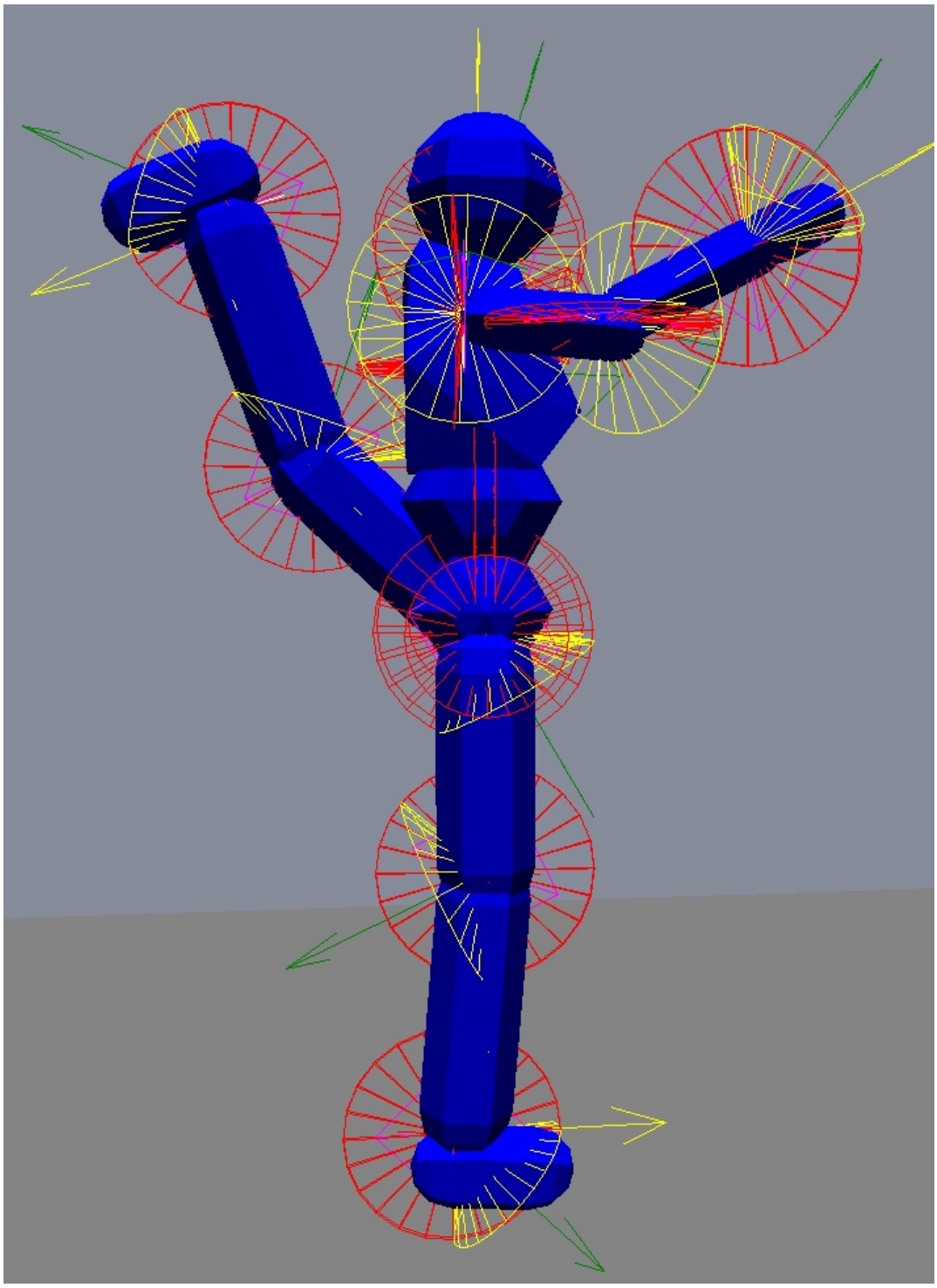


Figure 2.120: Hip constraint is violated

A common example is the "Jack-in-the-Box" case where a character is crouching in a position that violates the knee constraint limits. When the ragdoll is activated it quickly corrects the violating constraints, straightening the legs. The ragdoll springs upward. This issue can be resolved by opening the constraint limits when the ragdoll first spawns and then easing them back to their desired position over time. Havok offers the `hkpEaseConstraintsAction` to handle this automatically.

It should be noted that the `hkpEaseConstraintsAction` only works on constraint degrees of freedom that have limits. It will not work on lateral movement along the hinge axis of a hinge constraint. In this case, the hinge constraint will quickly resolve when entering the ragdoll state and the bones will be aligned exactly along the hinge. For this reason ragdoll constraints are the preferred constraint type.

**Constraint Separation at Runtime**

Key points:

- Tune the Inertia Tensors of the ragdoll with the `hkpConstraintStabilizationUtil::stabilizePhysicsSystemInerti` function.
- Increase Linear and Angular Damping.
- Correct bone separation with TOI constraints.

It is possible that there will be separations in the constraints of the ragdoll. Havok will do its best to correct these separations, but there are things that users can do to help with these situations as well. One cause of this separation is that one or more bones in the ragdoll get a large linear or angular velocity (see the "Shooting a Ragdoll" section of this doc). This can pull it away very quickly from the rest of the ragdoll, causing separation. Increasing the inertia tensors (which we mentioned in the "Setting up Rigid Body and Constraint Properties" section of this document) and increasing linear and angular damping on the rigid bodies can help with this. Havok offers the `hkpConstraintStabilizationUtil::stabilizePhysicsSystemInertias()` function to help evenly increase the inertia tensors for the entire ragdoll at runtime. Increasing these values makes the ragdoll stiffer, so they need to be tuned to values that strike a balance between movement and constraint separation.

The next cause is when some of the ragdoll comes into contact with another body at high velocities.

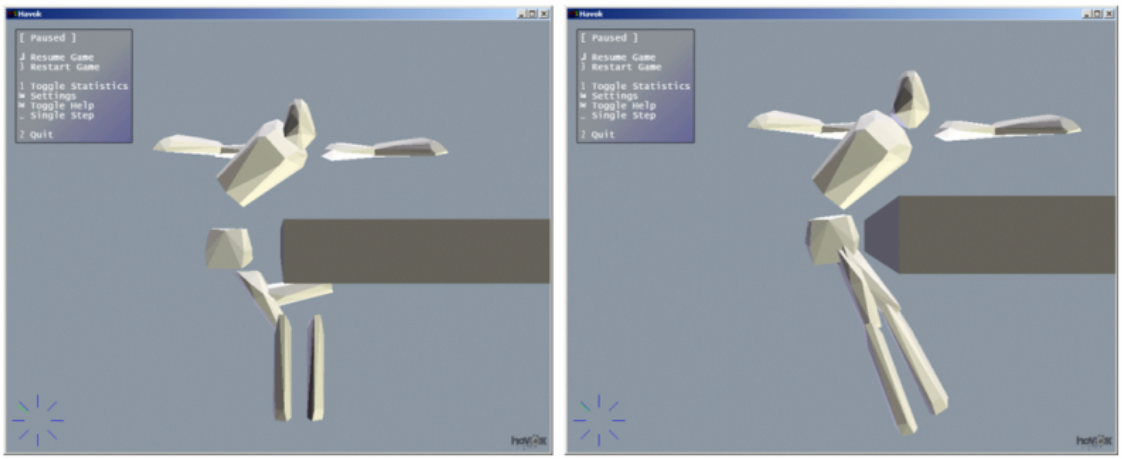



Figure 2.121: PSI constraint priority (left) versus TOI constraint priority (right)

Generally this is because in the default case ragdoll constraints are only solved in the main step, not the TOI step. If a ragdoll is going to be moving quickly and colliding with other bodies, increasing the constraint priority on the ragdoll's constraints can address this problem. With the TOI priority, the constraints will be solved during the TOI step. Note that this increases the performance cost of the ragdoll.

### Best Practices for Use Cases

#### Ragdoll Attachments

Lastly, let us mention some common use cases that come up for people. The first is how to deal with Ragdoll Attachments. These are bones not driven by animation at all, but which drive animation bones or other rendered objects, i.e. ponytails or grenades attached on a belt. When the character is animated, the non-attachment parts of the ragdoll are typically keyframed. These bones follow the animation exactly. If the animation causes high velocity movement for the ragdoll bones, then the attachments can get left behind (causing separation in the constraint). This can be mitigated by applying forces (or setting velocities) on the ragdoll attachments with some percentage of its parent's bone velocity (or force).

#### Blending into Ragdoll States

The second case is going from the animated state to the ragdoll state. Collapsing straight to ragdoll often looks unnatural. Users will generally have "death" animations which represent the boundary from animated to ragdoll to avoid this problem. As this animation plays, the `hkaRagdollPoweredConstraintController` can be used to easily blend from animation to ragdoll, giving a more visually appealing result. The strength of the powered constraint motors determines how closely it will match the animation. As the animation plays, decreasing the strength down to zero slowly will blend in more and more Physics-driven behavior and less of the animation.

#### Shooting a Ragdoll

The final use case is how to "shoot" a single bone of a ragdoll and not have it become unstable. Having ragdoll reactions from forces applied to individual bones is common especially in the shooter genre. If the full force of the bullet is applied directly to a single bone, instability and constraint separation are possible. A better way is to apply some of the force to desired bone, and then apply less and less force to the adjacent bones.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

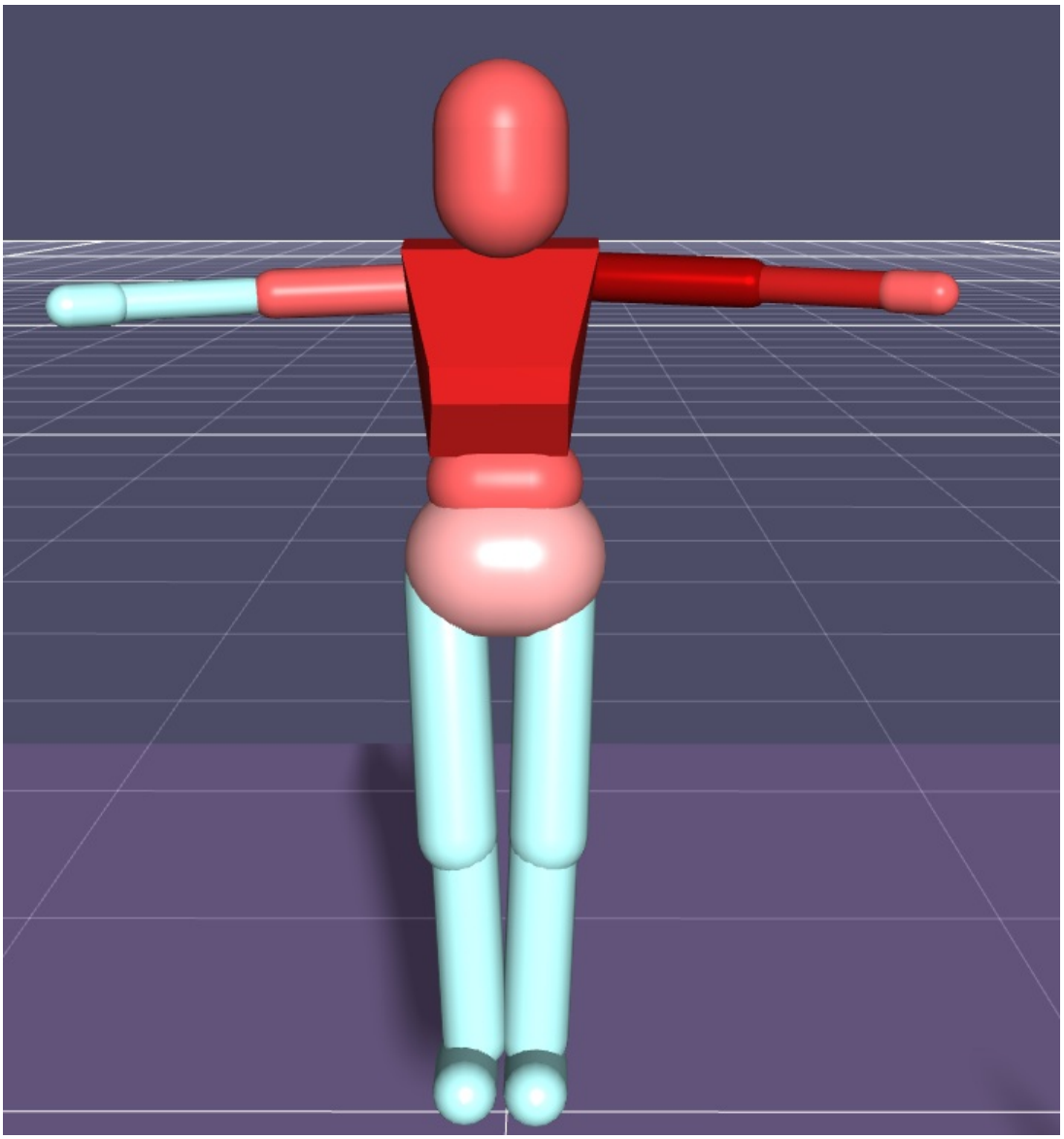


Figure 2.122: Large force applied to left arm and then distributed to nearby bones

This usually gives a better visual result as the entire body will react to the force. Manually adding extra angular velocity to the root and torso can also help when trying to get a "spin death" effect on a character.

# Chapter 3

# Havok Animation

## 3.1 Introduction

Welcome to the Havok Animation Manual. Havok's Physics and Animation solutions form an integrated game animation, dynamics and collision detection pipeline providing a complete solution for a game's animation and dynamics requirements. Havok Animation is a fully featured game animation runtime with customizable asset pipeline.

Havok Animation has a number of key features:

- Highly optimized and feature rich skeletal animation runtime for games on all Havok supported platforms. The animation system features a state of the art compression pipeline enabling more animation data to be packed into the available memory on each of the supported platforms.
- A customizable asset pipeline with runtime class reflection model for on target tools creation.
- Full integration with Havok Physics for character dynamics and control.
- Full integration with Havok Behavior for interactive character behavior authoring.

Together with Havok Physics and Havok Behavior, Havok Animation empowers users with a complete solution for next generation character animation and dynamics, including runtime animation sequencing and productivity tools.

## 3.2 Architecture Overview

The Havok Animation system uses the existing Havok runtime platform, including platform optimized math libraries, the `hkbase` library (for object and memory management), serialization and the Havok Exporter framework, providing both a complete end to end animation pipeline and an extensible framework to facilitate embedding the animation runtime and tools into an existing game production pipeline.

### 3.2.1 Object Overview / Glossary

This is a brief description of some of the more important classes in the runtime system. We use the acronyms AC and AR to indicate a class that is typically allocated at construction time (load time) or runtime (stack / heap allocated). Classes marked with AC are typically shared. Abstract classes are *emphasized*.

#### 3.2.1.1 Animation Classes

**hkaBone - AC**

- Describes a bone as it appears in the tool chain i.e. has a name, and rigging information. This is not the efficient structure we use at runtime to construct and blend pose information.

**hkQsTransform - AC, AR**

- This represents a decomposed 4x4 transform. It explicitly stores translation, rotation (as a quaternion) and scale along X, Y and Z. When combining or sampling animations we force them to drop down to this structure.

**hkaPose - AR**

- A class which holds a syncronized local and model space pose (as arrays of `hkQsTransform`s), and an array of float slots (as an array of hkReal). Allows you to work in either local or model space seamlessly and efficiently - it performs lazy updating.

**hkaAnnotationTrack - AC**

- A tag for an animated variable (usually a `hkQsTransform`). Contains a name and a set of time stamped annotations.

**(Hierarchy) - AC**

- Not actually a class in the system. See `hkaSkeleton` definition below. Defines a parenting relationship between bones. A root bone has no parent. Currently represented as an array of 16 bit integers.

**hkaSkeleton - AC**

- A set of hkaBones, a reference pose (e.g. T-Pose) and a Hierarchy, and a set of float slots (char*). Either one of the bone or float slot sets may be empty. May have multiple root bones.

*hkaAnimation - AC*

- An (abstract) interface.
- Represents a pose that changes over time. Has a finite period (in seconds). Can be sampled continuously. Can exist in any space for the bone data, but is usually in local space.

**hkaAnimationBinding - AC**

- Provides a mapping from animation tracks indices to skeleton indices for both bones (hkQsTransforms) and float slots (hkReal). Useful for partial animations (e.g. Upper body only).

*hkaAnimationControl - AR*

- An (abstract) interface.
- Provides runtime animation playback control. Controls the local time and overall weight for a playing animation.

**hkaAnimatedSkeleton - AR**

- Represents the runtime instance of a specific character. Keeps track of all active animations running on this instance. Samples and blends all active animations and extracted motion / locomotion. Provides support for per bone feathering and masking.

*hkaAnimatedReferenceFrame - AC*

- An (abstract) interface.
- This interface exposes the motion information for an animation, which is normally 'extracted' from e.g. the root bone.

#### 3.2.1.2 IK Classes

**hkaCcdIkSolver - AR**

- This is a general solver for chains of bones. It is an iterative solver. It has no support for limits.

**hkaFootPlacementIkSolver - AR**

- This is a specialized analytic IK solver designed to handle foot placement. It uses raycasting to find ground height information. Internally it uses the hkaTwoJointsIkSolver.

**hkaLookAtIkSolver - AR**

- This solver operates on a single bone and orients the bone so it faces a specified target. The solver has full support for cone limits.

**hkaTwoJointsIkSolver - AR**

- This analytic solver is designed to handle limbs such as legs and arms. It assumes the bones represent a ball and socket and hinge construction e.g. Hip and Knee or Shoulder and elbow. It has support for hinge limits. It has weighting support for gracefully easing in / out the result.

**hkaThreeJointsIkSolver - AR**

- This analytic solver is designed to handle three-jointed limbs. It assumes the first joint is resented by a ball and socket and that the second and third joints are represented by a hinge. It has does not support hinge limits and has no weighting support.

#### 3.2.1.3 Deformation Classes

**hkaMeshBinding - AC**

- Contains the bind pose required for skinning. Provides a mapping from the bone indexes in a skinned mesh to the bones of a given skeleton. Useful when the tool chain has to split meshes (e.g. For Hardware rendering with a bone limit).

*hkaMorphingDeformer - AR*

- An (abstract) interface.
- Takes 2 input streams (Vertex Buffers) and interpolates to an output stream. The interface forces the user to bind the streams to the deformer first - this allows us to do error checking and set up the streams for efficient processing. Can bind once deform often (if binding is a slow process).

*hkaSkinningDeformer - AR*

- An (abstract) interface.
- Takes a single input stream and skins to an output stream. The interface forces us to bind the streams to the deformer. User supplies input matrices when deforming. Bind once deform often.

### 3.2.2 Export Path

A critical component of any game production pipeline involves extracting information from the content creation tools (in particular the modelers), serializing this to an extensible stream format and efficiently loading, storing and accessing these assets at runtime. Havok's serialization framework is designed to integrate easily into existing pipelines, and Havok Animation provides a set of modeler export utilities, file formats and APIs which are fully customizable and extensible and which integrate seamlessly with the physics product.

#### 3.2.2.1 Design Philosophy

Our philosophy is to provide a set of very simple components and formats that fit easily into your existing pipeline. These are built on our robust serialization infrastructure and are provided with full source. Our asset processing is carefully divided into modeler specific and modeler independent sections. This separation allows you to extend and enhance this pipeline quickly without having to learn any

specific vendor API (including ours!). Once extended you leverage the full power of the system and get features like version control and batch processing for free.

In later releases we intend to extend this infrastructure to on-target asset viewing and tweaking.

#### 3.2.2.2 Serialization Infrastructure

Our core serialization infrastructure uses standard object reflection techniques. Since serialization is common to both Havok Animation and Havok Physics, a full explanation of how it works can be found in the Common Havok Components chapter of this manual.

#### 3.2.2.3 Thin Export Utilities and the Filter Pipeline

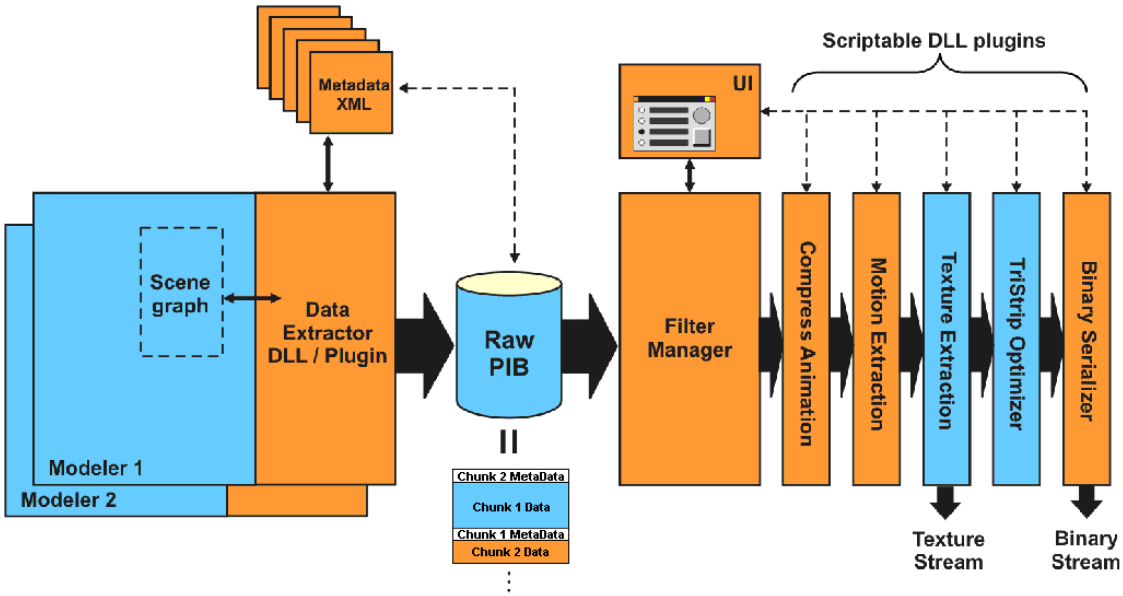



Figure 3.1: The Havok Asset Filter Pipeline

The asset pipeline that ships with Havok Animation is made up of a number of functional components as illustrated in the diagram. We provide "thin" export utilities for 3DS Max and Maya. These utilities have little user interface associated with them. Their job is to walk through the scene graph of the tool and extract specific data found into a single data stream. This data stream is then passed to pipeline of filters. Each filter in the pipeline can modify extent or remove data from the stream. Typical filters include scene transformation, rig and animation extraction or compression, data retargeting for a specific console or even data previewing. The section entitled 'The Havok Filter Manager' describes each of the current filters in detail.

The specific set of filters that process a given asset are interactively composed by an artist or technical director. This set of filters, together with any specific options, is automatically saved with the original assets. When the asset is next edited it retains the last set of filters and options. This simplifies both batch processing and asset maintenance. The filter manager also allows the user to load/save options from/to an external location.

## 3.3 Animation Runtime

### 3.3.1 Animation Overview

The Havok animation library provides a full runtime animation solution. Major animation runtime features include:

- Playback Control
- Sampling and Blending
- Compression/Decompression
- Motion Extraction
- Inverse Kinematics
- Retargeting
- Mirroring
- Pose Representation and Transform Utilities
- Havok Physics Integration
- Serialization and Tool Support (see the Havok Content Tools Chapter)

#### 3.3.1.1 Design Philosophy

Havok's animation runtime is designed specifically to address the growing difference between CPU performance and memory bandwidth. During the design we made the following tradeoffs:

- Where feasible the system trades CPU performance for memory savings.
- Data pipeline interruptions are avoided where possible; you won't see many callbacks in the API.
- We maximize shared data coming from the tool chain and minimize our runtime heap allocations.
- The user will want complete control over when and where animations are sampled and blended–we impose no scheduling on the animation pipeline and leave that to your game loop.

#### 3.3.1.2 Animation Runtime Modules

The figure below gives an overview of the modules in the animation runtime.

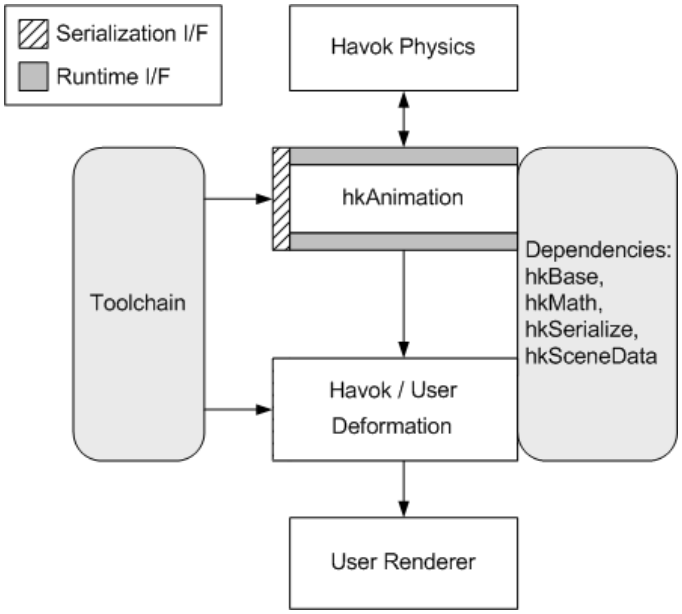



Figure 3.2: Runtime Animation Modules

Most object instances in the animation system can be shared. Animation, Rig and Mesh data, together with appropriate binding information come from the tool chain, through the serialization interface, and can be reused throughout your game. We have tried to keep the per instance data required for each character to a minimum. Only additional objects that deal with animation sampling (a control) and blending (an animated skeleton) are needed to have a fully animated character in your runtime.

The diagram below shows the most important relationships between classes in the animation runtime.

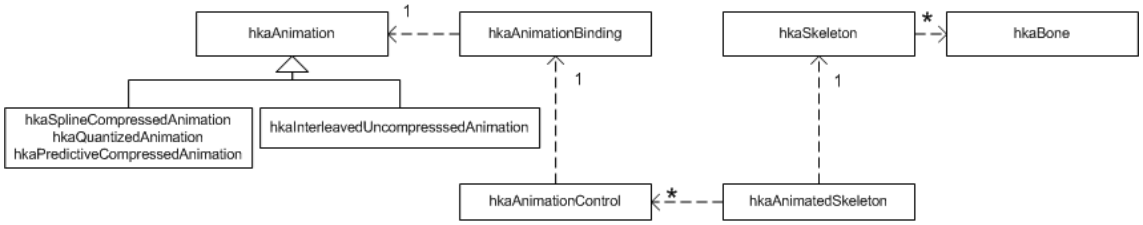


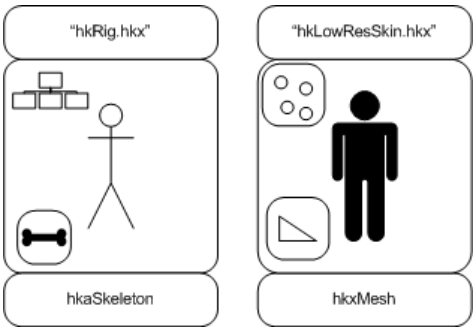

Figure 3.3: Animation Runtime Class Relationships

Only `hkaAnimationControl` and `hkaAnimatedSkeleton` are allocated at runtime. All other instances are created at load time and shared. The next sections go through each of these classes and their interaction in detail.

#### 3.3.1.3 Dataflow Example

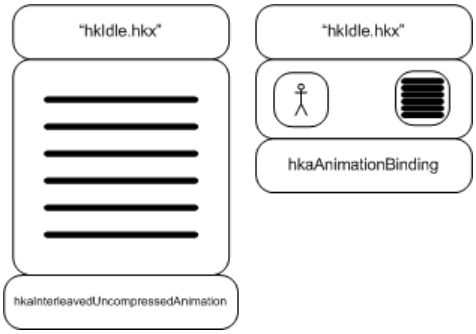
To illustrate how the various loaded and runtime-allocated objects can be used we'll walk through an example. The source code for this is in the `Demo\Demos\Animation\Api\MeshAndDeformation\Skinning` demo.

Initially we load a both a rig (`hkaSkeleton`), and a mesh (`hkxMesh`):



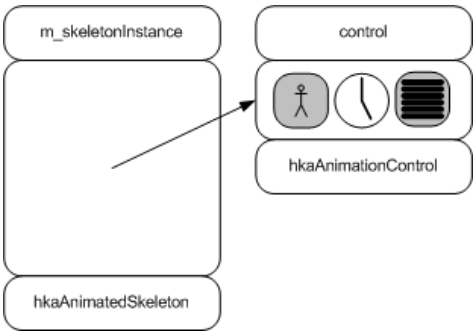

These stay in main memory as long as the character is being displayed/used. The `hkxMesh` represents a bind pose mesh. It contains a vertex buffer (shown as dots) and an index buffer (shown as triangles). The `hkaSkeleton` contains bones, a hierarchy, and bind pose info.

Next we bring the required animation (`hkaAnimation`) and the binding (`hkaAnimationBinding`) for the skeleton into main memory:




The binding shows how to map the animation to the target skeleton. Both of these reside in memory as long as this animation is required.

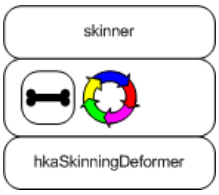
We create a `hkaAnimatedSkeleton` ('`m_skeletonInstance`' on the heap). This is only required when an animation is playing on the character. Once created, we use the `hkaAnimationBinding` to create a `hkaAnimationControl` ('`control`' on the heap). We then add `control` to the `hkaAnimatedSkeleton`.




Now `m_skeletonInstance` can be queried for its current pose via `sampleAndCombineAnimations()`. It will access the control (which stores time and weight information), sample the bound animation, and blend this to produce a local pose for the skeleton.

To skin this result we copy an instance of the original mesh. The index buffer is shared with the original mesh. If we're just going to skin and then render immediately we could create the copy of the vertex buffer on the stack.

Finally we create a deformer on the stack and bind it to the new output vertex buffer for the mesh instance and the original bind pose vertex buffer. The binding process may do complex things like download the vertices to the graphics card or vector unit. We take the sampled pose (from the `hkaAnimatedSkeleton`), convert it to world space, and combine its matrices with the original Skeleton inverse bind pose matrices (all locally on the stack). We check the mesh binding to arrange these matrices and then pass them to the `hkaSkinningDeformer`. The output vertex buffer is deformed and ready to be rendered.



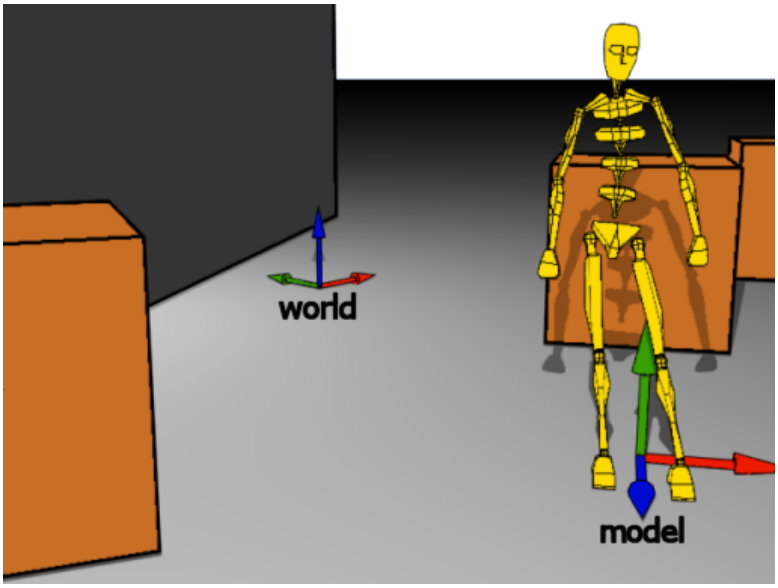

For each character we only require heap memory for the `hkaAnimatedSkeleton` and the space for each active controller. All other data is either temporary or shared (assuming we don't need to store the output vertex buffer).

### 3.3.2 Pose Representation

#### 3.3.2.1 World, Model and Local Spaces

Throughout this documentation we will be using terms like World Space, Model Space and Local Space to describe transforms and poses. In this section we are going to describe the convention used to name those spaces.

**World and Model Spaces**




**World space** specifies "where the world is", including orientation of the world (what direction is up, for example). At any time, since there is usually only one world, you only have one world space in your application ("the world space").

**Model space** is specific to a character in the application, and it specifies "where the character is", including orientation (what direction is up, forward, right, etc). Therefore, you have a model space for each character in the game ("the model space of the character").

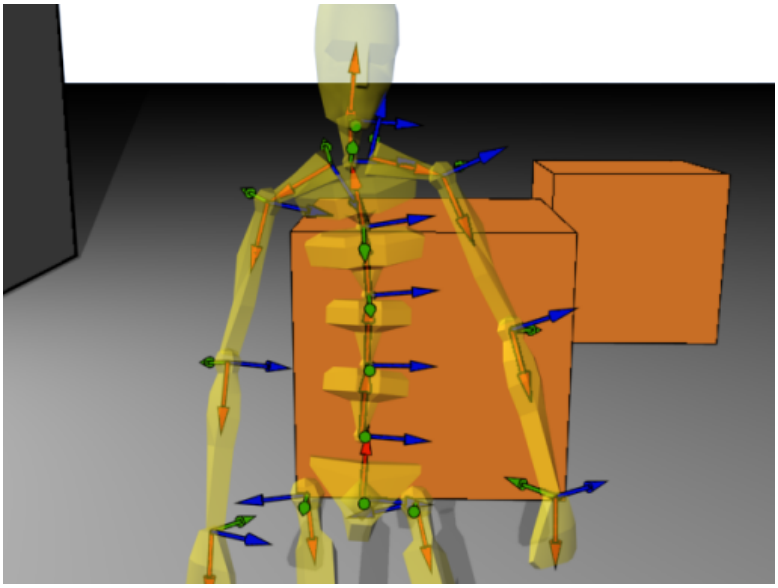
Notice that those two spaces are decoupled; for example, in world space the up direction can be the Z axis, while in model space the up direction can be Y instead (like in the diagram above, where the X, Y, and Z axes are represented with red, green, and blue). In order to place characters in the world, i.e, in order to specify "where the character is in the world", you need to describe the model space defined in respect to the world space. This is what we call the world-from-model transform (the `worldFromModel` transform[1]).

The world-from-model transform is usually tracked by the application. For example, it may be set based on output from the AI, the input from the user, the position of a physical character controller, the motion accumulated from the animations played on the character, or a combination of these.

### Bone Space

Apart from the world and model spaces, when working with skeletal animations, there are also spaces associated with each bone in the skeleton:

[1] The world-from-model transform is also the model-to-world transform. We usually use the "from" convention since, in Havok, transforms are pre-multiplied and therefore the "from" convention reads naturally from left to right: It's easier to read b_from_a = b_from_c * c_from_a rather than a_to_b = c_to_b * a_to_c



For each bone B, the associated space is called the **space of bone B** . Very often this is also called the **local space of bone B** , although this is somewhat redundant.

### World, Model and Local Space Poses

When working with animations, we are interested in describing the location and orientation of each bone, i.e., the space of each bone (the collection of all of these spaces is what we call a *pose*). As always, we need to define these spaces with respect to another space, by using a transform. The choice of the space determines whether our pose is a world, model or local space pose:

- If we define our bone spaces with respect to the world space, our pose is a **world space pose** : each transform in the pose is a `worldFromBone` transform.
- If we define our bone spaces with respect to the model space, our pose is a **model space pose** : each transform in the pose is a `modelFromBone` transform
- If we define our bone spaces with respect to the bone space of its parent bone, our pose is a **local space pose** [2] : each transform in the pose is a `parentBoneFromBone` transform. By definition, the parent space of the root (which has no parent) is the model space.

Notice that model space and local space representations of a pose contain the same information, just using a different representation - you can convert from a model space pose to a local space pose and vice versa without any extra transforms[3]. The hkaPose class allows you to work simultaneously in both representations. In order to convert from/to these representations to/from world space, you will need the `worldFromModel` transform.

### Converting between Spaces

The `hkaSkeletonUtils` class in the `hkaAnimation` library contains a number of very useful static methods for manipulating poses and skeletons. These methods allow you to:

[2] A possibly more adequate name would be *parent space pose*. But local space pose is the most commonly used name.
[3] Although you will need the hierarchy (the index of the parent bone for each bone) to do this conversion.

- Convert poses to and from World, Model and Local space.
- Perform weighted blending of poses and bone chains (i.e. partial poses).
- Enforce bone constraints in `hkaSkeleton`s in both Model and Local space (currently the only bone constraint is 'lock translation').
- Find bone ids by name.
- Populate bone chains given the bone ids of the start and end bone (where the end bone is deeper in the hierarchy).
- Find all of a bone's descendants by id.
- Check that a `hkaSkeleton`'s animation binding is valid.

Please refer to the Havok Reference Manual or `hkaSkeletonUtils.h` for more details.

#### 3.3.2.2 Using hkaPose

**Introduction**

When writing application code, you may find that very often you have to switch between the local and model space representation of the bones of a character[4]Some operations, like blending, work better in local space, while some others, like mapping from one skeleton to another, or IK, work in model space. Also, sometimes you may want to modify the model space transform of a bone while still keeping the local space transform of its children, hence working in both spaces at the same time. The `hkaPose` class helps you working with this dual (local+model space) representation of poses seamlessly and efficiently. Consider the following example:

**An Example**

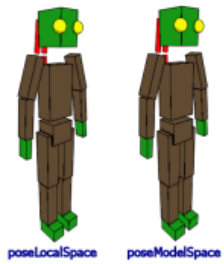
In this example, we'll want to apply Look At IK to a pose taken from an animated skeleton (`hkaAnimatedSkeleton`). We start by sampling the pose from that animated skeleton:

```
// We start with a pose in local space
hkLocalArray<hkQsTransform> poseLocalSpace(skeleton->m_numBones);
poseLocalSpace.setSizeUnchecked(skeleton->m_numBones);
animatedSkeleton->sampleAndCombineAnimations(poseLocalSpace.begin());
```

[4] For more information about these spaces, please check the World, Model and Local Spaces section at the beginning of the Animation Runtime chapter

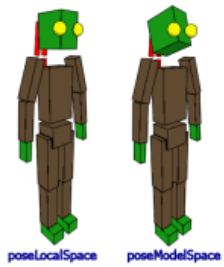
Now, we have checked the documentation of the Look At IK solver and noticed that it works in model space (it modifies a bone in model space). Thus, we need to convert the pose to model space:

```
// We convert it into model space
hkLocalArray<hkQsTransform> poseModelSpace(skeleton->m_numBones);
poseModelSpace.setSizeUnchecked(skeleton->m_numBones);
hkaSkeletonUtils::transformLocalPoseToModelPose(skeleton->m_numBones, skeleton->m_parentIndices,
  poseLocalSpace.begin(), poseModelSpace.begin());
```



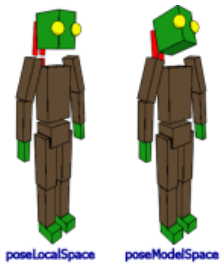

Good. We can now call the IK solver with the bones in the model space pose:

```
// We do IK on the head to look at the enemy
hkaLookAtIkSolver::Setup ikSetup;
ikSetup.m_fwdLS.set(0,1,0);
ikSetup.m_limitAxisMS.setRotatedDir(poseModelSpace[neckIndex].getRotation(), hkVector4(0,1,0));
ikSetup.m_limitAngle = HK_REAL_PI / 5.0f
hkaLookAtIkSolver::solve(ikSetup, enemyPositionMS, 1.0, poseModelSpace[headIndex]);
```



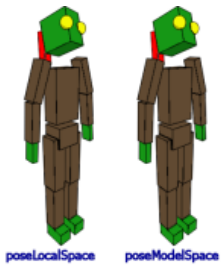

Well, the IK did his job and modified the head to point in the right direction. Notice, however, that we have only modified the model space transform of the head. What happens to children bones of the head, like the eyes or the pony tail? We want those to rotate with the head. At the moment, we only updated the model space transform of the head; we'll need to update the children accordingly:

```
// Ok, that moved the head, but what about the eyes and the ponytail?
// We want to keep the children local transform respect to their parent....
hkArray<hkInt16> childrenOfHead;
hkaSkeletonUtils::getDescendants(*skeleton, headIndex, childrenOfHead);
for (int i=0; i<childrenOfHead.getSize(); i++)
{
    hkInt16 parentIdx = skeleton->m_parentIndices[childrenOfHead[i]];
    poseModelSpace[childrenOfHead[i]].setMul(poseModelSpace[parentIdx], poseLocalSpace[childrenOfHead[i]])
      ;
}
```




Good. This has now updated the model space transform of those bones. However, the local space transform array hasn't been updated yet, so at the moment `poseLocalSpace` and `poseModelSpace` don't describe the same pose. If we need the new pose in local space, we'll have to update it too:

```
// Update the local space pose
hkaSkeletonUtils::transformModelPoseToLocalPose(skeleton->m_numBones, skeleton->m_parentIndices,
  poseModelSpace.begin(), poseLocalSpace.begin());
```




Fine, this will work; it will recalculate all the local space transforms according to the new model space pose. However, thinking about it, it would actually have been enough to do this:

```
// This would have been enough. Neck is the parent of Head
// neckFromHead = neckFromModel * modelFromHead = inverse(modelFromNeck) * modelFromHead
poseLocalSpace[headIndex].setMulInverseMul(poseModelSpace[neckIndex], poseModelSpace[headIndex]);
```

Since when we updated the children of the head, we already made sure they kept their original local transform. So the head's local transform was the only local transform to update. By noticing this, we only used one transform multiply instead of n (where n is the number of bones in our skeleton).

If you imagine now the case were we applied LookAt IK to not only the head but also both eyes, followed by other IK operations, mapping and blending, you will notice that tracking the transforms that have been modified and those that need to be updated may become a complicated task which requires a fair amount of bookkeeping and where conceptually simple changes, like changing the order of two IK operations, will easily break the code.

In order to facilitate operations with poses, Havok Animation provides the `hkaPose` class. This class has the following properties:

- It represents a pose for a skeleton.
- It operates with both local and model space representations of this pose.
- These representations are stored but only updated when required (lazy update).
- Changes on individual transforms are tracked and only the minimum number of transforms are flagged for update.
- Operations (like hierarchy traversing and updates) have been optimized and do not require allocations (temporary arrays, etc.).
- Direct access to both model and local space pose representations is also available.
- All data can be stored using stack-based allocation to speed up construction and destruction.

The example above, if using `hkaPose`, would look like this:

```
// We start by filling a pose with an array of transforms in local space
hkaPose thePose(skeleton);
animatedSkeleton->sampleAndCombineAnimations(thePose.getUnsyncedPoseLocalSpace().begin());
```

```
// We do IK on the head to look at the enemy
hkaLookAtIkSolver::Setup ikSetup;
ikSetup.m_fwdLS.set(0,1,0);
ikSetup.m_limitAxisMS.setRotatedDir(thePose.getBoneModelSpace(neckIndex).getRotation(), hkVector4(0,1,0));
ikSetup.m_limitAngle = HK_REAL_PI / 5.0f
hkaLookAtIkSolver::solve(ikSetup, enemyPositionMS, 1.0, thePose.accessBoneModelSpace(headIndex, hkaPose::
  PROPAGATE));
```

And that's it. Notice the following things:

- We don't need to convert the pose to model space at the beginning. When accessing the model space transform of the head (to pass it to the IK solver), the model space transform of the head (and its parents) gets calculated and updated in the pose. The model space transforms of other bones (like the children of the head) is not calculated; there is no need for it yet (and as we know they will change once the head is rotated, so it would be wasted effort).
- After the IK, we don't need to update the model space transform of the children of the head yet. By passing the *PROPAGATE* flag, we told the `hkaPose` that modifications on the head should propagate to the children. The `hkaPose` then knows that the local space transforms of those children are still valid and hence don't need to be modified. Their model space transformation won't be calculated until somebody asks for it (so, if we now do further IK, we haven't wasted the effort)
- Similarly, we don't need to update any local space transforms yet. If in the future we ask for any them, the `hkaPose` instance will know that only the local space transform for the head is not up-to-date, but all the rest are.
- Apart from the performance gain caused by the reduced number of operations, the code is now much simpler. There is no bookkeeping required since, from the application point of view, `hkaPose` is always up-to-date, both in model and local space.

The following sections will give you more detail on how hkaPose works and how to make the most of it.

**Overview**

`hkaPose` objects keep a dual (local space and model space) representation of a pose (a set of n transforms for the n bones of a skeleton). Both representations are kept in sync on demand (lazily), so only the minimum amount of calculations are done whenever the pose is modified or accessed. The figure below shows a conceptual representation of an `hkaPose` object:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Parent | | 0 | 0 | 2 | 3 | 4 | 5 | 6 | 1 | 8 |
| LocalPose | L | L | L | L | | L | L | L | L | L |
| LocalDirty | | | | | x | | | | | |
| ModelPose | M | | M | M | M | | | | | |
| ModelDirty | | x | | | | x | x | x | x | x |

In this example, most transforms in the local space representation of the pose are stored up-to-date (except for bone 4), while only a handful of model space transforms are (bones 0, 2, 3 and 4). But notice that both the local and model space transformation of any bone can be calculated at any time based on the information stored.

**Note:**

For ease-of-use the `hkaPose` class also contains a 'Float Slot Values' array into which float tracks may be sampled. Since all float slots are treated completely independently (there is no concept of 'local/model' space, and no hierarchy for float slots), the management of this part of the `hkaPose` class is trivial, and we will not go into it in detail here.

### Creating an hkaPose object

There are three ways to create an instance of `hkaPose`. All constructors require a pointer to the skeleton associated with the pose.

- Create an uninitialized pose:

```
hkaPose (const hkaSkeleton* skeleton);
```

  This constructor allocates the memory required to store the pose using calls to `hkArray.setSize()`, which in turn will call `hkAllocateChunk()`.

  This is the simplest constructor. Since the memory is allocated using normal allocation routines, it may be slower than the stack-based allocation constructor (described next).

  **Warning:**

  This and the next constructor create an uninitialized hkaPose instance. This means that none of the stored transforms in the pose are valid, and henceforth, the pose needs to be initialized before it's used. You should always call either `setToReferencePose()`, `setPoseLocalSpace()`, `setPoseModelSpace()` or use the write-only methods `getUnsyncedPoseLocalSpace()` or `getUnsyncedPoseModelSpace()` to initialize the pose after using these constructors. All these methods are explained later in this chapter.

- Create an uninitialized pose in preallocated memory:

```
hkaPose (const hkaSkeleton* skeleton, void* preallocatedMemory);
```

  This constructor creates an `hkaPose` instance and uses a preallocated memory buffer for storage. You can get the size of the required buffer by using the static method `getRequiredMemorySize (hkaSkeleton*)`. This is particularly useful if you want to use a fast memory allocation routine, like stack-based memory allocation (check the Havok Base Library documentation for more details). The most common usage is presented below:

```
// We pre-allocate the memory in an hkLocalBuffer
hkLocalBuffer<char> localBuffer( hkaPose::getRequiredMemorySize( skeleton ) );
hkaPose thePose(skeleton, localBuffer.begin());
```

- Create an initialized pose:

```
// enum PoseSpace { MODEL_SPACE, LOCAL_SPACE };
hkaPose (PoseSpace space, const hkaSkeleton* skeleton, const hkArray<hkQsTransform>& pose);
```

This constructor initializes an hkaPose instance with a pose defined either in local or model space (as defined by the *space* parameter). It is equivalent to calling the `hkaPose(skeleton)` constructor followed by either `setPoseLocalSpace(pose)` or `setPoseModelSpace(pose)`.

### Working with Single Bones

#### getBone() methods (direct read only)

You can retrieve the model or local space transform of any bone by using the following methods:

```
const hkQsTransform& getBoneLocalSpace (int boneIdx) const;

const hkQsTransform& getBoneModelSpace (int boneIdx) const;
```

If the transform in question was up-to-date, its value will be returned; otherwise it will be calculated (and stored, alongside any other bone transforms that had to be calculated).

**Tip:**
If your algorithm is going to be asking for the local or model transform of all the bones in the pose, consider the alternative of using the `getPose()` methods for optimized read-only access to a single pose representation. These methods are described in the Working with Multiple Bones section.

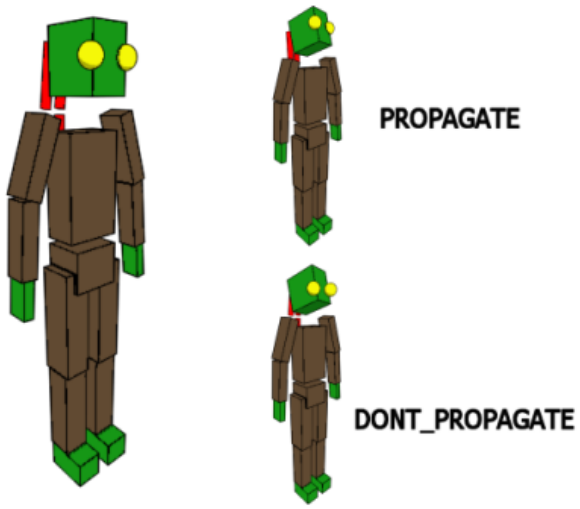
#### setBone() methods (write only)

You can also modify the local or model space transform of any bone using the methods:

```
void setBoneLocalSpace (int boneIdx, const hkQsTransform& boneLocal);

//enum PropagateOrNot { DONT_PROPAGATE, PROPAGATE};
void setBoneModelSpace (int boneIdx, const hkQsTransform& boneModel, PropagateOrNot propagateOrNot);
```

Setting the transform of a bone will store that transform in the `hkaPose` object, clean its "dirty" flag and possibly update or dirty the transforms of other bones affected by the change.

When setting the transform of a bone in model space, you can choose whether the model space transforms of its descendants are affected as well (*PROPAGATE*) or not (*DONT_PROPAGATE*), using the *propagate* parameter.




In the example above, we have set the transform of the head in model space. Notice how using the *PROPAGATE* flag modifies (flags as "dirty") the model space transforms of the descendants of the head (eyes and pony tail), while using the *DONT_PROPAGATE* flag keeps the model space transform of those bones (but, in turn, invalidates the local space transform of the children of the head).

When using `setBoneLocalSpace`, transformations are always propagated (i.e., the local space transforms of the children are always kept).

**Tip:**
If your algorithm is going to set the model or local space transforms of all bones in the pose, consider the alternative of using `setPose()` and `getUnsyncedPose()` methods for optimized write-only access (or the `getSyncedPose()` methods for optimized read+write access) to a single pose representation. These methods are described in the Working with Multiple Bones section.

### accessBone() methods (direct read+write)

You can also get direct access (a non-const reference) to a bone transform using the methods:

```
hkQsTransform& accessBoneLocalSpace (int boneIdx);
hkQsTransform& accessBoneModelSpace (int boneIdx, PropagateOrNot propagateOrNot = DONT_PROPAGATE);
```

Before returning, these methods will ensure that the transform in particular is up-to-date (read access). They will also assume that you are going to *immediately* change the value of the transform (write access), and so they will also go ahead and invalidate other transforms as required (the *propagateOrNot* flag works as it did in the `setBoneModelSpace()` method).

These methods can be used to perform some operations without requiring temporary variables, for example the following code:

```
hkQsTransform tempTransform = thePose.getBoneModelSpace(handIdx);
tempTransform.m_rotation.mul(extraRot);
thePose.setBoneModelSpace(handIdx, tempTransform, PROPAGATE);
```

This can be written as:

```
thePose.accessBoneModelSpace(handIdx, PROPAGATE).m_rotation.mul(extraRot);
```

The other common reason for using the `accessBone()` methods is to modify a pose through a method that operates on a transform. For example, the Look At IK solver modifies a transform rather than a full pose (as we saw in the introductory example):

```
// hkaLookAtIkSolver::solve() takes a model space transform as an in+out (read+write) parameter
hkaLookAtIkSolver::Setup ikSetup;
ikSetup.m_fwdLS.set(0,1,0);
ikSetup.m_limitAxisMS.setRotatedDir(thePose.getBoneModelSpace(neckIndex).getRotation(), hkVector4(0,1,0));
ikSetup.m_limitAngle = HK_REAL_PI / 5.0f
hkaLookAtIkSolver::solve(ikSetup, enemyPositionMS, 1.0, thePose.accessBoneModelSpace(headIndex, hkaPose::
  PROPAGATE));
```

**Warning:**
When working with `accessBone()` methods, it is important to ensure that the non-const reference to the bone transform is used inside a limited scope (like in the examples above). The golden rule is: **References to bone transforms inside the hkaPose object should only be used for a short scope, and should never be used after another hkaPose method is called** .

### Working with Multiple Bones

On some occasions, you may find that you want to operate on all bones in a single space: for example, when skinning you may require the model space transforms of all bones; or, when blending, you may need to access all the local space transforms. Also, you may need to interface to some Havok Animation (or your custom) components that take poses as arrays of transforms in a single space, rather than `hkaPose` objects.

In order to facilitate all this, there are different methods in hkaPose that allow you to get read, write, and read+write access to a single representation of all bones in the pose.

### getPose() methods (direct read only)

You can get a const reference to full local space or model space representation of a pose by using the methods:

```
const hkArray<hkQsTransform>& getPoseLocalSpace() const;
const hkArray<hkQsTransform>& getPoseModelSpace() const;
```

These methods will ensure the internal representation of the local or model space pose is up-to-date, and will return a reference to that representation (an array of transforms). This is particularly useful if you need to interface with an object or method that takes a pose specified as an array of transforms rather than an `hkaPose` object.

Notice that since these methods return a reference to internal data, this data is only valid as long as no

non-const methods are called on the `hkaPose` object.

### setPose() methods (write only)

You can set the model or local space transform of all bones in a pose by using the methods:

```
void setPoseLocalSpace (const hkArray<hkQsTransform>& poseLocal);
void setPoseModelSpace (const hkArray<hkQsTransform>& poseModel);
```

These methods will store the given pose (in local or model space) and invalidate the alternative representation.

### getSyncedPose() methods (direct read+write)

You can get a non-const reference to the whole pose in a single space (local or model) by using one of these methods:

```
hkArray<hkQsTransform>& getSyncedPoseLocalSpace();
hkArray<hkQsTransform>& getSyncedPoseModelSpace();
```

The `getSynced()` methods will ensure all transforms are up-to-date for that particular representation, and will return a non-const reference to that representation (as an array of transforms). Since the reference is non-const, it will assume that you are going to modify those transforms and, therefore, it will also invalidate the internal transforms stored for the alternative representation (so, for example, calling `getSyncedPoseLocalSpace()` will update all local space transform and invalidate all model space transforms).

**Warning:**
As with the single-bone version, `getSynced()` expects you to use the non-const reference returned for a limited scope, and never after further calls to other methods in the hkaPose object are made.

These methods are useful if a section of your algorithm is going to work and modify all (or most of the) bones in the pose using a single space, or when interfacing to a function/object that does so (takes an input+output pose as an array of transforms in a single space).

### getUnsyncedPose() methods (direct write only)

For the times you want write-only access to a full pose in a single space (local or model), you can use one of these methods:

```
hkArray<hkQsTransform>& getUnsyncedPoseLocalSpace();
hkArray<hkQsTransform>& getUnsyncedPoseModelSpace();
```

These methods differ from the `getSynced()` methods described above by the fact that they do not ensure that the particular representation requested is up-to-date before returning a reference to it. The value of those transforms is undefined and therefore **you must set all the transforms in the array before using the hkaPose object again** . These methods assume that you will initialize *all* transforms to the

right value, and therefore will clear the dirty flag for all transforms of the requested space while setting the dirty flag for the transforms of the alternative space.

The most common usage of these methods is for the initialization of hkaPose objects, as well as to pass poses to methods or functions that fill a full pose, given as an *out* parameter, for example:

```
// hkaAnimatedSkeleton::sampleAndCombineAnimations takes an array of local space transforms as an out (
  write only) parameter
hkaPose thePose(skeleton);
animatedSkeleton->sampleAndCombineAnimations(thePose.getUnsyncedPoseLocalSpace().begin(), HK_NULL);
// Or, more correctly if the skeleton contains float slots also
animatedSkeleton->sampleAndCombineAnimations(thePose.getUnsyncedPoseLocalSpace().begin(),
  thePose.getFloatSlotValues().begin());
```

### Other Methods

#### sync() methods

You can enforce the synchronization of the local or model space internal representations, or both of them, by using the methods:

```
void syncLocalSpace () const;
void syncModelSpace () const;
void syncAll () const;
```

#### setToReferencePose()

You can set or initialize a pose to the reference pose of its associated skeleton (`hkaSkeleton`) by using the method:

```
void setToReferencePose ();
```

#### enforceSkeletonConstraints()

You can ensure that all skeletal constraints in the pose are enforced by calling one of these methods:

```
void enforceSkeletonConstraintsLocalSpace ();
void enforceSkeletonConstraintsModelSpace ();
```

Currently, the only constraint that can be defined on a skeleton is whether particular bones can translate or not (`hkaBone::m_lockTranslation`). The methods above will check and ensure that no translations have been applied to bones with that flag.

The two methods perform the same operation, but they operate on different spaces. These methods are mostly used internally.

**Caveats**

When using `hkaPose` be careful when using the `getSynced` methods in parameter lists. For example:

```
hkArray<hkQsTransform>::const_iterator rPose = ragdollPose.getPoseModelSpace().begin();
 hkArray<hkQsTransform>::const_iterator aPose = animPose.getPoseLocalSpace().begin();
 hkArray<hkQsTransform>::iterator       aPoseM= animPose.getSyncedPoseModelSpace().begin();
 mapper->mapPose( rPose, aPose, aPoseM );
```

will behave as expected, whereas

```
mapper->mapPose(ragdollPose.getPoseModelSpace().begin(), animPose.getPoseLocalSpace().begin(),
  animPose.getSyncedPoseModelSpace().begin());
```

may behave unexpectedly if the compiler evaluates the third parameter before the second parameter. The call to `getSyncedPoseModelSpace` invalidates all the local space flags and the call to `getPoseLocalSpace` will reevaluate the local pose using the out-of-date model pose.

#### 3.3.2.3 Using hkQsTransform

In Havok Animation transformations are stored with a specialized representation, `hkQsTransform`[5]. `hkQsTransform`s have the following properties:

- They explicitly store three components: translation, rotation and scale.
  - Translation is stored as an `hkVector4`.
  - Rotation is stored as an `hkQuaternion`.
  - Scale is stored as an `hkVector4`.
- They define, among others, the following operations and elements.
  - Product: `hkQsTransform` x `hkQsTransform` -> `hkQsTransform`
  - Inverse: `hkQsTransform` -> `hkQsTransform`
  - Identity value (neutral element for the product operation)
  - Interpolation: `hkQsTransform` x `hkQsTransform` x `hkReal` -> `hkQsTransform`
  - Application to a vertex: `hkQsTransform` x `hkVector4` -> `hkVector4`
- They can be converted to and from `hkMatrix4` (but, as we will see, with possible loss of information).

`hkQsTransform`s can represent any orthogonal transformation, i.e., any transformation that does not involve skewing. The reason for that is that non-orthogonal transformations cannot be represented by a concatenation of translation, rotation and scale operations.

[5] The "Q" stands for Quaternion (used to store the rotation). The "S" stands for Scale, since these transforms can store scale.

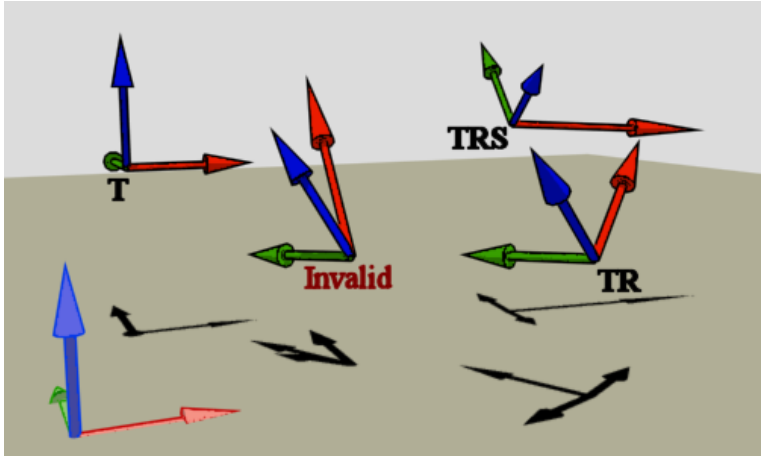



Figure 3.4: Example of different transformations (from a reference space, represented in lighter colors). Transforms **T** (translation only), **TR** (translation+rotation) and **TRS** (translation+rotation+scale) are all representable with hkQsTransform. However, the transform labelled in red cannot be represented with hkQsTransform since it contains skewing and therefore is non-orthogonal.

Notice that, since the set of valid `hkQsTransform`s (the set of all orthogonal transformations) is a subset of valid generic transforms (not necessarily orthogonal), care has to be taken when converting from one representation to the other, as that conversion may lose information (in particular about skewing)[6].

### Multiplying hkQsTransforms

#### Translation and Rotation

If A and B are two `hkQsTransform`s, multiplying them yields a result C where the translation (vector) and rotation (quaternion) components are:

- Translation(C) = Translation(A) + [Rotation(A) applied to Translation(B)]
- Rotation(C) = Rotation (A) * Rotation (B)

These operations are consistent with considering an `hkQsTransform` as the composition of a Translation (T_) and a Rotation (R_) matrices:

- C = Tc * Rc = A * B = (Ta * Ra) * (Tb * Rb)

In this case, Tc and Rc (the translation and rotation components of the result C) can be calculated as above.

#### Scale

Unfortunately, when introducing scale, the parallelism with matrices ends. This is because, if we consider the Translation, Rotation and Scale components to be matrices multiplied one after the other, then our result won't always be decomposable as an `hkQsTransform`; in other words, we would need a `hkQsTransform` C such as:

[6] In particular, many modelling tools allow the use of skewed transforms, since they use 4x4 matrix-based transformations. The Havok Animation Toolchain will always warn when performing conversions to hkQsTransform where the original 4x4 transform contains skew (as that skew will be lost).

- C = Tc * Rc * Sc = A * B = (Ta * Ra * Sa) * (Tb * Rb * Sb)

The problem is that, in the general case, it is not possible to find the translation, rotation and scale matrices Tc, Rc and Sc that satisfy the equation above. This is because concatenating rotations and scale operations can produce skewing, which yields a non-orthogonal transform, not representable with `hkQsTransform`.

So we need to find an alternative interpretation of the multiplication operation which keeps the results inside the space of orthogonal transforms. To do so, we define the resulting scale component of multiplying two `hkQsTransform`s A and B as:

- Scale(C).x = Scale(A).x * Scale(B).x
- Scale(C).y = Scale(A).y * Scale(B).y
- Scale(C).z = Scale(A).x * Scale(B).z

That is, we define the resulting scale vector as the component-wise multiplication of the scale vector. By doing this we ensure that our multiplication operation works consistently, keeps the results inside the space of valid `hkQsTransform`s, and follows the expected properties of a transform multiplication operation.

**Neutral Element (Identity)**

The neutral element (the *Identity* `hkQsTransform`) is defined such as

- A * Identity = Identity * A = A, for any `hkQsTransform` A

From the definition of the multiplication described above, the resulting neutral element (identity) is:

- Translation(Identity) = (0, 0, 0)
- Rotation(Identity) = Quaternion_Identity = 1
- Scale(Identity) = (1, 1, 1)

**Inverting hkQsTransforms**

The inversion operation is consistent with the way `hkQsTransform`s are multiplied and the neutral element is defined, and as such the inverse A' of an `hkQsTransform` A is defined as

- Translation(inv(A)) = - (inv(Rotation(A)) applied to Translation(A))
- Rotation(inv(A)) = inv(Rotation(A))
- Scale(inv(A)) = 1 / Scale(A) [for each x, y and z component]

### Interpolating hkQsTransforms

```
void hkQsTransform::setInterpolate4( const hkQsTransform& a, const hkQsTransform& b, hkSimdRealParameter t
  );
```

For convenience `hkQsTransform`s have an interpolation operation. Since this operation is used intensively in many animation-based applications, it has been implemented with speed in mind, rather than accuracy. The interpolation function is implemented as, given two `hkQsTransform`s A and B and an interpolation factor r (between 0 and 1), as returning C such as

- Translation(C) = Vector_Interpolation(Translation(A), Translation(B), r)
- Rotation(C) = Quaternion_Normalize(Vector_Interpolation(Rotation_Vector(A), Rotation_Vector(B), r))
- Scale(C) = Vector_Interpolation(Scale (A), Scale (B), r)

Vector4_Interpolation is the usual, component-wise, vector interpolation:

- Vector4_Interpolation(A, B, r) = A * (1-r) + B * (r)

Notice how we use this vector interpolation with quaternions as well, by converting them to four-component vectors and ensuring normalization after interpolation. This is much faster than performing SLERP, and returns similar results if the two rotations are in the same hemisphere.

### Blending hkQsTransforms

```
void hkQsTransform::setZero();
void hkQsTransform::blendAddMul(const hkQsTransform& other,  hkSimdRealParameter weight = 1.0f );
void hkQsTransform::blendNormalize( hkSimdRealParameter totalWeight = 1.0f);
```

Another convenience functionality in `hkQsTransform`s is the ability to blend transforms. As with interpolation, these operations are defined using vector operations on the three components. Blending of a transform B, with weight w, into a transform A results in a transform C is implemented as

- Translation(C) = Translation(A) + w * Translation(B)
- Rotation(C) = Rotation_Vector(A) + w * Rotation_Vector(B)
- Scale(C) = Scale(A) + w * Scale(B)

Notice how we don't automatically normalize the rotation component. This is because blending operations usually are performed with multiple transforms (starting with a "zero" transform) and weights, and therefore normalization is done at the end. We implement this blending normalization operation (of a blended transform A with total weight w) as:

- Translation(N) = (1/w) * Translation (A)

- Rotation(N) = Quaternion_Normalize( (1/w) * Rotation_Vector(A) )
- Scale(N) = (1/w) * Scale (A)

### Transforming vertices by hkQsTransforms

```
void hkVector4::setTransformedPos(const hkQsTransform& a, const hkVector4& b);
```

The `hkVector4` class defines the method `setTransformedPos()` which takes a position (vertex) and an `hkQsTransform`. For vertex transformation, we consider `hkQsTransform`s to represent a concatenation of Scale, Rotation and Translation operations on the vertex, such as (being P the vertex and T, R, S the components of the `hkQsTransform`):

- P' = Transform_Vertex(QsTransform(T,R,S), P) = Translate(T, Rotate (Scale(S, P)))

Or, if we consider the T, R and S components in matrix form, and P as a column vector

- P' = T * R * S * P

Notice that, for `hkQsTransform`, vertex transformation is a different operation to multiplication. This is in contrast to matrix-based transforms, where both operations are implemented as matrix multiplications. Because of this difference, some properties of Matrix4 involving these operations do not apply to `hkQsTransform`.

### Converting to and from general 4x4 matrices

You can convert `hkQsTransform`s to and from 4x4 matrices. 4x4 matrices can represent any general affine transform, while `hkQsTransform` can only represent orthogonal transforms.

### Converting to 4x4 matrix

Since the orthogonal transforms are a subset of affine transforms, you can convert to a general 4x4 matrix without loss of information, by using the method:

```
void hkQsTransform::get4x4ColumnMajor(hkReal* p) const;
```

The matrix4 is returned by filling an array of 16 floats, arranged in column-major format. That's the format used by the `hkMatrix4` class in Havok, so you can cast any `hkMatrix4` object into an array of `hkReal` and pass it to the method above.

### Converting from 4x4 matrix

Converting from a 4x4 matrix to a `hkQsTransform` is not always possible since not all affine transforms (representable by a 4x4 matrix) are orthogonal (representable by an `hkQsTransform`). The method:

```
hkBool hkQsTransform::set4x4ColumnMajor(const hkReal* p);
```

Will attempt to decompose the 4x4 matrix into the translation, rotation and scale components of an `hkQsTransform`. It will return true if the decomposition is fully successful (i.e., if the 4x4 is orthogonal); otherwise it will return false.

### Converting to and from hkTransforms

`hkTransform` is an alternative transform representation available in Havok. They usually represent orthonormal transformations, that is, transformations where there is only rotation and translation involved. This is because `hkTransform` is extensively used in Havok Physics, where scale is irrelevant.

However, and since the rotation component in `hkTransform` is stored as a 3x3 matrix, rather than a quaternion, you can set a "rotation" in an `hkTransform` that also contains scale (although we recommend that you use `hkMatrix4` in that case, and only use `hkTransform` for orthonormal transformations).

### Converting to hkTransforms

There are two conversions to `hkTransform`, depending on whether you want to ignore the scale component or not:

```
void hkQsTransform::copyToTransformNoScale (hkTransform& transformOut) const;
void hkQsTransform::copyToTransform (hkTransform& transformOut) const;
```

### Converting from hkTransforms

Similarly, there are two conversions to `hkTransform`. In this case, one will assume there is no scale (will assume the `hkTransform` holds an orthonormal transform), while the other won't. The first one (assuming no scale) is significantly faster but will fail if any scale is found in the `hkTransform`.

```
void hkQsTransform::setFromTransformNoScale (const hkTransform& transform);
void hkQsTransform::setFromTransform (const hkTransform& transform);
```

### General Properties of hkQsTransforms

Considering

- A, B, Q as being any `hkQsTransform`s
- M, N as being any affine 4x4 matrix transforms
- Q @ P as being "the transformation of vertex P by `hkQsTransform` Q"
- A * B as being "the multiplication of `hkQsTransform`s A and B"
- M * N as being "the multiplication of 4x4 matrices M and N"
- M4_To_QS (M) as being "the conversion from 4x4 matrix M to a `hkQsTransform`"

- QS_To_M4 (Q) as being "the conversion from `hkQsTransform` Q to a 4x4 matrix"
- Inv(Q) as being "the inverse of `hkQsTransform` Q"
- Inv(M) as being "the inverse of 4x4 matrix M"

The following are properties of all `hkQsTransform`:

- A * (B * C) = (A * B) * C
- A * Identity = Identity * A = A
- A * Inv(A) = Inv(A) * A = Identity
- A = M4_To_QS (QS_To_M4(A))

The following are **NOT** properties of all `hkQsTransform`s:

1. A * B = B * A
2. (A * B) @ P = A @ (B @ P)
3. Inv(Q) @ (Q @ P) = P
4. M = QS_To_M4 (M4_To_QS(M))
5. QS_To_M4 (Inv(Q)) = Inv(QS_To_M4 (Q))
6. A * B = M4_To_QS(QS_To_M4(A) * QS_To_M4(B))
7. M * N = QS_To_M4(M4_To_QS(M) * M4_To_QS(N))

(1) Transformations are not commutative. False for most cases.

(2,3) Due to the different nature of the "apply transform to vertex" and "multiply transforms" operations in `hkQsTransform`. Only true if no scale is involved.

(4) Since M can contain affine transformations not representable with `hkQsTransform`s (skew for example), information may be lost in the first conversion. Only true if M is orthogonal (no skewing).

(5) Due to the different interpretation of the inverse operation in `hkQsTransform`s and general 4x4 matrices. Only true if Q has no scale.

(6,7) Due to the different interpretation of the multiplication operation when scale is involved in `hkQsTransform` and general 4x4 matrices. Only true if A, B, M, N have no scale (are orthonormal).

**Tips when converting poses between hkQsTransform and 4x4 matrices**

The properties above, in particular 5, 6 and 7, raise the following:

**Important:**

When dealing with scale, and moving from 4x4 representation to `hkQsTransform` and vice versa, it is important to notice that operations (multiplication, inverse) in one space do not match in the other space, and therefore the results will differ depending on what representation is used.

So it is important to perform the conversion at the point where the best match is needed, because further operations will diverge in each space. This applies particularly when dealing with poses, since converting between local and model spaces involves multiplications and inversions.

For example, if you have a pose represented as an array of 4x4 matrices, each one defined in local space, and you need to represent that pose with `hkQsTransform`s, it is much better if you:

- First convert the local, 4x4 matrix-based, pose into model space (multiply each local transform by its parent)
- Then convert these model space pose of 4x4 transforms into a model space of `hkQsTransform`s by converting each transform
- Finally, if you need the local space pose, calculate it from the model space pose

By having done this, instead of directly converting the local, 4x4 matrix-based, pose to `hkQsTransform`s, we ensure that the two representations (4x4 matrix and `hkQsTransform`s) match best when considered in model space (what the artist sees in the modeler) instead of local space. This is what, for example, our export utilities do when exporting data from 3ds Max or Maya.

Similarly, if at runtime you have a local space pose using `hkQsTransform`s, and you need model space 4x4 matrices for skinning, you should convert the `hkQsTransform` local space pose to model space, and then convert the `hkQsTransform`s to 4x4 matrices. The rule is:

**Tip:**
In general, when dealing with poses, always do conversions in model space.

### 3.3.3 Animation and Rig Representation

#### 3.3.3.1 Animation Representation

The Havok Animation system supports animations of both transform data, and additional scalar data. Transform data is associated with the bones of a skeleton; skeletons also contain "slots" or "float slots" which are the analog of bones for floating point data. Most of our examples use only the bone data, usually derived from a skeletal animation system, and so an animation can be thought of as a character animation. This is the core use case for the Havok Animation system. However Havok Animation also supports arbitrary animated floating point channels, which can be used to extend a character animation with 'auxiliary' single DOF tracks. In this case, both bone and float tracks would be present (created, sampled and blended). A third use case would be to have no bone data at all, and use only single DOF float tracks, for example when representing morph target data for facial animation.

Even when no bone data is present, the same objects and object names are used for consistency, and we will refer to 'Skeletal' animations, and 'Skeletons' throughout this documentation.

**Bone And Float Representation**

Havok Animation handles two 'types' of data: transform data and float data. Transform data is associated with the bones of a skeleton and is used to define the pose of a character at a point in time. Float data is associated with the "Float slots" of a skeleton and allows for arbitrary floating point values to be animated. Both are treated independently of each other for creation, sampling and blending.

### Bone Data

Some animation systems represent a bone animation as a collection of separate channels or degrees of freedom. Using a predefined rig these channels can be transformed into either a local or world pose for a given character. This pose can then be handed to the rendering system for skinning.

When blending inputs from various sources you have to either map them back to the original rig or drop down to some other intermediate representation to do the blending. Sometimes mapping back to the original rig is difficult or costly e.g. An iterative IK system may work in general 6 DOF space. Or sometimes we need to blend animations which we originally authored using other sources e.g. MoCap data must be fitted to our rig.

Since we do not wish to impose any specific rig configuration on our animation system we choose to perform all blending in a 10 DOF[7] local pose space. We force each of our inputs, or `hkaAnimation`s, to produce a set of decomposed transforms, hkQsTransforms, in this local space. Each transform is decomposed into a format suitable for efficient blending. This set of transforms is used together with the animation binding to blend the local pose onto a given skeleton.

In Havok Animation all bone animations may be considered as a set of separate channels, however each of the bone channels contains the full 10 degrees of freedom we use to represent a bone. When we refer to a bone track we are referring to the full position, orientation and scale for a specific animated bone. Often we use the term 'transform track' to eliminate any ambiguity, and member/method names in the code may contain the string 'transform' to make things explicit, and to distinguish from 'float tracks as described below.

### Float Data

Single DOF floating point data is represented in Havok by a single DOF floating point track, or 'float track'. No assumptions about range or distribution are made (although uninitialized values will be set to 0), and blending is done using straight linear interpolation.

### Skeletal Animations

The `hkaAnimation` class provides the interface for all inputs to the animation system. The interface enforces that any implementation can provide a full set of transforms, and a full set of float values for any time period that lies within the length of the animation. To facilitate this, the interface explicitly stores the length of the animation; we refer to this as the "duration" of the animation. In addition this interface class stores useful information that has come from the tool chain. This includes a handle to any motion information that has been extracted (see Motion Extraction) together with naming and annotation data. This tool chain information is stored in an array of `hkaAnnotationTrack`s. If this array is not empty it will contain an entry for each bone track in the underlying animation.

We do not want to enforce either authoring or target sampling rates on our animations so we assume these can be sampled continuously. The core interface that enforces these restrictions is shown here.

```
virtual void sampleTracks(hkReal time, hkQsTransform* transformTracksOut, hkReal* floatTracksOut) const =
  0;
```

### Annotation Tracks

[7] 3 for position, 4 for orientation and 3 for scale.

Each bone track (but *not* each float track) is tagged with the original name used to identify it in the modeler. Often the names of the tracks in an animation will exactly match the names of the bones in the skeleton it is designed for. If you are retargeting an animation to a different skeleton or only partially animating a given skeleton it is useful to have the original tool chain names. This allows you to construct an appropriate animation binding which can be used to playback the animation on the new skeleton.

It is often useful to mark up track specific events in the modeler. Examples are 'Left ankle down at frame 5', 'Left toe up at frame 10' and similar foot placement events used to synchronize gait. Another example might annotate an event on the hand to say, at frame 5, 'handle grenade thrown now'. Different modelers have different techniques for marking up this annotation. At runtime this is baked into the annotation array which is stored in the animation track. This array contains a set of structures containing sorted events. Each event consists of a time specified in seconds and a text string which comes straight from the tool chain. The animation/annotation demo shows how you can access this data to drive game events.

**Interleaved Animations**

The simplest type of animation in Havok Animation is a `hkaInterleavedUncompressedAnimation`. This animation stores its data with an interleaved layout[8].

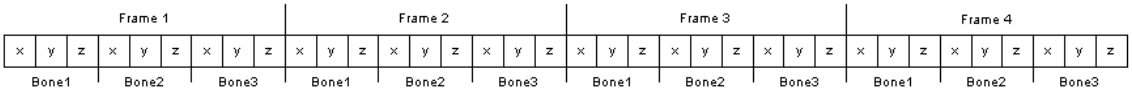



Figure 3.5: Animation Data in an Interleaved Layout

The float tracks are stored similarly in a separate block of data. Interleaving the data in this way gives a better memory access pattern and improved cache performance over storing the data per channel. This basic animation also stores the length of the total animation in seconds. We assume that each pose is uniformly spaced (in time). When fulfilling a request to sample the animations at a specified local time we:

- Find the samples immediately preceding and following the given time.
- Interpolate these poses to produce the final output pose.

**Note:**
When blending poses we simply linearly interpolate the position, rotation and scale for bone tracks. For the sake of efficiency we do not slerp rotations. In general the rotations for any bone will vary little from one frame to the next and lerp is appropriate. We also always linearly interpolate the float tracks.[9]

#### 3.3.3.2 Rig Representation

Havok Animation has no predefined rig structure. Instead we simply expect poses to appear in local space (parent relative). The `hkaSkeleton` class contains the hierarchy that defines the parent/child relationship between bones.

**Bones**

The `hkaBone` class only exists to provide you with some useful tool chain information. It contains the name of the bone (useful for retargeting as mentioned before). It also stores a flag which indicates if

[8] For simplicity we only show 3 of the 10 degrees of freedom of a single bone track here.

translation is used when playing animations on this skeleton. This flag is used by the skeleton mappers to ensure that bones are not stretched when mapping from one rig to another.

For float tracks there is no corresponding 'hkaFloat Bone' class. We extend the concept of bone to floats by introducing 'Float Slots'. A Float Slot is the conceptual destination for all sampled float data. A Float Slot has only a name, and implicitly an order inside a skeleton; see below.

**Skeletons**

The `hkaSkeleton` class contains:

- **Bone Information** : A set of bones and a parenting relationship (the parent indices). The indices array is stored as an array of signed 16 bit integers. Each bone in the set has a corresponding entry in the parent indices array. This entry gives the index of the bone's parent. Indices begin at 0. Root bones that have no parent are flagged with a parent index of -1.

  The skeleton also contains a reference pose. This pose is used during animation blending when no animation data exists for a given bone.

- **Float Information** : A set of float slots (names). There is no corresponding 'reference pose' for float slots. When no animation data exists for a given float slot, 0 will be assumed.

  **Note:**
  Unlike bones, all float slots are treated completely independently (there is no concept of 'local/model' space, and no hierarchy for float slots). Also there is no link between any given float slot and any given bone; in this sense bones and float slots are independent.

- **Partition Information** : An array of named bone subsets that contain the start bone and the number of bones in the subset. The partition information should cover all the bones in the skeleton, meaning that every bone should belong to exactly one partition.

### 3.3.4 Motion Extraction/Locomotion

#### 3.3.4.1 Overview

Most game engines use a simplified representation, a collision proxy, for characters. It is very important that the general motion of the proxy is tied appropriately to the underlying animation system. If this isn't done adequately the results often contain noticeable artifacts like foot sliding.

Naive pipelines force animators to animate on the spot and then match either character speed or the original animation to the movement of the proxy in-game. Usually these setups force artists to remove all the natural acceleration and deceleration from the cycle (because of foot sliding) leaving an artificial result. This is a programmer driven integration model.

Other pipelines allow the artist to author natural run cycles with full displacement. A constant linear velocity is removed from this animation and used directly to drive the character proxy at runtime. This artist driven model faithfully reproduces the original intended motion.

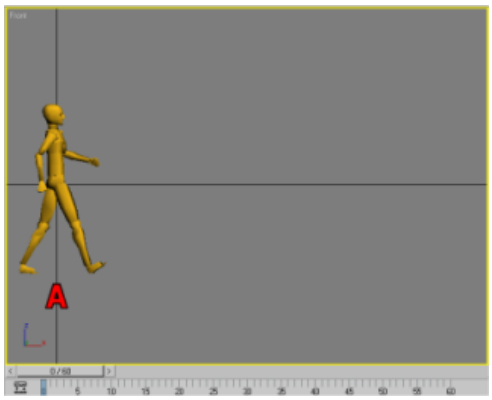
Havok Animation extends the second model to handle more complex forms of motion, including abrupt changes in motion and direction. This allows artists to fully author complex motions and have these reproduced and blended appropriately at runtime.

#### 3.3.4.2 What is Motion?

We use the term *motion* to specify the displacement of an animated skeleton (a character) during playback.

During the playback of an animation (a running cycle, or a hand waving animation), the position and orientation of the different bones of a skeleton will be animated. The motion of character is a concept that applies to the animated skeleton as a whole (instead of an animation track, which applies to individual bones), and basically is defined as the displacement of the animated skeleton as interpreted by the application. In other words, it represents how the overall animation of the individual bones affects the animation of the character's *reference frame* (this *reference frame* defines where the character *is* at any given point of time)

Take the example of a simple walk cycle. The original animation (in 3ds Max, for example) would usually start with a character located at position A and would end at position B, a few meters away from the starting position.



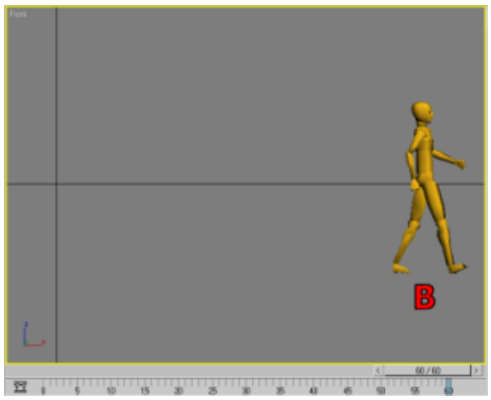




Table 3.1: Simple Motion caused by a Walk Cycle

If we loop this 60-frame (1 second) animation, though, what will happen is that the character will move from A to B, then snap back to A, move to B, snap again to A, etc. What we can do, of course, is make sure that once we reach the end of the animation, we then move our character to position B; that is, we add the translation vector B-A each time a full animation cycle is reached.

By doing this, we are starting to use the concept of motion, although in a very simplistic manner. The translation B-A is effectively the motion of our character after a single animation cycle.

The motion of animated character, therefore, is an abstraction of the concept of "how much a character moves" or, alternatively, "how much a character should move", during the animation. Motion can usually be estimated, as we did in our example, by looking at the animation for the *root node* of a character, and projecting it to a particular axis or plane.

We then define motion extraction as the process with which we split the animation (track) of the root of a character into a motion (track) and local animation (track).

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

**Note:**

Notice here that this is a 'splitting' of the information. All information is *preserved*, it is simply split into two tracks which will recombine at runtime to reconstruct the original animation. In particular, motion extraction is *not* intended to be used to 'shift' animations to the origin–the Havok Content Tools provide such functionality separately as a preprocess to animation creation in the Create Animation filter using the 'Move To Origin' checkboxes. Using these checkboxes zeroes components of the root animation track without storing the original, thus it does not preserve information.

For example, if in the example above we consider the motion of our character to be a constant velocity displacement between points A and B, our motion extraction would look like this:

**Note:**

In each of the following graphs we illustrate *two of the three* components of motion (X and Z) along the *same axis* of the graph and we differentiate them by color: The X-displacement will be RED and the Z-displacement will be BLUE.

Since our example character is facing (walking) down the X-axis and its root bone is at the pelvis, the bulk of the motion is in X, there is a small component in Z (as it oscillates up and down) and zero component in Y (no left-right motion in the pelvis).

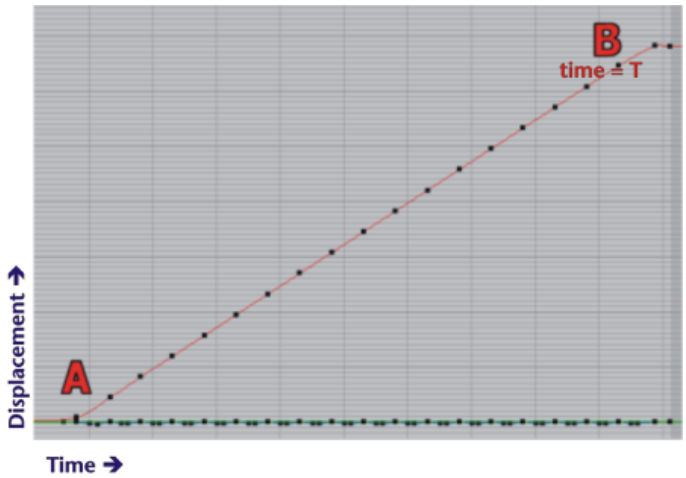
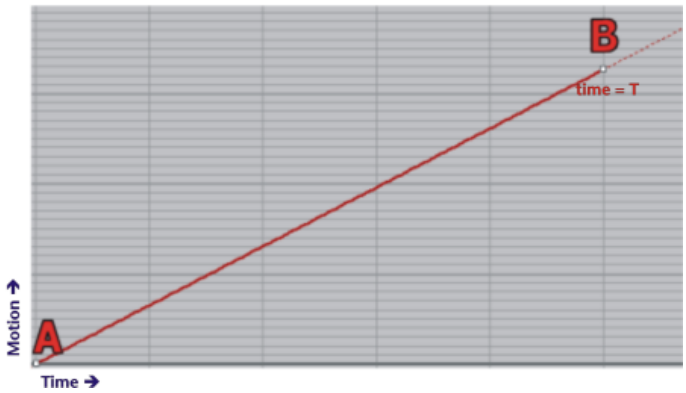
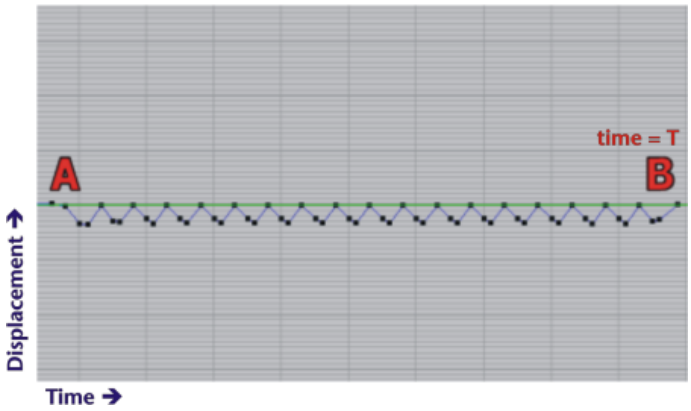
| | |
|---|---|
|  | The root track of our walk animation contains a displacement along the X-axis (in red) from A to B, plus some smaller displacements in the Z (up) axis (in blue), caused by the up/down hip movement. |
|  | After motion extraction, we get a motion track (depicted here) that represents the net displacement of the character (in the X-axis). *One of the most important things that differentiates motion from the rest of animation is the fact that motion accumulates during a looping playback. Notice how, after reaching B, the value of X in the motion track continues to increase, where the animation would however loop again to its values at time A.* |
|  | After motion extraction, the root track now only contains the smaller local displacements on the up (Z) axis (rescaled here to make it easier to see). |

Table 3.2: Character Motion at a Constant Velocity

Notice that after motion extraction, our animation will be a "walk in place", since all displacement in the forward (X) direction has been extracted, leaving only the smaller up-down motion of the pelvis.

**Using Multiple Samples**

In our example above, our motion along the X-axis is constant in velocity so we only needed two samples (one at A and one at B) to specify the motion (the motion was a straight line). There are some cases, however, where we may need to represent motion with more accuracy. Imagine this animation: our character walks a few steps, then stops, and then runs for a few steps. If we only look at the motion using start time A and end time B, our motion extraction would be:

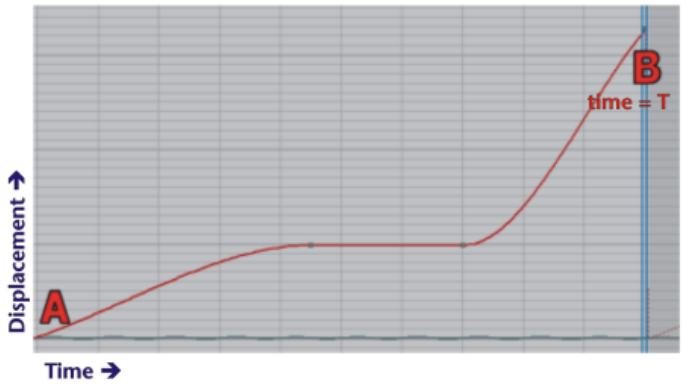
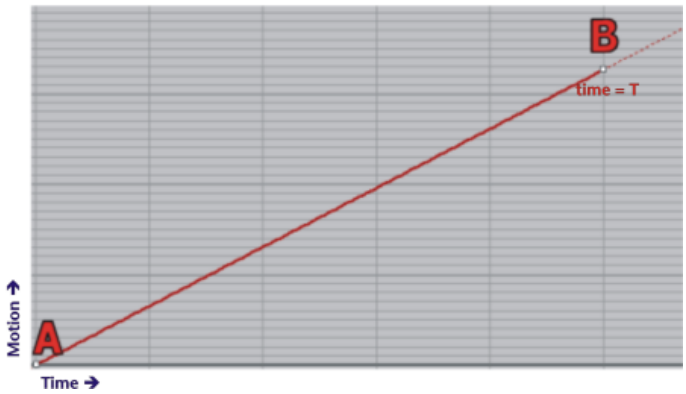
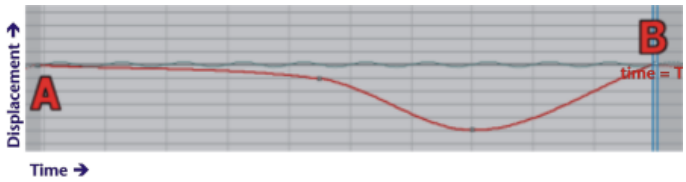
| | |
|---|---|
|  | The root track of our walk+stop+run animation contains mainly displacement along the X-axis (in red) from A to B, but this time it is not constant in velocity. |
|  | If we only sample the motion using the X-axis at points A and B as we did in the previous example, our motion will have again constant velocity over time. |
|  | After motion extraction, the root track now only contains the difference between the original animation and the extracted motion. Notice there are still big displacements on the forward (X) axis, resulting from the difference between the original animation and our constant velocity motion. |

Table 3.3: Character Motion at Multiple Velocities

As we said earlier in this section, one of the uses of motion is that it tells us "where the character is", trying to disregard small local variations in the animation of particular bones. In the example above, the root node would still have a considerable amount of animation in the X (forward) direction, since a constant velocity forward motion doesn't accurately reflect the movement of our character. This can sometimes be a problem, as we will see in some of our examples later on.

If, however, we use multiple samples for the motion (instead of just sampling A and B), we can construct a much better representation for the motion. In particular, if we use as many samples as the animation:

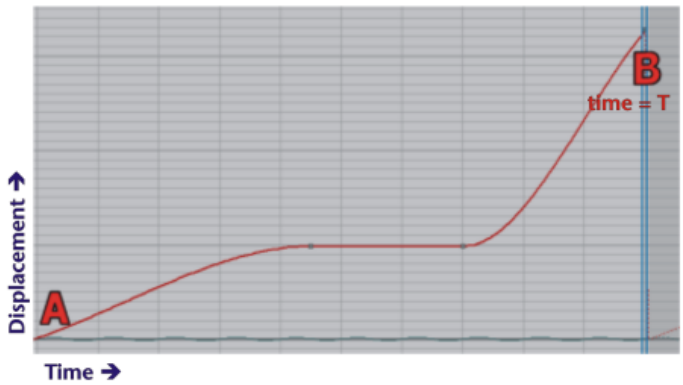
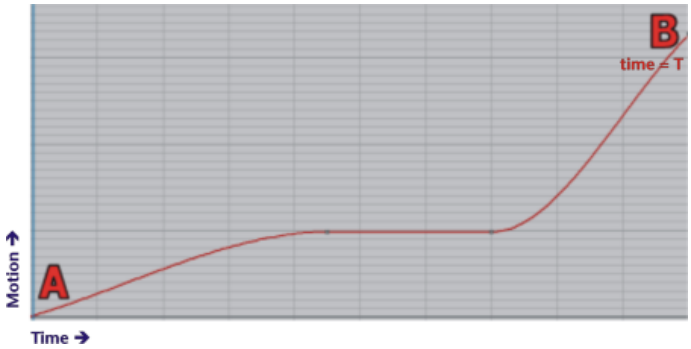
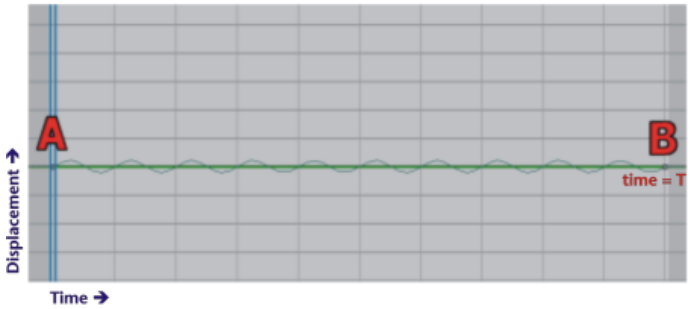
| | |
|---|---|
|  | Let's look again at our walk+stop+run animation. |
|  | We now sample the motion in the X-axis by taking samples every frame. We basically reconstruct the curve for the X component of the root's translation. |
|  | After motion extraction, the X translation component in the root track is now 0. The rest of components (Z, i.e. Up, in particular) only contain local displacements. |

Table 3.4: Motion Extraction using multiple samples

The motion track now gives us a much more accurate description of "where the character is" at any time during playback.

### Choosing the Motion Components

In the previous examples, we used a single component (X) to describe the motion extracted from the animation. This is because in our example, X was the "forward" direction in our scene, and the animations involved forward motion.

Since the choice of axis is arbitrary, from now on we will be talking about forward/back axis, right/left axis and up/down (or vertical) axis instead of X, Y or Z.

Not all animations have forward motion. The motion of a vertical jump, for example, could be described using the up/down axis; the motion of a forward jump may be described by using both the forward/back axis and the up/down axis. Finally, some animations, like a hand waving animation, may not have motion at all.

Also, apart from translations on particular axis, there is also a *turning motion* that we may want to extract from an animation. We define this turning motion (or *yaw motion*) as a rotation around the vertical (up/down) axis.

So, in total, we have 4 *motion components*:

- Translation on the forward/back direction.
- Translation on the right/left direction.
- Translation on the up/down direction.
- Turning (rotation) around the vertical axis.

The choice of which motion components should be extracted from the root animation is application-specific. However, most game applications have similar requirements, and therefore would follow these rules:

- If an animation is meant to loop seamlessly, you want to extract motion of those components where the initial and final poses differ (so the initial and final poses match).
- Even if a component loops seamlessly, the application may be interested in keeping track of motion for particular components during the animation cycle. In those cases, you want to extract the motion on those components using multiple samples.

**Example 1**

We have a jumping animation:

Figure 3.6: A Character's Jumping Animation

If we want this animation to loop seamlessly, we need to extract the motion in the forward direction, since the character ends up at some distance along the forward axis; otherwise the animation will snap back to the original position during playback.

Extracting motion along the vertical axis is not required for seamlessly looping, since the animation finishes with the character at the same height as the initial pose. However, in many situations we may want to extract the vertical motion from animations like this one. Imagine this situation:

We are using the animation to drive a character in a game. For collision detection, we are representing the character with a capsule, and we move this capsule representing the character according to the motion extracted from the character.

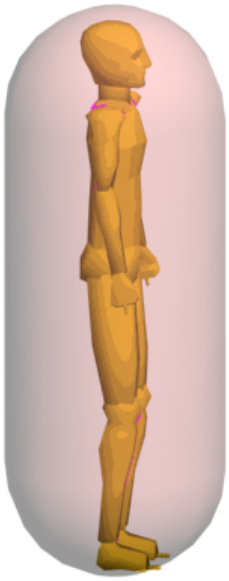


Figure 3.7: The Character with its Collision Detection Capsule

While with animations like walk or run cycles, considering only forward motion suits our needs, jump cycles are normally different. Consider the time when the animation has the character in the air:

Figure 3.8: The Capsule's Position when the Character jumps

Since our capsule representation follows the motion of the character, and we only extracted this motion along the forward axis, our game assumes the character is still at the same height (on the ground).

Imagine now that there is an obstacle on the way of the character (and that's why the player decided to jump, which triggered the jump animation). Since our capsule representation (which we use for collision detection) doesn't follow vertical movements of the character, the capsule will collide with the obstacle,

a collision will be reported and the player will be stopped.

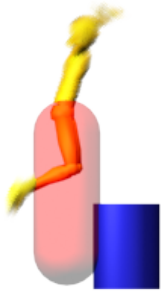


Figure 3.9: The Capsule Collides when the Character jumps over an obstacle

If, however, we allow vertical motion to be also extracted (and use multiple samples, otherwise the information about the jump is lost), we can use that motion to drive our capsule and therefore we don't get this false collision:

Figure 3.10: No Collision when Multiple Motion Samples are used

**Example 2**

Imagine we have an animation for a character turning around a corner:

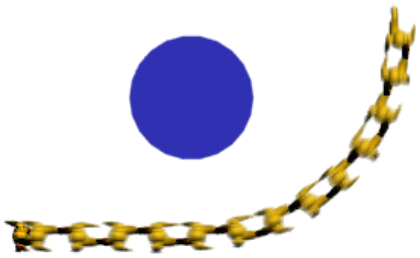


Figure 3.11: A Character's Turning Animation

In this case, we want to extract motion in both the forward/back axis and the right/left axis. We don't

need to extract up and down motion.

We need to use multiple samples in order to properly reproduce the motion curve of the character; if we use only two samples we would assume the motion is in a straight line (from the start to the end point) and therefore, if running an application like the previous example, our character representation (capsule) would report collisions with the blue obstacle in the picture.

Notice that for this animation we also want to extract turning (yaw) motion from the animation since the character actually turns 90 degrees from the start to the end (the animation finishes with the character facing a different direction); by doing this the end and start pose will match (and therefore loop seamlessly).

#### 3.3.4.3 Storing Extracted Motion

Sometimes the animator can extract the motion of an animation before the animation is exported. For example, in any of the examples above, the animator could remove the root track (or parts of it) in the modelling tool and therefore have the animation walk, run or jump in place.

While it is not uncommon practice to do this, the main disadvantage is the fact that valuable information is lost if the removed tracks (extracted motion) are not exported alongside the animation. If motion is extracted manually by animators, it becomes the responsibility of the programmers to ensure that proper motion is applied at runtime in order to reproduce suitable animation. Otherwise artifacts (sliding feet for example) may occur.

By extracting the motion during export (and therefore leaving the original animation untouched) the extracted motion (the motion track) is exported and stored alongside the animations and can therefore be used at runtime without loss of information.

#### 3.3.4.4 Applications of Motion Extraction

During the previous section we saw some of the reasons for extracting the motion from an animation. Here we'll review those and some other examples:

**Animation Playback**

When playing an animation of period T using a loop control, we need to ensure that the final pose at time T is the same as the initial pose at time 0 so the animation runs smoothly.

By extracting the necessary motion of the original animation, we can ensure that the final pose (at least for the root node) will match the initial pose.

Also, and since motion always accumulates (as opposed to animations, whose values repeat every period T), we can properly sequence or loop animations during playback, having the character move properly.

**Use the Extracted Motion to Drive a Character**

There are two main approaches that can be taken in order to control and display the movement of a character in a game:

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

1. The application uses the user input to drive a character controller, which calculates a new position and the proper animation (and weights) to play. The animation system then calculates a new pose. The new pose and position are then used to display the character.

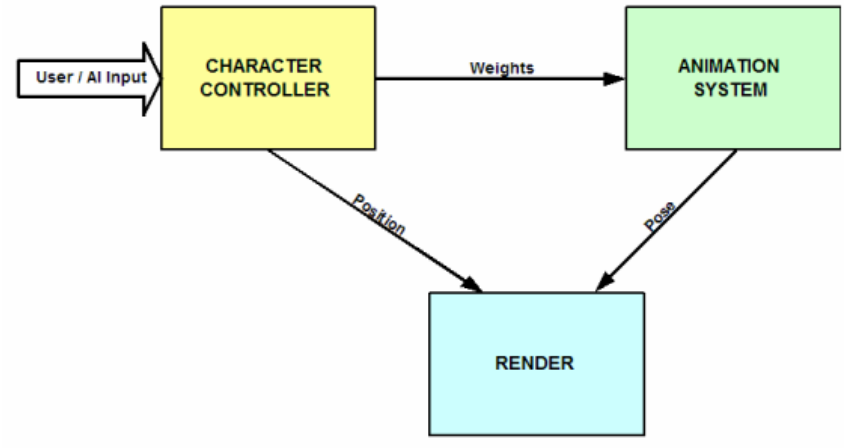



Figure 3.12: Animation System Driven by a Character Controller

The advantage of this first approach is that the application has total control over the position of the character, and it doesn't depend heavily on the animation system. This approach is commonly used in, for example, network-based first person shooters, where ensuring the correct positioning of the players is more crucial, and artifacts in the animation (sliding feet for example) are easily tolerated.

2. The application uses the user input to drive the animation system, which calculates a new pose and desired motion. This desired motion is used by the character controller to drive the character to a new position. The character is then displayed using the new pose and position.

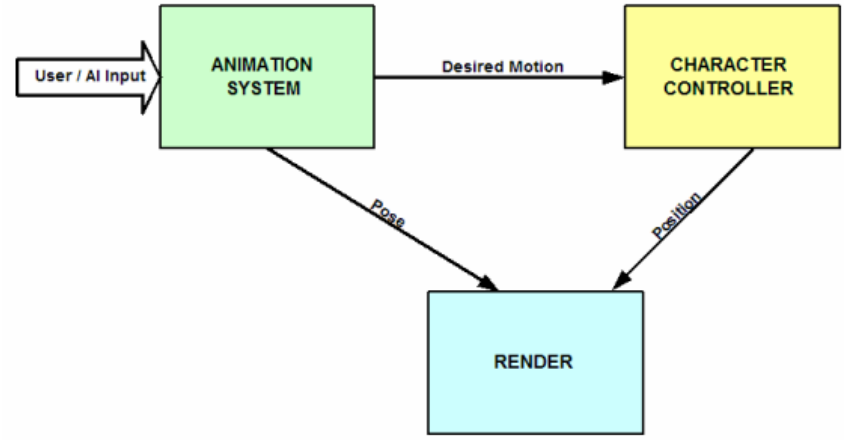



Figure 3.13: Character Controller Driven by an Animation System

The advantage of this second approach is that, by driving the character based on the motion extracted from the animation we ensure that the final result (position+pose) closely matches what the animator intended. For example, replacing a slow walking cycle with a fast walking cycle will automatically drive the character controller faster without requiring any adjustments in the controller code. This approach is particularly suitable for third-person games where emphasis is put on high-quality, smooth animation playback.

Notice that the two diagrams above are an oversimplification of the logic in game; they are just meant to emphasize the responsibilities of each component regarding the control of the character's position in

both approaches.

#### 3.3.4.5 Motion in the Havok Animation System

Usually motion extraction is done through using the Havok Toolchain (see the Motion Extraction filter in the Havok Filter Manager chapter). At run time, the classes involved are as follows:

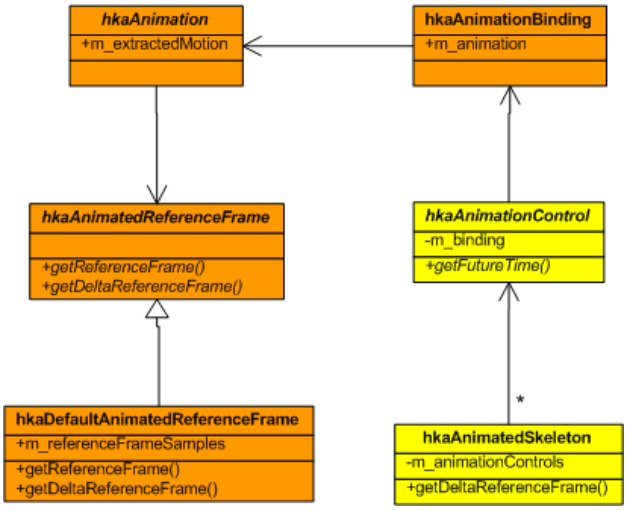



Figure 3.14: Motion Extraction Classes (at Runtime)

The classes in orange (on the left) are created (exported) by the toolchain; the classes in yellow are run-time classes only.

Some comments on how motion is handled by the different classes:

- `hkaAnimation`: Has an `m_extractedMotion` member (possibly NULL) that points to an instance of `hkaAnimatedReferenceFrame` (the extracted motion).
- `hkaAnimatedReferenceFrame`: Defines the `getReferenceFrame()`, which returns the location of the reference frame (the extracted motion) at a given point in time (between 0 and the period of the original animation), and the `getDeltaReferenceFrame()` method, which returns the movement of the reference frame (the accumulated, or delta, motion) between two times.
- `hkaDefaultAnimatedReferenceFrame`: Implements both `getReferenceFrame()` and `getDeltaReferenceFrame()` by querying the samples that were extracted and stored from the original animation during toolchain operations (motion extraction).
- `hkaAnimationControl`: Keeps track of the current time and calculates how a time increase would affect the local time of an animation (it may handle looping, etc.).
- `hkaAnimatedSkeleton`: Defines and implements `getDeltaReferenceFrame()` which samples the motion from all playing animations (through their respective animation controls) and blends them accordingly.

**Extracting Motion using the Havok Toolchain**

The Havok Toolchain provides a *Motion Extraction Filter* that performs motion extraction operations in interleaved (`hkaInterleavedUncompressedAnimation`) animations. For more information about this filter, please check the Toolchain documentation.

**Extracting Motion at Run-Time**

While it is most common to perform motion extraction during toolchain operations, it is possible to also do so at run-time. You can create an instance of `hkaDefaultAnimatedReferenceFrame` based on an `hkaInterleavedUncompressedAnimation`, and then extract that from the animation using `hkaAnimatedReferenceFrameUtils`.

### 3.3.5 Animation Compression

This section describes animation compression features available within the Havok Animation SDK. Without the use of compression the space requirements of animation assets may exhaust the memory resources available in gaming consoles. Animation compression allows for these assets to stored in a compact form which significantly reducing the amount of memory required. The goal of animation compression is to produce a result that is perceptually similar to the raw input animation while achieving a significant savings in the size of the representation.

The Havok Animation SDK supports three compression schemes. Each compression technique implements *lossy* animation compression in which there is a tradeoff between fidelity to the original animation versus the size of the compressed animation in bytes. Each of the compression schemes allows the author control over the settings which define the space verses quality trade off. This trade off is conceptually similar to those encountered in popular "lossy" multimedia file representations for images, video or music. In addition to the trade-off between fidelity and compressed size, there is also a tradeoff between compressed size and decompression speed. Schemes that compress the animation to a smaller size are typically (but not always) slower to decompress.

The three compression schemes are:

- Spline: Provides a good balance of compressed size and decompression speed on all platforms. *This is the only scheme that allows you to decompress individual bones.* The Spline compression scheme is described in subsubsection 3.3.5.1.
- Quantized: Provides the best decompression speed on platforms with appropriate SIMD support, but the compressed size is not as small as the other schemes. The Quantized compression scheme is described in subsubsection 3.3.5.2.
- Predictive: Provides the smallest compressed size, and decompresses faster than Spline on platforms with appropriate SIMD support. The Predictive compression scheme is described in subsubsection 3.3.5.3.

All of the compression schemes share the following features:

- Full control over the accuracy of the approximation.
- Smoothly degrading quality from lossless to lossy compression settings.
- Multiple decompression threads may be executed in parallel.
- Optimization for several platforms.

Runtime performance statistics for each of the available compression schemes are detailed in the Platform Guide for the appropriate platform. You should carefully consider the decompression speed for your target platform when choosing a compression scheme.

#### 3.3.5.1 Spline Compression

Havok's spline compression technology provides state of the art compression of animation assets. Compression ratios of 10x–20x may be expected for near-lossless compression, while compression rates of 30x or more may be expected for lossy compressions. Unlike the other compression schemes, spline compression allows per-bone and per-track decompression.

The fundamental technique of spline compression is the fitting of low order piecewise polynomials to regularly sampled input animation data. The resulting polynomials are stored as NURBS, which are a common curve representation. NURBS allow for a smooth fit to input data while preserving data discontinuities when appropriate.

The spline compression pipeline consists of three stages: Track Analysis, Spline Approximation and Quantization. This process is Illustrated below in **??**. These stages are are detailed in **??**, **??**, and **??** respectively.

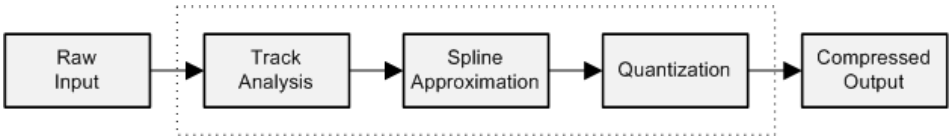



Figure 3.15: Spline Compression Pipeline

Havok's spline compression technology is implemented by the `hkaSplineCompressedAnimation` class. Spline compression settings may be declared uniformly for all tracks using the `hkaSplineCompressedAnimation::TrackCompressionParams` class, or individually for each track using the `hkaSplineCompressedAnimation::PerTrackCompressionParams` class. Spline compression settings are uniquely defined by three values: tolerance, degree and quantization type. These three settings may be specified independently for translation, rotation, scale and float data.

The tolerance value defines the accuracy to which the track analysis and spline approximation steps approximate the input data. This setting typically has the greatest control over the end quality of the compressed animation. The smaller the tolerance value, the closer the resulting spline approximation will follow the input data. The larger the tolerance value, the smaller the compressed output memory footprint will be.

The degree setting determines the polynomial degree of the spline representation. Piecewise cubic curves (degree 3) produce the smoothest results, while piecewise linear (degree 1) curves may provide a more compact representation at the cost of producing a less smooth result.

Quantization determines the number of bits used to represent scalar values in the spline data stream. Spline animation compresses input data in blocks, typically 256 frames of data. The block size is described in section **??**.

**Track Analysis**

The track analysis stage of the spline compression pipeline identifies unchanging components of the input animation and stores this data in an efficient and compact manner. Translation, scale and float data

is analyzed component-wise (x,y,z for translation and scale) to determine whether the maximum and minimum values are within tolerance of the mean. Any components which pass this test are denoted as *static*, and are represented by the mean value for the entire block. Rotations are treated similarly, but all four components must pass the tolerance test for the rotation to be considered static. This requirement is due to the fact that rotations are quantized as quaternion values and individual (x,y,z,w) components may not individually be marked as static. Additionally, tracks which are found to be within tolerance of special *identity* values: (0,0,0) for translation, (1,1,1) for scale and (0,0,0,1) for rotation are represented by a single bit in the spline data stream. All output of the track analysis stage of spline compression are guaranteed to be within tolerance of the input data as measured by the Infinity (Supremum) Norm.

**Spline Approximation**

The spline approximation step approximates input data as a series of piecewise polynomial curves using a standard NURBS representation. NURBS curves consist of a temporal component called knots, and a spatial component called control points. The spline control points are the same dimensionality as their respective input types (translation: 3, rotation: 4, scale: 3, float: 1) minus any degrees of freedom marked as static by track analysis. The specific knots and control points used to represent the compressed data are found by a spline optimization and fitting routine which is specialized for compressing animation data. The spline fitting routine guarantees that the resulting spline curve is within tolerance of the input data as measured by the Euclidean (L2) Norm. Compression is achieved as typically many fewer control points are necessary to represent the input curve than the number of input sample points.

**Quantization**

Once the control points of the spline compressed animation have been created, they are quantized to a specified number of bits. The quantization values are defined in the `hkaSplineCompressedAnimation::TrackCompressionParams` class. Translation, scale and float values may be quantized as either 8 or 16 bit values. The error introduced by n bits of quantization is of the order 2^(-n). Rotation values are stored and quantized as compact quaternion representations. As quaternions are always 4-dimensional unit vectors, a significant savings may be had by storing only 3 components and implying the value of the 4th. Quaternions may be compressed as 32, 40, or 48 bits. 40-bit quaternions are considered the default and other values should be chosen only as special needs dictate. The 32-bit quaternion representation uses a polar decomposition which introduces a speed penalty. Additional bit sizes including 16, 24, and 128 bits are available in the SDK, but are not exposed through the content tools. The error introduced by quaternion quantization to n bits is of the order of 2^(-n/3). An additional step in which quaternion control points are renormalized to unit vectors introduces a small additional error.

It is the user's responsibility to ensure that the errors introduced by the tolerance and quantization settings are roughly of the same order. For example, 8 bit quantization introduces an error of approximately 0.004 and a tolerance setting in the range [0.01,0.001] is quite appropriate in this case. However a tolerance of 0.00001 would be of limited use as the error introduced by compression will be quantization dominated. In typical cases the animator can avoid this problem by simply avoiding opposing trends in quantization and tolerance settings; tightening tolerances while simultaneously decreasing the number of quantization bits is generally not productive.

**Per Track Compression Settings**

The `hkaSplineCompressedAnimation` class offers two constructors using the `hkaSplineCompressedAnimation::TrackCompressionParams` or the `hkaSplineCompressedAnimation::PerTrackCompressionParams` classes for uniform or per-track compression settings, respectively. Each constructor also takes a single instance of the `hkaSplineCompressedAnimation::AnimationCompressionParams` class to define settings which affect the entire animation. The `hkaSplineCompressedAnimation::PerTrackCompressionParams` class itself

contains an array of `hkaSplineCompressedAnimation::TrackCompressionParams` instances which defines a palette of compression parameters. Additional arrays map each transform and float track to the appropriate compression parameters. The idea of having a compression palette was envisioned to support uses cases such as Level Of Detail compression (with a "major" bones compression setting and a "minor" bones compression setting) or an upper-lower body split. The number of palettes created is arbitrary (though having more palettes than tracks serves no useful purpose). To apply a compression setting to a specific bone, the track to bone mapping must be reversed to determine the appropriate track for the current animation. The content tools provide this functionality in the form of a GUI to the user.

**Block Size**

Spline compression operates on a "block" of data at a time. Block size is 256 frames of input animation by default. Animation sequences longer than this are automatically broken into multiple blocks without the need for user intervention. For most platforms it is not necessary to change the default block size setting. The exception is the PlayStation®3 SPU implementation of spline compression where certain memory limitations need to be respected. Please refer to the PlayStation®3 Guide for more information.

**Individual Bone/Track Sampling**

Spline compressed animations can allow individual transform tracks to be sampled through the `hkaSplineCompressedAnimation::sampleIndividualTransformTracks` member function, and individual bone poses to be sampled through the `hkaAnimatedSkeleton::sampleAndCombineIndividualBones` member function. Allowing individual tracks / bones to be decompressed requires a small additional memory overhead, thus this feature is considered optional and disabled by default. This feature may be enabled through the `hkaSplineCompressedAnimation::AnimationCompressionParams` class.

#### 3.3.5.2 Quantized Compression

Havok's Quantized compression technology provides for very fast decompression of compressed assets. Quantized compression is designed to produce compression ratios similar to Spline compression for near-lossless tolerance settings, while providing decompression performance that is several times faster on platforms that have appropriate SIMD support.

The Quantized compression pipeline consists of two stages: Track Analysis and Quantization, as illustrated below:

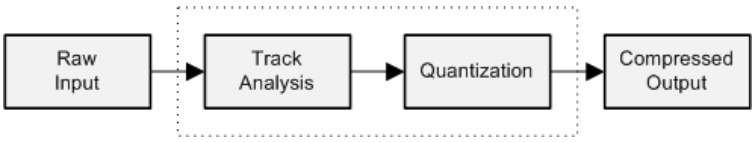



Figure 3.16: Quantized Compression Pipeline

These stages are detailed in **??** and **??** respectively. Unlike Havok Animation's other compression schemes, the lossy compression step is omitted.

Havok Animation's Quantized compression technology is implemented by the `hkaQuantizedAnimation` class. Quantized compression settings may be declared uniformly for all tracks using the `hkaQuantizedAnimation::TrackCompressionParams` class, or individually for each track using the `hkaQuantizedAnimation::PerTrackCompressionParams` class. Quantized compression settings are uniquely defined by a single value: tolerance. This setting may be specified independently for trans-

lation, rotation, scale and float data.

The tolerance value defines the accuracy to which the track analysis step approximates the input data. The smaller the tolerance value, the closer the resulting approximation will follow the input data. The larger the tolerance value, the smaller the compressed output memory footprint will be.

**Track Analysis**

The track analysis stage of the Quantized compression pipeline identifies unchanging components of the input animation and stores this data in an efficient and compact manner. Translation, scale and float data is analyzed component-wise (x,y,z for translation and scale) to determine whether the maximum and minimum values are within tolerance of the mean. Any components which pass this test are denoted as *static*, and are represented by the mean value for the entire block. Rotations are treated similarly, but all four components must pass the tolerance test for the rotation to be considered static. This requirement is due to the fact that rotations are quantized as quaternion values and individual (x,y,z,w) components may not individually be marked as static. Tracks which have been identified as static are further checked to see if they are within tolerance of the reference pose of the character. This allows for a great savings for bone properties which do not deviate from the reference pose (in many use cases bone lengths and scales do not deviate at all from the reference pose).

**Quantization**

Quantization is fixed at 16 bits for translation and scale components, and for float values. Quaternions are quantized at 48 bits per quaternion. This gives a bound on accuracy of approximately O(10^-5).

**Per Track Compression Settings**

The `hkaQuantizedAnimation` class offers two constructors using the `hkaQuantizedAnimation::TrackCompressionParams` or the `hkaQuantizedAnimation::PerTrackCompressionParams` classes for uniform or per-track compression settings, respectively. The `hkaQuantizedAnimation::PerTrackCompressionParams` class itself contains an array of `hkaQuantizedAnimation::TrackCompressionParams` instances which defines a palette of compression parameters. Additional arrays map each transform and float track to the appropriate compression parameters. The idea of having a compression palette was envisioned to support uses cases such as Level Of Detail compression (with a "major" bones compression setting and a "minor" bones compression setting) or an upper-lower body split. The number of palettes created is arbitrary (though having more palettes than tracks serves no useful purpose). To apply a compression setting to a specific bone, the track to bone mapping must be reversed to determine the appropriate track for the current animation. The content tools provide this functionality in the form of a GUI to the user.

#### 3.3.5.3 Predictive Compression

Havok's Predictive compression technology was designed to make use of modern SIMD architectures. It is similar to the Quantized scheme but with a lossless compression step after the quantization step. Typically, Predictive compression produces a smaller compressed size than Spline or Quantized compression. On platforms with an appropriate SIMD implementation it decompresses faster than Spline compression. On such platforms, it is recommended that you use Predictive compression instead of Spline compression unless you need per-bone decompression, which only Spline compression provides.

The Predictive compression pipeline consists of three stages: Track Analysis, Quantization and Lossless Compression, as illustrated below:

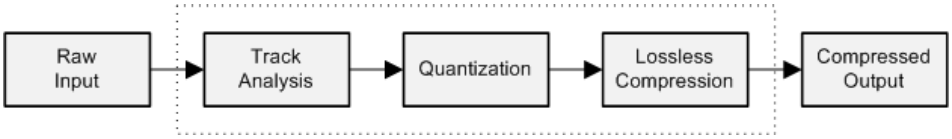



Figure 3.17: Predictive Compression Pipeline

These stages are detailed in **??**, **??** and **??** respectively.

Havok Animation's Predictive compression technology is implemented by the `hkaPredictiveCompressedAnimation` class.

For the most part, the compression pipeline deals with floating point *channels*. Each element of the translation, scale and rotation is treated independently as a channel. In the case of rotations, the W component of the quaternion is dropped and reconstructed from the other components.

### Compression Settings

Predictive compression settings are defined by two types of tolerances:

- Static Tolerance: If a channel can be treated as constant over all frames without creating error beyond this tolerance then it will be. This tolerance is used during the Track Analysis stage.
- Dynamic Tolerance: Non-static channels will be quantized so that the compressed data never differs from the original data by more than this tolerance. This tolerance is used during the Quantization stage.

For both types of tolerances, the tolerance value defines the accuracy to which the compressed data approximates the input data. The smaller the tolerance value, the closer the resulting approximation will follow the input data. The larger the tolerance value, the smaller the compressed output memory footprint will be.

You can set each of these tolerances independently for Translation, Scale, Rotation and Float data.

Predictive compression settings may be declared uniformly for all tracks using the `hkaPredictiveCompressedAnimation::TrackCompressionParams` class, or individually for each track using the `hkaPredictiveCompressedAnimation::CompressionParams` class.

**Note:**
The Dynamic Tolerance should be set to less than or equal to the Static Tolerance.

### Track Analysis

The purpose of the Track Analysis stage is to identify any channels that are constant over all frames so that they do not even need to pass through the Quantization and Lossless Compression stages.

The track analysis stage identifies each floating point channel as either Reference, Static or Dynamic:

- Reference: The channel does not change at all and is identical to the reference pose (or reference floats), so it does not need to be represented at all in the compressed data. The components of a quaternion are not allowed to be Reference unless all of them are. A track is considered Reference if the value never deviates from the reference value by more than the Static Tolerance.

- Static: The channel does not change at all, so it can be represented as a single value for all frames in the compressed data. A Static channel is represented by the mean value over all of the frames. A track is considered Static if the value never deviates from the mean by more than the Static Tolerance.
- Dynamic: If neither of the above applies then the track data is dynamic and must be quantized and compressed by the later stages of the pipeline.

**Quantization**

The Quantization stage is only applied to the channels that were determined to be Dynamic in the Track Analysis stage. The purpose of the Quantization stage is to map the range of values that the channel covers to an integer interval such that the integer values are as close as possible to zero. Values closer to zero compress better during the Lossless Compression stage. The Dynamic Tolerance value is used to determine the smallest possible number of bins that the data will be quantized into. The following figure shows how the data is quantized for different Dynamic Tolerance values:

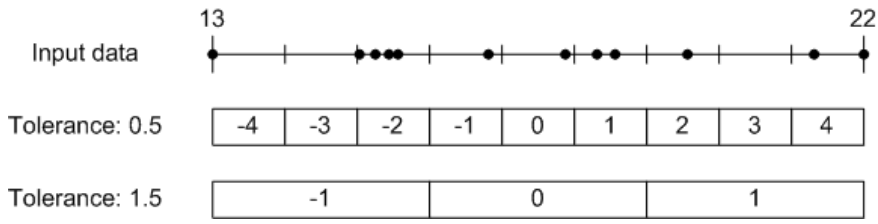



Figure 3.18: Quantization for Predictive Compression

In the figure above, the original channel data falls into the floating point interval [13, 22]. If the Dynamic Tolerance is set to 0.5, the input data will be mapped to the integer range [-4, 4]. Since each bin covers 1 unit in the original data range, the error due to quantization will be no greater than 0.5. Similarly, if the Dynamic Tolerance is set to 1.5 only three bins will be used. The fewer the bins the smaller the integers representing them are, which leads to better compression during the Lossless Compression stage.

**Note:**

Setting the Dynamic Tolerance for the Rotation channels to 0 is treated as a special case. Normally, during the Quantization stage, two real numbers are stored for each channel: the center and width of the interval that was quantized. For example, in the figure above, the center would be (13 + 22) / 2 = 17.5 and the width would be 22 - 13 = 9. Using the interval center and width the compressed integers can be mapped back to the original interval. For quaternions, each value has a known range of [-1,1]. If you set the Dynamic Tolerance for Rotations to 0 the system does not store the center and width of the interval and instead uses the interval [-1,1].

**Lossless Compression**

The Lossless Compression stage compresses and decompresses 16x16 blocks of 16-bit integers, where each column of a block is a frame and each row of a block is a channel. Each channel is decompressed using a quadratic prediction based on the last three frames. The difference between the predicted value and the actual value for smooth data is close to zero, so encoding these differences requires fewer bits than encoding the data itself.

#### 3.3.5.4 Example Compression Ratios

The following table provides the compression statistics for 15 example animations using an error threshold of 0.0001 for all channels. The skeleton and animations have 38 bones. The total uncompressed size of

the animations is 3146 KB.

| Compression Scheme | Total Compressed Size of 15 Animations (KB) | Compression Ratio (Original Size / Compressed Size) |
|---|---|---|
| Quantized | 406 | 7.7 |
| Spline | 216 | 14.5 |
| Predictive | 158 | 19.8 |

These statistics were gathered using the `SampleOnlyTimingsDemo` and `SampleAndBlendTimingsDemo` demos. You can use these demos to test the compression ratio and performance for your animations.

### 3.3.6 Playback

The state of a character in the Havok Animation library is defined by the `hkaAnimatedSkeleton` class. An instance of `hkaAnimatedSkeleton` contains one or more Playback Controls which each reference a single `hkaAnimation` instance and specify a blend type. Each individual animation is sampled and blended together into a pose.

**Note:**
A complimentary sampling and blending framework designed to maximize performance using `hkaQuantizedAnimation` is presented in section Playback of hkaQuantizedAnimation.

#### 3.3.6.1 Sampling and Blending of Bones and Float Slots

The methods described below allow the pose of a character to be determined at a given point in time.

```
void hkaAnimatedSkeleton::sampleAndCombineAnimations( hkQsTransform* poseLocalSpaceOut, hkReal*
  floatSlotsOut ) const;

void hkaAnimatedSkeleton::sampleAndCombinePartialAnimations( hkQsTransform* poseLocalSpaceOut, hkUint32
  maxBones, hkReal* floatSlotsOut, hkUint32 maxFloatSlots ) const;

void hkaAnimatedSkeleton::sampleAndCombineSingleBone( hkQsTransform* localSpaceOut, hkInt16 bone ) const;
void hkaAnimatedSkeleton::sampleAndCombineSingleSlot( hkReal* floatOut, hkInt16 slot ) const;
void hkaAnimatedSkeleton::sampleAndCombineIndividualBones( hkQsTransform* localSpaceOut, hkInt16* bones,
  hkUint32 numBones ) const;
void hkaAnimatedSkeleton::sampleAndCombineIndividualSlots( hkReal* floatOut, hkInt16* slots, hkUint32
  numSlots ) const;
```

**Sampling a Complete Pose**

The `sampleAndCombineAnimations()` method allows the pose of a character, as defined by a series of blended animations, to be sampled at a given time. Calling `sampleAndCombineAnimations` queries all active controls and samples their animations. This animation data is then blended into a single local pose.

**Sampling a Range of Bones**

This is an advanced feature which supports LOD of a range of bones which have been topologically sorted by the user (the Content Tools enforce one such sorting: depth-first sort). This range (0 to N) of bones is guaranteed to be sampled, but the efficiency of this method is dependent on the bone ordering in the

corresponding `hkaSkeleton`. It is the user's responsibility to convert these bone transforms to world space (if desired), and to ensure that a full chain of bones starting from the root bone have been chosen. It is advised that the user contact Havok support before using this function.

#### Sampling Individual Bones

The `"Single" and "Individual" sampling` functions allow the user to sample an arbitrary set of bones in the animation at a given time from a set of blended animations. One anticipated use case is supporting Level Of Detail animation in which only major bones such as the spine, arms and legs are sampled, but minor bones such as fingers and toes are not. Another anticipated use case is to sample only a small subset of bones which are considered important, such as those belonging to the chain of bones from the root to the tip of the hand swinging a weapon. Note that the bones sampled are in local space. It is the user's responsibility to convert these bone transforms to world space (if desired), and to ensure that a full chain of bones starting from the root bone have been chosen. The corresponding Slot versions of these functions operate on float slot data.

**Important:**

This feature is currently only supported for blends of Interleaved, Spline Compressed, and Mirrored versions these animations, respectively. This feature must be enabled in the content tools for Spline Compressed animations. This feature is currently not supported for Quantized Compressed or Predictive Compressed animations.

### 3.3.6.2 Sampling of Transform Tracks and Float Tracks

The tracks of an animation may be sampled directly from the `hkaAnimation` class. Sampling from the animation allows direct access to the local space transforms of each track present in the animation at a given time. The features described below are useful for users implementing their own animation controls and blending functionality.

```
void sampleTracks(hkReal time, hkQsTransform* transformTracksOut, hkReal* floatTracksOut) const = 0;
void samplePartialTracks(hkReal time, hkUint32 maxNumTransformTracks, hkQsTransform* transformTracksOut,
  hkUint32 maxNumFloatTracks, hkReal* floatTracksOut) const;
void sampleSingleTransformTrack( hkReal time, hkInt16 track, hkQsTransform* transformOut ) const;
void sampleSingleFloatTrack( hkReal time, hkInt16 track, hkReal* out ) const;
void sampleIndividualTransformTracks( hkReal time, const hkInt16* tracks, hkUint32 numTracks,
  hkQsTransform* transformOut ) const = 0;
void sampleIndividualFloatTracks( hkReal time, const hkInt16* tracks, hkUint32 numTracks, hkReal* out )
  const = 0;
```

#### Sampling All Tracks

The `sampleTracks()` interface is the simplest of the sampling member functions. Given a local time, the value of all transform and float tracks are computed. The associated `hkaAnimationBinding` class may be used to map the sampled tracks to the appropriate bone/float slots.

#### Sampling A Range of Tracks

The "Partial" sampling function allows the first N tracks to be sampled. It is the user's responsibility to convert these bone transforms to world space (if desired), and to ensure that a full chain of bones starting from the root bone have been chosen. It is advised that the user contact Havok support before using this function.

#### Sampling Individual Tracks

The "Single" and "Individual" sampling functions allow the user to sample an arbitrary set of tracks in the animation at a given time. One anticipated use case is supporting Level Of Detail animation in which only tracks corresponding to major bones such as the spline arms and legs are sampled, but tracks corresponding to minor bones such as fingers and toes are not. Another anticipated use case is to sample only a small subset of tracks which are considered important, such as those corresponding to the chain of bones from the root to the tip of the hand swinging a weapon. Note that the transforms sampled are in local space. It is the user's responsibility to convert these transforms to world space (if desired), and to ensure that a full set of tracks corresponding to a chain of bones from the root bone have been chosen.

**Important:**
This feature is currently only supported for Interleaved, Spline Compressed, and Mirrored versions these animations, respectively. This feature must be enabled in the content tools for Spline Compressed animations. This feature is currently not supported for Quantized Compressed or Predictive Compressed animations.

### 3.3.6.3 Playback Control

The `hkaAnimationControl` interface class provides access to the information needed to sample and blend a specific animation. If an animation is playing then it must have an associated control. Currently controls cannot be shared and must be created for each instance of a playing animation.

Each control returns a state structure which contains:

- The local time used to sample an animation.
- A master weight to use when blending this animation.
- Access to the animation binding for this animation.
- Two sets of track weights used for per track feathering/masking (one for transform tracks, one for float tracks).

#### The Default Playback Control

The default control, `hkaDefaultAnimationControl`, is the default implementation for our control interface. This control automatically handles looping when stepped beyond the start or period of the animation. It also provides:

- Playback speed control.
- Direct access to the master weight for the playing animation.
- Direct access to the sampling time for the playing animation.
- Overflow and underflow callbacks (animation local time has past the end or beginning of the animation, respectively).
- Ease curve support.

#### Ease Curve Support

Ease curves are most frequently used when transitioning between animations at runtime. Easing modulates the standard control weight for an animation and provides a smooth transition from one animation to the next. Our ease modulation is always enabled.

To control the ease in or ease out curves you specify the control points of a standard Bezier curve. This parametrization has some nice intuitive curves as shown below.

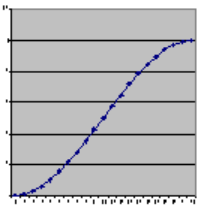
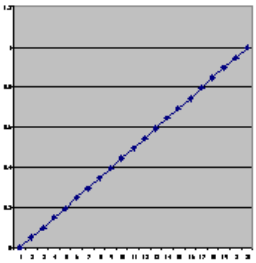
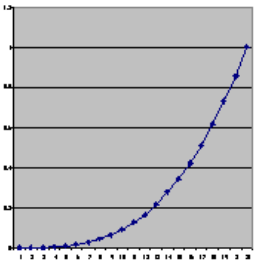
| Curve Type | Graph | Parameters |
|---|---|---|
| Smooth / Sinusoidal |  | [ 0, 0, 1, 1 ] |
| Linear |  | [ 0, 0.33, 0.66, 1 ] |
| Exponential |  | [ 0, 0, 0, 1 ] |

Table 3.5: Bezier Curves for Easing In/Out of Animations

Values for the control points are usually specified between 0 and 1, although other values can be used.

When easing in/out you specify the duration over which the ease occurs. Conceptually this horizontally stretches/squashes the curves. If you ease in an animation that is already easing in the curve continues from its current value rather than snapping back to the start of the curve.

If you ease back in an animation that is currently easing out then the ease-in continues from where the ease-out left off, if the ease curves are symmetric.

#### 3.3.6.4 Animation Blending

The `hkaAnimatedSkeleton` class aggregates all playing animation controls for a specific instance of a character. It is responsible for advancing these controls, sampling their underlying animations, and combining these sampled poses into a single local pose that will ultimately be used for rendering.

It also provides support for sampling and blending any motion that has been extracted from each animation. This separation of motion blending and animation blending is efficient. It allows the character motion to be checked independently from the rest of the animation system–this check may occur frequently per frame in, say, an NPC's state machine.

The interface also allows you to specify additional per track weights. These allow you to feather all animations playing on a particular bone or float slot. This could be used to dull effects from a limb that has been shot.

**Normal Blending**

Normal blending is the most common use case and corresponds to 'averaging' the contribution of multiple tracks, as follows:

```
For each active animation control bound to this animated skeleton
     If the binding specifies normal blending
       Use the control to sample the animation using the current control state
       For each track in the animation
           If this animation track is bound to a bone
               Weight = Motion Weight *  Control Weight * Track Weight
               Blend transform/float into bone/float slots according to Weight, and accumulate Weights

 For each bone in the skeleton
     If the total weight on this bone is below bind pose threshold
         Blend in the bind pose for the skeleton

 Normalize transforms and floats by sum of weights of that track.
```

**Note:**
If the total weight of all contributing controls on any particular bone drops below a user specified threshold (the *bind pose threshold*) then we automatically blend in the reference pose transform for that bone. This eliminates any popping artifact caused by animations suddenly ceasing or feathering that is too aggressive.

As in Interleaved Animations, we use linear interpolation to perform the blending. To be precise, blending happens as follows:

We add a weighted multiple of each of the position, rotation and scale components separately (with a hemisphere check for the quaternion representing the rotational component), and similarly with the float values. After all contributions have been added we 'normalize' the resulting output pose by division by the total weight (for position, scale and float components) or by quaternion normalization (for rotation components). This has the added advantage of being order-independent–animation controls can be added to the skeleton in any order and are guaranteed to produce the same outcome.

**Note:**
The blend hint is not used for *float tracks* which are always blended the same way.

**Additive Blending**

Additive blending allows detail animations to be layered onto an existing base animation to add subtle detail to the final pose. To produce the detail animation use either the runtime utilities found in `animation/animation/animation/util/hkaAdditiveAnimationUtility.h` or use the `CreateAdditiveAnimationFilter`. The animation is created by extracting the detail from a base pose or animation. One common option removes the first frame of an animation from each of the subsequent frames leaving just the detail. When using the utility or the toolchain you have the following options when choosing the base:

- Use the first frame of an animation
- Use the reference pose from a corresponding skeleton
- Use raw animated data. This data must be same size as the original data (runtime only).

Once the detail animation or additive animation data has been created we need to inform the blending routines in `hkaAnimatedSkeleton` that this data should be layered on during blending. This is done by setting the blend hint flag in the `hkaAnimationBinding` for the given additive animation. This blend hint flag currently has two values, *NORMAL* and *ADDITIVE*. Normal blending follows the pseudo code above. For additive blending the routine below is run

```
For each active animation control bound to this animated skeleton
     If the binding specifies additive blending
       Use the control to sample the animation using the current control state
       For each track in the animation
           If this animation track is bound to a bone
               Weight = Motion Weight *  Control Weight * Track Weight
               add in in bone according to Weight
```

This is very similar to normal blending except the type of blend operation is different and no renormalization occurs afterward. During sampling all normal animations are blended first and then all additive animation are layered on. For an example of additive blending see the demo `demos/AnimationApi/Blending/Additive`; this demo shows how to create an additive animation at runtime.

**Note:**
If you are using the runtime utilities to create additive animations you must remember to manually set the blend hint for the corresponding animation binding. Also if you are creating additive animations directly in your modeler (instead of using the `CreateAdditiveAnimation` filter) you will have to manually set the blend hint at runtime.

```
// Switch the binding to additive so this animation will be blended differently in sample and combine.
 m_binding[HK_ADDITIVE_ANIM]->m_blendHint = hkaAnimationBinding::ADDITIVE;
```

**Note:**
Additive blending is not explicitly supported for float tracks–float tracks are always blended 'additively'.

#### 3.3.6.5 Playback of hkaQuantizedAnimation

The `hkaQuantizedAnimation` framework is designed to complement or replace the `hkaAnimatedSkeleton` framework. This framework introduces three new main classes:

- `hkaQuantizedAnimation`
- `hkaBlender`
- `hkaQuantizedSampleAndCombineJob`

These classes allow the user to fully sample and blend characters with an emphasis towards maximizing decompression and blending speed on SIMD platforms.

**hkaQuantizedAnimation**

The `hkaQuantizedAnimation` compression pipeline is introduced in Quantized Compression section. The `hkaQuantizedAnimation` class is optimized for decompression speed. This class supports two sampling interfaces:

- Inherited functions from `hkaAnimation` such as `sampleTracks()` or `samplePartialTracks()`
- The new `sampleFullPose()` interface

The `hkaQuantizedAnimation` implementation is designed to work as efficiently as possible with the `sampleFullPose()` interface. Using the inherited `sampleTracks()` interface will result in a performance penalty. Note that when using an `hkaQuantizedAnimation` instance within the `hkaAnimatedSkeleton`, the `hkaAnimatedSkeleton` will call `sampleFullPose()` if and only if all animations associated with the `hkaAnimatedSkeleton` are of type `hkaQuantizedAnimation`; mixing of types other than `hkaQuantizedAnimation` will force use of the `sampleTracks()` interface and reduce performance.

The `sampleFullPose()` interface produces both output values and output weights for all bones and floats in the character, regardless of whether a full or partial body is animated. The output weights will typically be integer values: 1.0 for bones that are present in the animation, and 0.0 for bones that are not present in the animation. Blending a full body animation with a partial body animation using the `hkaBlender` will result in fractional weights being created.

**hkaBlender**

The hkaBlender provides an optimized set of utility functions for blending poses together on SIMD enabled platforms. Each blending function provides the appropriate analogue of `dst = ( 1 - alpha ) * srcL + ( alpha ) * srcR`. The hkaBlender functions properly on non-SIMD platforms though the speed gains are not as dramatic. Note that *all* input and output arrays are expected to be 16-byte-aligned and have allocations rounded up to the nearest multiple of four in size when operating on SIMD platforms.

The interface for the hkaBlender provides the following functionality:

- Blending of arrays of `hkReal`, `hkVector4`, `hkQuaternion`, or `hkQsTransform`.
  - Blending may be performed weighted or unweighted (per-bone weights).

  - Blending supports normal, additive, or subtractive modes.
  - Rotations may be slerped or lerped together (lerping is faster, but requires normalization).

- Multiplication (or inverse multiplication) of arrays of `hkReal`, `hkVector4`, `hkQuaternion` or `hkQsTransform`.
- Normalization of arrays of `hkQuaternion` or `hkQsTransform`.
- Rotation of arrays of `hkVector4` by arrays of `hkQuaternion` or `hkQsTransform`.
- Conversion of localToModel or modelToLocal.

The blending functions above may be called explicitly by the user if desired. These functions are called internally when using the `hkaQuantizedSampleAndCombineJob`.

Bone and float blend weights are carried in separate tracks, each of which is the size of the number of bones or floats, respectively. The algebra used for computing a new blend parameter and weight given and original blend parameter and left and right weights is shown below.

```
//
// Weighted blending proceeds by computing a new blend parameter beta, and new weight w,
// that are a function of the original blend parameter alpha and the given left and right weights
//
// If weightL > weightR then want beta < alpha.  Let beta = alpha * weightR / weightL
// If weightR > weightL then want (1-beta) < (1-alpha).  Let (1-beta) = (1-alpha) * weightL / weightR
//
// Lastly, handle zero weights...
//
//               |  weightL > 0                      |  weightL == 0
// ---------------------------------------------------------------------------
// weightR > 0   |  beta = (see logic above)         |  w = weightR
//               |  w = blend(weightL,weightR,beta)  |  beta = 1
// ---------------------------------------------------------------------------
// weightR == 0  |  w = weightL                      |  w = 0
//               |  beta = 0                         |  beta = alpha
//
```

### hkaQuantizedSampleAndCombineJob

The `hkaQuantizedSampleAndCombineJob` provides a universal job description for sampling and blending a list of `hkaQuantizedAnimation` instances for a single character. This class is used to describe blends on any system architecture. The `hkaQuantizedSampleAndCombineJob` consists of two parts:

- Per-character input settings and output buffer descriptions
- A list of animations to be decompressed and blended, including per-animation blend flags

For a full description of the `hkaQuantizedSampleAndCombineJob` class, please review the `hkaQuantizedSampleAndCombineJob.h` class header file.

An `hkaQuantizedSampleAndCombineJob` instance may be created manually by the user, or automatically by referencing an `hkaAnimatedSkeleton` instance. Creating an `hkaQuantizedSampleAndCombineJob` instance manually gives the user full control over the job description in addition to the ability to use all possible blend flags, such as subtractive blending which is not supported by the `hkaAnimatedSkeleton`

class. The hkaQuantizedSampleAndCombineJob is designed to interface cleanly with user systems for setting up blend lists.

To create an hkaQuantizedSampleAndCombineJob instance by referencing an hkaAnimatedSkeleton instance, use hkaMultithreadedAnimationUtils::allocateQuantizedSampleAndCombineJob() utility function which takes an hkaAnimatedSkeleton instance as input and produces an hkaQuantizedSampleAndCombineJob as output. Memory allocated for this job must later be freed by calling hkaMultithreadedAnimationUtils::deallocateQuantizedSampleAndCombineJob() on the newly created hkaQuantizedSampleAndCombineJob instance.

An hkaQuantizedSampleAndCombineJob instance may be executed in a single thread by calling executeQuantizedSampleAndCombineJob(), which is declared in hkaCommonAnimation.h. To execute an hkaQuantizedSampleAndCombineJob instance in parallel on multiple threads simply add the hkaQuantizedSampleAndCombineJob to an existing hkJobQueue like any other hkJob.

Note that the hkaQuantizedSampleAndCombineJob works with animations of type hkaQuantizedAnimation only. When using other animation types (such as hkaSplineCompressedAnimation) or when mixing multiple animation types, you must use the hkaAnimationSampleAndCombineJob instead. While compatible with the hkaQuantizedAnimation type, the hkaAnimationSampleAndCombineJob job type does not fully take advantage of the speed potential of the hkaQuantizedAnimation type or the hkaBlender blending routines.

#### 3.3.6.6 Mirroring Animations

**Overview**

Mirroring animation allows animators to synthesize a new animation from an existing one in which motion is transferred to opposite sides of the body. An example of an original animation and its mirrored counterpart are shown below in **??**. Note how the opposite hand is waving in the mirrored animation yet asymmetries of the model and its textures are preserved. The Havok logo and black knee patch remain on the character's left side in both poses.

Figure 3.19: Example Mirrored Animation

Mirrored animation can also be used as a form of compression by using a left handed data set to represent both left and right handed motion simultaneously. The overhead of storing mirroring information is typically very small compared to the animation track data itself, allowing a series of left and right handed animations to be stored in roughly one half the space of exporting both animations directly.

**hkaMirroredSkeleton and hkaMirroredAnimation**

Two classes are used to create a mirrored animation: `hkaMirroredSkeleton` and `hkaMirroredAnimation`. One instance of `hkaMirroredSkeleton` needs to be created for each skeleton (rig) which is to be mirrored. Just as multiple `hkaAnimation` instances can share the same `hkaSkeleton` instance, multiple `hkaMirroredAnimation` instances can share the same `hkaMirroredSkeleton` instance.

The `hkaMirroredSkeleton` class, shown below, encapsulates all settings relevant to mirroring an animation. Mirroring an animation requires user setup in three categories.

- A mapping of bone indices from one side of the body to the other (e.g. left_forearm âĘŤ right_forearm)
  - This can be done by specifying the mapping manually, or using bone names to perform the mapping automatically.
- Geometric setup of mirroring
  - The geometric setup of mirroring is most easily accomplished by providing an example symmetric pose (such as the T-pose commonly used for skinning). The world axis corresponds to the normal to the mirror plane in the model space of the character in the given symmetric pose. Another way to imagine this axis is to choose the direction which is perpendicular to both the "up" and "forward" axes of the character in model space (e.g. the direction that points "left" or "right").

  - The tolerance setting may be used to correct small asymmetries in the provided symmetric pose, assuming that canonical axes (X,Y,Z) were chosen as joint axes. A value of 0.0 disables this axis snapping. Typical tolerance values are 0.01 or less.

- Mapping of annotation names (if desired)

Public member functions, shown below, are provided to allow the user to setup mirroring using a variety of input parameters.

```
class hkaMirroredSkeleton
{
public:
    hkaMirroredSkeleton( hkaSkeleton *skeleton );
    ~hkaMirroredSkeleton();

    // BONE PAIRING

    /// Allows the user to set bone pairing information
    void setBonePair( hkInt16 bi, hkInt16 bj );
    void computeBonePairingFromNames( const hkArray< hkString > &ltag, const hkArray< hkString > &rtag );

    // MIRRORING SETUP

    /// Given a worldAxis (typically X=[1,0,0,0] Y=[0,1,0,0]
    /// Z=[0,0,1,0]) and an initially symmetric pose, compute all
    /// necessary mirroring setup.  Note that bone pairing setup
    /// must be complete before this function should be called.
    void setAllBoneInvariantsFromSymmetricPose( const hkQuaternion& worldAxis, hkReal tolerance, const
      hkQsTransform *poseModelSpace );
    void setAllBoneInvariantsFromSymmetricPose( const hkQuaternion& worldAxis, hkReal tolerance, const
      class hkaPose & );
    void setAllBoneInvariantsFromReferencePose( const hkQuaternion& worldAxis, hkReal tolerance );

    // ANNOTATION PAIRING

    /// Set annotation pair
    void setAnnotationPair( hkaAnnotationTrack::Annotation* ai, hkaAnnotationTrack::Annotation* aj );
    void setAnnotationNamePair( const char* namei, const char* namej );
};
```

Once the `hkaMirroredSkeleton` class is created, it can be added to an `hkaMirroredAnimation`. Note that is important to access a mirrored animation through a binding created by a call to `createMirroredBinding()`. This will ensure that mirrored bones are properly mapped in the resulting animation. When used with this binding, the `hkaMirrroredAnimation` class then behaves the same as any other `hkaAnimation` instance.

```
class hkaMirroredAnimation : public hkaAnimation
{
public:

    hkaMirroredAnimation( hkaAnimation* originalAnimation,
                                  hkaAnimationBinding* originalBinding,
                                  hkaMirroredSkeleton* mirroredSkeleton );

    ~hkaMirroredAnimation();

    hkaAnimationBinding *createMirroredBinding();
    static void HK_CALL destroyMirroredBinding( hkaAnimationBinding *binding );
};
```

**Ensuring Proper Mirroring Setup**

If a character does not appear to mirror correctly, ensure that the following settings are correct:

- Bone Pairing
  - Mirroring will not produce desired results if bone pairs are set up incorrectly. To verify bone pairs, try printing the names of bones and associated pairs to the console, checking for any inconsistencies.

- Geometric Setup
  - If the mirrored character is not skinned correctly, this typically points to an incorrect choice of world axis given in the `setAllBoneInvariantsFrom*()` call. Verify that your settings are correct by examining the asset in the modeling tool and choosing the correct axis (see the description of mirroring setup above for a description of the correct axis). Typically this axis will be X, Y, or Z.
  - If the character is skinned correctly, but the mirrored version faces in the wrong direction, or upside down, ensure that the given symmetric pose faces in the same direction as a "forward" animation (such as a forward walk). If the given symmetric pose faces in another direction, this rotation offset will show up improperly in the final result.

## 3.3.7 Multithreading

The Havok Animation SDK can be used in a multithreaded environment in a straightforward manner as no data needs to be written to by multiple threads.

### 3.3.7.1 Animation Jobs

Havok Animation uses the same task, job and job queue structure shared by other Havok components and supports four types of jobs:

- `hkaAnimationSampleOnlyJob`: Samples N animations into N output buffers. This is the multithreaded equivalent of a call to `hkaAnimation::sampleTracks()` (or `hkaAnimation::samplePartialTracks()` if partial sampling is being used). This job is useful if you are doing your own blending. This job type is being phased out and will be deprecated in a future release. It is recommended that you use `hkaSampleBlendJob` instead. This job type does not support the newer `hkaQuantizedAnimation` or `hkaPredictiveCompressedAnimation`.
- `hkaAnimationSampleAndCombineJob`: Samples all animation controls of a single `hkaAnimatedSkeleton` into a single output buffer. This is the multithreaded equivalent of a call to `hkaAnimatedSkeleton::sampleAndCombineAnimations()` (or `hkaAnimatedSkeleton::sampleAndCombinePartialAnimations()` if partial sampling is being used). This job is useful if you are using the `hkaAnimatedSkeleton` interface. This job type is being phased out and will be deprecated in a future release. It is recommended that you use

hkaSampleBlendJob instead. This job type does not support the newer hkaQuantizedAnimation or hkaPredictiveCompressedAnimation.

**Note:**

An optional post-conversion of local-space to model-space is available as part of the hkaAnimationSampleAndCombineJob.

- hkaQuantizedSampleAndCombineJob: This is analogous to the hkaAnimationSampleAndCombineJob except that it is optimized for the use of hkaQuantizedAnimations. It cannot be used if any of the animations being blended are not hkaQuantizedAnimations. This job type is useful if you want the absolute best performance and are willing to sacrifice some memory.
- hkaSampleBlendJob: This job is intended to replace the hkaSampleAnimationJob and the hkaAnimationSampleAndCombineJob. Features of this job type include the following:
  - It supports all animation compression types.
  - It can be used to decompress and blend an hkaAnimatedSkeleton like the hkaAnimationSampleAndCombineJob. If skeleton mapping is being used by the hkaAnimatedSkeleton the remapping will be done by the job.
  - It can be used to decompress N animations into N output buffers like the hkaAnimationSampleOnlyJob. But unlike the hkaAnimationSampleOnlyJob, the output is in bone space, not track space. The output poses can also be optionally converted to model space.

The uses of each job type are illustrated in the SampleAndBlendMultithreadingDemo and SampleOnlyMultithreadingDemo, found in the Multithreading folder of the Animation Api demos.

**Using the hkaSampleBlendJob for Sampling and Blending**

To initialize a hkaSampleBlendJob to do sampling and blending of an hkaAnimatedSkeleton you simply call one of the build() interfaces.

**Using the hkaSampleBlendJob for Sampling Only**

To initialize a hkaSampleBlendJob to just sample a set of animations you must first call initSampleOnly() once and then call addAnimation() to add each animation to the job.

#### 3.3.7.2 Scheduling, Processing and Data Management

Scheduling of the tasks is left entirely up to the user as (unlike with a Havok Physics 'World') there is no global 'container' of animation or animated skeletons which are active at a given time. Per-character and per-bone LOD is done in the same way as for non-multithreaded sampling and sampling and blending: the user explicitly specifies which animations and how many bones/float slots to sample when creating the required animation job.

When run on *shared-memory architectures* the data is operated on directly, and the methods called are the same as the equivalent singlethreaded methods. Thus most of the overhead for job setup (conversion to 'chunks' for DMA, etc.) is avoided.

When run on *non-shared-memory architectures* the input data will be DMA'd across to a be accessible on a different CPU, the 'withChunk' versions of sampling will be used. There is a small overhead for job creation (conversion to 'chunks' for DMA, etc.).

#### 3.3.7.3 Synchronous and Asynchronous Job Processing

Havok Animation allows full flexibility for addition of new jobs to the job queue and notification of completed jobs. Knowing your use-case and likely distribution of animation jobs you may pick the best scheme for ensuring all threads are used with maximum efficiency.

**Addition of new jobs.**

The job queue may be set to force termination of a worker thread when empty or continuous querying for new jobs using the `hkJobQueue::WaitPolicy`. By default the policy is *WAIT_UNTIL_ALL_WORK_COMPLETE* which is useful if you know ahead of time how many jobs there will be and (batch) add all jobs at once, waiting until the task is complete before further processing the results (for example IK fixup, custom blending). This is synchronous processing.

If however you are continuously creating new jobs or processing the results of jobs before all jobs are complete, then you may set the wait policy to *WAIT_INDEFINITELY* in which case the processing is asynchronous.

**Notification of job completion**

All job types have a `hkaJobDoneNotifier` that contains both a semaphore member (which may be shared) and a flag member, which may be used to wait for notification of job completion. For synchronous processing you may not need to use either of these if you have ensured there is enough work to be done by all threads evenly.

For asynchronous processing you may wish to use one of these notifications to determine when a job is done so that you may further process the results.

The use of each processing is illustrated in the both the `Animation/Api/Multithreading/SampleAndBlendMultithreadingDe` and the `Animation/Api/Multithreading/SampleOnlyMultithreadingDemo`.

### 3.3.8 Skeleton Mappers

#### 3.3.8.1 Overview

Skeleton mappers may be used to convert a pose from one skeleton to another. Two basic use cases are supported:

- Converting poses between a low and high resolution version of the same skeleton. This is known as "Ragdoll Mapping".
- Converting poses between two different characters. This is known as "Animation Retargeting".

The remainder of this section is specific for Ragdoll Mapping, though many concepts are also common to Animation Retargeting. Animation Retargeting is described fully in subsection 3.3.9.

There are situations where you may want to represent the same character using different skeletons. A common usage of this is when integrating animation with physics : while the animated character can have a high bone count (let's say 50), usually its physical representation (ragdoll) will have fewer (let's say 15)

In those situations, we want to be able to convert a pose from one representation to the other. For example, we may want to convert the animation (50 bones) pose to a physics/ragdoll pose (15 bones) so the ragdoll starts with the same pose the animation had. Similarly, we may want to convert a ragdoll pose (15 bones) to an animation pose (50 bones) so we can put skin around the ragdoll.

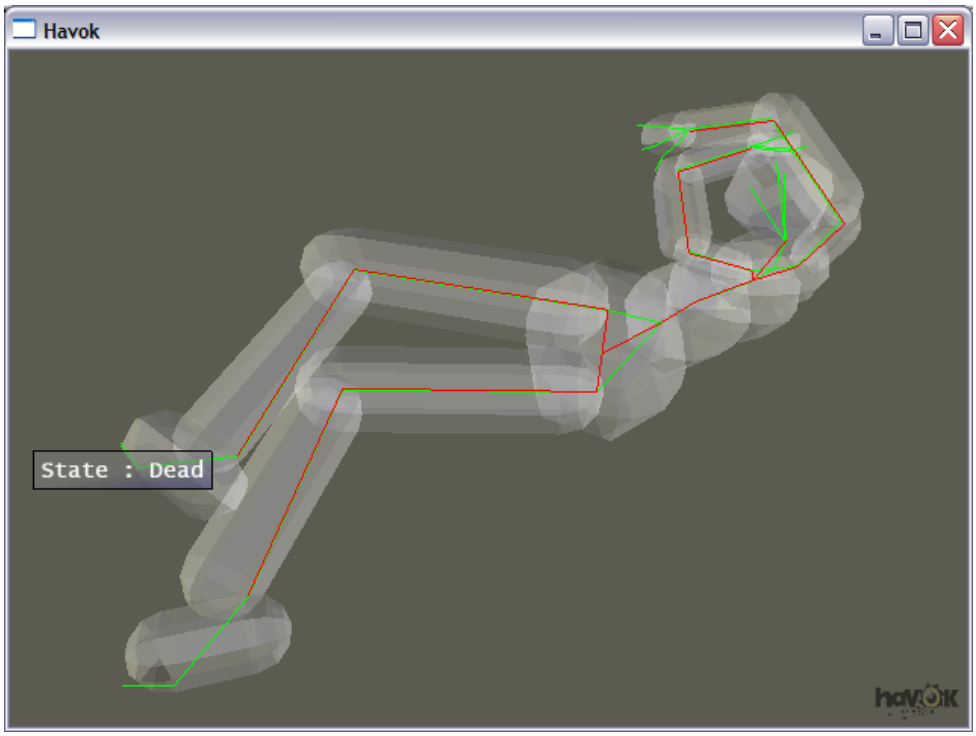
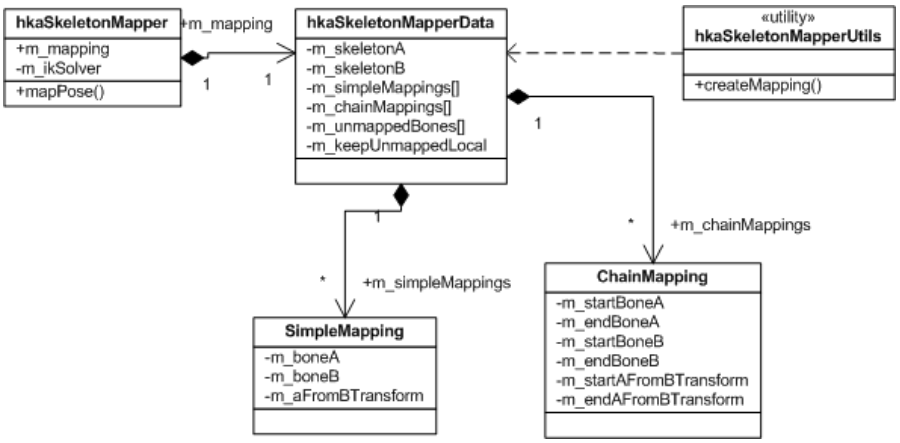



Figure 3.20: Here we map the physics pose (in red) to a higher bone count pose (in green), which we can then use for skinning the ragdoll.

The run-time classes involved on doing this kind of retargeting (where the skeletons represent the same character, with different resolutions) are called skeleton mappers (`hkaSkeletonMapper`).

#### 3.3.8.2 hkaSkeletonMapper and hkaSkeletonMapperData



The `hkaSkeletonMapper` is a run-time class that is responsible for transforming a pose from one skeleton (A) to another skeleton (B). All the information required by this hkaSkeletonMapper is stored in a hkaSkeletonMapperData structure, which is passed on construction. As we will see later, the `hkaSkeletonMapperUtils` class can be used to create this data based on any two skeletons.

**How is it done**

The skeleton mapper does its job of mapping a pose between two skeletons through its method `mapPose()`.

The `mapPose()` method works with two poses: *poseA* is an input pose–it is not modified)–while *poseB* is an input/output pose–it will be modified based on the *poseA*, but the previous values may also be used (for example, for bones that are not present in A).

This process is done in model space, were bones in the output pose (B) are gradually driven by the input pose (A). This job is done by looking at the information in the `hkaSkeletonMapperData`, which contains three main sets of mapping information: simple mappings, chain mappings, and unmapped bones.

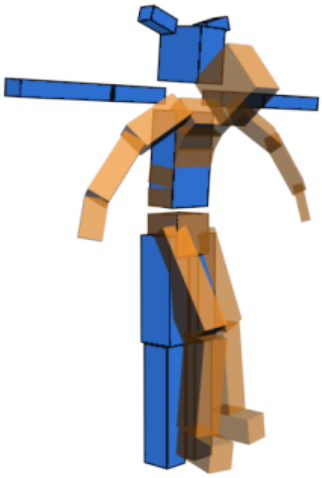


Figure 3.21: The two poses used by the mapper: the input pose A (in orange) and the output pose B (in blue) that will be modified.

**Simple Mappings**

Simple mappings are one-to-one bone mappings. A simple mapping specifies that the transform for bone B should be driven by the transform in A. An extra transform is also used for the cases where the pivots of these two bones have different location or orientation.

Figure 3.22: Simple mappings are one-to-one bone mappings, and are applied first.

**Chain Mappings**

Chain mappings are more complex mappings, used to map chains of bones from one skeleton to the other. The best example is the spine: while one skeleton may use 5 bones for the spine, another skeleton may use only 2 or just one. The mapper treats these chains a single problem to solve.

The way chain mappings are solved is by matching the start of the chains in both skeletons and then modifying the chain bones in B (output pose) by trying to reach the same position for the end bone as in A, without modifying the local orientation of the bones in the chain.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

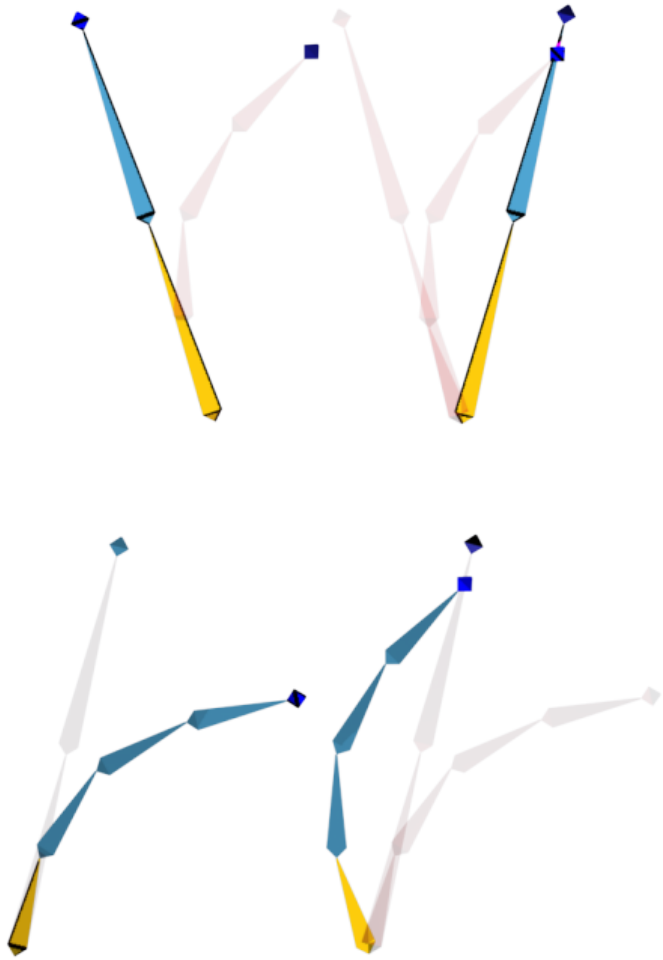


Figure 3.23: When solving chain mappings we try to reach the desired end target position by just rotating the whole chain to align it with the line going from the start bone to the target position. By doing so, the results are reversible.

Figure 3.24: Chain Mappings are applied after simple mappings - in the example, the spine is aligned so it points towards the head (its end effector)

**Unmapped Bones**

Not all bones can be mapped by simple mappings or chain mappings. Sometimes the output pose B has a higher resolution than the input pose A, and therefore some bones cannot be driven by pose A. The mapper data has a list of unmapped bones, and can do two things with them (based on the value of the member `m_mapping.m_keepUnmappedLocal`):

- Keep those bones in the model pose they had in pose B
- Keep those bones in the local pose they had in pose B

Keeping the bones in their world pose would, in the example we are working with, leave the two antennae where they are in the picture above (we will see in the next step, though, that skeletal constraints may move them to their proper location -but wouldn't change their orientation). Keeping the local pose of the unmapped bones, however, will keep these unmapped bones local transform and, therefore, they will move with their parent (the head in the example).

Figure 3.25: Keeping unmapped bones local ensures that the bones in pose B that have no equivalent in pose A follow their parent

The most important bones to unlock are those that are responsible for movement of the character such as the root or pelvis node. Additionally bones that have siblings, typically the clavicles and hips, can also be unlocked to improve quality when mapping from a ragdoll skeleton to a high resolution skeleton. The Havok Content Tools allow for individual bones to be selectively locked or unlocked.

**Enforcing Skeleton Constraints**

During the final step, which is one of the most important, the mapper goes through the transforms in the output pose and ensures that none of them breaks any of the constraints for that bone defined in the skeleton.

Currently, the only skeleton constraint that can be specified for a bone is "lock translation"–whether this bone can translate from its parent or not. If "lock translation" is true, the only translation allowed is that defined in the initial transform of the bone.

By default, all translations are allowed in a skeleton, so it is important that you lock the translations for all bones in your skeleton where you don't want the mapper to add extra translations. In the most common case, you only want translations allowed in the root bone(s) (center of mass/root/pelvis). In some rigs, you may need to allow translations for other bones as well (see "Triangle Pelvis Setup" for another example).

There are many reasons why you want to limit the amount of translations added by the mapping. In the case of mapping from animation to physics, for example, adding extra translations to the physics pose would likely mean setting a pose where physical constraints are broken, causing popping artifacts or instability. Similarly, adding unwanted translations when mapping to an animation pose may cause artifacts on the skinning (stretching, etc.).

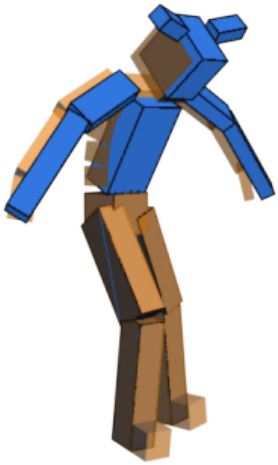


Figure 3.26: Enforcing skeleton constraints will make sure that no unwanted translations were added to the output pose. In the example, since the single-bone spine reaches further than the 5-bone spine (because it cannot bend), the head and arms no longer match pose A in world.

**The Triangle Pelvis Setup**

Since, as we've seen, the skeleton mapper works in world space (modifies subsets of the pose independently), it can handle the situation where the input and output skeletons have different hierarchies.

The most common example is what is know as the "Triangle Pelvis" setup in Character Studio.

Character Studio allows the artist to optionally use this setup for their bipeds. When doing so, the thighs in 3ds Max are parented to the first link in the spine instead of the pelvis (their "natural" parent) [10].

It is important to notice that skeletons using that structure should keep the "lock translations" flag off (should allow translations) for both thigh bones since, although they don't translate with respect to the pelvis (their "natural" parent), they may do with respect to the spine (their actual parent).

[10] The reason seems to be that the Physique skin modifier works better with this "unnatural" parenting.

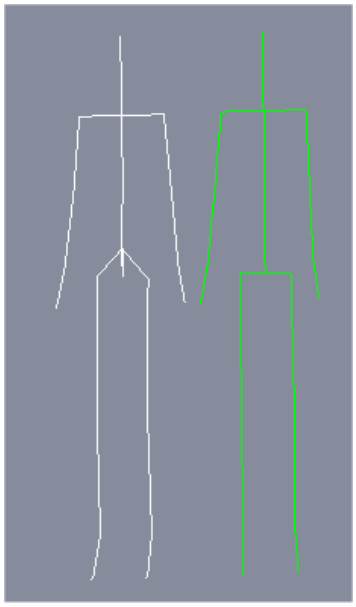


Figure 3.27: The skeleton on the left has triangle pelvis setup - the legs are parented to the spine. The skeleton on the right has a natural setup - the legs are parented to the pelvis. The mapper works in world space and therefore can map poses from one skeleton to the other - but it is important to allow translations on the thighs in the skeleton with triangle pelvis setup.

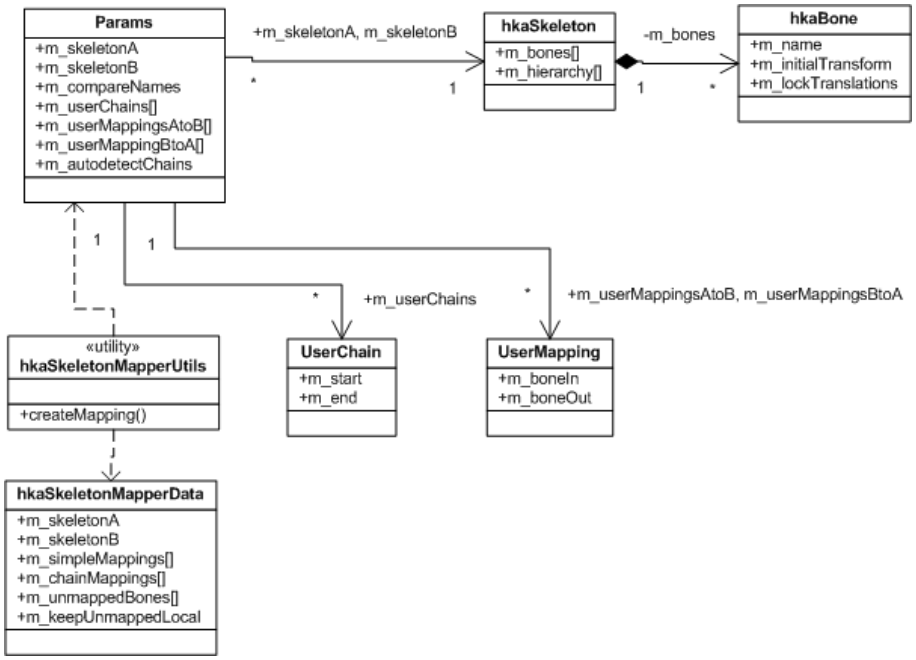
**Twist Bones for Ragdolls**

Twist bones do not follow the standard hierarchy of a character, therefore special care should be taken when mapping them. A high resolution skeleton may contain twist bones, but typical ragdoll skeletons do not. For example a character's bicep may be represented by a single animation bone "UpperArm" and two twist bones "UpArm Twist" and "UpArm Twist1" in the high resolution skeleton. Here "UpperArm" and "UpArm Twist" will be siblings, while "UpArm Twist1" is the child of "UpArm Twist". In the ragdoll skeleton there will typically be a single bone "Ragdoll_UpperArm". When setting up a high to low mapping, the high res bone "Upper Arm" should drive the ragdoll bone "Ragdoll_UpperArm". When setting up a low to high mapping, the low res bone "Ragdoll_UpperArm" should drive both the high res bones "UpperArm" and "UpArm Twist"; "UpArm Twist1" will typically not be driven.

#### 3.3.8.3 Setting Up Mapping Data

In the previous section we saw how the mapper works at run time, and how it uses a combination of simple mappings, chain mappings, a list of unmapped bones and skeletal constraints data to do so.

All this information is passed on construction using an `hkaSkeletonMapperData` object. While you can create this data by hand, Havok provides a utility, `hkaSkeletonMapperUtils`, to facilitate this setup.




This utility works based on the names of the bones from two skeletons A and B, and creates two sets of mapping data, one for going from A to B, the other one for going from B to A.

### Bone Names

Each bone of a skeleton has a name parameter. The utility will use these names in order to match bones for the mapping. Since in most cases the names of the bones in the two skeletons will be similar, but not identical (for example, the head can be called "Girl Head" in one skeleton, but "Ragdoll_Head" on the other, a user-defined string compare function (`m_compareNames`) is passed to the `hkaSkeletonMapperUtils`.

### The Initial Pose

In order to generate the proper mapping information, the `hkaSkeletonMapperUtils` utility compares the initial pose of the two skeletons (the pose generated by the initial transform of each bone) in world space. It is, therefore, very important that both skeletons are properly aligned in that initial pose.

### Simple Mappings

The utility will create a simple mapping for all pairs of bones in A and B whose name match[11]. Their world transforms will then be matched, and the offset will be stored and used for this mapping at runtime. Notice that, for optimal mapping, you want this offset to have no translation (otherwise a rotation in one skeleton will map to a rotation and a translation in the other skeleton); as a precaution, a warning will generated by Havok if large translations are found.

You can also force the utility to create simple mappings (from A to B or from B to A) for bones whose names don't necessarily match[12].

### Chain Mappings

[11] Unless they are part of a user chain.

[12] One example of this can be found when mapping from a low res skeleton to a high res skeleton, where the high res skeleton has an extra "forearm twist" bone to help with skinning. This bone has no match in the low res skeleton, but it should be driven by the "forearm" bone when mapping from the low res to high res.

By default, the utility will try to autodetect chain mappings. If `m_autodetectChains` is on, a chain mapping will be created when:

- Two simple mappings have been created for bones (which we call *Start* and *End*)
- You can go from *Start* to *End* through the hierarchy of each skeleton without finding another bone that has already been mapped (the bones you go through form the chain on each skeleton)
- At least one of the chains has length >=3 (including *Start* and *End*)

You can also force the utility to create specific chains, by adding `UserChain` objects. You can do this easily by specifying the names of the start and end bones[13].

**Skeletal Constraints**

It is important to remember that maintaining skeletal constraints is one of the jobs of the mapper at run time, and it's one of the main reasons for using the mapper. Therefore, make sure that the bones in both skeletons have their `m_lockTranslation` flag set up properly.

As a precaution, the mapper (`hkaSkeletonMapper`) will generate a warning on construction if the target skeleton has no translations locked.

### 3.3.9 Retargeting Animations

The Animation Retargeting feature allows users to take animations originally authored for one character and plausibly transfer that animation at runtime to a different character. This feature enables asset reuse by allowing a single animation asset to be shared between two or more characters. Screenshots below show retargeting examples of retargeting both similar characters in **??** and retargeting dissimilar characters in **??**.

[13] For example, if one skeleton has a spine of 3 links (spine1, spine2, spine3, head) and the other skeleton has a spine of 5 links (spine1..spine5, head), the system by default will create a chain from spine3 to head. However, commonly you want to force a chain from spine1 to head.

Figure 3.28: Retargeting Several Characters

Figure 3.29: Retargeting Dissimilar Characters

Havok Animation's Retargeting feature consists of a setup component and a runtime component. Setup may be performed either at load time using the Havok Animation SDK or at content creation time using the Havok Content Tools. Retargeting is applied to an original animation pose at runtime to create a pose appropriate for the retargeted character. The SDK provides flexibility in choosing how and when retargeting is applied to maximize benefit. Both the character pose and extracted motion can be

retargeted, allowing larger retargeted characters to run faster and jump higher than smaller characters.

#### 3.3.9.1 Example Supported Use Cases

Animation Retargeting is designed to have general scope and support many different use cases. Use case examples shown in the Havok Animation Demos include:

- Characters with similar hierarchies, but differing bone lengths **??**
  - Example: Tall characters and short characters
- Characters with a skeleton (rig) which is a subset or superset of another character
  - Example: Human and human with wings
- Characters that share a common "trunk" of bones
  - Example: Human with wings and human with lizard tail
- Characters with animated attachments
  - Example: Character holding staff and character holding sword
- Characters with extracted motion
  - Example: Larger characters walk faster, jump higher than smaller characters
- Characters with differing numbers of bones in a chain
  - Example: Character with five spine bones and character with two spine bones
- Characters that differ in both size and structure **??**
  - Example: Human and Bipedal Monster (see below for scope of retargeting)

Animation Retargeting is fully integrated with other Havok Animation features. Retargeting supports

- Partial Animations
- Blending
- Extracted Motion
- Compressed animations
- Mirrored animations

**Note:**
Runtime Level of Detail (LOD) sampling is currently not supported for retargeted animations. Partial Animations, however, are fully supported

#### 3.3.9.2 Scope of Retargeting

In general, the more similar the original and retargeted characters are in shape and structure, the more plausible the quality of the retargeted results. Retargeting will provide an exact result when the original and retargeted characters are identical, or a uniformly scaled multiple of one another. The result will also be exact for shared bones between two otherwise identical or uniformly scaled characters (e.g. a human and the same human with wings).

For other use cases the results of retargeting will be approximate. Structurally similar characters generally retarget well, though large differences in relative bone lengths can affect perceived quality. For example retargeting a human character's animation to another human character will generally produce pleasing results, retargeting the human character's animation to an ape with short legs and long arms may produce acceptable results, and retargeting the human character's animation to a fish has the least chance of success.

Note that Havok Animation's retargeting feature does not attempt to understand the differences in style of locomotion between characters. Retargeting a human walk cycle to a horse may result in the horse walking upright on its hind legs, but will generally not result in a galloping motion.

The Animation Retargeting feature also does not attempt to specially handle characters with differences in the parenting of bones. A real-world example of this would be retargeting between two characters, in which one character has a "Triangle Pelvis" setup and the other does not. As the Animation Retargeting feature does not attempt to understand the differences in the parenting of the hip bones, the results will have a lower chance of success than when retargeting between two characters with the same kind of pelvis setup.

For specific limitations associated with the retargeting feature, please see **??**.

#### 3.3.9.3 Retargeting Setup

During setup corresponding bones are chosen between the original and retargeted characters. Setup depends only on the original and retargeted character's skeletons and respective reference poses, not any animation data. Matching of individual bone for simple maps may be done manually, or automatically by matching bone names. Chains of bones may also be chosen between characters. Differences in position, orientation and length are determined for corresponding bones and bone chains in the reference pose of each character. At run time these offsets will be applied to the original pose to create the retargeted pose.

The reference pose of each character is used to determine the differences between corresponding bones. The reference pose is typically the initial pose of the character when the `hkaSkeleton` object is created, and need not be the same as the skin binding pose. Changes in the reference pose of each character can be used to modify the Retargeting setup. For example, moving the arms of one character further from the body in the reference pose will result in that character's arms being further from the body in all retargeted poses.

Each bone in the retargeted character may be driven either by one or zero bones of the original character. Bones which are not retargeted are not updated during retargeting.

Simple mappings relate a single bone of Skeleton A to a single bone of Skeleton B. Chain mappings may also be created which map a singly-linked chain of bones in Skeleton A to a singly-linked chain of bones in Skeleton B. Chain mappings allow for Retargeting between characters that have a different

number of bones in a chain, for example a different number of spine bones. Chain mappings are more computationally expensive than simple mappings, and should not be used when simple mappings alone are sufficient. Unlike the chain mappings used for ragdoll mapping, chain mappings used for retargeting should not include any simply mapped bones (ragdoll mapping requires that the beginning and end of chains be simply mapped).

**Note:**

Handling Twist Bones. Typically when creating skeleton mappers for both "A->B" and "B->A" use cases, the simple mappings will be the same. One use case in which they will not be the same is when only one `hkaSkeleton` contains Twist bones and the other does not. Twist bones are a special kind of bone created by some modelers which do not follow the usual conventions for bones used in skeletal hierarchies. Suppose that Skeleton A contains bones "UpperArm" and "UpArm Twist", while Skeleton B only contains the bone "UpperArm". In this use case the "UpperArm" bone of Skeleton A should drive the "UpperArm" bone of Skeleton B in the "A->B" mapping, while the "UpperArm" bone of Skeleton B should drive both the "UpperArm" and "UpArm Twist" bones of Skeleton A. This is similar to the situation shown in **??** where the "UpperArm" bones of each `hkaSkeleton` are driven by chain mappings and the "UpArm Twist" bones of Skeleton A are additionally driven by the "UpperArm" bones of Skeleton B. If both characters contain corresponding Twist bones then no special treatment is necessary, the twist bones should be mapped to one another in the same manner as non-Twist bones.

Two additional parameters are exposed, "AutoPos Tol" and "Rot Tol (Degrees)". See **??** for further explanation.

#### Retargeting Setup Using the Havok Animation SDK

Retargeting may be setup by manually creating a `hkaSkeletonMapper` object at runtime. In situations where the correspondence between the bones of retargeted characters may be determined procedurally, the `hkaSkeletonMapperUtils::createMapping()` utility function may be used to create new `hkaSkeletonMapper` objects. This utility takes `hkaSkeletonMapperUtils::Params` as input and creates two `hkaSkeletonMapperData` objects as output, one for retargeting Skeleton A to Skeleton B, and one for retargeting Skeleton B to Skeleton A. The `hkaSkeletonMapperData` class provides all the setup necessary to create an `hkaSkeletonMapper` object to perform retargeting with.

For further information about setting up hkaSkeletonMapper objects, please see subsubsection 3.3.8.2. When using an hkaSkeletonMapper for retargeting, ensure that `hkaSkeletonMapperUtils::Params.m_mappingType` is set to *hkaSkeletonMapperData::HK_RETARGETING_MAPPING*.

Extracted motion may also be retargeted using the `hkaSkeletonMapper` class. When using the `hkaSkeletonMapperUtils::createMapping()` utility, the `ExtractedMotionParams` class may be used to specify how extracted motion is scaled. Options include using the size of the bounding box of the character, using a specified bone length, or simply specifying scaling factors.

#### Retargeting Setup Caveats

Very short bones can cause quality issues when retargeting. The retargeting feature takes bone length and changes in bone length into account when retargeting. Consider for retargeting purposes, a bone in Skeleton A which is one millimeter long, and the corresponding bone in Skeleton B which is one centimeter long. Both bones are quite short compared to the size of a typical human character, and may be visually indistinguishable to the eye. However to the retargeting feature the one centimeter long bone is ten times longer than the one millimeter bone! Changes in position of the millimeter long bone of Skeleton A will be magnified tenfold when retargeted to Skeleton B. This situation may arise frequently with root or locater bones which are intended to be at the origin, but are not quite aligned exactly to the zero point. This could result in large changes in velocity as the source character moves away from the origin. This specific situation may be avoided by choosing the "Move Reference Pose To the Origin" feature when creating skeletons. In general the "AutoPos Tol" (`hkaSkeletonMapperUtils::m_positionMatchTolerance` in

the SDK) setting may be used to avoid this situation for all bones. This value is the length in world units below which bones are considered to have zero length. Any bones shorter than this value are considered to have the exact same length of zero. Additionally, bones of zero length should not be retargeted to bones of non-zero length (essentially this leads to a divide by zero in implementation). The `hkaSkeletonMapperUtils::createMapping()` fuction will warn if it finds two such mapped bones.

The quality of retargeting also degrades as the difference in orientation between the bones of the original and retargeted characters grows large. One use case not supported by the retargeting feature is two visually identical skeletons that have different underlying bone orientations. (For example if the "Affect Pivot Only" option of the modeler has been used to alter bone orientations of one character significantly). For this reason retargeting between two characters provides the best results when the differences in bone orientation are small between the reference poses of each character. Although there may be some retargetings in which large differences in angle are acceptable, typical corresponding bones should not differ in orientation by more than a few tens of degrees (30 degrees max is a good rule of thumb). The "Rot Tol (Degrees)" setting (`hkaSkeletonMapperUtils::m_rotationMatchTolerance` in the SDK) can be used to warn the user if two bones differ by more than a small change in angle. Additionally all retargeted characters should all have the same up and forward directions in the reference pose.

#### 3.3.9.4 Retargeting at Runtime

Animation Retargeting supports two basic runtime use cases based on where Retargeting occurs in the sampling and blending process. Each use case is described below. Users should consider the tradeoffs of flexibility and cost when choosing which use case best fits their needs.

- Retargeting a fully sampled pose from one character to another
  - Example: Retargeting the pose of a cow to the pose of a horse.
  - Concept: Equivelent to retargeting at the root of a blend tree.
  - Runtime cost: Proportional to the number of retargeted characters.
  - Flexibility: Can retarget full poses from one character to another.
  - Usage: `hkaSkeletonMapper::mapPose()`
  - Demo: `BasicPoseRetargetingDemo.cpp`
- Retargeting animation tracks to later be blended
  - Example: Retargeting a cow walk to a horse walk for later blending with other horse animations.
  - Concept: Equivelent to retargeting at the leaves of a blend tree.
  - Runtime cost: Proportional to the number of retargeted animations.
  - Flexibility: Can mix retargeted animations from several different characters.
  - Usage: `hkaDefaultAnimationControl::setSkeletonMapper()`
  - Demo: `BlendedRetargetingDemo.cpp`

**Note:**

When retargeting for any use case, it is important to initialize the destination pose to a known good pose (typically the reference pose) before retargeting. This is necessary for two reasons:

- Initializing unmapped bones (unmapped bones will not be filled in by retargeting)
- Initializing all bones involved in chain mappings. While simple mappings only write to the retargeted pose, chain mapping both read from and write to the retargeted pose.

**Retargeting Extracted Motion**

Extracted motion may also be retargeted. When using the `hkaDefaultAnimationControl` class (with an assigned `hkaSkeletonMapper`) extracted motion is retargeted automatically. When not using the `hkaDefaultAnimationControl` class, `hkaSkeletonMapper::mapExtractedMotion()` may be called directly.

### 3.3.10 Skeleton Partitioning

Skeleton Partitioning is an advanced feature of the Animation SDK that allows for more optimal sampling, retargeting, and blending. Skeleton Partitioning requires two key pieces of information to be present:

1. The named skeleton partitions, `hkaSkeleton::m_partitions`, and
2. Which partitions are used by the animation, `hkaAnimationBinding::m_partitionIndices`.

To produce a skeleton with named partitions, the user can directly add new named partitions to the `hkaSkeleton::m_partitions` or use the utilities in the `CreateSkeletonFilter`. To produce partitioned animations, the user can either use the runtime utilities found in `Animation/Animation/Animation/Util/hkaPartitionedAnimationUtility.h` or use the `CreatePartitionedAnimationFilter`. Partitioned animations are created from full animations by taking the transforms from only the specified partitions. For example, a skeleton might be partitioned to have bone subsets for the legs, torso (excluding the arms), left arm, and right arm. A full animation of a left punch can then be created so that it only includes the transforms for the left arm and torso (excluding all other transforms).

#### 3.3.10.1 Sampling And Blending with Partitions

There are no special steps that need to be taken in order to sample with partitions. All sampling methods referenced in the Sampling and Blending of Bones and Float Slots still apply to the sampling of animations using partitions. However, there are some special considerations to take into account that might allow for optimal use when using the Havok Animation SDK:

- Calling the `hkaAnimation::sampleTracks()` function, the `hkaAnimation::samplePartialTracks()` function or the sample-only mode of the `hkaSampleBlendJob` will result in a pose that only contains the transforms for the partitions with which the animation was built (`hkaAnimationBinding::m_partitionIndices`). The transforms will be packed tightly into the pose buffer (even if the partitions are separated in the skeleton). Pose buffers should be allocated with this in mind. Poses can be mapped back to a full pose using the mapping functions in the `hkaPartitionedAnimationUtility`.
- Calling the `hkaAnimatedSkeleton::sampleAndCombineAnimations()` or `hkaAnimatedSkeleton::sampleAndCombinePartialAnimations()` functions, the `hkaSampleAndBlendJob`, or the `hkaSampleAndCombineJobs` will return a full pose for the skeleton or the *maxBones* for the partial animations case. The reference pose will be filled in for bones not included in the sampled partitions. Pose buffers should be allocated with this in mind.

#### 3.3.10.2 Mirroring with Partitions

Mirroring across partitions is available and works with the methods described in the Mirroring Animations section. An additional step of setting up a partition mirror map must be done to properly mirror the partitions. The map can be setup by directly initializing the `hkaMirroredSkeleton::m_partitionPairMap` or a map can be created automatically from the partition names using the `hkaMirroredSkeleton::computePartitionPairingFromNames()` function. If partitions are present in the `hkaMirroredSkeleton` and in the `hkaMirroredAnimation`, mapping will happen automatically.

**Note:**
Bones within a single partition must only be mapped to bones within the corresponding paired partition. It is not possible to have one partition mapped to bones in multiple partitions.

**Note:**
If a partition map is not initialized and the `hkaMirroredAnimation` contains partitions, an invalid output pose is possible.

#### 3.3.10.3 Retargeting with Partitions

Retargeting can be executed on a per-partition basis and works with the methods described in the Retargeting Animations section. Much like the mirroring of animations, the retargeting of animations using partitions requires the additional step of initializing a map between the partitions in Skeleton A and the partitions in Skeleton B. This map can be set up using the `hkaSkeletonMapperUtils::createMapping()` function and specifying the `hkaSkeletonMapperUtils::Params::m_autodetectPartitionMap` to be true or using the `CreateMappingFilter`.

In order to map partitions from an animation, the partitions in the pose to be mapped must be provided to the `hkaSkeletonMapper::mapPoseLocalSpace()` function or the `hkaAnimationMappingJob`. The user has a choice in way they want the result to be returned: either filled into the correct places of a full pose, or as a tightly packed buffer containing only the transforms of the partitions. The mapped transforms will be placed in the resultant pose buffer such that they match the order of the partitions of the skeleton, not so they match the order of the partitions being mapped.

**Note:**
Bones within a single partition must only be mapped to bones within the corresponding paired partition. For instance, it is not possible to have a chain mapping that crosses multiple partitions.

### 3.3.11 Inverse Kinematics

Havok Animation provides different inverse kinematics algorithms for run-time. By using inverse kinematics, you can modify a pose according to external information based on the environment. Examples of usage are:

- Make the character look at particular objects in the scene.
- Place the feet of the character firmly on an uneven ground surface.
- Reach for a doorknob.

Some of the algorithms provided by Havok Animation are specialized for some of these situations (for example, foot placement or look-at controllers), while more generic solvers (like the CCD solver) can be used for arbitrary situations.

The IK solvers provided by Havok Animation are:

- Cyclical Coordinate Descent (CCD) Solver: Iterative solver that operates with chains of bones of any length.
- Two Joints Solver: Analytical solver that operates on two joints: a ball-socket joint and a limited hinge joint (useful for limbs like arms and legs).
- Three Joints Solver: Iterative solver that operates on three joints: a limited ball-socket joint (2 rotation axis) and two limited hinge joints (useful for objects like robotic arms or spider legs).
- Look At Solver: Analytical solver that operates on a single, limited, ball-socket joint.
- Foot Placement Solver: A high-level IK solver that tracks changes in ground height and incline and modifies the extension of a leg and the orientation of a foot accordingly.

#### 3.3.11.1 The CCD IK Solver: hkaCcdIkSolver

The CCD solver method employs an iterative IK process that improves the accuracy of the bone chain configuration with each pass. By repeating the CCD procedure on the bone chain for a given number of iterations the chain will either reach its target position or be as close as possible. There is a direct correlation between the number of iterations performed and the accuracy of the solver.

**Note:**

The CCD solver has been implemented under the assumption that it will be used for simple animation fix ups and as such is not suitable for complex, highly constrained limb movement. The CCD solver satisfies a positional constraint only and does not attempt to achieve a given end effector orientation.
Although this produces agreeable results most of the time, it is worth noting that unexpected bone chain configurations may occasionally occur. This is due to the fact that the solver gives complete freedom to the bone rotations and thus they may move in an unintuitive manner. An arm, for example, may rotate in a physically impossible fashion. The simple CCD solver does not currently support joint constraints to prevent these undesired configurations and some care may need to be taken to ensure they are not produced.

The CCD solver takes two parameters for construction, namely the number of *iterations* to perform and the *gain* to use during solving. The number of iterations used directly relates to the quality of the final configuration (although there will be a point whereby further iterations do not visibly improve the result). The gain influences how quickly the solver converges on a solution and effectively dampens the movement of the bones to try reduce oscillations about the desired target. It may be worthwhile to modify the default values for these parameters to try find a sweet spot between CPU time and accuracy required for your needs.

At run time, the CCD solver modifies a pose (wrapped with an `hkaPose` object) based on a series of constraints:

```
virtual hkBool solve ( const hkArray<IkConstraint>& constraints, hkaPose& poseInOut );
```

The constraint array contains a set of `IkConstraint` structures. This structure is shown below:

```
struct IkConstraint
{
    /// The start bone in this chain
    hkInt16 m_startBone;

    /// The end bone in this chain
    hkInt16 m_endBone;

    /// The target position for the end bone, in model space
    hkVector4 m_targetMS;
};
```

A constraint is simply a set or chain of bones to be manipulated (defined by the `m_startBone` and `m_endBone`) with a target or desired position for the `m_endBone` (given by the `m_targetMS` member). The solver will take each constraint in the array in turn and determine new transforms for each bone in the chain.

#### 3.3.11.2 The Two Joints IK Solver - hkaTwoJointsIkSolver

The Two Joints solver (`hkaTwoJointsIkSolver`) is an optimized solver for IK configurations where the only joints involved (the only joints modified for the IK) are:

1. A ball-and-socket (spherical) joint
2. A hinge joint, with a range of allowed angles around the hinge axis of [a,b], where a>=0 and b<=180 degrees.

We find this configuration in many situations, in particular limbs (arms and legs). For example, in an arm, the spherical joint would be the shoulder, and the hinge joint would be the elbow.

As well as the pose to be modified, the solver takes as its input:

- The indices of the bones for the first (spherical) and second (hinge) joint
- The index of the end effector bone
- The hinge axis of the second joint, specified so positive rotations (using the right hand rule) extend the joint.
- Limit angles (specified with cosine) of the hinge joint (defaults are 0 deg [1.0] and 180 deg [-1.0])
- Gain values for modifications applied to both joints. The modifications calculated by the IK solver will be scaled by these values (a value of 0.0 won't modify the pose, a value of 1.0 will apply the IK fully). You can use these gains in order to "ease in" and "ease out" the IK.
- The target position, in model space, of the end effector bone

The figure below shows the Two Joints IK solver in action:

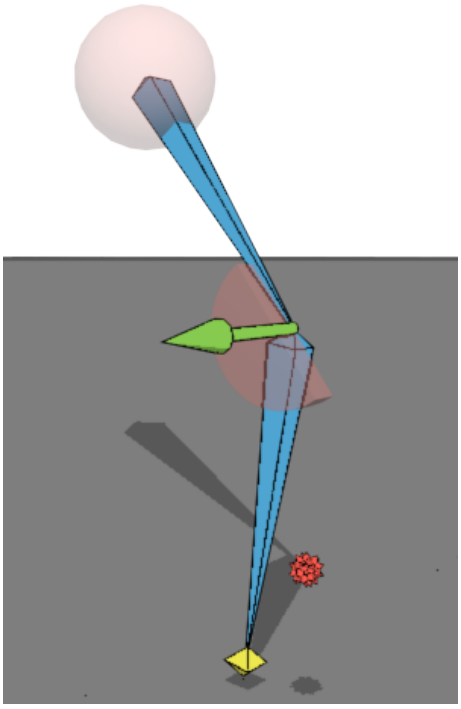
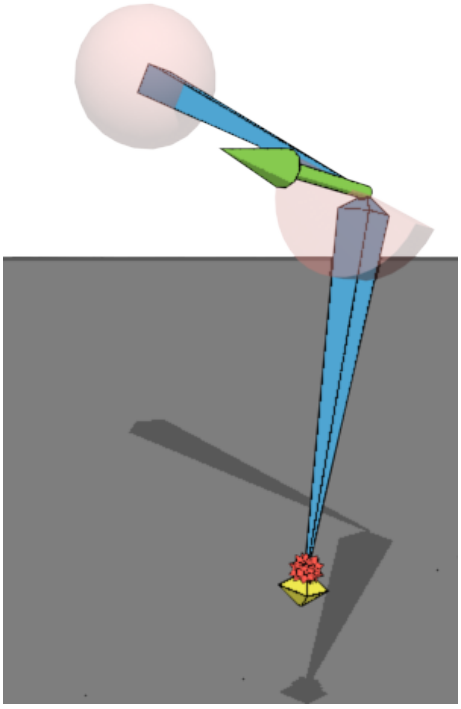
| | |
|---|---|
|  |  |
| The input pose | The output pose |

Table 3.6: The Two Joint IK Solver in Action

The solver will always apply the minimal modification in order to reach the desired target. That is, from all possible solutions to the IK problem, the closest solution to the input is chosen. Thus, the original pose passed to the solver influences the output pose, as shown below.

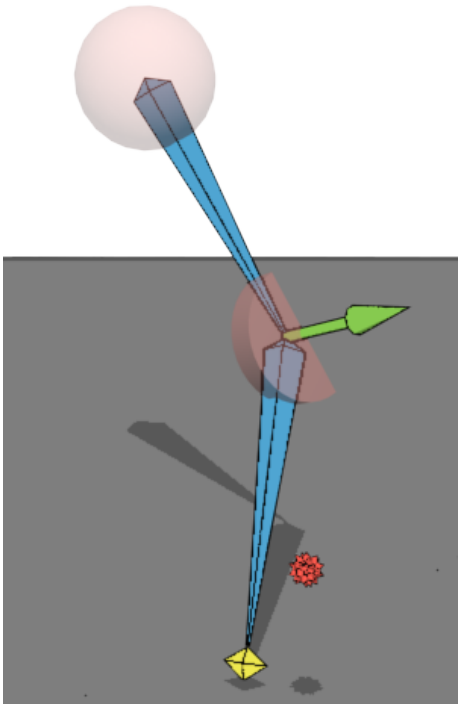
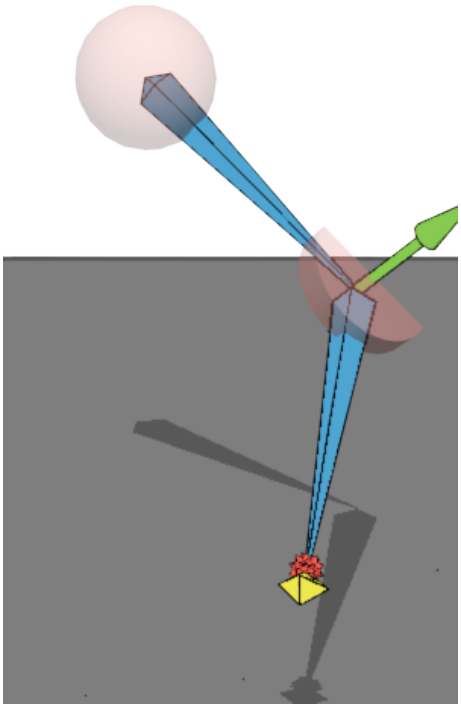
| | |
|---|---|
|  |  |
| The input pose | The output pose |

Table 3.7: Different input poses will yield different (but still valid) results

Notice that the first (spherical) and second (hinge) joint do not necessary need to be immediate parent and child in the bone hierarchy; that is, as long as the second joint is deeper in the hierarchy than the first joint (and the target joint is deeper in the hierarchy than the second joint), the IK solver will operate correctly. Any joints in between the first, second and target joint will be kept rigid (their local transform won't change), as illustrated below:

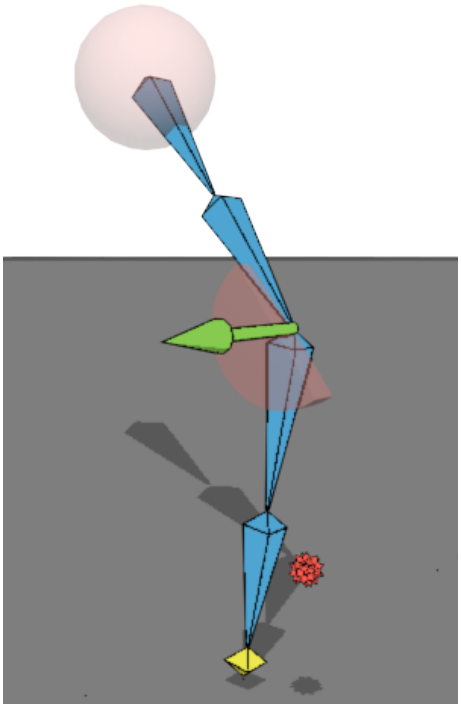
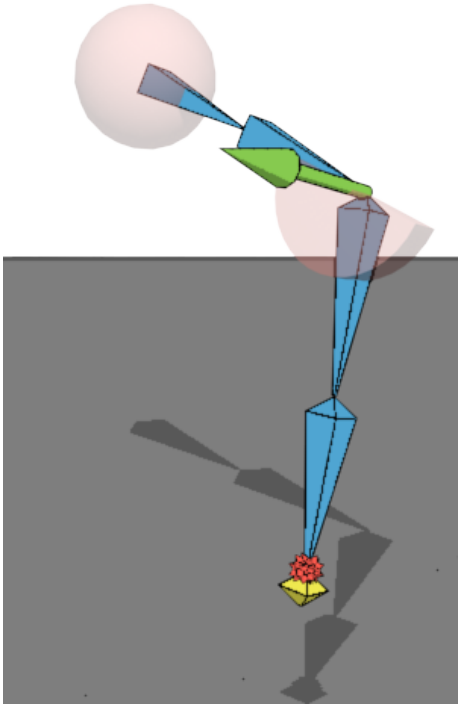
| | |
|---|---|
|  |  |
| The input pose | The output pose |

Table 3.8: The Two Joint IK solver can operate on chains of any length, but it will only modify two joints

#### 3.3.11.3 The Three Joints IK Solver: hkaThreeJointsIkSolver

The Three Joints IK Solver (`hkaThreeJointsIkSolver`) is a fast iterative solver for IK configurations where the joints involved (the only joints modified for the IK) are:

1. A first hinge joint with two rotation axes. The first axis defines rotation of the whole bones plane and the second axis is perpendicular to this plane.
2. A second hinge joint with a rotation axis perpendicular to the plane.
3. A third hinge joint with a rotation axis perpendicular to the plane.

We can find this configuration in many situations, in particular unnatural limbs (insects or creatures legs, robotic manipulators, space robotic arms). For example, in a spider leg, the first joint would be the joint between the body and leg, the two hinge joints would be the elbows and the end effector (target point) represents contact point with the ground. A major restriction of the solver is the fact, that all joints and end effector must lay in a plane.

As well as the pose to be modified, the solver takes as its input:

- The indices of the bones for the first (2 axes hinge), second (hinge) and third (hinge) joints

- The index of the end effector bone
- The hinge axis of the first joint, around which the whole plane is allowed to rotate. This axis must lay in the bones plane.
- The target position, in model space, of the end effector bone

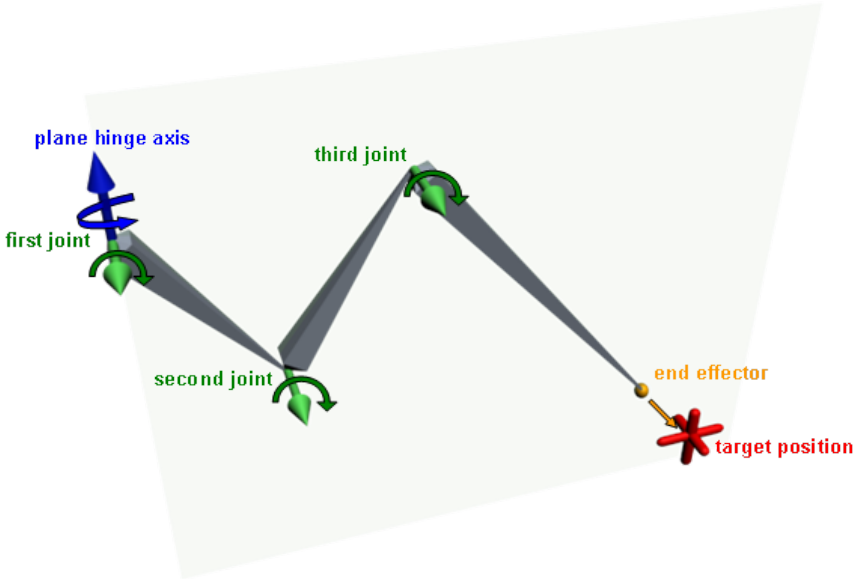



Figure 3.30: The solver configuration in 3D

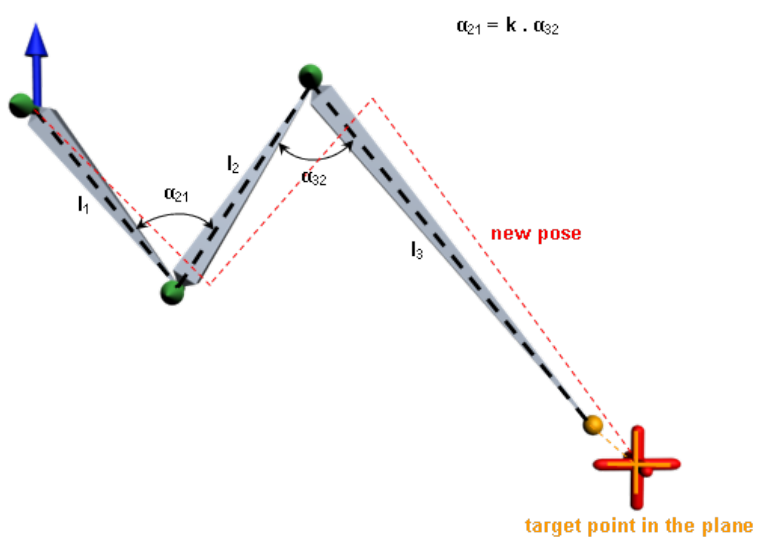



Figure 3.31: The 2D solution in the plane

The solver determines a new pose with the end effector at the target position in the three steps:

1. The solver rotates the whole bones plane around the first joint hinge axis (blue) to the target point position.
2. The solver calculates three new rotations in each joint around hinge axes (green) perpendicular to the bones plane to reach a desired position in the plane (red configuration).
3. The final output pose is updated.

In general, the mechanism in this configuration is kinematically under-estimated and the additional condition has to be defined. Our solution utilizes the restriction that the angles between the first and second bones (Îś21) and second and third bones (Îś32) are linearly dependent (Îś21 = k Îś32). The constant k is calculated from the initial pose in the constructor. The core computation is performed by iteratively solving in the plane.

The target position has to occupy the operational space of the configuration. The operational space is defined as a sphere with the center in the first joint and with inner radius (collapsed configuration) and with outer radius (maximal reachable configuration). In a situation where the target point is out of the operational space (too close or too far from the first joint), its projection to the sphere is automatically calculated to ensure smoothness of the iterative solution.

The solver will always apply the minimal modification in order to reach the desired target. That is, from all solutions satisfying additional condition to this IK problem, the closest solution to the input is chosen.

Considering the nature of this kinematic problem and its solution, the restrictions of the solver are the following:

- All joints and the end effector must lie in a plane
- The angles between the first and second bones and the second and third bones are linearly dependent.
- The target position, in the model space, has to be changed continuously. It is recommended to apply only small steps between the previous and current target position.
- The target position, in model space, has to occupy the solver operational space.

#### 3.3.11.4 The Look At IK Solver: hkaLookAtIkSolver

The Look At solver (`hkaLookAtIkSolver`) is a very simple IK solver that modifies the rotation of a single joint/bone so the bone points towards a particular point in model space. The output is limited by a cone of movement. Alongside the transform of the bone, the following information is passed to the Look At IK Solver

- A "forward" axis, defined in the local space of the bone. This axis is the axis that will be aligned towards the target point.
- A "eyes" position vector in the local space of the bone (head). Usually it defines a point between eyes. The default value is (0,0,0).
- A limit axis, defined in model space. This axis is the center axis of the cone that limits the movement of the bone.
- A limit angle, which alongside the "cone" axis defines the cone of allowed movement for the bone

- A gain value. The rotation calculated by the solver will be scaled down by this value (so a value of 0.0 won't modify the bone, while a value of 1.0 will apply the IK fully). You can use this value in order to ease in and ease out the Look At IK.
- A target position, defined in model space. The IK will try to align the target vector to the direction defined by the vector between the "eyes" and target positions.

The following illustrations show the Look At IK Solver in operation, both when the target direction is inside the limiting cone movement and when it's outside.

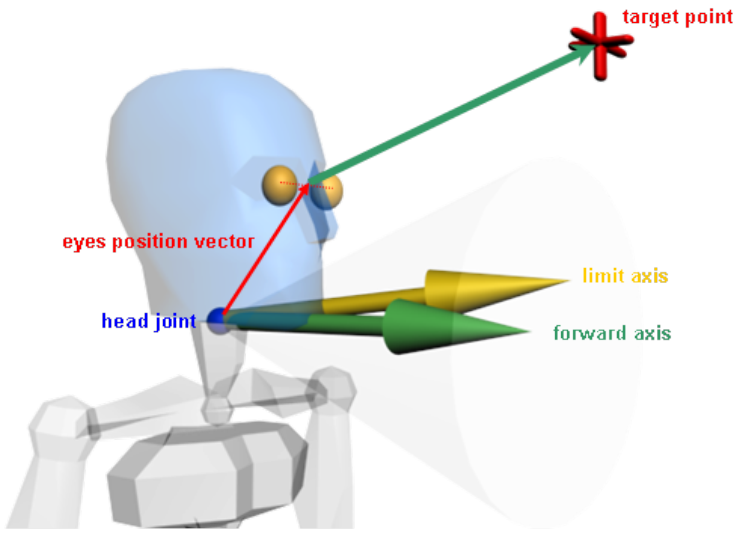



Figure 3.32: The input to the Look At IK Solver (target is inside the limiting cone)

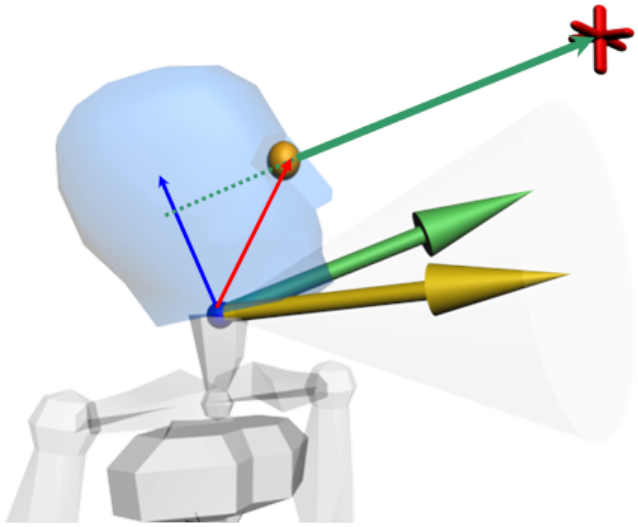


Figure 3.33: The output of the solver (target is inside the limiting cone)

Notice how in this second case the "forward" axis (in green) is kept inside the limiting cone (in yellow):

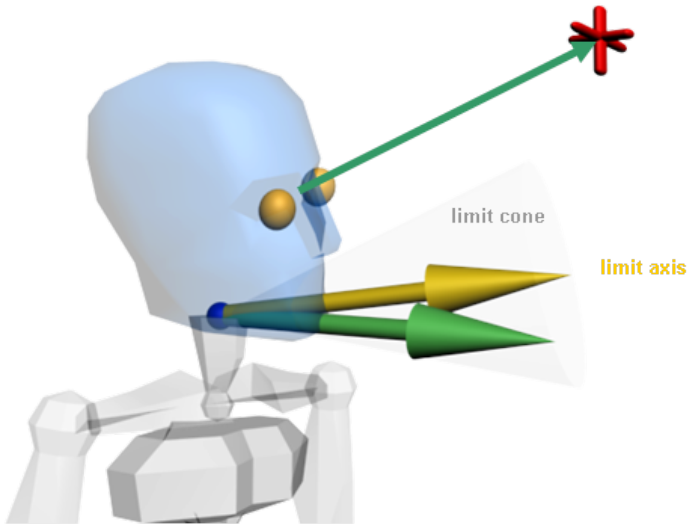



Figure 3.34: The input to the Look At IK Solver (target is outside the limiting cone)

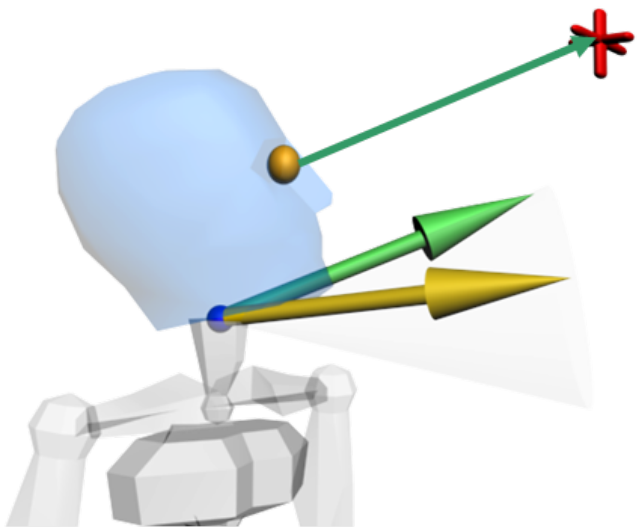


Figure 3.35: The output of the solver (target is outside the limiting cone)

### Applying Look At to Hierarchies of Bones

The Look At IK Solver differs from other solvers since it operates on a single bone transform rather than a full pose. If your character has any children bones attached to the bone used for look at, you may want to modify them accordingly. For example, if you are modifying the head with the Look At solver, you probably want any eyes, antennae or ponytails to move rigidly with the head bone. You can use an `hkaPose` object to do so, as shown below:

```
// "pose" is an hkaPose object that contains the current pose of our character
hkaLookAtIkSolver::Setup setup;
setup.m_fwdLS.set(0,1,0);
setup.m_eyePositionLS(0.12f,0.1f,0.0f);
setup.m_limitAxisMS.setRotatedDir(pose.getBoneModelSpace(m_neckIndex).getRotation(), hkVector4(0,1,0));
setup.m_limitAngle = HK_REAL_PI / 5.0f;
// By using the PROPAGATE flag, all children of the head bone will be modified alongside the head
// (their local transforms will remain constant)
hkaLookAtIkSolver::solve (setup, m_targetPosition, 1.0f, pose.accessBoneModelSpace(m_headIndex, hkaPose::
  PROPAGATE));
```

### Applying Individual Left/Right and Up/Down Limits

In some use cases it is desireable to specify separate limits for the left/right and up/down motion of a bone. For example a character's neck might have a +/- 75 degree range of motion in the side to side direction, be able to look up 45 degrees, and look down only 30 degrees. An illustration of this is shown in the figure below; here the forward vector is constrained to lie within a box inside of the limiting cone. To align the box, an up direction must also be specified. This may be done by specifying an additional `hkaLookAtIkSolver::RangeLimits` data structure pointer to the `hkaLookAtIkSolver::solve` function. These individual limits will override the single limit specified in the `hkaLookAtIkSolver::setup` data structure.

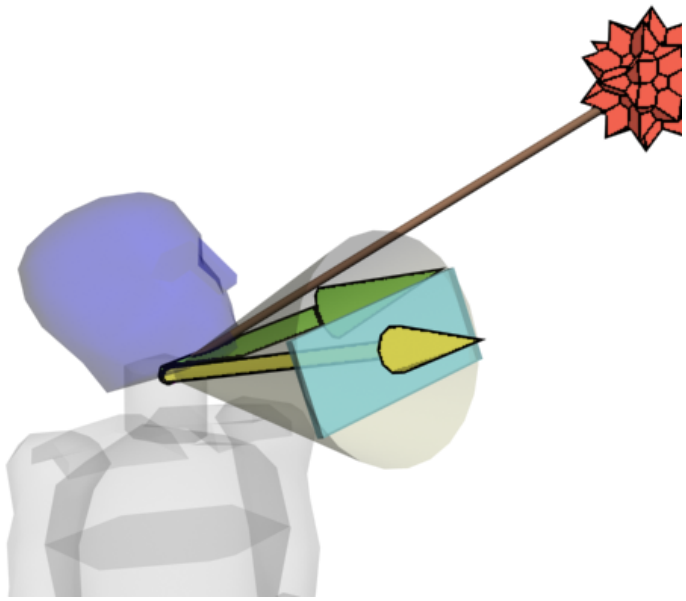


Figure 3.36: Individual Angle Limits in Look At IK

**Applying Look At to Multiple Bones**

The Look At Solver may also be applied to multiple bones withing a hierarchy. For example Look At may be applied both to the head (neck) bone as well as to the bones which control the eyes. For a typical humanoid the neck has a more limited range of motion than the eyes. When looking at an object 90 degrees to the right, the neck may be limited to only rotate 45 degrees and the eyes could rotate an additional 45 degrees to stare directly at the goal. This effect may be accomplished by solving for parent bones before solving again for child bones. An example of this feature is demonstrated in the `LookAtDemo.cpp` file.

#### 3.3.11.5 The Foot Placement IK Solver: hkaFootPlacementIkSolver

The Foot Placement solver is a specialized solver that can be used to modify the extension of legs and orientation of feet according to changes in ground height and slope. This solver is a little more sophisticated than the rest of the IK solvers, particularly because it keeps a state between calls and because it actually interacts with the environment (through a raycast interface) in order to operate.

The solver operates on a single leg, and therefore you will need to instantiate one solver for each leg. The solver operates with the following data:

- Setup Data: Passed on construction, it defines the structure of the character: skeleton, joints involved, limits and axis.
- Input Data: Passed to the solver on each step, it remains unmodified. It contains information about gains, the location of the character, the original transform (position and orientation) of the foot (the one the solver will base its results on) and a raycast interface for the solver to use.
- Pose: The solver modifies an instance of `hkaPose`.
- Output Data: Apart from modifying the pose, the solver returns some other information, like the vertical error detected, that can be used by the application (for example, to update the pelvis of

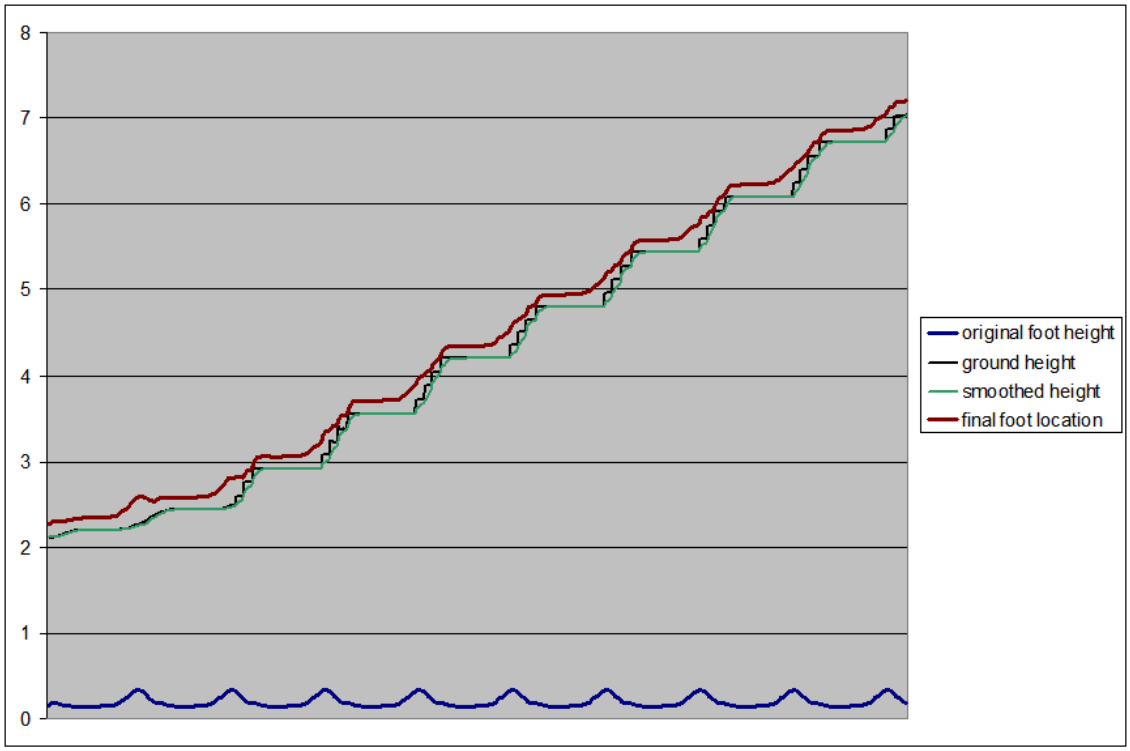
the character).

### Overview

In simplified terms, the solver works in the following manner:

1. Using raycast(s), it cast the current position of the foot towards the ground, from where it gets a height and an incline
2. From the original transform of the foot, it calculates the height above ground the foot is supposed to be on (if the surface was flat).
3. Based on that and other gain values, it smooths out sudden changes in the height detected by the raycast.
4. Based on this smoothed height of the ground, and the original foot height above the ground, it places the foot at the right location (using a Two-Joints IK solver)
5. Similarly, based on the incline of the ground, and the original orientation of the foot, it reorients the foot accordingly.

The following graph, extracted from an actual application using the Foot Placement solver, should illustrate the above.




The graph was taken from a character walking a set of stairs. In blue, at the bottom, you can see the original height (in model space) of the foot as driven by the animation system. In black, you can see the height (in world space) detected by raycasting the foot against the landscape. In green, you can see this

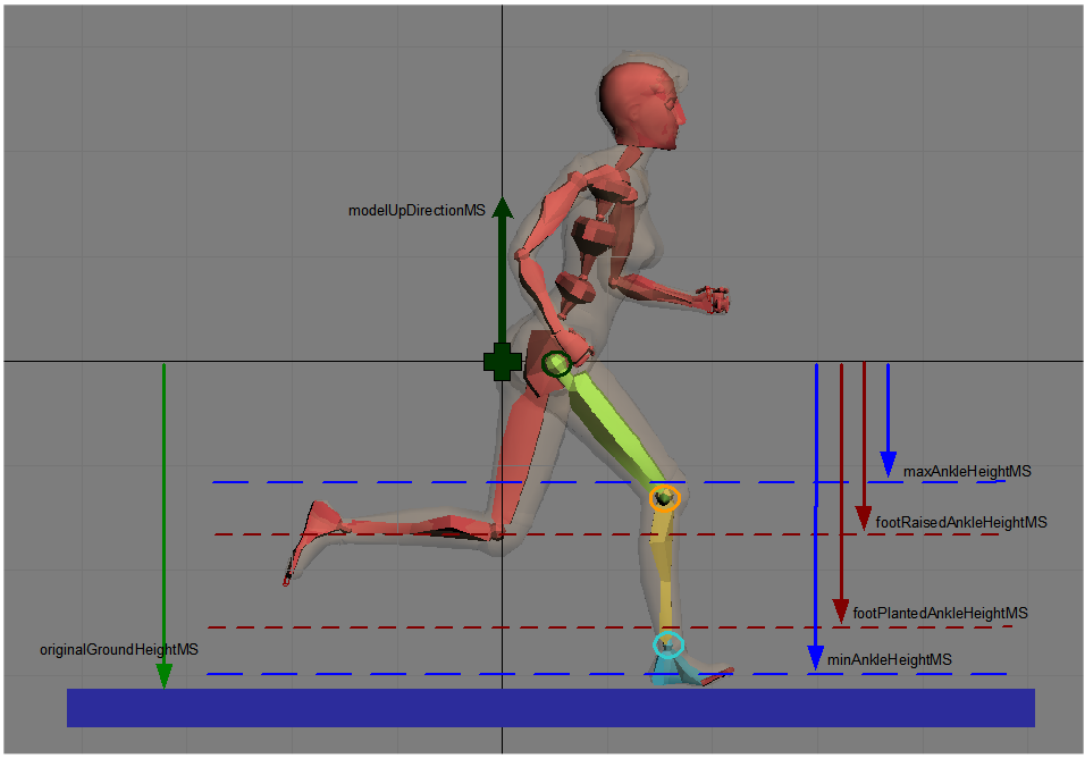
height smoothed out; notice how it matches the ground height at the times where the foot is planted. Finally, in brown, you can see the resulting height of the foot, which is based on both the original foot height above the ground, and the smoothed ground height.

### Setup Data

The Foot Placement solver takes some setup data on construction. This setup data contains information that usually doesn't change at run time, like limits and axis and bone configurations. This information is:

- The skeleton of the character to which we are going to apply this solver
- The indices of the three joints involved: hip, knee and ankle (in the diagram below, circled in green, orange and blue)
- The rotation axis of the hinge in the knee joint, specified in local space of the knee (in the diagram below, the green axis pointing outwards from the knee.
- The location of the end of the foot, specified in the local space of the ankle. This is used for raycast, and is described in more detail later on (it's the light green circle in the second diagram below). If this location is (0,0,0), the length of the foot will be ignored (a single ray cast will be used).

(Notice that the parameters above, with the exception of the "end of foot" parameter, are the same used by the Two-Joints IK solver.)

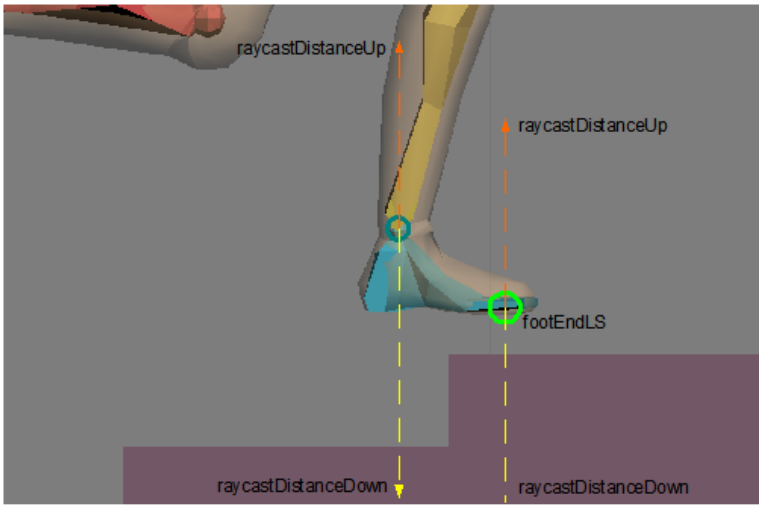



- The UP direction of the world (the application), specified in world space. This is the direction of UP in the game environment (pointing from the ground to the sky). It is used for the raycasts.

- The UP direction of the model, specified in model space (and shown in green in the diagram above). This is the direction of UP in the space where the model (character) was defined. It may not match the UP direction of the application: for example, a character may be setup with Z as the up axis (maybe setup in 3ds Max), while it may walk around an environment where Y is up (maybe created in Maya).
- The original ground height in model space. This is the height of the (imaginary) ground used to author the model, i.e, the vertical distance between the origin (0,0,0) in model space (a.k.a. the reference frame) and this imaginary ground.

  **Note:**
  The location of this origin (and hence, the value of this parameter) will be different depending on the way your character is setup (root in the ground, root at the pelvis) and how motion is extracted. In the example above, this distance (and the ankle heights described later on) would be a negative value since the ground is located below the origin in our model space (shown as a green cross). If your character and animation place the model space origin (the reference frame) at the ground, then this distance is 0 (and the ankle heights will be positive).

- The ankle height (specified from the model space origin) below which we consider the foot is planted. This is used in order to calculate gains used for smoothing the foot placement (as described in the overview)
- Similarly, the ankle height above which we consider the foot is raised.
- A maximum and minimum height that the ankle cannot go above/below. The foot placement will ensure that the height of the ankle never exceeds this range.
- Maximum and minimum angles allowed for the knee (specified with their cosine). These are the same parameters used in the Two-Joints IK solver, and can be used to avoid knee popping artifacts.




- Two distances, `raycastDistanceUp` and `raycastDistanceDown`, that define the start and end of the rays that will be cast against the environment in order to place the foot. These distances are both positive and will be applied to both the start of the foot (the location of the ankle) and the end of the foot (as specified by `footEndLS`).

  **Note:**
  The reason two raycasts can be used instead of one is in order to take the length of the foot into account when doing foot placement over discontinuous terrain (like stairs), as in the example above. Casting a single ray would leave the foot going through the steps. But by using a second ray near the end of the foot the foot placement solver will detect this and the foot will then be placed on the step above.
  If the end of the foot is not specified (i.e., if `footEndLS` is left as 0,0,0), then a single raycast will be used.

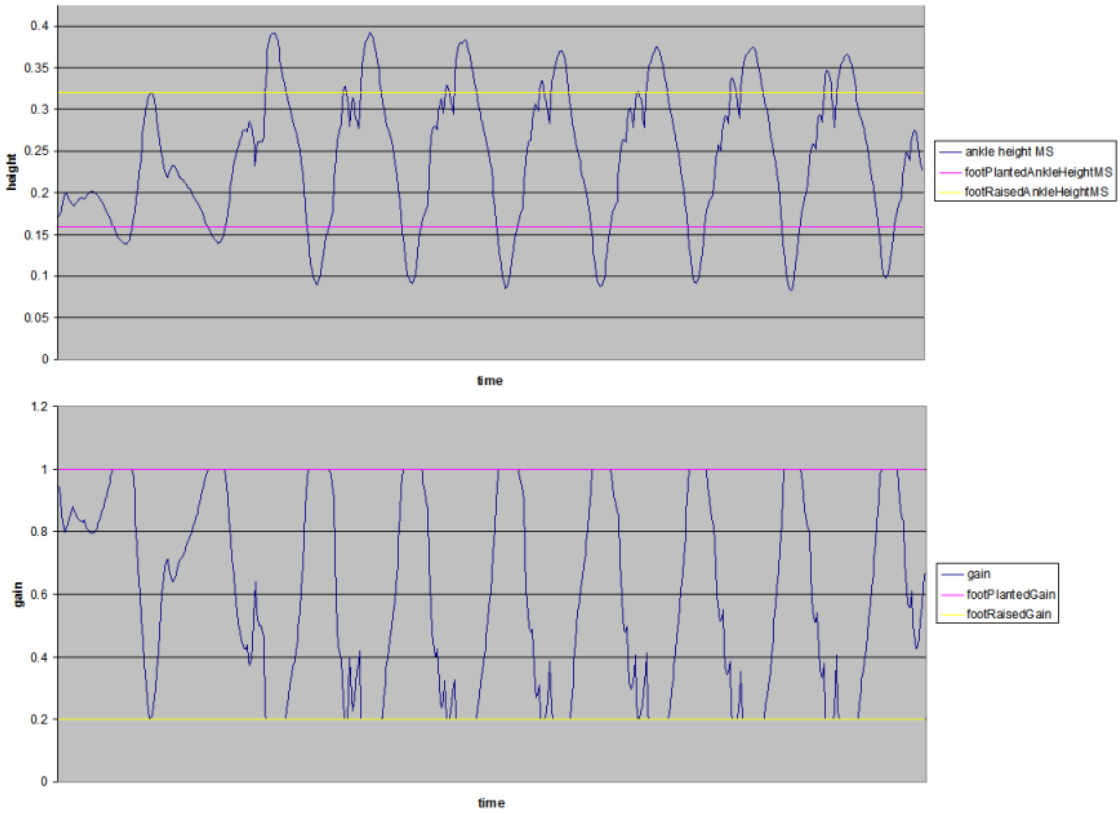
**Input Data**

Alongside the pose to modify (passed as an `hkaPose` object, `poseInOut`), the following information is passed every time you want the Foot Placement solver to operate:

- The original transform of the ankle (in model space). The foot placement will base its results on this transform, rather than the transform of the ankle in the pose (`poseInOut`) passed to the solver. This is because, in some cases, you may want to do the fixing up based on an "original pose" different than the actual pose passed as a parameter. In the most common case, though, this transform will be the same as the model space transform of the ankle in `poseInOut`.
- The `worldFromModel` (model to world) transform. This transform describes the position and orientation of the character (the model) in the world and it's usually the same transform used throughout the application (for skinning, game logic, rag doll, etc).
- A flag (`footPlacementOn`) indicating whether we want the foot placement to switch itself on or off. The foot placement solver will ease itself in or ease itself out smoothly, and will keep itself on or off, based on this parameter.
- A raycast interface, defined by the abstract class `hkaRaycastInterface`. The implementation should be able to cast rays of limited length into the environment.
- An on/off gain, used by the solver to transition smoothly when switched on or off (based on changed in the `footPlacementOn` flag)

- The gain used to smooth positive changes in ground height (i.e. when the ground is ascending)
- The gain used to smooth negative changes in ground height (i.e. when the ground is descending). The reason for having two different gains is that you may want to have higher gains when going up since penetration artifacts are more likely to happen when going up than when going down.

The final gain used to smooth changes in ground height is a combination (a multiplication) of the above gains (either the ascending or descending, depending on the landscape) and another gain value calculated based on how "planted" the foot is. This value is calculated based on the desired ankle height (as defined by the original transform of the ankle), the `footPlanted` and `footRaised` ankle heights (passed at setup time) and two other gain values, `footPlantedGain` and `footRaisedGain`, passed as part of the input structure.




As the graphics above show, this gain will be calculated as follows:

- For heights above the `footRaisedAnkleHeightMS` setup value, the `footRaisedGain` will be used.
- For heights below the `footPlantedAnkleHeightMS` setup value, the `footPlantedGain` will be used.
- For values in between, an interpolation between `footRaisedGain` and `footPlantedGain` will be used.

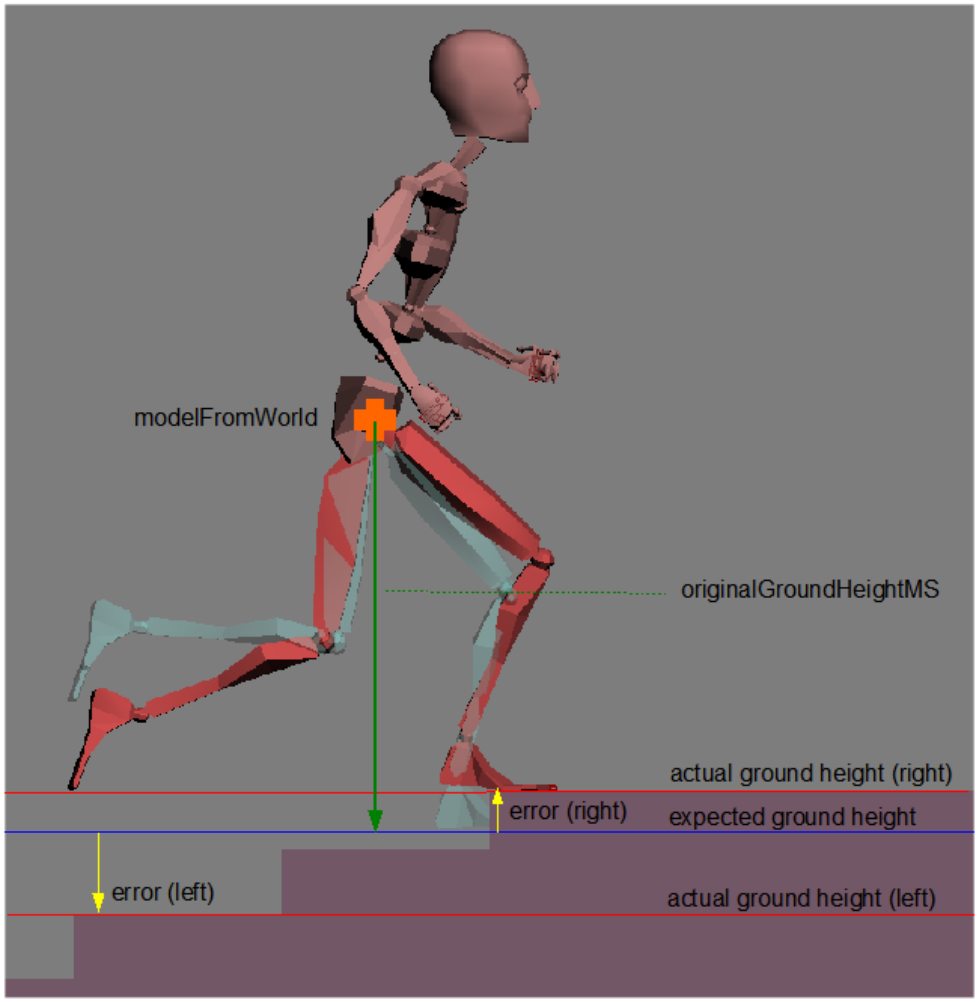
**Note:**

Again, remember the final gain used to smooth landscape changes is the product of this gain and the ascending/descending gain.

### Output Data

The Foot Placement solver also returns some extra information that can be used by the application:

- A *vertical error*. This is the difference between the actual height of the ground as detected by the raycast and the height of the ground as expected by the animation (calculated by using the `worldFromModel` transform and the setup value `originalGroundHeightMS`). This is illustrated below




The bones in blue represent the pose before the foot placement is applied (twice, once for each leg), and the bones in red represent the pose after the foot placement is applied. You can see the vertical error returned for each leg in yellow. Notice that, in the example, the right leg is returning a positive error (since the ground is higher than expected) while the left leg is returning a negative error (the ground is lower than expected).

You can use these error values in order to update the vertical position of your character over time, so it reacts to the environment according to changes in ground height (i.e, the error gets reduced).

**Tip:**

When doing so, you may want to favor negative errors (ground lower than expected) over positive errors. This is because negative errors will try to extend the leg (to reach down), while positive errors will try to contract the leg. It is easier to correct negative errors by updating the position of the character (i.e, lowering the pelvis) than by using IK to extend the leg, since in most poses the leg is already extended and thus the available range is quite limited (while the range to contract the leg is far larger).

- A Boolean, `hitSomething`. This will be set to true if the foot placement was able to detect ground below the feet (through ray casting). Otherwise, false is set. This can used by application logic: for example, to detect when the character is not supported by its feet. Note that when detecting the ground the variable also checks that the ground is actually close enough to touch the foot (the raycasts can extend beyond the foot). The distance check is controlled by the `minAnkleHeightMS` setup variable.

### 3.3.12 Deformation (Skinning And Morphing)

The current deformers fall into two categories: skinning deformers (`hkaSkinningDeformer`)and morphing deformers (`hkaMorphingDeformer`). The deformers are then split into two sub groups: one is where you are using a SIMD (SSE on PC, etc.) enabled `hkMath` library and one where you are not (a plain FPU version). The SIMD versions require that *at least* your input data members (position, normals) are aligned to 16 bytes (16 bytes == 4 floats == 1 simd vector) and works faster if your output data members are aligned as well. It autodetects the data alignment and uses an appropriate algorithm automatically. If you choose to use the SIMD deformers when you don't have SIMD enabled you may well discover that it is slower than just using the plain old FPU version due to needless copies etc.

**Important:**

It is important to note that the deformers are there to be used by Havok's demos but are not production quality. It is assumed that deforming will be done most commonly by your graphics engine, usually in hardware on GPUs or VUs. That hardware deformation is usually performed at the same time as per vertex lighting operations, so Havok cannot provide optimized deformers for all such game specific usage.

Havok animation provides both tool chain and runtime support for animated meshes. This breaks down into support for skinned meshes and support for attachments. We explicitly break these into distinct sections as they are often treated separately by different sections of the rendering pipeline. Both types of objects refer to our base mesh stored in the `hkxScene` description so we begin with a discussion of that.

#### 3.3.12.1 Meshes

Meshes exported through our export utilities are made from one or more mesh sections. Each mesh section is composed from a material description, a vertex buffer and an index buffer. The section represents some segmentation of the mesh and although the description itself allows complete flexibility, our export utilities typically divide based on material.

#### 3.3.12.2 Vertex Buffers

Mesh data is typically held in the vertices on a per vertex level. Our vertex description is completely flexible specifying only a raw data buffer and number of elements and a data description (a `hkClass`) of the layout of a buffer element.

We provide data descriptions for the common vertex formats we export. The names of these classes are generated from the component they contain. For example the class `hkxVertexP4N4C1T2` contains a position as 4 floating point values, a normal as 4 floats, a color value, and 2 texture coordinates. To create your own format simply write the header file for the class description, and include the `HK_DECLARE_REFLECTION` macro. Once the serialization scripts are run on the header it will produce a class description which can be used to construct a vertex buffer.

**Note:**

All positions, normals, binormals and tangents are exported in world space.

#### 3.3.12.3 Skinned Meshes

In our scene data all meshes are represented in local space (i.e. the positions, normals and tangents etc. are all in local space). When skinning, meshes are often represented or stored in world space, and the inverse of the binding pose is used to construct a matrix palette for skinning. Storing the skinned meshes in this format means we only need to store each vertex in the skin once in local space, while storing the inverse bind pose in world space saves us from having to store vertex data multiple times in the local space of each bone the vertex is bound to. Since meshes are exported in local space the transformation from local to world space is implicitly included in our inverse bind pose (given in `hkaMeshBinding`).

Skinned meshes are exported in 2 phases. In the first phase the export utility places the character in its bind pose and exports the mesh in local space. The exported mesh will contain weight and bone indices for each of the vertices. Today we export a maximum of 4 weights per vertex, and these weights are quantized and stored as bytes to reduce mesh size. These skinned meshes are usually accessed through the `hkxSkinBinding` array in the `hkxScene` class. Each skin binding contains:

- A pointer to the underlying mesh in bind pose space
- A list of the nodes it is bound to
- A copy of the original bind pose of the bones
- The skin world transform at the time of binding (since the skin mesh vertices are stored relative to this)

Usually this information is adequate to deform a skinned mesh using matrix palette skinning. However, often the user will redefine the rig in the filter pipeline and hence this description requires extra data to map to a new skeleton.

This extra data is stored in the `hkaMeshBinding` and this is the class we usually use at runtime. This contains a mapping from the bones in the original rig as seen and stored in the modeler and the rig as defined by the user in the filter pipeline. Including the `CreateSkin` filter in the filter stack creates one of these mesh bindings. A `hkaMeshBinding`'s `m_mappings` array can be null, indicating that the mapping is just a trivial one to one map (skeleton bone indices are used as is by the mesh indices). If the `m_mappings` array is of size one, then the mappings held in that structure apply to all mesh sections and are the mapping between the mesh indices to the skeleton indices (as the mesh uses less than the full set of skeleton bones for instance). If the `m_mappings` array is more than size one, then there is a mapping for each primitive (each index buffer, in each `hkxMeshSection`) in order. This happens when you break up a mesh based on limited bone influences per mesh primitive (e.g: 20 bone limit per `DrawPrimitive` call in DirectX when using a vertex shader that takes a palette of 20 bones). The `CreateSkin` filter can do this limitation and segmentation for you using the `hkxMeshSectionUtil` functions.

The `hkaMeshBinding` also combines the skin world transform (`hkxSkinBinding::m_initSkinTransform[[worldFromSkin]]`) and the binding pose transform array (`hkxSkinBinding::m_bindPose[[worldFromBone]]`) into a single transform array, `hkaMeshBinding::m_boneFromSkinMeshTransforms[[bonefromSkin]]`.

#### 3.3.12.4 Attachments

During asset filtering the `CreateSkin` filter adds attachments to the animation container. These attachments are found by examining the nodes that form the skeleton rig. If any of these nodes refer to scene objects a `hkaBoneAttachment` is created. Each attachment has:

- The bone index of the bone it is bound to
- The world transform of the attachment
- A description of what has been attached.

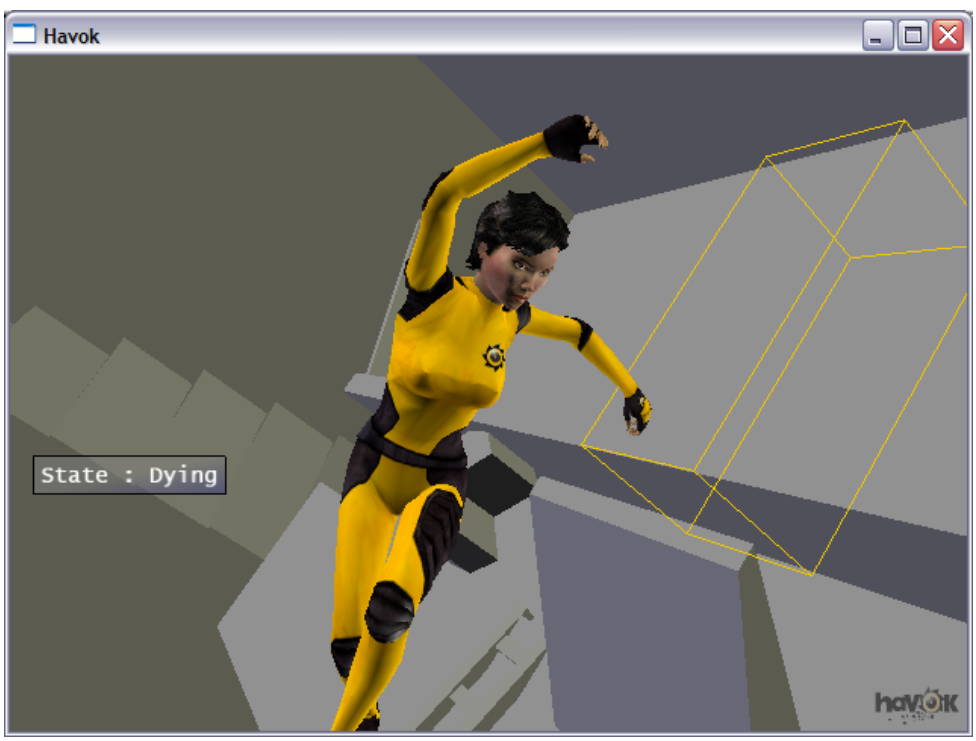
The description of what has been attached is given as a `hkVariant`. This allows anything to be attached but in practice we only attach lights, cameras and meshes.

**Note:**
Though our meshes are stored in local space it is common to use the world space position of a given bone to drive the model space matrix from the attached mesh. Using the `m_boneFromAttachment` transform is the most common way to do this.

## 3.4 Integrating Havok Animation with Havok Ragdolls

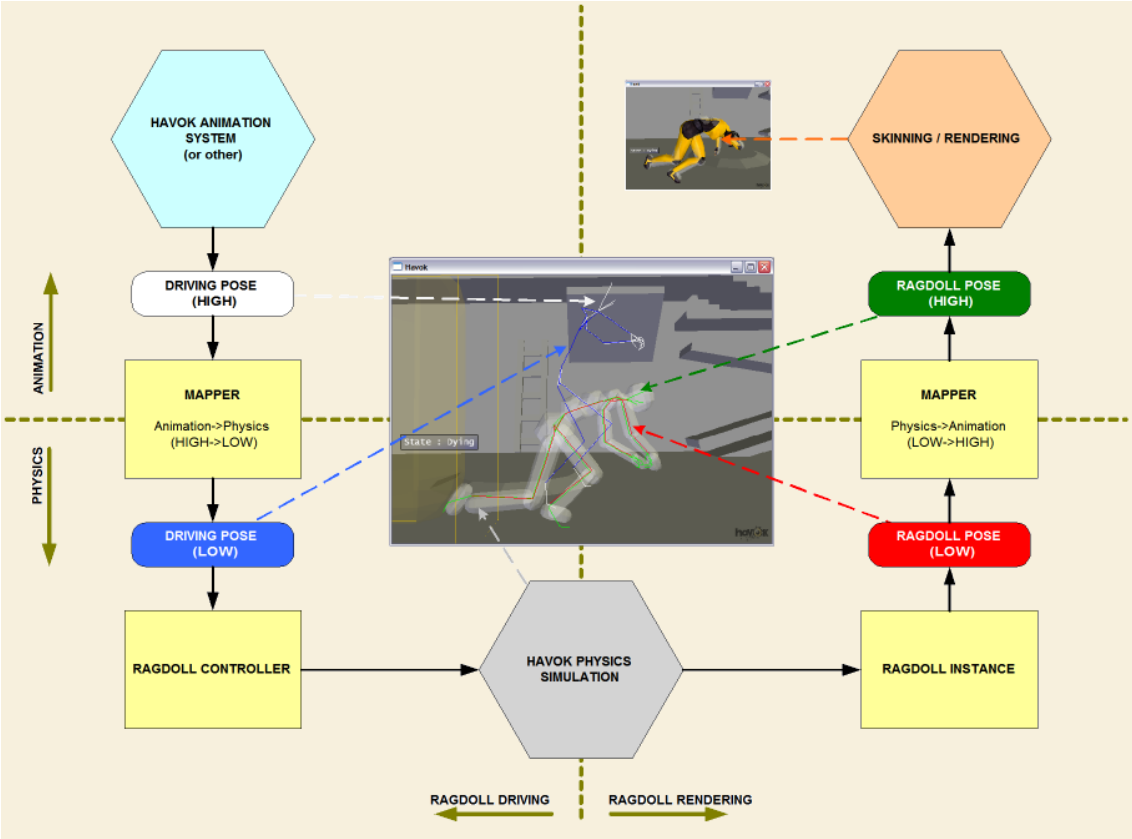
### 3.4.1 Introduction

This chapter describes the different tools provided by the Havok Animation and Havok Physics SDK in order to integrate physically-simulated characters (ragdolls) into an animation pipeline.




### 3.4.2 A Typical Scenario

The following diagram shows the typical flow of information in a system where ragdolls have been integrated with the animation system.




In the application above, the animation system is used to drive physical ragdolls, i.e., make the physical representation of the character achieve a desired pose. Also, the current physical pose of the physical ragdoll is then used to render the visual representation of the character (skin it).

Therefore, we have two quite distinct processes here:

- **Ragdoll Driving** (left side of the diagram) : Given a "desired" pose taken from the animation system, feed the physics simulation with some input that will drive the physical ragdoll towards that desired pose.
- **Ragdoll Rendering** (right side of the diagram) : Given the "current" pose taken from the physics simulation, feed the animation system with a pose that can then be rendered / skinned.

Notice that, for many reasons (mostly performance), the physical representation and the animation representation of a character are usually different. While an animation skeleton for a character may have 50 bones or more, we will usually represent this character with a ragdoll skeleton of less than 15 bones.

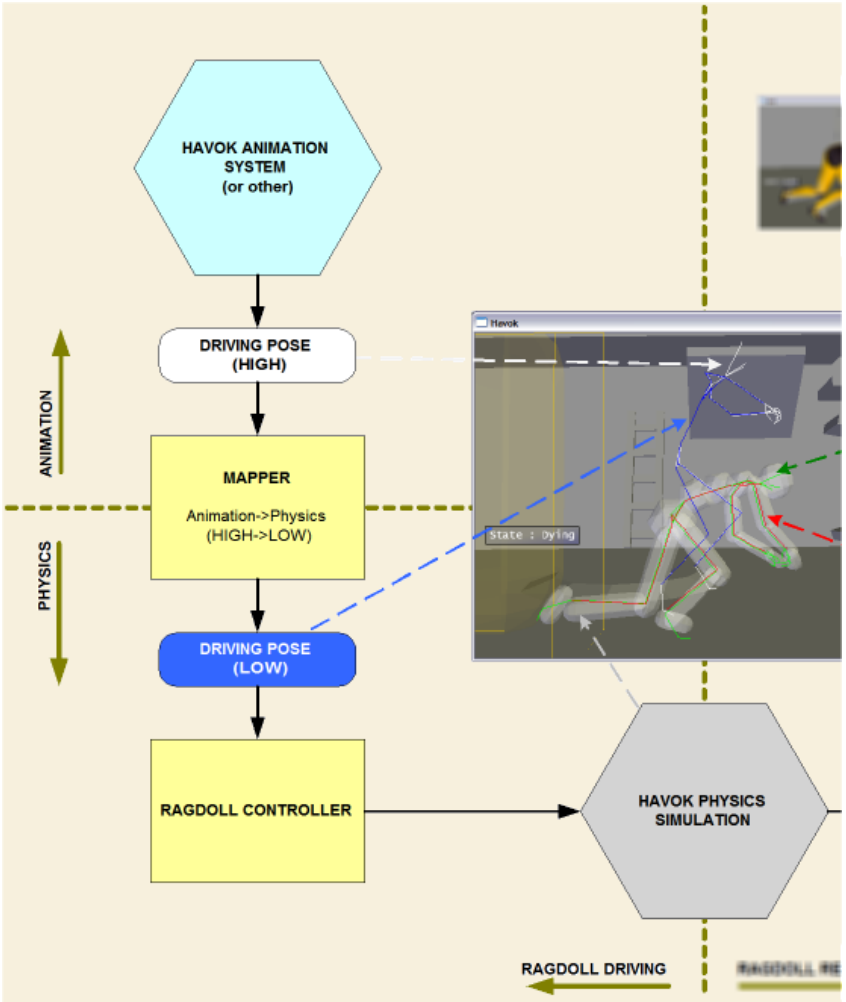
Therefore, we can distinguish as well between two different spaces, depending on the kind of poses we work with:

- **Animation Space** (top part of the diagram) : Poses have a higher-bone count
- **Physics Space** (bottom part of the diagram) : Poses have a lower bone count (a bone for each rigid body)

A lot of the work involved on ragdoll integration is associated with moving between these two spaces. Mapper ( **hkaSkeletonMapper** ) objects are used in order to do so.

Let's take a look at how the information flows both when driving and when rendering ragdolls.

#### 3.4.2.1 Driving Ragdolls



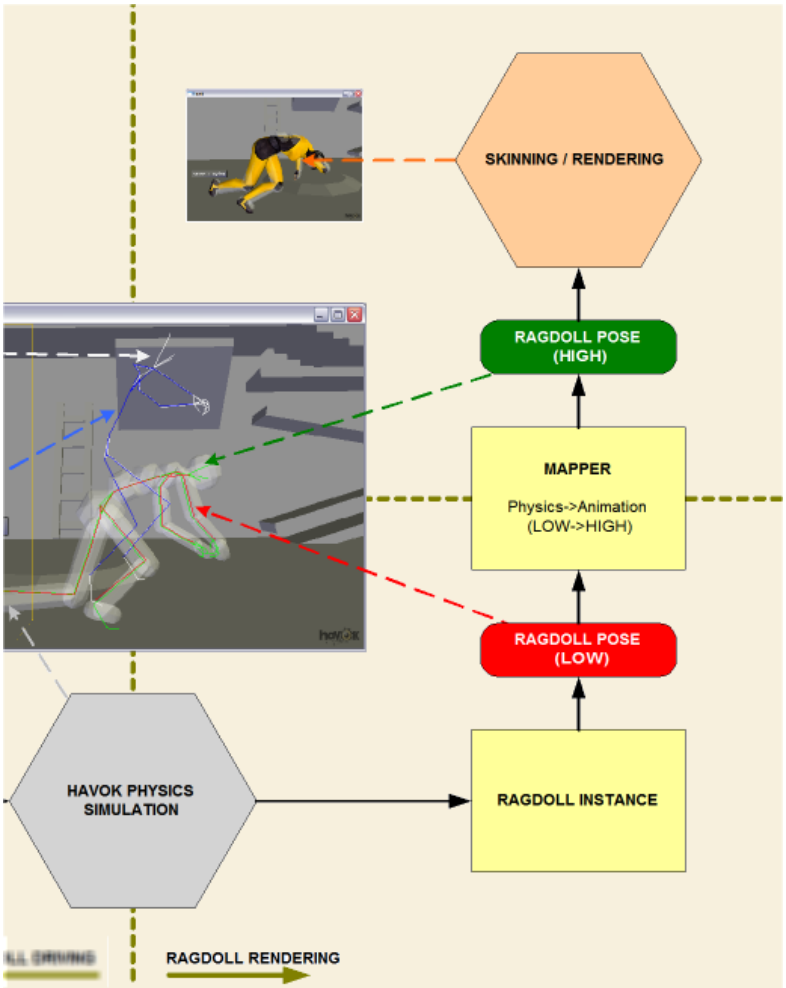

We start with a high bone count (animation) pose given to us by the animation system. This pose is depicted in white in the diagram, and we call it the "driving" pose.

When driving ragdolls, we want to convert this pose to a low bone count (physics) pose. We do this using a mapper ( **hkaSkeletonMapper** ). The mapper here will map from "high" to "low" bone count poses. The output is, then, a low bone count pose (depicted in blue). This pose is also a "driving" pose, but now using a low bone count representation.

Given this pose, we now want the physical representation of the character (the ragdoll) to try to match the pose. We use a ragdoll controller for that. The output of this object is a set of forces / torques / other input to the physics system.

We don't always need to pass the driving pose through a ragdoll controller. For example, we may want to force the rigid bodies to match the given pose (rather than driving them towards it). In this case, we can pass this pose straight to the rigid bodies positions and orientations by calling **hkaRagdollInstance::setPose()** .

#### 3.4.2.2 Rendering Ragdolls




When rendering ragdolls, we start with a set of rigid bodies simulated by the physics. We can get a pose based on those rigid bodies by using a ragdoll instance, and calling **hkaRagdollInstance::getPose()** . This ragdoll pose has a low-bone count, and it's depicted in red in the diagram above.

Since skins are usually bound to the original high bone count skeleton, we need to convert this low bone count pose to a high bone count pose. Again, we use a mapper to do so. The output of this mapper is a high bone count pose, depicted in green.

Finally, we can use this pose to render and skin the character, or possibly modify it even further (blend other animations, apply IK).

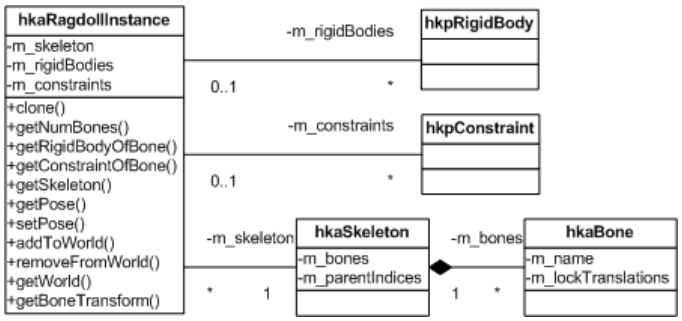
### 3.4.3 Ragdoll Instances

Most of the functionality relating to the manipulation of ragdolls is encapsulated in the `hkaRagdollInstance` class.

A ragdoll instance keeps pointers to the rigid bodies and constraints that make up a ragdoll. It also holds a pointer to the `hkaSkeleton` instance that describes the structure of this ragdoll.

Having all this information allows the ragdoll instance to provide the following functionality:

- Accessing the rigid bodies and constraints representing each bone / joint.
- Accessing the associated skeleton and its hierarchy.
- Retrieving the current pose of the ragdoll in the physical world.
- Setting the pose of the ragdoll in the physical world.
- Cloning the ragdoll.
- Adding and removing the ragdoll to/from the physical world.




#### 3.4.3.1 Creating a Ragdoll Instance

You can create ragdoll instance during tool chain operations using the Create Ragdoll filter. In order to setup a ragdoll instance at run-time, you need the following components:

- A set of rigid bodies and constraints, representing the bones and the joints of the ragdoll. As we'll see in the next section, these rigid bodies and constraints have to be properly set up.
- A skeleton (`hkaSkeleton`), containing as many bones as rigid bodies in the ragdoll.

This information (an array of rigid bodies, an array of constraints and a pointer to a skeleton) is passed to the constructor of an `hkaRagdollInstance`:

```
hkaRagdollInstance ( const hkArrayBase<hkpRigidBody*>& rigidBodies, const hkArrayBase<
  hkpConstraintInstance*>& constraints, const hkaSkeleton* skeleton );
```

### Rigid Bodies and Constraints

In order for the `hkaRagdollInstance` object to optimally work with the rigid bodies and constraints at runtime, it requires that the following:

- The rigid bodies must be properly ordered from parent to children. The first rigid body has to be the root rigid body (usually the pelvis).
- If there are n rigid bodies, there must be n-1 constraints (connecting those rigid bodies). Constraints also have to be ordered so the child rigid body (rigid body A) of constraint *i* is rigid body *i+1*.
- The parenting of constraints need to be correct. Rigid body A in a constraint must always be the "child" rigid body, while rigid body B must always be the "parent" rigid body.
- The local transformation of the rigid bodies must match the pivot point of the bone they represent.
- Since the constraints represent joints, the pivot point of a constraint should match the pivot point of the joint (the pivot point of the child bone / child rigid body).

These requirements ensure that operations on ragdolls can be done, and can be done efficiently. Havok provides ragdoll utilities to assist in this. Using **hkaRagdollUtils::reorderAndAlignForRagdoll()** will ensure all the above requirements:

```
static hkResult hkaRagdollUtils::reorderAndAlignForRagdoll (hkArray<hkpRigidBody*> &rigidBodiesInOut,
  hkArray<hkpConstraintInstance*> &constraintsInOut);
```

The only requirements on the input physics system in `reorderAndAlignForRagdoll()` are:

- There must be n rigid bodies and n-1 constraints.
- Constraints must be properly parented.
- Constraints' pivots must be properly placed (at the joint).

If the above are satisfied, the `reorderAndAlignForRagdoll()` method will re-order the rigid bodies and constraints and modify the rigid bodies so their local transforms match the pivot of the associated constraints.

### Disabling Inter-Bone Collisions

It is normally desirable to filter out collisions between bodies constrained to each other. This both avoids possible conflicts between bodies with constraint pivots inside their collision shape and improves performance by removing unnecessary collision processing between bodies which should never actually collide. Ragdoll limbs represented as constrained rigid bodies are typical in this regard, and care should be taken to ensure that optimal collision filtering is set up between adjacent limbs (i.e. limbs sharing a constraint).

The required filtering can be implemented using either the hkpGroupFilter (uses the bodies' filter info) or the hkpConstraintCollisionFilter (uses knowledge of the constraints currently present in the world) - see their documentation in the collision filtering chapter for details.

#### The Ragdoll Skeleton

In order to create a ragdoll instance, you also need to provide a pointer to an **hkaSkeleton** associated with the ragdoll. This skeleton is used at run-time in order to access hierarchy information efficiently, as well as to store some per-bone information (name, initial pose and skeletal constraints).

It is easy to create an skeleton from the rigid bodies and constraints. First, you need to ensure that the rigid body and constraint arrays satisfy the requirements described in the previous section. Then use the utility **hkaRagdollUtils::constructSkeletonForRagdoll()** to create an appropriate skeleton.

```
static hkaSkeleton* hkaRagdollUtils::constructSkeletonForRagdoll (const hkArray<hkpRigidBody*> &
  rigidBodies, const hkArray<hkpConstraintInstance*> &constraints);
```

The names of the bones stored in this new skeleton will be copied from the names of the rigid bodies. It is important that your rigid bodies are properly named before calling this method.

The initial skeleton pose is taken from the current position of the rigid bodies at the time of the call. In general, the skeleton should be created at setup time with the rigid bodies aligned in a T-Pose.

Finally, it is important to notice that the **constructSkeletonForRagdoll()** method will allocate some memory in order to hold the array of bones as well as the bone names. You can destroy this skeleton as well as this associated memory by calling:

```
static void hkaRagdollUtils::destroySkeleton (hkaSkeleton* skeleton);
```

#### 3.4.3.2 Getting and Setting a Pose

The `hkaRagdollInstance` class also allows you to set and retrieve the current pose of a ragdoll, using the following methods:

```
void hkaRagdollInstance::getPoseModelSpace (hkQsTransform* poseModelSpaceOut, const hkQsTransform&
  worldFromModel) const;
void hkaRagdollInstance::setPoseModelSpace (const hkQsTransform* poseModelSpaceIn, const hkQsTransform&
  worldFromModel);
```

The poses are specified in model space, i.e., as a set of *model-from-bone* transforms. You also need to specify how this model space is related to the world space (since that's the space used by Havok Physics to place the rigid bodies), passing an extra `worldFromModel` transform. For more information about these spaces and what they represent, please refer to the World, Model and Local Spaces section.

**Note:**

`setPoseModelSpace()` is intended for (re)initialization but not intended to be called continuously during simulation as it can be slow and it will zero all body velocities. To influence (or "drive") the pose of an `hkaRagdollInstance` over time during simulation, use `hkaRagdollRigidBodyController::driveToPose()`, or call `hkpKeyFrameUtility::applyHardKeyFrame()` on each rigid body instead.

### 3.4.4 Mappers

A mapper is responsible for converting a pose from one skeleton to another. The run-time class that does this is the **hkaSkeletonMapper** class. Mappers are described in detail in the Animation chapter in this manual. They play a really important role in ragdoll integration, so please read the documentation about Skeleton Mappers.

### 3.4.5 Ragdoll Controllers

Ragdoll controllers are responsible for driving a ragdoll instance to a desired pose. This can be achieved by different means (powered constraints, applying velocities to rigid bodies), so multiple implementations of ragdoll controllers are possible.

There are three ragdoll controllers provided with the Havok SDK:

- **The Ragdoll Rigid Body Controller** : operates on the rigid bodies (bones).
- **The Ragdoll Powered Constraint Controller** : operates on powered constraints (joints).
- **The Ragdoll Powered Chained (Constraint) Controller** : operates on powered chained constraints (joints).

#### 3.4.5.1 The Ragdoll Rigid Body Controller

The ragdoll rigid body controller ( **hkaRagdollRigidBodyController** ) tries to drive the rigid bodies of the ragdoll to the desired pose by setting their linear and angular velocities.

This controller wraps the functionality in the `hkKeyframeHierarchyUtility`, and specializes it for use with ragdolls. You can create a ragdoll rigid body controller by either associating it with a ragdoll instance, or by explicitly passing the rigid bodies and hierarchy involved:

```
hkaRagdollRigidBodyController ( hkaRagdollInstance* ragdoll );
```

The setup for the controller is available through a public member, `m_controlDataPalette`. This is an instance of `hkaKeyFrameHierarchyUtility::ControlData`, and stores information about gains and ranges used by the controller (follow the link to the class for details on its members)

```
hkArray<hkaKeyFrameHierarchyUtility::ControlData> m_controlDataPalette;
```

While driving the ragdoll using this controller, you should use the driveToPose() method:

```
void driveToPose ( hkReal deltaTime, const hkQsTransform* poseLocalSpace, const hkQsTransform&
  worldFromModel, hkaKeyFrameHierarchyUtility::Output* stressOut );
```

The call requires a timestep since the forces and velocities applied to the rigid bodies are time-step dependent. `driveToPose()` returns information about the error (stress) on the controller for each rigid body through the output parameter `stressOut`.

You will need to call `driveToPose()` every step of the simulation where you want the ragdoll to be driven. You can stop driving the ragdoll simply by not calling `driveToPose()`. If, after that, you want to start driving the ragdoll again (or if you move the ragdoll's position or velocity outside the controller), you'll need to call `reinitialize()` to let the controller now (since it keeps state information based on the last step).

```
void reinitialize();
```

#### 3.4.5.2 The Ragdoll Powered Constraint Controller

This controller is implemented through a static utility class (`hkaRagdollPoweredConstraintController`). The powered constraint controller tries to drive the rigid bodies to the desired pose by using the motors of the powered constraints that attach those rigid bodies.

Therefore, this controller is only useful if the constraints in the ragdoll are powered constraints (powered hinges or powered ragdoll constraints). Fortunately, you can convert a hinge or ragdoll constraint to its powered counterpart by using the utility method `hkpConstraintUtils::convertToPowered()` (in the `hkpUtilities` library).

```
static hkpConstraintInstance* hkpConstraintUtils::convertToPowered (const hkpConstraintInstance*
  originalConstraint, hkpConstraintMotor* constraintMotor);
```

The `hkaRagdollPoweredConstraintController` class provides the following static methods:

```
static void driveToPose( hkaRagdollInstance* ragdoll, const hkQsTransform* poseLocalSpace );
static void startMotors( hkaRagdollInstance* ragdoll );
static void stopMotors( hkaRagdollInstance* ragdoll );
```

The main method is `driveToPose()`. It takes a ragdoll instance and a pose, specified in local space. This method will go through the different joints and set the target of each motor to the desired pose.

The other two utility methods, `startMotors()` and `stopMotors()`, will go through the different motors associated with the joints (through the powered constraints) and enable / disable them accordingly.

The ragdoll powered constraint controller will generally work with all kinds of motors, yet the `hkpPositionConstraintMotor` is usually preferred because of its increased stiffness. Because each motor implementation has different values that control strength, damping, etc.. if you want to change the behavior of the motors (for example, weaken or strengthen them) you will have to do so in your application code (where you can correctly cast each motor to its proper type).

#### 3.4.5.3 The Ragdoll Powered Chained Controller

The ragdoll powered chain controller is a specialized variant of the ragdoll powered constraint controller to be used when your ragdolls contain powered chained constraints. It is controlled in exactly the same manner as the powered constraint controller (their interfaces are identical).

#### 3.4.5.4 Which Controller to Use

As always, the choice of the ragdoll controller depends on the actual application. In many cases, you may want to use both types at different stages of the simulation. The following table gives a comparison of both controllers in different areas.

| Rigid Body Controller | Powered Constraint Controller |
|---|---|
| Works on any ragdoll setup, regardless of the constraints used. | Requires the use of powered constraints. Powered constraints are slower to simulate than normal constraints. |
| Keeps state between calls. | Stateless (access through static methods). |
| Time-step dependant. | Independent of time step. |
| Needs to be applied explicitly every step (through `driveToPose()`). | Can be switched on/off, but it's implicitly applied every step by the physics. So, once on, it's always applied. `driveToPose()` just changes the current target pose. |
| Works in world space (bone transforms) | Works in local space (joints). |
| Drives all bones, including the root. | It doesn't drive the root. |
| Since it works by setting velocities in world space, it will drive the ragdoll to the desired world space pose - forces applied to it will have little effect. | Since it works in local space (and doesn't drive the root), it gives more realistic results in many situations.For example, you can apply impulses and forces to the ragdoll (caused by, let's say, an explosion) - the ragdoll joint controller will still drive the joints while the ragdoll flies through the air. |

Table 3.9: Using Bone and Joint Controllers

To see these controllers used in context see the Animation Examples and Use Cases section.

### 3.4.6 The Detect Ragdoll Penetration Utility

Havok integration of physics and animation into the animation pipeline allows for continuous change between an animation-driven character to a fully dynamic character (ragdoll). In many of these "state-change" situations it's very useful to know the penetration status of rigid bodies/bones. The detect ragdoll penetration utility (`hkaDetectRagdollPenetration`) checks for these penetrations and returns the current status of all ragdoll's bones for an specific input pose.

#### 3.4.6.1 Overview

There are situations where you may want to exactly detect the penetration of a ragdoll into the landscape. For example, your character is walking against the wall and suddenly he is shot by an enemy. In the walking state, the character ragdoll is driven by an animation and the motion of each rigid bodies is keyframed. After the shot, the character state is swapped to dead (ragdoll) and the character is fully driven by dynamics.

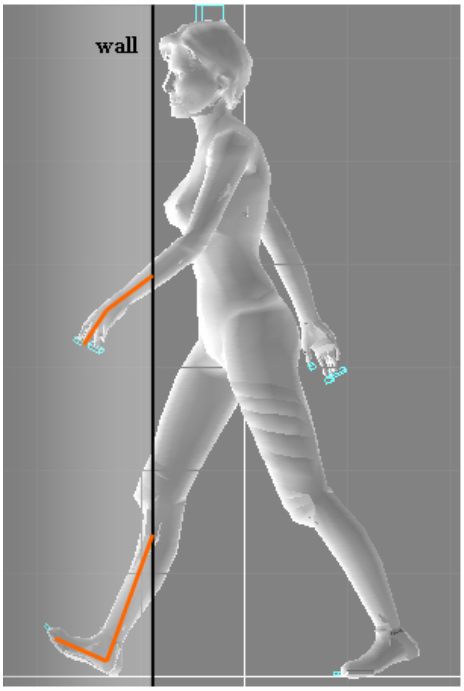



Figure 3.37: Character's limbs penetrate to landscape during keyframed to dynamics state change

If some of the rigid bodies (usually limbs) are in the "dark side" of the landscape at this moment, problematic situations can occur, which if they are not correctly handled may cause improper behavior as the collision detection and constraints may "fight" each other - causing the ragdoll to get "stuck" in the landscape and/or jitter. A recovery method for ragdoll penetration is shown in **the Driving Ragdolls section** .

Continuous physics guarantees that if the ragdoll starts in a correct (not going through the landscape) pose, it should remain in a correct pose over time, i.e., it should not go through the landscape at any time. Given that, a good way to ensure proper behaviour is to follow this process:

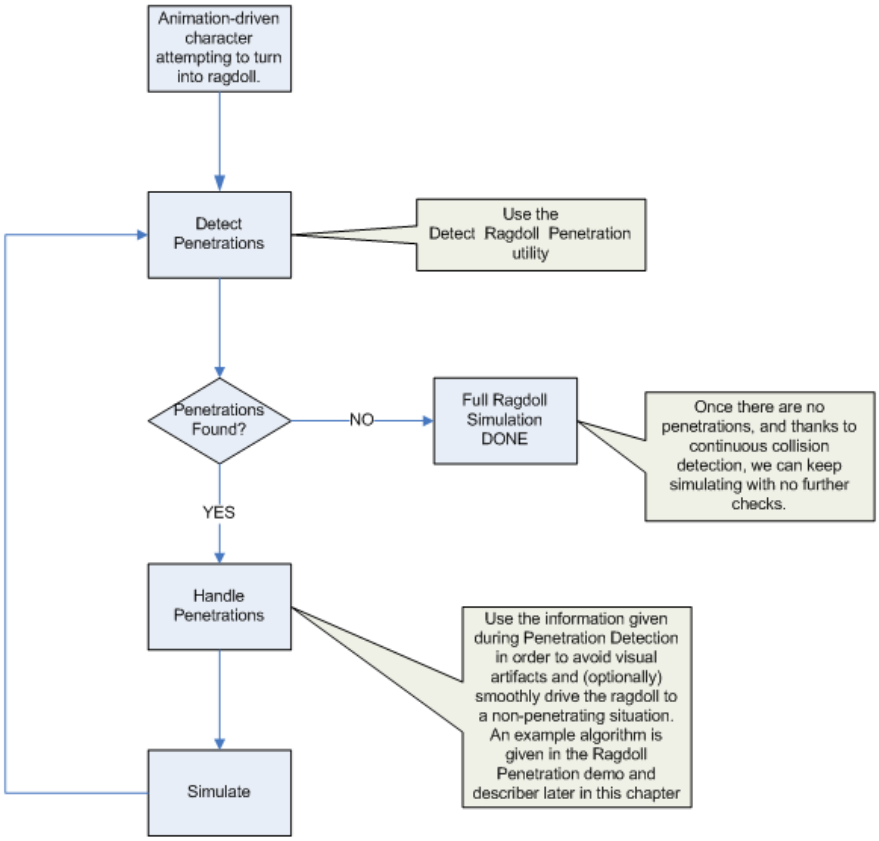



Figure 3.38: A process to handle ragdoll penetrations

Notice that the detection can also be used preemptively - for example, you can decide not to turn a character into a ragdoll if the penetration check fails.

#### 3.4.6.2 The hkaDetectRagdollPenetration utility

**hkaDetectRagdollPenetration** utility (available in the `hkaRagdoll` library) detects the penetration of all bones of the ragdoll's skeleton with the landscape. The detection routine simply performs a raycasting of all bones against the landscape until all bones are not penetrating.

Since only the bone positions in space are required, this utility can be applied to an arbitrary skeletal pose, and not just to ragdolls. Each ray is defined by the start and end point of the bone in world coordinates. The utility will associate one of three states to each bone: *non-penetrating*, *penetrating* or *descendant* (of a penetrating bone). These three states describe the penetration of the actual pose better than a simple yes/no penetration state per bone.

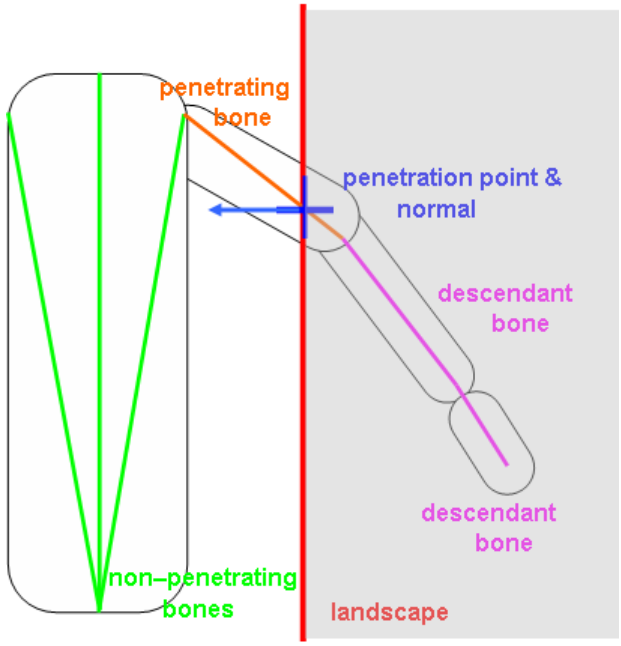



Figure 3.39: Status of particular bones during penetration

The utility is optimized for performance and stores an internal state - after the first check, it will only check (raycast) for bones that were previously penetrating. Since a state is kept, the utility must be instantiated. The setup structure passed to the constructor contains:

- A pointer to a skeleton (`hkaSkeleton`) : this is usually the skeleton of the `hkaRagdollInstance` we are working with. The utility will also work with any non-ragdoll skeleton (so you can use it to detect penetrations of any characters)
- A pointer to raycast interface (`hkaRagdollRaycastInterface`) : this interface is used to perform the raycasts - you can use Havok's optimized raycasting system or a custom raycasting solution if desired. The interface also gives you a chance to perform any setup on the raycast (for example, assigning particular collision filter information in order to select what objects should be raycast against).
- A pointer to an AABB phantom (`hkpAabbPhantom`), which should encapsulate the whole ragdoll. This phantom is used in order to optimize the raycasting when using Havok's raycasting. The utility will also gradually update the phantom as the pose changes and bones stop penetrating.

```
// setup DetectRagdollPenetration object
hkaDetectRagdollPenetration::Setup setup;
{
    setup.m_ragdollSkeleton = m_ragdollInstance->getSkeleton();
    setup.m_raycastInterface = m_ragdollRaycastInterface;
    setup.m_ragdollPhantom = m_ragdollPhantom;
}
m_detectRagdollPenetration = new hkaDetectRagdollPenetration(setup);
```

The principal `detectPenetration()` method takes a `hkaPose` **in world space** as input and performs raycasting of all bones with the landscape. The detection routine fills an output structure, containing

information for each penetrating bone as an array of `BonePenetration` structures. Each bone penetration structure contains the begin and end indices of the bone, the penetration point in world coordinates and the normal of the penetrated surface.

```
// ModelPose is set directly to WorldPose used for penetration detection
hkaPose ragdollPose( m_ragdollInstance->getSkeleton() );
m_ragdollInstance->getPoseWorldSpace( ragdollPose.accessUnsyncedPoseModelSpace().begin() );

// Method detectPenetration returns true if there is a penetration, or false if there isn't
hkBool penetrationStatus; penetrationStatus = m_detectRagdollPenetration->detectPenetration( ragdollPose,
  m_penetratedBones );
```

The penetration state of each bone is internally stored in the utility. As we mentioned before, each bone can have three states: no penetrating (HK_NOP), penetrating (HK_YESP) or descendant of a penetrating bone (HK_YESP_DESCENDANT). The class provides the method `getBoneStatusArray()` to get a reference to the bone state array.

```
// get boneStatusArray from hkDetectPenetration object
const hkArray<hkaDetectRagdollPenetration::BonePenetrationStatus>& boneStatusArray =
  m_detectRagdollPenetration->getBoneStatusArray();
```

#### 3.4.6.3 An Example of Ragdoll Penetration Recovery

As described in the overview, once you detect some bone penetrations, you need an algorithm to handle them in order to avoid/reduce any visual artifacts. The RagdollPenetrationDemo shows a very effective algorithm to recover from ragdoll penetrations and smoothly drive the ragdoll "out" of the landscape.

The algorithm operates based on the results of the hkaDetectRagdollPenetration utility, varying its action depending on the state of each bone:

- if HK_NOP (not penetrating bone/rigid body): enable collision with the landscape (if it was disabled).
- if HK_YESP_DESCENDANT (descendant of the penetrating bone/rigid body): disable collision with the landscape.
- if HK_YESP (penetrating bone/rigid body) : disable collision with the landscape and apply correction impulse. The direction of the impulse is the same as the penetration point normal. The magnitude of the impulse is calculated using the following formula:

  *Impulse = m * ( vd - vc*cos(alpha) )*

  where $m$ is the mass of the penetrating rigid body, *vd* is the desired velocity (a tunable constant), *vc* is the current velocity magnitude at the penetration point and *alpha* is the angle between the current velocity and the normal at the penetration point.

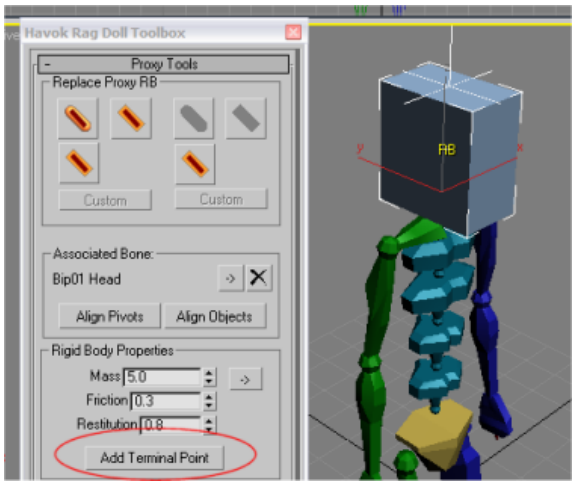
The RagdollPenetrationDemo illustrates the effective use of this utility.

#### 3.4.6.4 Toolchain Implications

Since penetration detection is done through raycasting from each bone to their children, one of the consequences is that leaf bones of a skeleton cannot be checked for penetrations. This raises a problems with ragdoll skeletons, where leaf bones are commonly "normal" bones (rigid bodies) with volume and could therefore intersect the landscape.

The `hkaRagdollInstance` class allows for its associated `hkaSkeleton` to have more bones than rigid bodies, and therefore allows for extra terminal bones to be added to the ragdoll skeleton, specifying the dimension of the leaf rigid bodies (for example, the tip of the feet, hands and head). This means that an `hkaRagdollInstance` may have $n$ rigid bodies, but the associated skeleton may have $m$ (where $m>=n$) bones.

The Havok Content Tools have been adapted to facilitate the creation of ragdoll terminal bones:




**Caution:**

When creating ragdolls with terminal bones, make sure you create the ragdoll skeleton using the Create Skeleton filter and then use it for the **Use Skeleton** option in the Create Ragdoll filter. Using the **Automatic** mode in the Create Ragdoll filter would ignore any terminal bones (as it builds the skeleton from the rigid bodies, and not vice-versa).

## 3.5 Animation Examples and Use Cases

### 3.5.1 Animation Showcase

The Havok Animation showcase demo brings several useful features of Havok Physics and Animation together in one usecase example. These include:

- Animation blending and playback.
- Integration with the Havok character controller.
- Foot, hand and look-at IK.

- Local hit reactions using a ragdoll controller that works in model space.
- Getting back up using pose matching.
- Collision filtering.

After a brief introduction on the demo setup, the following chapter describes each of these in detail.

#### 3.5.1.1 Showcase Setup

The Animation Showcase demo uses different representations for the main character. When running around and colliding with the environment, we use the Havok character controller. A capsule is used as the proxy shape (yellow in the image below). You can display this in the demo from the Options menu [Display->ShowProxy].

This proxy is responsible for resolving collisions in the landscape and for motion extraction.

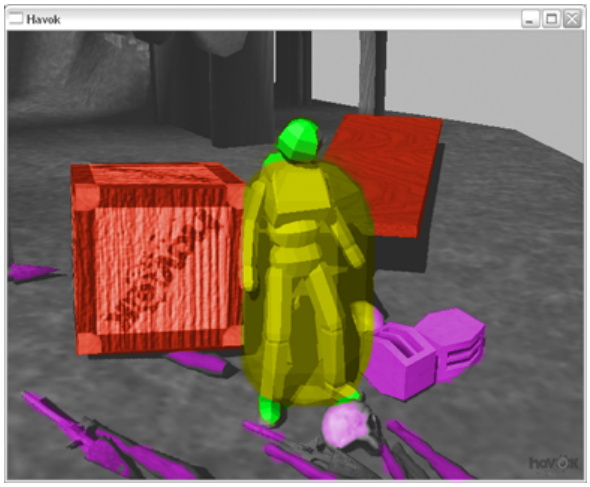



Figure 3.40: Collision layers in the demo

The proxy is also responsible for interaction with dynamic game objects. In our demo, these objects are the see-saw and the large crates (shown above in red). For these objects, the interaction needs to be well defined. We restrict the movement of the proxy by controlling the force that it applies to objects. This gives the character a sense of strength.

The other dynamic objects in the scene (marked in purple above) are flagged as debris items. These do not interact with, or affect the movement of the character proxy. They do interact with the ragdoll representation (shown in green).

The ragdoll representation is a constrained rigid body system. Under normal operation the pose from the animation system is used to drive the positions of the rigid bodies using a ragdoll rigid body controller (`hkaRagdollRigidBodyController`). Since these bodies are driven by the animation, they collide realistically with the debris.

In particular, their velocity comes directly from the animation system. By default the legs are keyframed and the upper body is dynamic. This means the upper body will react to collisions from both the static environment and dynamic objects, while legs power right through objects (i.e. their motion is not

impacted by collisions). If you run against the wall or pick up a dynamic object and hit the character with it you will see this reaction.

#### 3.5.1.2 Collision Filtering

When the character falls and turns into a powered ragdoll it changes the motion state of the bodies and their collision filter information to allow them to interact with everything (including debris). In total, we use eight distinct layers in the demo to control interaction. These are

- Landscape Layer : contains the static geometry for the level. In this demo, we have both visual geometry and simplified collision geometry.
- Proxy Layer : contains the proxy for the character controller.
- Debris Layer : contains the small dynamic objects in the scene and the bricks in the wall on the top level. Each of these have different masses.
- Moveable Environment : contains the see-saw and the large crates.
- Ragdoll Keyframed : contains the bodies of the ragdoll that are currently keyframed (the lower body).
- Raycast : this layer is used to control which items to raycast against when performing foot and hand IK.
- Ragdoll Dynamic : contains the rigid bodies for the ragdoll which are not keyframed. When the ragdoll falls all ragdoll bodies are switched into this layer.
- Picking : a layer that control which items are effected by the mouse pointer.

The interactions are shown here. Layer pairs with collisions enabled are shown in green. Layer pairs with interactions disabled are shown in red.

| | Landscape | Proxy | Debris | Moveable | Ragdoll Keyframed | Raycast | Ragdoll Dynamic | Picking |
|---|---|---|---|---|---|---|---|---|
| Landscape | | | | | | | | |
| Proxy | | | | | | | | |
| Debris | | | | | | | | |
| Moveable | | | | | | | | |
| Ragdoll Keyframed | | | | | | | | |
| Raycast | | | | | | | | |
| Ragdoll Dynamic | | | | | | | | |
| Picking | | | | | | | | |

Figure 3.41: Layers enabled in the demo

Some layers are disabled for game play and control reasons. In particular the moveable objects, like the see-saw and the large boxes, only interact with the proxy. The proxy pushes and interacts with objects in a more controlled way than the driven ragdoll. These larger objects would typically be involved in gameplay and changes in animation content should not change their response.

#### 3.5.1.3 Animation Playback and Blending

Animations in the demo are played back using the low-level mixing and blending functionality of `hkaAnimation`. These low level calls are made through 2 higher level interfaces. The first is an animation command processor (`AnimationMachine`). This defines a very simple data protocol which translates messages into calls to animation controls. Another interface, the animation state machine (`AnimStateContext`, `AnimStateManager`), generates lists of these commands as it transitions between states.

#### 3.5.1.4 Integration with the Havok Character Controller

The character proxy is driven by animation at each frame to produce a new position. This process is described in the Motion Extraction section. Since the landscape has significant discontinuities, especially on the stairs, the character movement contains lots of high frequency noise. If we simply take the position of the proxy and use it as the reference frame for our animation then all this noise will be seen in the final animation.

To combat this noise we decouple the proxy slightly from the animation. We allow it to drift (up to a limit) in both the vertical and horizontal direction. An error is fed back into the proxy on the next step. The decoupling is best viewed by turning on the character proxy in the display options and running up and down the stairs.

#### 3.5.1.5 Foot, Hand and Look-At IK

The parameter and setup for the foot, hand and look at IK are detailed in the Inverse Kinematics section. One useful trick shown in the demo is how the error from the IK solver is fed directly back into the pelvis' vertical position. In doing so the character automatically adjusts the pelvis according to the results of the IK.

The feedback loop is very straightforward. The IK returns a positive error if the leg is compressed and a negative error if it does not reach its target. We compute a delta movement based on the maximum compression and stretch. Experimentally we have found that biasing the delta more towards the stretch gives a better visual result. This delta is damped and then applied to the pelvis.

```
// w is close to 1.0.
const hkReal weightedError = w * hkMath::min2(leftError, rightError) + (1.0f - w) * hkMath::max2(leftError
  , rightError);
hkReal pelvisDelta = weightedError * m_options.m_IK.m_pelvisFeedback; // damp it
verticalDisplacementInOut += pelvisDelta;
```

#### 3.5.1.6 Local Hit Reactions

As previously mentioned, we drive the ragdoll representation of the character using the current animation. If dynamic objects collide with the ragdoll the controller will briefly diverge from its desired location, the collision will be resolved and the controller returns the ragdoll to the desired pose. This can be seen by picking one of the debris objects (bricks, pulleys, skulls) and using the mouse spring to make it collide with the character while he is standing idle.

The idle animation keeps the character standing upright. Normal collision detection and resolution takes place and the ragdoll bones move. The controller then attempts to correct the movement and drive the ragdoll back to the idle pose. Obviously, the key is in choosing appropriate parameters for the controller strength. If the controller is too strong then we will see little movement in the character when objects collide. If it is too weak then the controller will not follow the animation properly.

In the demo, the bodies of the ragdoll are automatically colored to show the stress on each of the bones. This is a measure of how far the ragdoll has diverged from the animation. It can be used to drive sound, damage or trigger a full ragdoll sequence. For details on how this controller works and parameter choices please see `hkaRagdollRigidBodyController` and the utility class it relies on `hkaKeyFrameHierarchyUtility`.

#### 3.5.1.7 Animation Driven Ragdoll Deaths

When the character is triggered into full ragdoll mode we switch all the bodies to dynamic and allow the physics to drive the animation. However, using just physics gives a relatively lifeless performance. The ragdolls appear limp and floppy as if the character is unconscious. To remedy this, we use another ragdoll controller (`hkaRagdollPoweredConstraintController`) to layer an animation onto the local joints of the ragdoll.

Even though the ragdoll is tumbling and falling through the air, the controller drives the joints to match the currently playing animation. In the demo, if you enable GetUp->DieAnimation it plays a death animation (which brings the character into a ball) as soon as we transition to ragdoll. As can be seen the general gait of the character matches the animation even though his overall orientation and position are different.

**This use of animations can greatly enhance the visual behavior of a ragdolled character.**

#### 3.5.1.8 Getting Back Up Using Pose Matching

The same controller is used to make the ragdoll seamlessly get back up again. As the character is in the air tumbling through the scene we are constantly examining its pose and attempting to find a pose it can get back up from. The controller drives to this desired pose so that when the character has settled after the fall they are in a configuration close to the initial pose of a get-up animation.

This demo uses just 3 get-up animations. In each of these an artist has marked frames in the animation using annotations. When these animations are loaded we scan the annotations for the marked frames. For each marked frame we sample the animation at that point and add it to a list of valid poses to get up from (see `AnimationShowcase::initPoseMatching()`).

The pose matching utility `hkaPoseMatchingUtility` find the closest poses that match the current ragdoll orientation and drives to that.

An artist controls the get-up point by placing annotations in the modeller. These annotations are set in Max using note tracks and in Maya by creating a custom attribute. Further information on the animation content tools is available in the Havok Content Tools documentation.

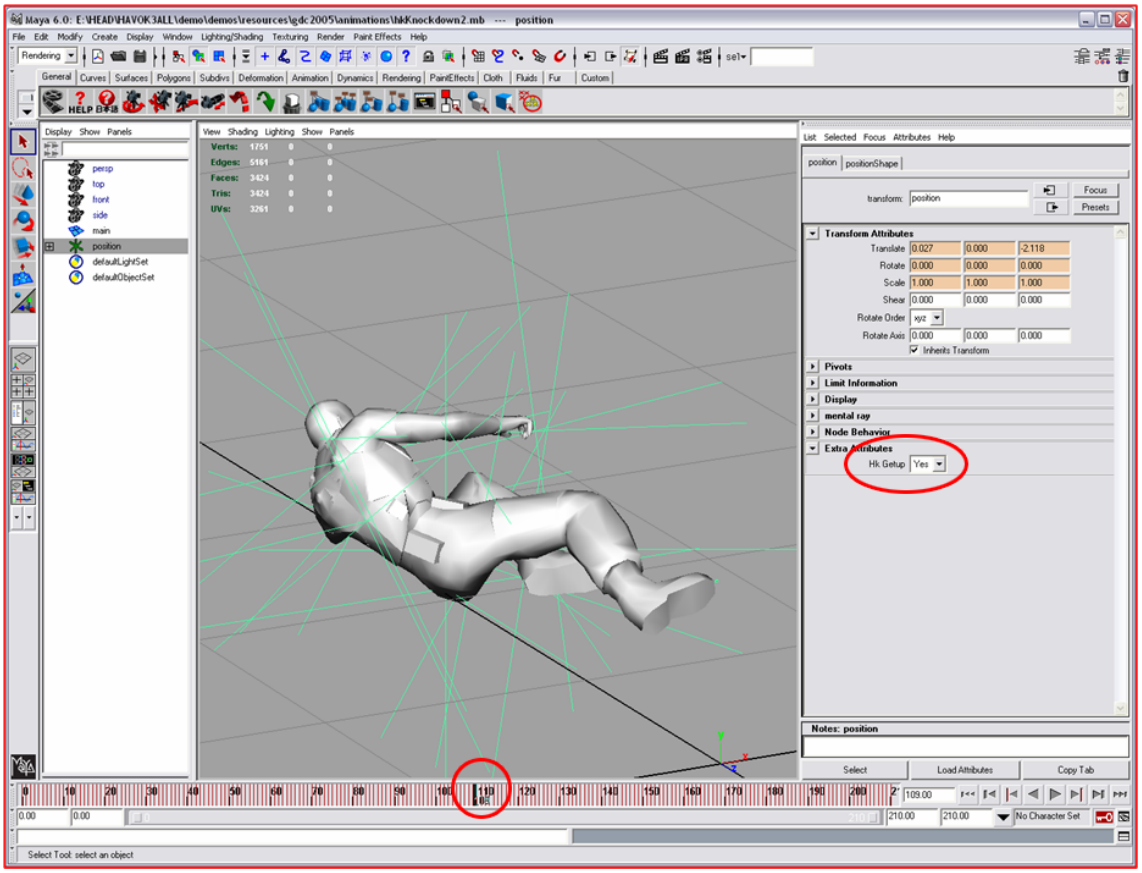



Figure 3.42: Annotating poses in the modeller

Our method for searching the pose database and matching against the current ragdoll configuration is simple and efficient. We choose three reference points on the body and use their relative orientation to find the closest matching pose. The reference points we choose include the main upper body axis (from pelvis to neck) which is usually the most discriminating axis.

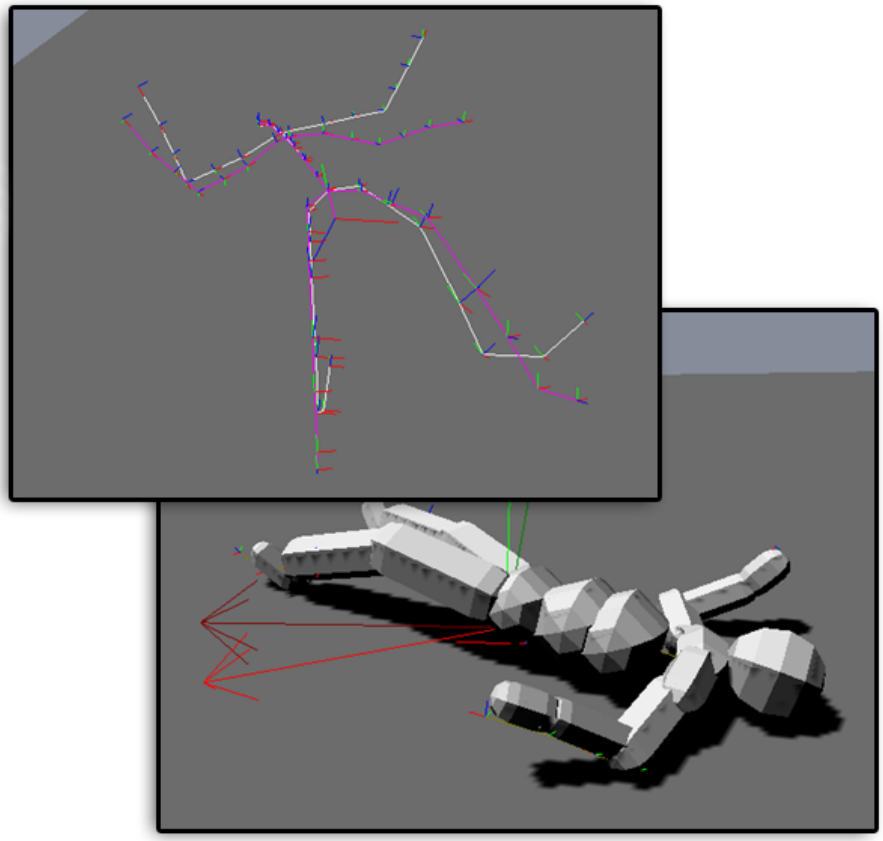


Figure 3.43: Pose Matching reference frame

An advantage of this scheme is that it is very efficient to search for a given pose. It requires we build a frame from the reference points in model space and compare this with the other frames held in the database. This reduces to constructing a quaternion and performing a quaternion dot product with the pre-computed frames.

The image above displays an example of pose matching. The actual animation pose we are attempting to drive to is shown in white. The current pose of the ragdoll is overlaid in purple. As can be seen the match is close but not exact. It does not need to be exact because during the get-up sequence we blend between the current ragdoll pose and our animation making the transition perfectly smooth.

**Controlling Pose Switching**

As the ragdoll falls we are constantly matching new poses to get up from. If our resting pose is very close to the boundary between two matching poses in the database then the ragdoll can toggle between them continuously. This usually occurs when the ragdoll does not settle quickly in one of the desired states. To eliminate this we must either change the pose matching function to a more discriminating one or add hysteresis (i.e. lag) to the code.

In the showcase demo we add hysteresis. When we find a new pose we do not make the switch until that pose remains the best match for a fixed time period. This prevents the toggling between similar states since it allows the ragdoll to settle. In addition we also stop looking for better poses once the ragdoll has settled. We simply check the velocity of the pelvis to make this decision. See the method `AnimationShowcase::matchPose()` for details.

# Chapter 4

# Havok Content Tools

## 4.1 Guide To The Havok Content Tools Documentation

### 4.1.1 Introduction

The Havok Content Tools encompass a large and powerful set of functionality, and with it comes extensive documentation. While the documentation has been structured into clearly defined areas, depending on your background, choice of modeller, experience using Havok and project role, you will often find yourself focusing on different sections which will not necessarily be in sequential order.

The purpose of this chapter is to give you, the reader, an indication of the sections of the documentation that we believe are most relevant depending on your role and the Havok product(s) you are using. Here we'll list the documentation you should not miss, but we still encourage you, of course, to browse the rest of the documentation.

We've divided the guide into two main sections, depending on project role: documentation for content creators (artists), and documentation for tools programmers.

- Content Creators: Anybody using the Havok Content Tools to model, animate and/or setup assets to be exported to and used by Havok's runtime.
- Tools Programmers: Anybody attempting to integrate the Havok Content Tools into an in-house tools pipeline, extending them or automating them.

Each section is divided further depending on the Havok products being used. If you are using more than one Havok product, please check the guide for each product.

### 4.1.2 Documentation for Content Creators

Most of the Havok Content Tools documentation is targeted at content creators, artists, and technical artists using our tools to generate content to be eventually exported and used in Havok's runtime. Here we describe the most relevant sections of the documentation grouped by product.

#### 4.1.2.1 All Havok Products

**Make sure you've already read**
HCT Documentation Guide > Introduction

The following basic documentation is essential reading, relevant regardless of the Havok product being used:

- Introduction > Architecture: Gives you an overview of how the Havok Content Tools fit into the game content creation pipeline.
- The Havok Filter Pipeline > The Filter Manager: This is one of the main components of the Havok Content Tools, and where most content processing is set up.
- The Havok Filter Pipeline > Core Filters: Elemental asset processing units, common to all products.
- The Preview Tool: Used to visualize and debug created content before it is exported.

There is also essential reading depending on whether you are working with 3ds Max, Maya or XSI/Softimage:

- *The Scene Exporter* (3ds Max | Maya | Softimage): The core Havok component for each modeler, connecting content creation with content processing.
- *Export Basics Tutorial* (3ds Max | Maya | Softimage): Although it uses an Animation use case, this tutorial covers the basic concepts regarding asset export. A must-read regardless of product.

Finally, if you run into any trouble with any of the Havok Content Tools components, don't forget the following section:

- Troubleshooting: Divided into multiple sections, it covers common mistakes, errors and known issues you may encounter while using the Havok Content Tools.

#### 4.1.2.2 Havok Animation

**Make sure you've already read**
HCT Documentation Guide > Content Creators > All Products

Common concepts related to Havok Animation are covered as subtopics in the Common Concepts chapter, in particular:

- Common Concepts > Animation > Repositioning Animations and Motion Extraction
- Common Concepts > Animation > Annotations
- Common Concepts > Animation > Controlling Compression
- Common Concepts > Animation > Previewing Compression
- Common Concepts > Animation > Controlling File Size

Most of the content creation for Havok Animation (creating animations and rigs) is accomplished using standard, out-of-the-box tools already provided by each one of the modellers. However, Havok extends those tools with animation-specific modeler extensions:

- *Animation Tools* (3ds Max | Maya | Softimage): How to create Havok Animation-specific data, like Annotations.

Animation content processing (filters) are covered by:

- The Havok Filter Pipeline > Animation Filters: Describes all filters related to animation.

And everything is put together in a tutorial:

- The *Animation and Export Basics Tutorial* (3ds Max | Maya | Softimage) is a very good introduction to animation export, covering the most commonly used processing tools.

If you are using Havok Physics 2012 as well, please check also the Havok Physics 2012 + Animation documentation guide.

#### 4.1.2.3 Havok Physics 2012

**Make sure you've already read**
HCT Documentation Guide > Content Creators > All Havok Products

Common concepts related to Havok Physics 2012 are covered as subtopics in the Common Concepts chapter, in particular:

- Common Concepts > Physics 2012 > Rigid Body Concepts
- Common Concepts > Physics 2012 > Constraint Concepts

Havok extends each modeler with plugins that allow you to setup and visualize physics objects:

- *Physics 2012 Tools* (3ds Max | Maya | Softimage): Describes how to create rigid body systems in the modeller.

Physics 2012 processing filters are covered in:

- The Havok Filter Pipeline > Physics 2012 Filters: Describes all filters related to physics processing.

Concepts are put in practice with a couple of tutorials:

- *Tutorial > Physics 2012 Basics* (3ds Max | Maya | Softimage): Covering the basics of rigid body and constraint creation.
- *Tutorial > More on Rigid Bodies* (3ds Max | Maya | Softimage): More advanced rigid body behavior.

If you are using Havok Animation as well, please check also the Havok Physics 2012 + Animation documentation guide.

#### 4.1.2.4 Havok Physics 2012 + Animation

**Make sure you've already read**
HCT Documentation Guide > Content Creators > All Products
HCT Documentation Guide > Content Creators > Havok Animation
HCT Documentation Guide > Content Creators > Havok Physics 2012

Users of both Havok Physics 2012 and Havok Animation can take advantage of specific functionality in the Havok Content Tools, particularly in the area of ragdolls.

Common concepts are covered in:

- Common Concepts > Physics 2012 and Animation > Local Frames

Of particular interest are the modeler tools used to setup ragdolls, described in:

- *Physics 2012 Tools > The Rag Doll Toolbox* (3ds Max | Maya | Softimage)

Two filters in particular are most relevant when setting up rag dolls:

- Physics 2012 Filters > Create Rag Doll Filter
- Animation Filters > Create Mapping Filter

Essential reading for anybody working in ragdoll setup is the following tutorial:

- *Tutorial > Rag Doll Toolbox* (3ds Max | Maya | Softimage): Goes through the whole process of setting up a ragdoll and exporting it to the game.

### 4.1.3 Documentation for Tools Programmers

The Havok Content Tools are built for easy integration into existing asset pipelines. Their architecture is also modular, allowing for easy extension and automation of both content creation and processing.

In order to successfully integrate and extend the Havok Content Tools, it is important to understand the basic functionality and workflow from a user (content creator) perspective as well. Therefore we recommend you read the sections outlined in the Documentation for Content Creators in addition to the sections mentioned here.

#### 4.1.3.1 All Havok Products

**Make sure you've already read**
HCT Documentation Guide > Introduction

It is essential to understand the basic architecture of the Havok Content Tools. This is described in:

- Introduction > Architecture: Gives you an overview of how the Havok Content Tools fit into the game content creation pipeline.
- The Havok Filter Pipeline > The Filter Manager: This is one of the main components of the Havok Content Tools, and where most content processing is set up.

A whole chapter of the Havok Content Tools documentation is dedicated to extending the Havok Content Tools:

- Integrating with the Havok Content Tools

Programmers should also read the *Havok SDK User Guide*, in particular the *Serialization* section, as well as any sections relevant to each individual Havok product. The SDK documentation is not installed with the Havok Content Tools but can be found as part of the Havok SDK packages.

## 4.2 Introduction and Architecture

### 4.2.1 Introduction

A critical component of any game production pipeline involves extracting information from the content creation tools, serializing this to an extensible stream format and efficiently loading, processing, storing and finally accessing these assets at runtime. The Havok Content Tools and Havok's serialization framework provide an extensive and extensible set of tools for asset creation, export, processing and loading. This document focuses on asset creation, export and processing. For details on asset loading (within a game), please consult the documentation for the Havok Serialization Framework.

This document first presents an introduction to the core elements of the Havok Content Tools:

- Introduction and Architecture : This section introduces the Havok Content Tools and gives an overview of how they operate and fit into an asset pipeline.
- The Havok Filter Pipeline : Presents the tools available for asset processing and serialization and how they can be combined to achieve powerful results.

Several sections are then presented covering the content creation and export tools for each supported modeler:

- 3ds Max Tools : Presents the set of Havok Content Tools for 3ds Max, in particular the 3ds Max scene exporter and the 3ds Max tools for content creation. Includes tutorials for 3ds Max users.
- Maya Tools : Presents the set of Havok Content Tools for Maya, in particular the Maya scene exporter and the Maya tools for content creation. Includes tutorials for Maya users.

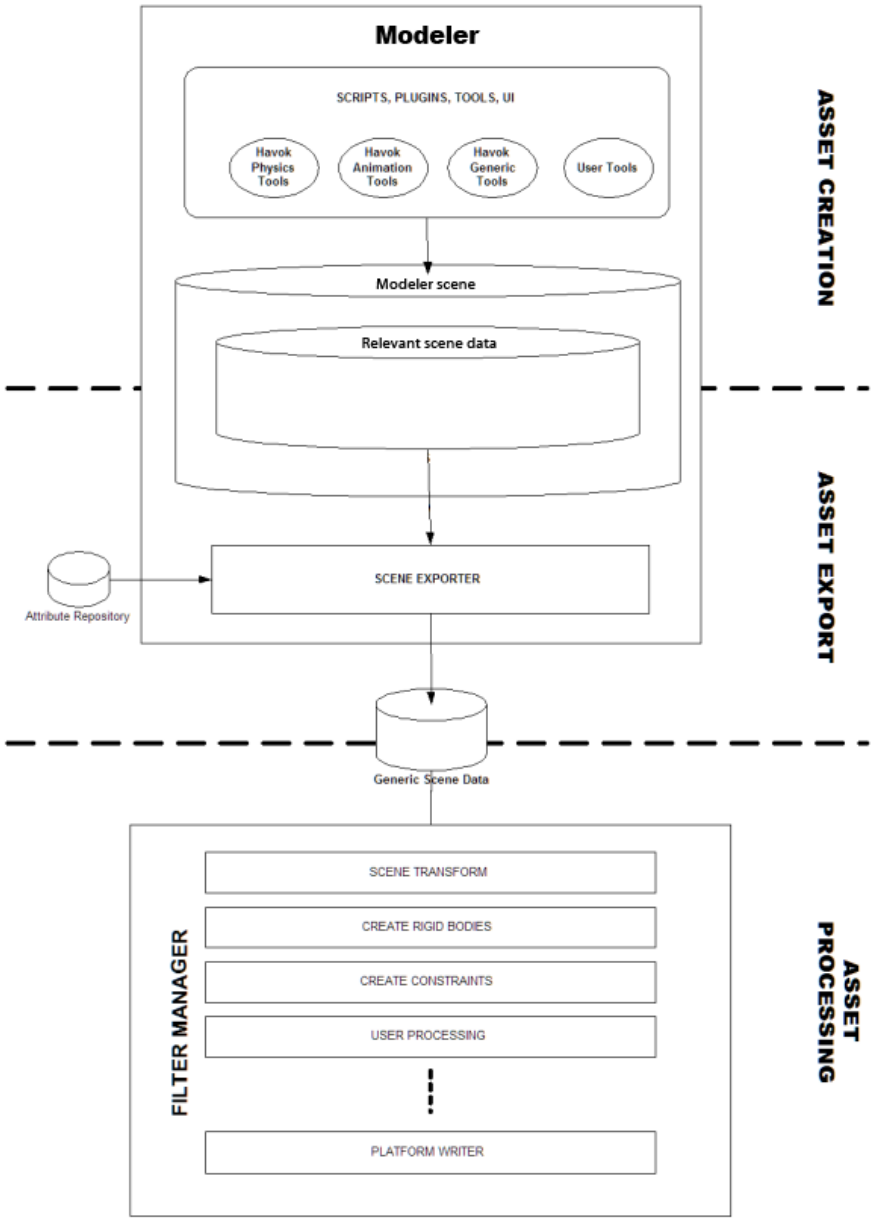
- XSI Tools : Presents the set of Havok Content Tools for XSI, in particular the XSI scene exporter and the XSI tools for content creation. Includes tutorials for XSI users. *Note that the XSI toolchain for Havok Content Tools is deprecated and as such will not be receiving product updates and bug fixes as often as the 3ds Max and Maya toolchains.*
- Common Concepts : All supported modeling packages share some concepts related to physics, cloth, destruction and animation setup, which are described in this section.

Finally, advanced topics are presented covering customization and troubleshooting of the Havok Content Tools:

- Integrating with the Havok Content Tools : A detailed guide on the interfaces exposed by each component of the Havok Content Tools and how best to extend and integrate with them.
- Troubleshooting : A list of common problems and gotchas and how to solve them.

### 4.2.2 Architecture

The following diagram outlines the architecture behind the Havok Content Tools:




The tools can be divided into three functional categories: Asset Creation/Setup, Asset Export and Asset Processing.

#### 4.2.2.1 Asset Creation Tools

These are tools created specifically for each modeler and are typically used by artists, animators or designers. Tools provided in this area therefore focus on usability, flexibility and having a good workflow integrated with the rest of the modeler's tools. These tools do not have any dependencies with Havok or external libraries, they simply set up data within the modeler in such a way as it will be interpreted in a useful way by a Havok Scene Exporter.

Examples of Asset Creation Tools are:

- Havok Physics Tools
  - Custom nodes, modifiers, parameter sets and attributes that can be added to the scene or to existing nodes in order to assign physical properties to objects.
  - Tools to manipulate and visualize otherwise complex concepts like constraint limits and spaces.
  - Higher level tools such as those to create rigid bodies for animation bones, associate them and align them.
- Havok Animation Tools
  - Currently, all supported modelers offer enough built-in support for animation concepts so we do not ship any custom animation extensions. Future examples could be tools to add: annotations to tracks; custom IK information to bones; etc...
- Havok Generic Tools
  - Tools which provide an interface to the other Havok tools, by providing toolbars, menus, etc...
- User Tools
  - Since asset creation tools only interface with the scene and do not have any dependencies with Havok or external libraries, it is very easy for users to create custom tools that sit in this area. For example, users could provide their own scripts to: automate repetitive tasks; add game-specific data; etc..

The various content creation tools provided by Havok are described in detail in the 3ds Max Tools, Maya Tools and XSI Tools sections of this manual.

#### 4.2.2.2 Asset Export Tools

The responsibility of the Havok Scene Exporters (one provided for each modeler) is to navigate the scene and extract relevant scene data, which is then converted into a generic (modeler-independent) Havok scene data format. This scene data (and how it relates to the actual modeler scene) is the key link between the asset creation, export and processing tools. In general, the scene data contains:

- The scene hierarchy (of nodes)
- The animation of each node's transform
- The meshes, lights, cameras that may be associated with each node
- The materials and textures that may be associated with each mesh
- Extra attributes that may be associated with each node (possibly animated)

The Havok Scene Exporters are described in detail in the 3ds Max Tools, Maya Tools and XSI Tools sections of this manual.

The Integrating with the Havok Content Tools section of this manual gives more details about the format of this generic scene data.

#### 4.2.2.3 Asset Processing Tools

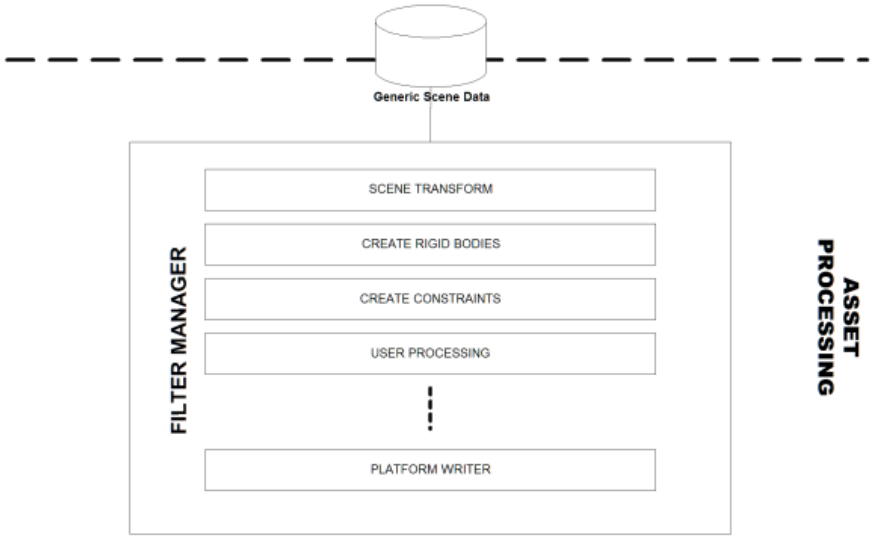
The output of a Havok Scene Exporter is passed as the input to the asset processing tools. Since this input is generic scene data, the asset processing can operate in a totally modeler-independent fashion. Once an asset processing tool (a *filter*, as described later on) has been written, it can operate regardless of whichever modeler used to generate the original content.

By combining these asset processing tools in various ways, powerful and varied results can be achieved.

The following section, The Havok Filter Pipeline, describes the individual processing tools in detail.

## 4.3 The Havok Filter Pipeline

### 4.3.1 The Filter Manager

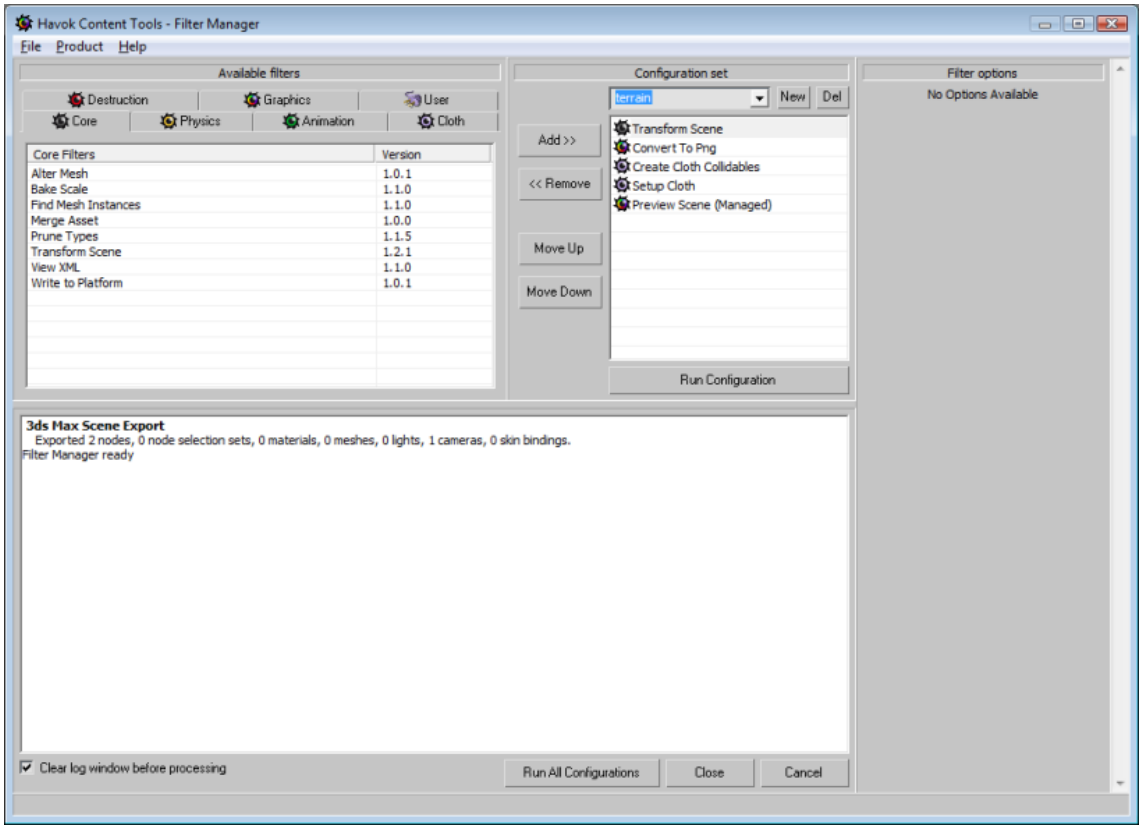



Asset processing in the Havok Content Tools is organized as a sequential combination of operators, where the output of each operator is taken as the input of the next one. These individual operators are known as *filters* in the Havok Filter Pipeline. There are multiple types of filter: some modify objects; some create new objects based on existing ones; some delete objects. Filters can also perform operations which have no effect on the content, such as previewing the scene or writing it to a file. Havok provides a large set of filters, which are individually described later in this chapter. One of the advantages of this scheme is that the small granularity and the loose coupling between filters makes them particularly easy to write and extend. The Writing Your Own Filter section at the end of this document gives more details on how to do so.

Before looking at the individual filters, let's take a look at the *filter manager*. This is the application which allows the user to select, organize and modify the options of one or more sets of filters. The filter manager is therefore the "control center" from which asset processing is managed.

#### 4.3.1.1 The Filter Manager Interface

The following interface is presented to the user after running any of the Havok Scene Exporters (or from the standalone filter manager, described in the following section):




The interface is divided into several panes:

- **Available Filters** : Shows a list of the available filters, grouped into the following categories:
  - **Core** : Filters that operate on generic scene objects (that are not related to either physics or animation).
  - **Physics** : Filters that create or operate on physics objects (rigid bodies, constraints, etc.).
  - **Animation** : Filters that create or operate on animation objects (skeletons, skin, motion, etc.).
  - **Graphics** : Filters related to graphics elements (materials, etc.). Also contains the scene previewer.
  - **User** : This category is left open for custom filters written by users.
- **Configuration Set** : Lists the set of configurations and the filter setup for the currently selected configuration. Filters are executed in the order that they appear in the list.
- **Filter Options** : Many filters have options which affect the filter's behavior. Whenever a filter is selected in the filter configuration list, any options it may have are shown in this pane.

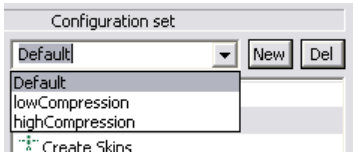
- **Log Window** : This is used to report messages, warnings, etc. while setting up the filter configurations and while processing. Any serious messages are reported in an orange/red color; if any such messages occur they should be investigated further.

There are also some buttons in the filter manager UI:

- **Add** >> : Adds the filter currently selected in the *Available Filters* list to the current filters list. Double-clicking on a filter performs the same operation.
- << **Remove** : Removes the current filter selected in the list. Pressing **DEL** performs the same operation.
- **Move Up** : Moves the currently selected filter in the list one slot up. Pressing the " **-** " key performs the same operation.
- **Move Down** : Moves the currently selected filter in the list one slot down. Pressing the " + " key performs the same operation.
- **Run Configuration** : Executes the current configuration.
- **Run All Configurations** : Executes all configurations consecutively.
- **Close** : Saves all changes to the configurations and closes the filter manager.
- **Cancel** : Closes the filter manager, prompting for confirmation to save any changes made to the configuration(s). Pressing **ESCAPE** , or closing the window performs the same operation.

**Configuration Sets**

A set of filters in a specific order and with specific options is known as a *configuration* in the Havok Filter Pipeline. It is possible to have multiple filter configurations associated with a single asset. The **Configuration Set** pane contains a dropdown list showing the active configuration, of which there is always at least one by default:




Each configuration contains its own set of filters, and operates on its own copy of the asset data. This allows the same original asset data to be processed in different ways without having to repeatedly change the filters being used. For example, multiple configurations can be used to process animation data using different amounts or types of compression.

To change the active configuration, simply select another from the dropdown list. To rename a configuration, edit its name using the box. To delete a configuration, press the **Del** button. To add a new configuration, press the **New** button.

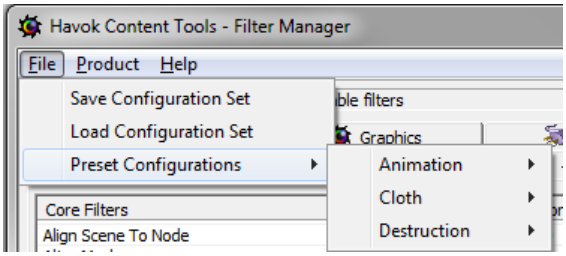
Whenever a new configuration is added, it becomes the active configuration, and begins as a copy of the previous configuration.

**Saving & Loading Configuration Sets**

When invoked from a modeler, the filter manager configurations are automatically saved as part of the modeler asset file. In addition, configuration sets can be saved and loaded into a standalone file (also called an "options file", or .hko file), using the **File** > **Save Configuration Set** and **File** > **Load Configuration Set** menu options.

This allows the same filter setups to be used again and again, or tweaked as desired and resaved. They can even be used to override or upgrade the filters saved with individual assets. See the section on Overriding and Upgrading Filter Settings for more information on how to do this.

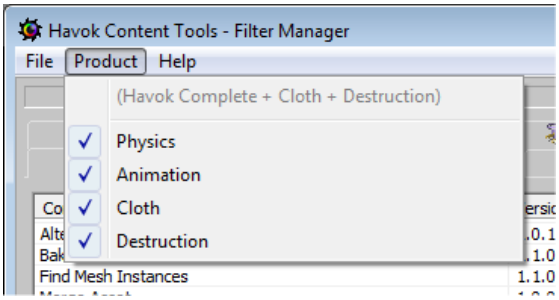
Additionally, commonly used configurations have been provided for various Havok products and can be loaded using the **File** > **Preset Configurations** menu options:



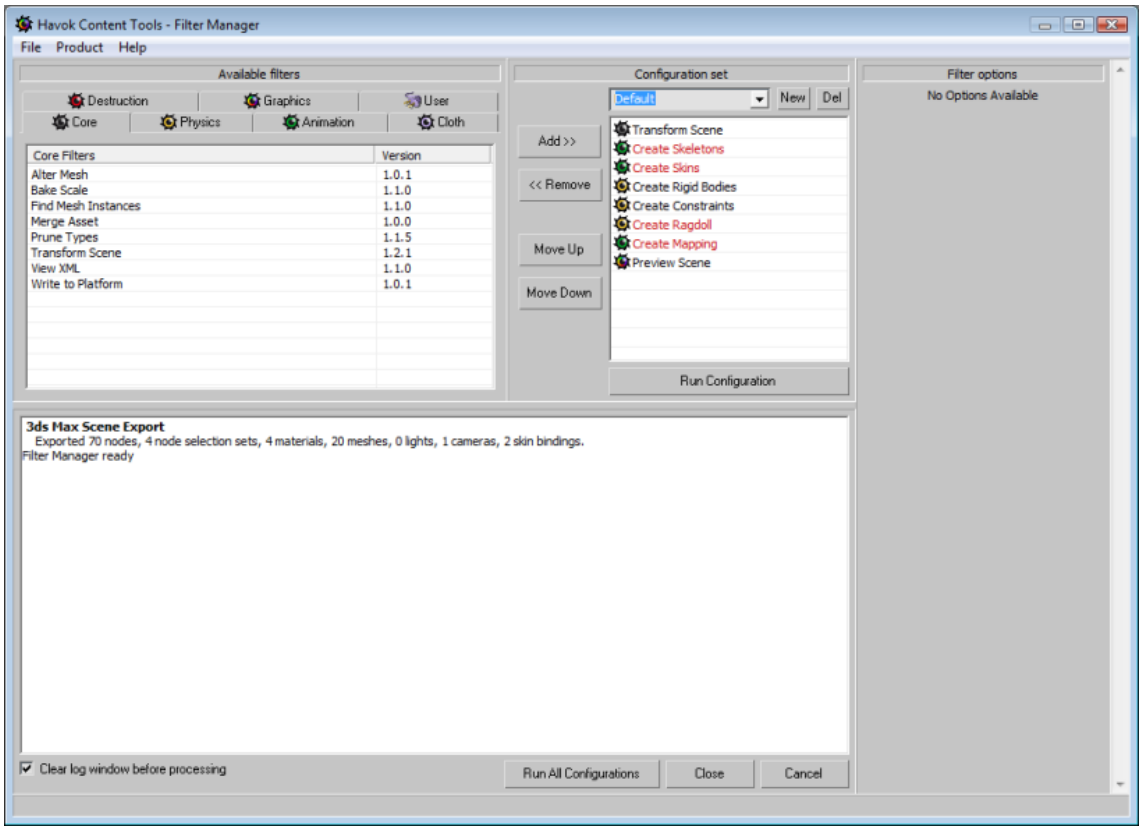

### Product Selection

Filters produce output that may depend on different Havok SDK products to allow them to be loaded at runtime. For example: Scene Transform works for all Havok products; Create Bodies generates objects which require the Havok Physics/Complete SDK for loading at runtime; the Create Ragdoll filter generates objects which require both the Havok Physics and Animation SDKs for loading at runtime.

All filters are always available for use within the filter pipeline, regardless of the runtime product which your company has licensed. Therefore, it is useful to be aware of which filters may be unsupported by your Havok product. You can select your specific Havok product by using the **Product** menu in the filter manager:




This product selection is stored in the registry and therefore is a user setting–its value remains between each session of the filter manager. Filters which are not supported at runtime by the selected product will appear in red in the filter manager, indicating that they should not be used:




On execution of any setup which includes runtime-unsupported filters, a warning will appear in the log window for each unsupported filter.

#### 4.3.1.2 The Standalone Filter Manager

The filter manager is implemented as a DLL; each modeler invokes the same filter manager through the DLL interface. It is also possible to invoke the filter manager through a standalone Havok application, known as the *Standalone Filter Manager*.

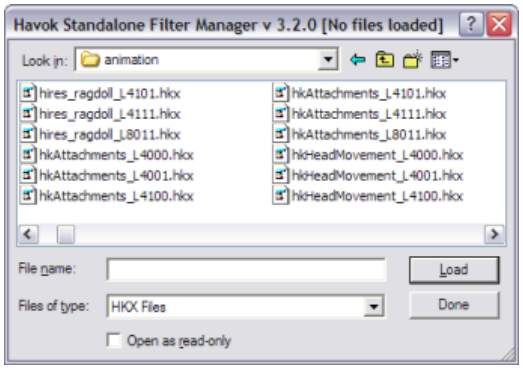
The input data for this application is taken from one or more Havok serialized (`.hkx`) files. These files are usually the output of some processing previously performed by a filter manager invoked from a modeler.

The standalone filter manager has many possible uses:

- It can be used as a "preview" tool (by loading an asset and just using the Preview Tool filter).
- It can be used to batch process assets without the need to use a modeler.

**Using the Standalone Filter Manager in Interactive Mode**

To use the Standalone Filter Manager in interactive mode simply launch the associated executable (a shortcut is placed in the Start menu by the Havok Content Tools installer). A splash screen will appear for a short period and you will be presented with a file open dialog box.




Select a single file or multiple files to load and press the **Load** button. Each time you press the **Load** button the title in the dialog is updated to reflect the number of files selected for loading. Once you have successfully selected one or more files press the **Done** button to complete the loading process.

The files are loaded in turn using our standard serialization. If any file fails to load a warning is printed in the console window.

Successfully loaded files are then merged together into a single root level container and this is passed to the filter manager. The filter manager should now appear and interactively allow you to adjust the current filter stack.

### Default Options

When `.hkx` files are loaded into the Standalone Filter Manager there are no filter configurations available, so a default configuration set is used instead. This configuration, called 'HKX Preview', contains a single Preview Tool filter, allowing the user to view the `.hkx` file contents.

This default setup is stored in an XML file called "`defaults.hko`", placed in the same folder as the Standalone Filter Manager. This allows the default options to be modified or replaced as necessary. See the HKO Files section for more information on creating or modifying options.

Once loaded by the Standalone Filter Manager, the filter configurations can be edited and/or processed as per usual.

### Error Codes

The Standalone Filter Manager will return a code in both batch and interactive mode. It will always return greater than or equal to `0` for a successful state. The following codes can be returned:

- `-2`: Input file failed to load
- `-1`: Filter manager failed
- `0`: Success
- `1`: Success with warnings

### Command Line Arguments

The Standalone Filter Manager can also be launched in command line mode. Its usage is as follows:

```
USAGE: hctStandaloneFilterManager.exe
        [-s settingFilename.hko -p assetPath -o outputPath -i
        -l logPath.txt -t -v1] file1.hkx [fileN.hkx]
```

This launches the Standalone Filter Manager in command line mode. The specified files (`file1..N`) are loaded and merged before being passed to the filter manager. The filter manager is automatically invoked and processing begins in batch mode. If you wish to specify files on the command line and still run the previewer in interactive mode, you can use the *-i* flag to force interactive mode.

The optional *-s* flag specifies a filter set (i.e., a `.hko` file) to use. `.hko` files are usually created by launching the tool interactively, creating one or more filter stacks and choosing the 'Save Configuration Set' option. If this flag is omitted the default options file will be used.

The optional *-p* flag specifies the asset path, described further in the following section.

The optional *-o* flag specifies the output path. If not specified, the current working directory is used.

The optional *-i* flag forces interactive mode.

The optional *-l* flag specifies the log file.

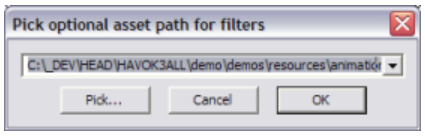
The optional *-t* flag specifies whether the output from the filter manager is printed to the standard output or not.

The optional *-v* flag specifies the level of verbose output to the log file (0: No log written; 1: Warnings and Errors only; 2: Full output is written).

The main reason for using the command line mode is to efficiently process assets and create `.hkx` files. To this end, the filter set should contain a Platform Writer filter using `\$(output)` in its **Filename** option so that it takes the output path specified by the *-o* flag into account.

### Asset Paths

When `.hkx` files are created using relative paths in the filters, they are relative to the asset path. Thus if you load several files you have several asset paths to choose from, or if you have an `.hkx` file that should not use its asset path for this run, then you can specify your path. The asset path dialog pops up after you have chosen your file set with the list of asset paths contained in those files. You can choose one of them, edit them, or just browse for one if you can't remember it. If you choose **Cancel** , the pipeline will run with an empty asset path, so any relative paths will be in relation to the current working directory.




### Examples

The Standalone Filter Manager was originally designed to allow you to view `.hkx` files without having to resort to loading the original art asset in the modeler. However, since it uses the full power of the filter manager it can also be used directly in the tool chain as an asset processing tool. Some use cases would

be:

- Use a View Xml filter to verify that the expected file elements are present in the asset.
- Load separate rigs, animation and skin data to verify that they work together.
- Use any of the motion extraction or compression filters on raw animation to test their results in isolation.
- Batch process assets to transform a scene from a left-handed to right-handed system.
- Batch process XML production assets to produce final game assets in binary-platform-specific formats.

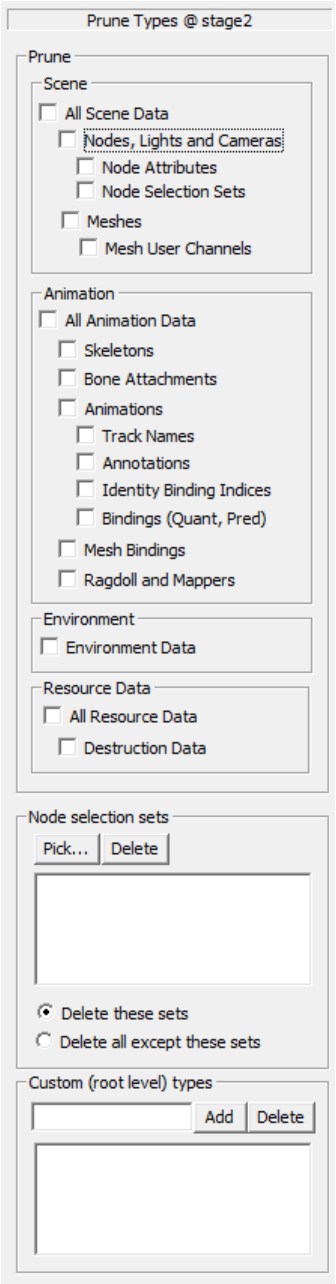
### 4.3.2 Core Filters

Core filters are those which generally modify or examine basic scene data. They can always be used regardless of which Havok Product is licensed for use at runtime.

#### 4.3.2.1 Prune Types

**Components Required** : Common

This filter is used to remove subsets of data by type, so they don't get exported, previewed, etc. It is useful to reduce the size of the exported file by removing unnecessary object types.




Here you can select some generic types of data to be removed from the scene:

- **All Scene Data** : Fully removes the `hkxScene` object.
  - **Nodes, Lights and Cameras** : Removes nodes, lights, cameras, materials and textures, retaining some basic scene information.
    - **Node Attributes** : It removes all attributes associated to nodes (`hkxNode`).
    - **Node Selection Sets** : It removes all node selection sets.
  - **Meshes** : Removes mesh data (meshes and skin bindings).
    - **Mesh User Channels** : Removes any vertex/edge/triangle selection data associated to meshes.

- **All Animation Data** : Fully removes the `hkaAnimationContainer` object, ragdoll instances and skeleton mappers.
  - **Skeletons** : Removes any skeletons (`hkaSkeleton` objects).
  - **Bone Attachments** : Remove any bone attachments (`hkaBoneAttachment` objects).
  - **Animations** : Removes any animations (`hkaAnimation` objects) and their associated bindings.
    - **Track Names** : Removes track names from the animations.

      **Warning:**

      Do not check this option if you plan to bind animations to skeletons at runtime (since track names are required to do so).
    - **Annotations** : Removes annotations from the animations.
    - **Identity Binding Indices** : Removes the index arrays of a binding if the mapping is equivalent to the identity (but keeps the binding). At runtime these arrays will be implicitly read as if they were an identity mapping.
    - **Bindings (Quant, Pred)** : Removes the `hkaAnimationBinding` instances associated with instances of `hkaQuantizedAnimation` and `hkaPredictiveCompressedAnimation`. Those animation types don't need a binding because they are represented in bone space.
  - **Mesh Bindings** : Removes any mesh bindings (`hkaMeshBinding` objects).
  - **Ragdoll and Mappers** : Removes any ragdoll instances and skeleton mappers (`hkaRagdollInstance` and `hkaSkeletonMapper` objects).
- **Environment Data** : Removes any `hkxEnvironment` object. Note that these objects are ignored by the Platform Writer filter, so it is usually not necessary to prune them.
- **All Resource Data** : Completely removes the "Resource Data" subtree.
  - **Destruction Data** : Removes only the Destruction-related objects from the "Resource Data" subtree.
- **Node selection sets** : Allows you to add a list of user-defined scene node selections to be removed. Such selections can be easily created in modelers (named as "Selections Sets" in Max, "Sets" in Maya and "Groups" in XSI).
- **Custom ( root level ) types** : Allows you to add a list of custom type names (like `hkpPhysicsData` or `hkaRagdollInstance`). The filter will remove any objects of the given types found inside the named variants at root level (check Scene Data for details).

A common usage of the Prune Types filter is to separate out static data which is common to a single character or scene from corresponding dynamic data. For a single character a user may wish to export many animations separately, without duplicating the skeleton (rig) data in each exported file. This can be easily accomplished by first creating a Prune Types filter which prunes all data, *except* the skeleton information, and exporting just the skeleton information to a common file; next each animation may be exported in turn by pruning everything *except* the animation data. The resulting files will be smaller than if the skeleton data were included in each exported animation.

**Warning:**

Care must be taken to prune only and all the object data not needed. Please see the section Controlling File Size for how to produce files of minimal size.

#### 4.3.2.2 Merge Asset

**Components Required** : Common

This filter takes an HKX file, and merges its contents with the current asset. This is useful for example if you are exporting a rag doll and would like test its behavior in some already exported landscape.

The path to a HKX file which is to be merged should be specified in the filter options.

#### 4.3.2.3 Bake Scale

**Components Required** : Common

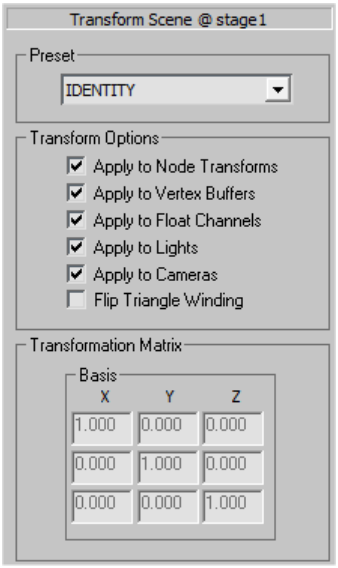
This filter iterates through each of the exported scene nodes, removing any scale found in the transform and applying it to the associated mesh/skin (if any). The scaling is removed by decomposing the original transform. This usually means that changes in scale are converted to changes in child node positions.

Use this filter when your exported assets should not include scaling. This filter can be useful in order to associate meshes to rigid bodies (since rigid bodies do not have scale). The animation system gracefully handles scaling.

#### 4.3.2.4 Transform Scene

**Components Required** : Common

Often, the modeler and the runtime/renderer will use different conventions for scale and handedness. This filter transforms your asset to a different coordinate system, or to a different unit system, for example.




The filter allows you to choose the types of data you wish to operate on. Usually you will select all of the ' **Apply** ' options. In general when you perform a change in handedness (say using MIRRORING) you should also check the **Flip Triangle Winding** box.

The filter has a number of presets:

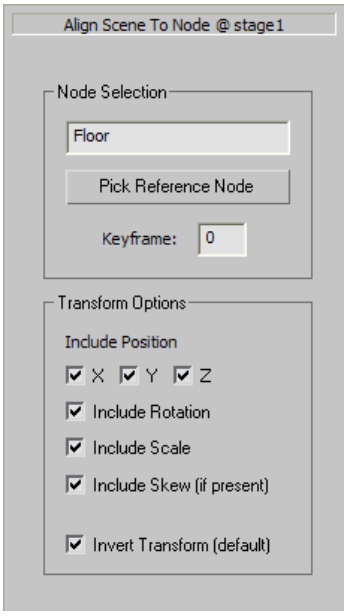
- `MIRROR_(X/Y/Z)`: Flips data
- `SCALE_INCHES_TO_METERS` (Usually used for Max, which models in inches by default)
- `SCALE_CMS_TO_METERS` (Usually used by Maya, which models in cms by default)
- `SCALE_FEET_TO_METERS`
- Custom: This option allows you to specify the complete transformation matrix to be applied to the scene. The matrix is pre-multiplied.

This filter will transform the scene and therefore affect the input of all filters proceeding filters in the pipeline, so it should generally appear as the first filter in a given configuration. For example, if this filter sits at the top of the stack then `MIRROR_X` will flip the nodes which will in turn ensure the Create Animations filter creates a mirrored animation.

#### 4.3.2.5 Align Scene To Node

**Components Required** : Common

This filter provides the ability to transform the scene by placing a chosen node at the origin.




Each node in the scene is transformed by the inverse world space transform of the selected node. This effectively places the chosen node at the origin (the original transform and its inverse cancel out) and all other nodes move rigidly with it.

Users may specify the node name directly, or choose it from the scene hierarchy by choosing the "Pick Reference Node" button. The node may be at any level of the scene hierarchy. A keyframe number may be entered, specifying the time at which the node's transform will be sampled.

Selected components of the transform may also be enabled or disabled:

- Position (X/Y/Z)
- Rotation
- Scale
- Skew (if present)

Additionally the inversion of the transform may be disabled. This will apply the direct world space transform of the selected node to the entire scene.

The Align Scene To Node filter may be used any time a transform needs to be applied to the scene for further processing, such as aligning a character to a canonical axis.

**Application to Animation and Extracted Motion**

The Align Scene To Node filter may be useful for positioning and orienting animations. At times the animator may create an animation at an arbitrary position and orientation in space. If a node exists in the scene representing the projection of the character on the ground, the Align Scene To Node filter can be used to place this node at the origin and orient the character along a canonical axis. The "ground projection" node may be or not be part of the character's bone hierarchy.

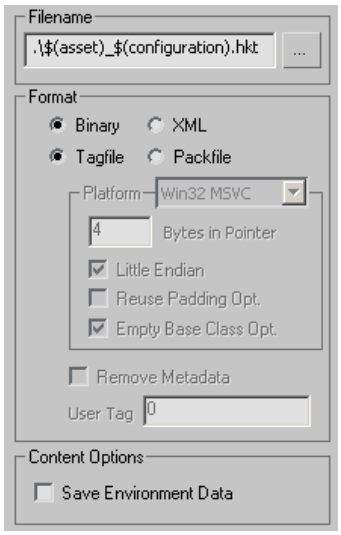
The Align Scene To Node filter can be used if an Alternative Reference is specified in the Extract Motion Filter and the animation does not begin at the origin.

Note that there is some overlap in functionality between the Align Scene To Node filter, the Rotate Animation filter and the "Move to origin" option of the Create Animations filter. It is the users responsibility to ensure that later filters do not improperly override the work done by the Align Scene To Node filter.

#### 4.3.2.6 Write To Platform

**Components Required** : Common

This filter allows you to write out the contents of the stream as raw XML, a platform-specific binary format, or platform-independent portable tagfiles (the default). The target filename together with the target platform details are selectable in the options.




- **Filename** : The filename specifies where to dump the stream contents. Use the (...) button to open a standard file save dialog and specify the filename. NOTE: The path for the filename is always stored relative to the original asset. This means that if the asset is moved the output will be placed in a sensible relative location. This makes transferring assets between machines and batch processing far easier.

  You may use environment variables in this string. The strings '`\$(asset)`' and '`\$(configuration)`' are usually defined. Any instances of '`\$(asset)`' found in the filename will be replaced with the asset's name (the file extension is removed). Similarly, any instances of '`\$(configuration)`' will be replaced with the configuration's name. If an object was selected in the modeler before exporting, then a '`\$(selected)`' string is also available. Of course, you may define your own environment variables within the modeler. See the relevant modeler's export options for more.

  In the cases where an environment variable may or may not be defined, you can also use the '`\$(variable:default_value)`' format; if the variable is not present, then the specified default value will be used.

  Environment variables are useful for creating files when batch processing assets. Each write to platform belonging to a different asset/configuration can easily construct a unique filename using these variables.

- **Format** : This allows you to specify the output format. You can choose between human-readable XML or binary format. When Tagfile or XML is selected all other options in the "Format" group will be disabled as they have no influence. If Packfile and Binary are selected the controls will be enabled to let you specify platform specific settings. The controls are:
  - **Platform** : The binary target is both platform and compiler dependent (since some compilers do different types of optimization). The "Bytes in Pointer", "Little Endian", "Reuse Padding Opt." and "Empty Base Class Opt." will be automatically set when choosing a platform, unless "Custom" is chosen.
  - **Remove Metadata** : This option will remove all `hkClass` data from the output file, reducing the file size. Notice that some variants will have to be interpreted by the user since the associated `hkClass` won't be available. Check the Serialization section in this manual for details.
  - **User Tag** : This value is an optional tag (an integer value) that can be associated with binary files (it is stored with the header of the binary file).

    **Note:**

    The proposed file extensions are `.hkt` and `.hkx` for tagfiles and packfiles respectively. The filter's UI will try to match the file extension with the file format if appropriate, i.e. if the filename for a tagfile is `myfile.hkt` it will be changed to `myfile.hkx` when selecting a packfile format and vice versa. If you choose a different file extension it won't be modified in either case.

- **Save Environment Data** : This option will include any defined environment variables in the output file. This is usually not required since the environment variables are generally used within the filter pipeline only.

**Warning:**

The final *file size* of the asset when written may not represent the minimal asset size in memory at runtime). Please see the section Controlling File Size for how to produce files of minimal size.

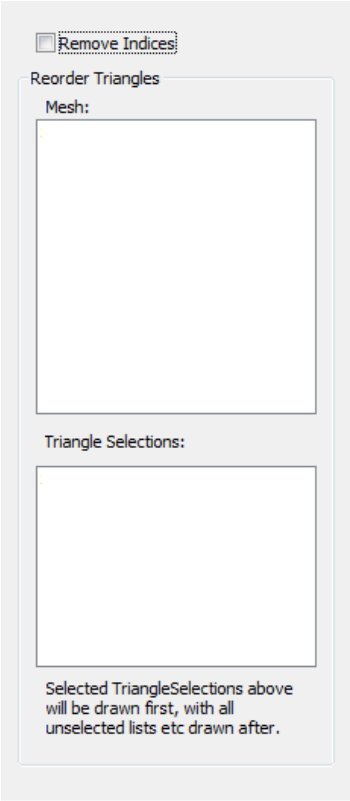
#### File Conversion Utilities

Unlike tagfiles, packfiles are platform-specific and therefore need to be created for each platform being supported. This can be done by using the **Write to Platform** filter several times in a row, or alternatively can be done from the command line using the Python scripts and/or `AssetCc` utilities provided in the `BatchProcess` subdirectory of your Content Tools installation. These utilities can also be used to convert tagfiles to packfiles, and vice versa. See the Serialization chapter of the Havok Common user guide for more information on the different file formats.

### 4.3.2.7 Alter Mesh

**Components Required** : Common

This filter can alter the mesh layout. It can remove triangle indices from meshes, and rewrites the vertex position data using a triplet for each triangle which is useful for rendering engines and platforms that do not support triangle indices. It can also reorder the triangles such that given selections are first in the mesh, which is useful for enforcing triangle draw order to preserve alpha effects, or reduce needless overdraw.

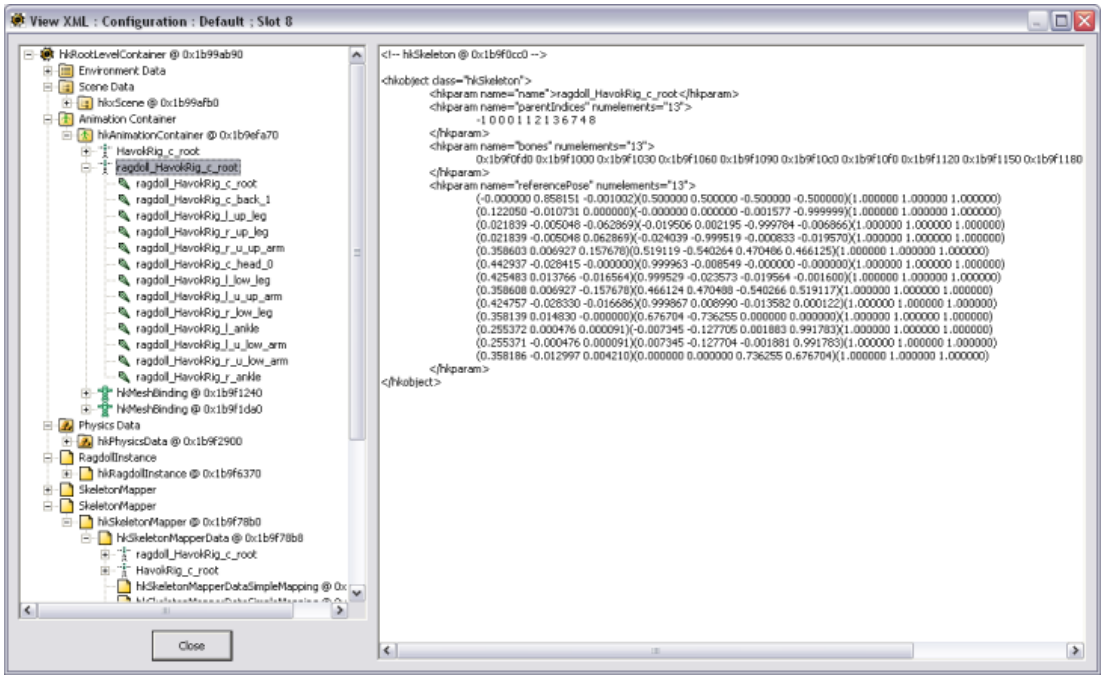



The triangle reordering works by looking for triangle selection sets that were exported from the modeler. The meshes are all listed in the first list box, and when you select a mesh its selection sets are shown in the lower list box. Select the selection sets you wish to be first. The order is per mesh, so just select each mesh you wish to alter and its corresponding preferred first sets. If you select multiple triangle selection sets for a given mesh, the order will be in the order seen in the list box, so will depend on the export order.

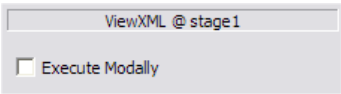
#### 4.3.2.8 View XML

**Components Required** : Common

This filter, when executed, launches an interactive dialog which allows you to navigate the current data in the filter pipeline. Selecting any item displays an XML description of its contents.




This filter is very useful when you want to check the contents of a scene at any stage during processing. Since it uses Havok's serialization infrastructure, it can handle any data, even if it contains custom classes which have been created by custom filters.




By default the filter executes in a non-modal mode, that is, processing is not halted when the View XML window is opened. This allows you, for example, to have multiple View XML windows open at the same time.

Selecting the **Execute Modally** option will make the View XML operate modally; that is, processing will be halted until the View XML window is closed.

#### 4.3.2.9 Find Mesh Instances

**Components Required** : Common

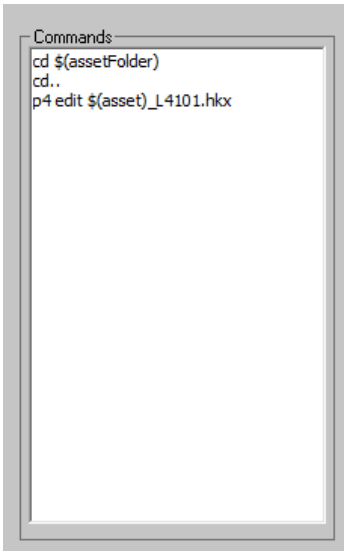
This filter examines all meshes in the scene data, looking for those which contain the same data. Any meshes which are considered to be duplicates of each other are replaced by a single instance of the mesh, and any pointers referencing the meshes are updated. By removing the duplicate mesh, the file size is kept down.

This is useful if you are using instancing/copying of objects to set up a scene in the modeler.

#### 4.3.2.10 Execute Command

**Components Required** : Common

This filter provides a command-line interface to allow the execution of a series of arbitrary commands. The command line output is redirected to the log window and an error value is outputted after a command fails. This filter could be used if you need to check out an asset before writing to it and another filter could be used to commit the changes afterwards.

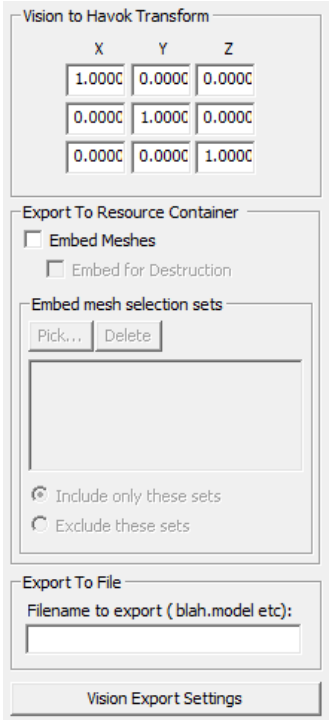



Each command string should be entered on a new line and you may use environment variables within the string (see the section on the Write to Platform Filter for more information on how to do this).

#### 4.3.2.11 Process Vision Data

**Components Required** : Common

This filter attempts to create Vision meshes (i.e. `VDynamicMesh`) for all graphics nodes in the scene. The Vision meshes are stored as `hkxBlobMeshShape` objects in a new resource container called `Vision Meshes`. Note the initial mesh data is not removed from the asset file; if you only need the Vision mesh data you should add a `Prune Types` filter to remove the original meshes. Make sure the *Process Vision Data* filter is placed and executed after any other filter that could potentially generate new meshes.




The filter features a **Vision to Havok Transform** although the recommended way of transforming the scene is to use a Scene Transform filter. This is because soon the **Vision to Havok Transform** will soon be declared deprecated and removed. A Scene Transform filter with the correct transformation must be placed before the **Process Vision Data filter** in order to transform the data correctly into Vision space from modeller space.

Clicking on the **Vision Export Settings** button will bring up the same options as to what you would get if you used **File** > **Export All** method.

Please refer to "Exporters - Getting Started" in the Havok Vision documentation for full details of the **File** > **Export All** method, and make sure to read the "Using Havok Content Tools To Migrate And Export Vision files" section if you are migrating from vExporter2 to Havok Content Tools.

### 4.3.3 Physics Filters

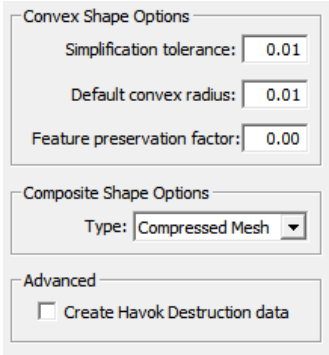
The Havok Physics Content Tools allows you to export Physics data in your asset files through the Havok Content Tools filter pipeline. See the modeller plugins documentation to see how to create Physics Data in the modeller for export.

#### 4.3.3.1 Create Bodies

**Components Required** : Havok Physics

The Create Bodies filter is the main filter for Havok Physics assets creation. Its main responsibility is to create bodies and shapes from the modeller data. The filter will take all nodes having a Rigid Body attribute in the modeller and create their associated Shape, Motion, Material and Motion Properties. The

result will be serialized into a `hknpPhysicsScene` containing one `hknpPhysicsSystem` (and thus bodies, motions, etc. construction informations) and the created shapes. See *Physics Systems and Scenes* in the Havok Physics manual for more information on these containers.




**Options**

The filter's options mainly control the way shapes are created.

**Default convex radius** is the value (in meters) of the convex radius that will be applied when building convex shapes (see Convex Shapes). It is possible to override this value in the modeller to set a custom convex radius per shape. Note that convex radius does not apply to capsules and spheres shapes. (Default value: 1 cm)

**Feature Preservation Factor** is a parameter between 0 and 1 that controls how convex shapes are created. When creating a convex shape, the original convex hull is shrunk to ensure that the resulting shape (with convex radius) will fit in the original hull (see the Convex Shapes section in the Physics Manual). However, for shapes with acute corners, the vertices can be displaced by a large distance, which can "soften" these edges and noticeably change the aspect of the shape. Setting this parameter to 1 will try to reduce the convex radius to keep the vertex displacement under the initial radius. Values between 0 and 1 linearly blend between these two extremes. (Default value : 0)

**Simplification tolerance** specifies the maximum allowed error between the simplified hull and the unsimplified hull of the input vertices. (Default value: 1 cm)

**Composite shape type** allows you to choose which kind of composite shape will be created by the filter. A composite shape is automatically created when several shape nodes are under a Rigid Body node in the modeller (created using e.g. the "Create Compound" or "Rigid Body with Proxy" tool in the modeller). In that case they are grouped into a composite shape. Available choices are Compressed Mesh, Static Compound or Dynamic Compound. Please refer to the Composite Shapes section to learn more about these shapes. Note that when creating a Compressed Mesh, all meshes (i.e. non-convex shapes) from the modeller are merged into one mesh shape. (Default value: Compressed Mesh)

**Create Havok Destruction Data** pre-processes the Destruction information and creates the Destruction Breakable shapes. This option is required only if you want to use the Havok Destruction filters. Please consult the Havok Destruction manual for more information. (Default value: false)

#### 4.3.3.2 Create Constraints

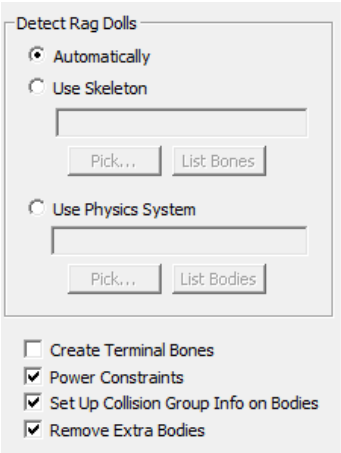
**Components Required** : Havok Physics

The Create Constraints filter creates constraints between Physics bodies from the modeller data. The filter requires Physics bodies to be present, so you need to run the Create Bodies filter before the Create Constraints filter, or have the bodies already created in the input data. This filter has no options.

#### 4.3.3.3 Create Ragdolls

**Components Required** : Havok Physics + Havok Animation

The Create Ragdoll filter allows you to create a Ragdoll, which bridges Havok Physics and Animation together. This filter requires bodies and constraints to be created (so you will have to run Create Bodies and Create Constraints before creating ragdolls). This filter will replace the Physics System Data generated by the previous filters by a Ragdoll Data which links the bodies and constraints to an Animation skeleton.




**Options**

**Detect Ragdolls** selects how the filter attempts to find what should be used to create a ragdoll in the existing data. (Default value: Automatically)

- **Automatically** : Attempt to create a ragdoll out of every physics system present in the data (note that if you used the Create Bodies and Create Constraint filters there should be only one system). It will create one skeleton bone per rigid body and arrange them in a hierarchy that mirrors the hierarchy formed by bodies and constraints.
- **Use Skeleton** : Create a ragdoll based on an existing skeleton (created using the Create Skeleton filter in Animation), selecting the physics system by matching the skeleton bones' names with the body names. It is usually the most useful option if the rigid bodies were originally created from a hierarchy of nodes (as they usually are).
- **Use Physics System** : Create a skeleton from the given physics system based on the body and constraint hierarchy in that system.

**Create terminal bones** will create extra bones at the end of the skeleton hierarchy (e.g. hands, feet and head). (Default value: false)

**Power constraints** adds a motor to power the constraints (which is useful for blending with animation, e.g. for driving the ragdoll to a specific pose). Note that only Ragdoll and Hinges constraints can be powered. (Default value: true)

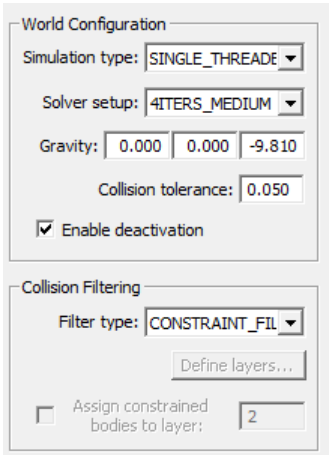
**Set up collision group info on bodies** automatically marks bodies with a collision group filter information so constrained bodies won't collide with each other. (Default value: true)

**Remove extra bodies** deletes bodies that were in the physics system used to create the ragdoll but were not matched with any animation bone. Use this option to remove objects in the scene that are not part of the ragdoll (e.g. ground or landscapes) when you are exporting it. (Default value: false)

#### 4.3.3.4 Create World

**Components Required** : Havok Physics

The Create World filter allows you to override the default options when constructing the Physics world at runtime.




**Options**

**Simulation type** allows you to configure how the simulation is run. (Default value: Multi-threaded)

- **Single Threaded** : Forces the physics simulation to run on one thread.
- **Multi-threaded** : The simulation will run on several threads using a dynamic space splitter to distribute the task to the threads.
- **Grid** : The simulation will run on several threads using a fixed space splitter to distribute the task to the threads.

**Solver setup** configures the solver, allowing you to choose the number of sub-iterations and the "hardness" of the solver. See the Solver Details section in the Physics Manual. (Default value: 4ITERS_MEDIUM)

**Gravity** sets the value of the gravity vector. (Default value: (0, 0, -9.81))

**Collision tolerance** is the minimum distance at which speculative contact points are generated. A higher distance means less potential tunneling but added CPU cost. (Default value: 5 cm)

**Enable deactivation** allows bodies at rest to get deactivated by the engine, greatly reducing the cost of the simulation. (Default value: true)

**Filter type** sets the collision filter type. (Default value: constraint filter)

- **No filter** : No collisions will be filtered.
- **Constraint filter** : Disables collision between bodies participating in the same constraint.
- **Group Filter** : Disable collisions between groups of bodies (you need to set a per-body collision filter info). The Define layer... button will allow you to set which groups collide against which.

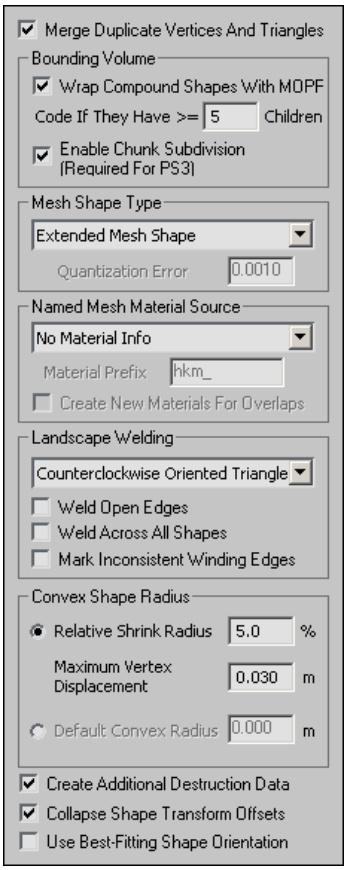
### 4.3.4 Physics 2012 Filters

Physics 2012 filters are those which create/modify Havok Physics 2012 objects. Therefore these should generally only be used if your company has licensed a Havok product which includes the Physics 2012 component.

#### 4.3.4.1 Create Rigid Bodies

**Components Required** : Havok Physics 2012

This filter interprets rigid body data in the scene data (usually set up using the Havok Physics 2012 Tools within a modeler) and creates appropriate runtime rigid body and shape objects based on that data.




If the " **Merge Duplicate Vertices and Triangles** " option is enabled, the filter will look for duplicated vertices (and triangles) in meshes and ignore them when creating Havok shapes. This reduces the memory overhead and improves runtime performance of collision detection.

In the **Bounding Volume** section you can choose to wrap shapes with MOPP code:

- Using the " **Wrap compound shapes with MOPP Code** " option will tell the filter to use Havok MOPP technology with compound shapes with a high number of children (the threshold being specified by the user), which in many cases may speed up simulation.
- The " **Enable Chunk Subdivision** " option tells the MOPP compiler to organize the MOPP code into smaller chunks so it can be processed on the PlayStation®3 SPU.

In the **Mesh Shape Type** section, you can choose between the following mesh types when exporting meshes:

- " **BV Compressed Mesh Shape** " (`hkpBvCompressedMeshShape`). A memory-efficient mesh shape that can store meshes and convex pieces in one shape. For compound rigid bodies, the filter will add any convex shapes belonging to the rigid body to the BV compressed mesh shape. In addition, this shape manages its own bounding volume information and does not need MOPP code to be built around it.
- " **Extended Mesh Shape** " (`hkpStorageExtendedMeshShape`). Extended mesh shapes are a bit more memory-intensive than simple mesh shapes (below), but they can also hold convex shapes

alongside the landscape information. For compound rigid bodies, the filter will add any convex shapes belonging to the rigid body to the extended mesh shape.

- " **Simple Mesh Shape** " (`hkpSimpleMeshShape`).

Mesh shapes can retain material information from the modeler to various extents depending on the mesh type selected:

- BV compressed meshes can store a different material name per primitive (triangle or convex piece), each primitive containing an index to a string table storing the material names within the class.
- Extended meshes can also store material information in a palette, but only from triangle primitives.
- Simple mesh shapes can only store a material index in the triangles.

For more information about the mesh shapes, please check the *Havok Physics 2012 SDK documentation*.

In the **Named Mesh Material Source** section you select the material name source from the drop-down list:

- If the " **Mesh Section Materials** " option is selected, the filter will use the materials from mesh sections as named mesh materials. Multiple materials can be assigned to a mesh in Max by defining groups of faces with the same SubID, and applying a Multi/Sub-Object material; in Maya by connecting face sets to shading groups which define the materials and in XSI by defining polygon clusters and assigning materials to them.
- If the " **Triangle Selection Sets** " option is selected, the filter will use triangle selections from Havok Mesh Channel Toolbox as named mesh materials. You may also specify a " **Material Prefix** " for triangle selection names so that only those selections with the given prefix will be exported. To mark triangles with a mesh material, use the Triangle Selection Tool in Max, the Face Selection Tool in Maya and the Vertex Cluster Tool in XSI.
- The " **Create New Materials For Overlap** " option makes the filter treat areas where named triangle selections overlap as if they had a new material. The new material's name will be the composition of the overlapping selections' names. The default is to consider these areas as being marked with the first material encountered during processing. In either case, an overlap will be reported.

In the **Landscape Welding** section you select the type of welding to be applied to Mesh Shapes:

- Setting welding information adds extra information to Mesh Shapes regarding welding, improving collision detection and resolution across triangle boundaries. In order to calculate this information, the filter needs to know what side of the triangles is considered to be *UP*. It will use the ordering of the triangle vertices in order to define this *UP* side. By default, most modelers use counter-clockwise orientation (where vertices are ordered in counter-clockwise order when viewed from the *UP* side).
- The " **Weld Open Edges** " option defines how unconnected edges are treated. If this option is checked, all open edges are assumed that there will be a connecting edge in a different mesh and all contact point normals will be rotated to the triangle normal. If the option is unchecked, the triangle edge will collide like a normal edge.

- The " **Weld Across All Shapes** " option defines whether welding is calculated for each mesh individually, or whether neighboring meshes are used in the calculation.

The options in the **Convex Shape Radius** section allow you to choose between two different types of Convex Radius calculation as the default global behavior:

- The " **Relative Shrink Radius** " defines the relative size of the shape which is used to optimize collision detection. Basically the vertices of the shape are shrunken by this relative value and then the entire shape is expanded back again by setting the convex radius. This value is closely tied to "Max. Vertex Displacement". Once an overriding "Extra Radius" is specified in a particular Physics Shape then the relative shrinking is disabled for this Rigid Body and the absolute value from the shape will be used instead.
- The " **Max. Vertex Displacement** " enables a functionality which replaces the shape shrinker filter. If this option is active, the shape will be shrunken at construction time by the size of the shape multiplied by "Relative Shrink Radius". However if the shape has very sharp edges, shrinking the shape only by a small amount could result in vertices being moved by a big distance. This parameter limits the shrink value so that no vertex is moved more than this value. The amount the shape is shrunk is used as the convex radius to 'grow' the shape to its original size.
- The " **Default Convex Radius** " value allows for globally setting the default value for the Extra Radius available in some physics shapes like Box or Sphere. This value is only used if you have not set the convex radius in the physics shape in the modeler. This value normally increases the size of the shape (by the radius) and is normally used in combination with the shape shrinker, which shrinks it back to the normal size. This 'strange' trick is used to improve collision detection performance. However it is advisable to use a different system to optimize your shapes and not use the shape shrinker filter, but use the "Max Vertex Displacement" & "Relative Shrink Radius" value pair instead.

The " **Create Additional Destruction Data** " switch is only used in combination with Havok Destruction. If you don't use Destruction, you should leave it turned off to save space in the exported asset files. When you do use Destruction it is essential though to enable this switch as Destruction depends on the additional exported data. Note that even with that switch turned off, some Destruction-related data will end up in the asset file's "Resource Data" subtree. You can prune that using the Prune Types filter.

When the " **Collapse Shape Transform Offsets** " option is enabled, the filter will attempt to collapse intermediate transforms into leaf shapes (for the case of convex shapes and mesh shapes). If the option is disabled, the filter will keep shapes in their local space (the space of the node representing the shape). This facilitates instancing (sharing) of shapes. However, if the shape and its rigid body are in different location in space, extra transform shapes are required to represent that offset. Enabling this option will remove those extra transform shapes by baking them into the convex/mesh shape. The option has no effect on bounding shapes (box, sphere, capsule and cylinder).

If the " **Use best-fitting shape orientation** " option is enabled, the filter will look for a best-fitting orientation for box, capsule and cylinder shapes rather than aligning those shapes to the transform of the scene node. Enable this option if you want the shapes to be based solely on the vertices of the mesh, regardless of the axes of the node transform. Normally this option is not required.

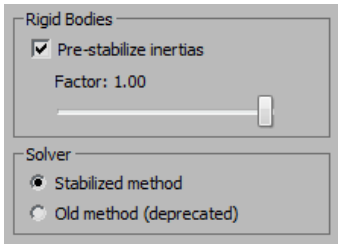
The Create Rigid Bodies filter will also search for attribute groups named "`hkProperties`" associated with nodes flagged as rigid bodies, and convert any attributes in them to runtime `hkpProperty` objects associated with the runtime `hkpRigidBody` object. Since runtime properties are pairs of (ID, value) where both ID and value are integers, the filter will convert attributes (name, value) by hashing the name of the attribute (the hashing method is exposed in the `hkpProperty` class). Check the Integration section for details on how to setup and export custom attributes associated with scene nodes.

#### 4.3.4.2 Create Constraints

**Components Required** : Havok Physics 2012

This filter interprets physical constraint data in the scene (usually set up using the Havok Physics 2012 Tools within a modeler) and creates the appropriate runtime constraint objects based on that data.

The rigid bodies connected by constraints must be created by the time this filter is processed, and therefore this filter needs to placed after a Create Rigid Bodies filter.



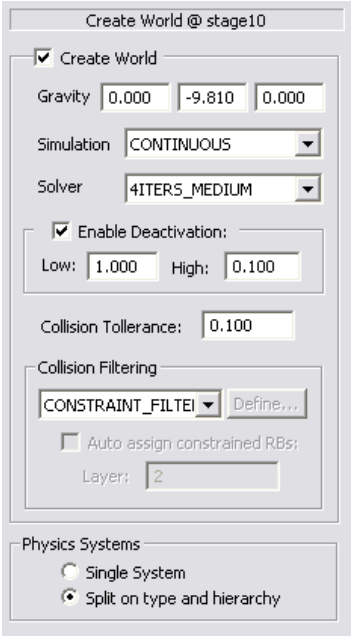

- **Pre-stabilize inertias** : Analyzes each constrained body in the scene and modifies its inertia tensor such that the constraint solver has a better chance of achieving a stable solution. Currently only affects rigid bodies constrained by ball-and-socket atoms (i.e. ball-and-socket, hinge, limited hinge and ragdoll).
- **Factor** : If the *stabilize inertias* option is enabled, it controls the amount by which the inertias will be increased. Generally, the inertia is increased by `factor * rigidBodyMass * armLength`. Therefore, this parameter has values between 0 (rigid bodies maintain their original inertias) and 1 (rigid bodies have their inertias increased by the maximum amount necessary to achieve a stable simulation).
- **Solver** : This option enables you to choose the solving algorithm to be applied while simulating the constraints in the current asset file. For best results, this should be set to *Stabilized method*. However, if you experience performance issues and are willing to trade simulation stability for speed, you may select the *Old method (deprecated)*. The latter will use the constraints solver existent in versions prior to 2010.2.

#### 4.3.4.3 Create World

**Components Required** : Havok Physics 2012

This filter can be used to set up information about the world (gravity, etc.). It can also be used to group rigid bodies into physical systems and assign collision groups to them.

This filter is not essential when creating physical assets (a default world will be created in the Preview Tool filter if necessary). If this filter is used, it must be placed after any Create Rigid Bodies or Create Constraints filters in order to operate properly.




- **Create World** : If this box is checked, a world setup will be added to the asset. Properties like gravity and other simulation settings can be specified in the options dialog. Please consult the Havok Dynamics chapter for details on these parameters and how they may affect the simulation.

  Among the world options that can be set up, the **Collision Filtering** section should be noted. Collision filters are used in order to determine which collisions between rigid bodies should be ignored. Many algorithms (collision filters) are provided in the Havok Physics 2012 SDK to do so. The options here allow you to choose between three options:

  - `NO_FILTER`: No collisions will be ignored
  - `CONSTRAINT_FILTER`: Collisions between constrained rigid bodies will be ignored
  - `GROUP_FILTER`: Collisions between rigid bodies with matching IDs will be ignored

  The Group Filter implementation is thoroughly described in the Havok Physics 2012 SDK. It offers a good compromise between flexibility and efficiency (the `CONSTRAINT_FILTER` option, for example, can be quite slow). By cleverly assigning proper IDs to rigid bodies, multiple behaviors can be achieved.

  If the **Auto assign constrained RBs** box is checked, IDs will be assigned to rigid bodies so the `GROUP_FILTER` will disable collisions between constrained rigid bodies, as the `CONSTRAINT_FILTER` does (but at a smaller cost). You can also define an specific layer ID for all the rigid bodies in the current asset, as well as defining (through the **Define...** button) which layers should collide with which other layer. Again, check the Physics 2012 SDK documentation on more details about how this filter works and how it interprets the different rigid body IDs.

- This filter also allows you to split the rigid bodies and constraints currently in the scene into multiple Physics 2012 Systems (collections of constraints and rigid bodies). Selecting the " **Split on type and hierarchy** " options will split the scene by looking at how rigid bodies are connected between them and whether they are fixed or not. Selecting the " **Single System** " option will put all rigid bodies and constraints in a single system.

#### 4.3.4.4 Create Constraint Chains

**Components Required** : Havok Physics 2012

This filter allows you to create constraint chains within the filter pipeline. To create a constraint chain you must specify the start and end bodies in the chain. See the Physics 2012 SDK documentation for more on constraint chains.

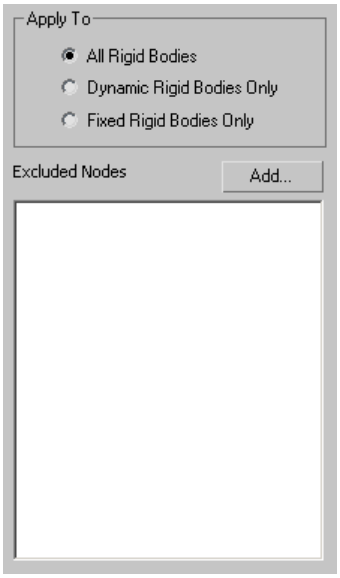
**Note:**
Currently this filter creates chains for powered constraint only. In the filter pipeline only the constraints constructed by a Create Ragdoll filter are powered.

#### 4.3.4.5 Shrink Shapes

**Warning:**
The Shrink Shapes filter should not be used any longer as it is inconsistent in its behavior (e.g. it does not handle Extended Mesh Shapes at all). You should use the new Automatic Shrinking provided in the Create Rigid Body filter.

**Components Required** : Havok Physics 2012

This filter goes through each convex shape in the scene and shrinks the original shape, so that when this convex radius is added the total size of the shape matches the original size of the model.

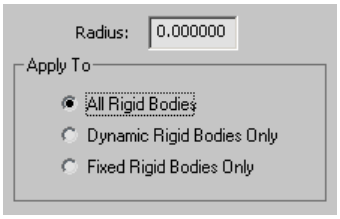



The filter allows you to specify which types of rigid body should be affected. In addition, you can exclude individual objects by using the **Excluded Nodes** list box; those objects will not be shrunk.

#### 4.3.4.6 Set Shape Radius

**Components Required** : Havok Physics 2012

This filter allows you to override the convex radius values of any rigid bodies in a scene.

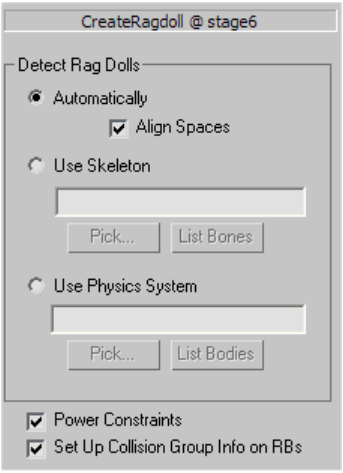



The filter goes through each convex shape in the scene and sets the convex radius to the value specified in the filter options. The filter allows you to specify which types of rigid body should be affected.

#### 4.3.4.7 Create Rag Doll

**Components Required** : Havok Physics 2012 + Havok Animation (Havok Complete)

Havok Complete provides the ability of treating rag dolls as special objects (`hkaRagdollInstance`), tracking the rigid bodies and constraints involved, as well as a skeletal representation of the ragdoll.




The filter can operate in three modes, which determine how a ragdoll is "detected":

- **Automatically** : Provided mostly for compatibility with older assets, this mode will assume that the first physics system in the scene contains the rigid bodies and constraints that make up the ragdoll. It will create a custom skeleton based on the hierarchy of rigid bodies and constraints.
  - **Align Spaces** : When using automatic mode, you can switch on this option to move the pivot of each rigid body so it matches the location of the child space of the constraint. It will also align the parent space to the child space. Usually this is not necessary if you have created the asset using the Havok Content Tools (as they'll usually be aligned already) but it is useful if you are processing assets created with older tools (the legacy physics exporters for example).
- **Use Skeleton** : A ragdoll will be created by finding rigid bodies with the same names as the bones of the selected skeleton. If the rigid bodies were originally created from a hierarchy of nodes (as they usually are) this is the most useful option, as it allows full control over the components of the ragdoll. It requires that a Create Skeleton filter is used beforehand in order to define the ragdoll skeleton.
- **Use Physics 2012 System** : Allows you to specify the rigid bodies and constraints that make up the rag doll by selecting a physics system in the scene. The skeleton will be automatically created based on the hierarchy of rigid bodies and constraints in that system.

The filter also provides further functionality to improve the ragdoll behavior:

- **Power Constraints** : Constraints in a ragdoll are often powered so they can be driven (to do physics+animation blending for example). If this option is enabled, constraints in the ragdoll will be powered.

  **Note:**
  Only limited hinges and ragdoll constraints can be powered.
  If any constraint is already powered, this checkbox won't have any effect on it.

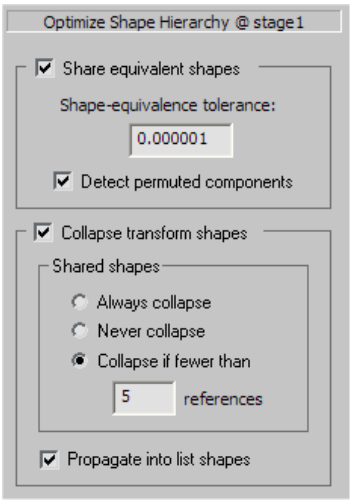
- **Set Up Collision Group Info on RBs** : Assigns collision IDs to the rigid bodies in the ragdoll so, if an `hkpGroupFilter` is used (check the Havok Physics 2012 SDK for details), constrained rigid bodies won't collide with each other. (Similar to the 'Auto assign constrained RBs' option in the Create World filter.)

Note that this filter operates on rigid bodies and constraints and therefore needs to be added after the Create Rigid Bodies and Create Constraints Filters. When using the **Use Skeleton** option, it also needs to be added after a Create Skeleton filter which sets up the skeleton of the ragdoll.

#### 4.3.4.8 Optimize Shape Hierarchy

**Components Required** : Havok Physics 2012

This filter attempts to optimize the shape hierarchies of all the rigid bodies in the scene, reducing the number of shapes used and/or the depth of shape hierarchies. Please check the Havok Dynamics chapter for details on shapes hierarchies and collision detection.




There are two (optional) steps, "Share equivalent shapes" and "Collapse transform shapes".

- **Share equivalent shapes** : If this box is checked, the filter identifies identical (or near-identical) shape structures in one or multiple rigid bodies and changes the hierarchy so such shapes are shared. It replaces "cloned" shapes or shapes structures with "instanced" ones.
  - **Shape-equivalence tolerance** : specifies a tolerance used to decide whether two shapes should be considered identical (and thus replaced with one, shared shape)
  - **Detect permuted components** : When this option is selected the filter will check for permutations of vertices/triangles/planes when comparing `hkpSimpleMeshShape` or `hkpConvexVerticesShape` objects. This will possibly detect more instanced shapes but due to the increased amount of comparisons it will take longer time. When the option is off vertices, triangles and planes are compared using the same order.
- **Collapse transform shapes** : If this box is checked, the filter attempts to collapse transform (and convex transform or convex translate) shapes into their child shape. For example, a transform shape on top of a capsule shape can be collapsed into a new capsule shape with transformed vertices.
  - Shared shapes: When a shape is shared by multiple, different, transform shapes, collapsing each transform shape into it will create a different shape for each transform. Although this reduces the shape hierarchy depth, if the original shared shape was large, having multiple instances of it can increase memory usage. Therefore, a trade off between reduced hierarchy depth and reduced memory usage is required. The options are:

    * **Always collapse** : Transform shapes will be collapsed even if they point to a shared shape.
    * **Never collapse** : Transform shapes won't be collapsed into shared shapes.
    * **Collapse if fewer than X references** : Transform shapes will not be collapsed if they point to a shape shared by $X$ or more shapes/rigid bodies. Specifying a number <=2 is equivalent to **Never Collapse** .
  - **Propagate into lists shapes** : When this option is selected, the filter will propagate any transform shapes applied to list shapes into the children of the list. This may reduce the depth on some of the branches if the transform can be collapsed in any of the children; however, if the transform cannot be collapsed into the children, this may actually increase the total number of shapes.

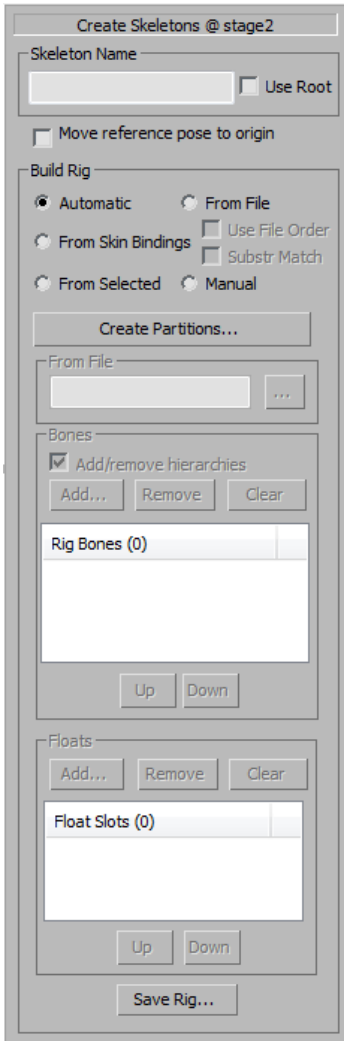
### 4.3.5 Animation Filters

Animation filters are those that create/modify Havok Animation objects. Therefore these should generally only be used if your company has licensed a Havok product that includes the Animation component.

#### 4.3.5.1 Create Skeletons

**Components Required** : Havok Animation

This filter is fundamental to the animation filter pipeline, as skeleton objects are involved in all animation tasks. This filter constructs a skeleton (an `hkaSkeleton` object) from a set of nodes in the scene. The skeleton consists of two main data components: bones and float slots. Bones correspond to node (transforms) in the scene and have an associated hierarchy matching the scene hierarchy. Float slots correspond to *animated float data* in the scene and have no hierarchy; see note below for how this data is exported from the modeler.




**Skeleton Name** : By default the skeleton will take its name from the root bone. You may override this by unchecking the **Use Root** checkbox and entering a name in the adjacent text field. Note that the skeleton name is not required for most runtime code but it will be inherited (copied) by skin bindings, bone attachments and animation bindings created by later filters using this skeleton. Thus it may be useful at runtime for associating this skeleton with the correct skins, attachments or animations originally created with it, for example after loading from different files or locations.

If **Move Reference Pose to Origin** is checked, the translation of the root bone for the created skeleton will be set to zero (as if it whole skeleton was placed at the origin).

The rest of the options determine how the rig is constructed, i.e. which nodes and which animated floats in the scene should be used to create the skeleton. See the Note below for details on 'Float' slot addition.

- **Automatic** : This constructs the rig by examining the scene for any skinned meshes. It takes any node associated with the first skin found and travels up the scene graph until it reaches the root node. From this node (just under the root node) it walks back down the scene graph and recursively adds *all* child nodes. Note that *no* Floats (Float Slots) are added automatically from this option.
- **From Skin Bindings** : This method selects all nodes referenced by the skin bindings in the scene, then searches for and adds any nodes in the scene that connect the referenced nodes together. This

usually creates less bones than the " **Automatic** " mode. Note that *no* floats (Float Slots) are added automatically from this option.

- **From Selected** : This constructs the rig from the nodes selected in the modeler when the filter manager was run. Note that *no* floats (Float Slots) are added automatically from this option.

- **From File** : The file loaded is a text file containing the names of bones, float slots, and partitions that should be contained in the rig. By specifying the skeleton definition in an external file, it can easily be shared across assets with differing filter stacks. *The path for the filename is stored relative to the original asset by default.* (To use absolute paths, set the modeler environment variable `assetPath=""`.) With this option selected, the control allowing you to specify the filename becomes enabled. To create a new definition file from a skeleton which has been set up, use the **Save Rig...** button.

  **File Format** :

  - Skeleton Definition File Format (default): This file format is the format used when clicking " **Save Rig...** ". The first line of the file must be "`[HAVOK SKELETON DEFINITION FILE]`" to distinguish it from the Old File Format below. Due to the fact that nearly any character is allowable in bone/float names, the new line character is used to tokenize the pertinent information. This information is contained within sections beginning with predefined headers and ending with "`[END]`". The predefined headers are: "`[BONES START]`", "`[FLOATS START]`", "`[PARTITION START]`". Any lines of text outside of these defined sections are considered to be comments and are therefore ignored.
    * `[BONES START]`/`[END]`: Each line of text between these signifiers is interpreted as a bone in the skeleton.
    * `[FLOATS START]`/`[END]`: Each line of text between these signifiers is interpreted as a float slot in the skeleton.
    * `[PARTITION START]`/`[END]`: Each start/end partition section is considered to be a single partition definition containing three lines of data. The first line is the partition name, the second line is the start bone name, and the third line is the end bone name. If partitions are being used, it's required that the entire skeleton is covered by partition definitions.
  - Old File Format: This is a very simple format in which each line of the file corresponds to a different rig bone or float slot (partitions are not supported). The code will first look for the text as a bone. If the bone is not found in the scene, the text will be added as a float slot.

  **Options** :

  - **Use File Order** : If this option is checked and the 'From File' mode is used, then the bone order will not be automatically reordered by the filter to match the scene hierarchy; the bone order will exactly match that specified in the file.

    **Warning:**

    **Use File Order** allows the user complete control of the bone order. Bone orders which are not *topologically sorted* (every bone's parent occurs before the child in the list) are not allowed as they will cause problems with partial sampling, the `hkaPose` class and ragdoll mapping. It is up to the user to ensure that the bone order they specify in the file is topologically sorted, otherwise an animation will not be created.

  - **Substr Match** : This option regards the contents of the skeleton definition file as substrings which are to be matched against bone names in the scene. This option can allow a single skeleton definition file to be used with multiple characters following a similar naming convention. For example if a Giant and a Dwarf character each share a similar rig in which the bones are prefixed by the names "Giant_" and "Dwarf_" and each contains a bone "Giant_Root" and "Dwarf_Root" respectively, then the name "Root" may be used in the skeleton definition file if this option is used. This option should be used with care as spurious matches are possible. If multiple characters exist in the same scene, it may be desirable to only export selected bones to the Havok Content Tools at one time.

- **Manual** : This allows you to construct the rig by directly specifying nodes in the scene. With this option selected, the controls allowing manual selection become enabled. To add a node click on the corresponding **Add...** button. This will bring up the *Select Node* dialog which allows you to choose a transform node (for Bones) or an animated float node (for Floats). For bones, if the **Add/remove hierarchies** check box is checked then all children of this node will be added to the list of bones. To remove a bone/float simply highlight it in the list at any time and click **Remove** ; if the **Add/remove hierarchies** check box is checked then all children are also removed. To empty the list, press **Clear** .

**Note:**

The float slots for the skeleton specified in the Create Skeletons filter by the 'Floats' list will normally correspond to the names of (some subset of) the animated float data present in the scene. While the animated *bone* data (as transforms associated with scene nodes) is always exported out of the modeler into the hkxScene object parsed by the filters, the user must explicitly request animated *float* data to be exported via one of two methods: Custom Attributes (see appropriate Max, Maya or XSI descriptions) or using Attribute Selection.
Using either of these two methods, the float data will be pulled out as Attributes of type hkxAnimatedFloat inside Attribute Groups associated with the corresponding scene nodes. You can then select these attributes by creating the rig **Manual** or using **From File** .

**Note:**

Preview of float slots (using the Preview Tool Filter) can be done by enabling the 'Float Tracks' display under the 'View' menu. Float slot names and the current slot value will be displayed as text in the viewport.

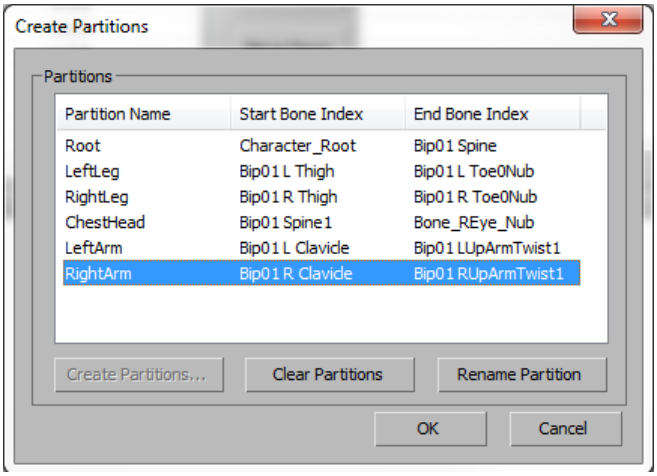
The **Create Partitions...** button opens the Create Partitions Dialog. This is an optional step that allows for the creation of named subsets of bones within the skeleton.

**Note:**

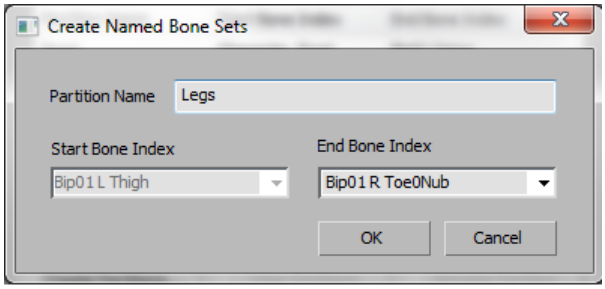
Editing partitions defined in an external file will not automatically be saved to that file and will be lost when the filter loses focus. You must use " **Save Rig...** ". It is only allowed in order to make it easy to create new setups from old ones.

### Create Partitions Dialog

This dialog allows for the creation of contiguous bone subsets in the animation. In combination with the Create Partitioned Animation filter, a partial animation of only specified subsets is easy to set up.




The **Create Partitions...** button will open the Create Named Bone Set Dialog.



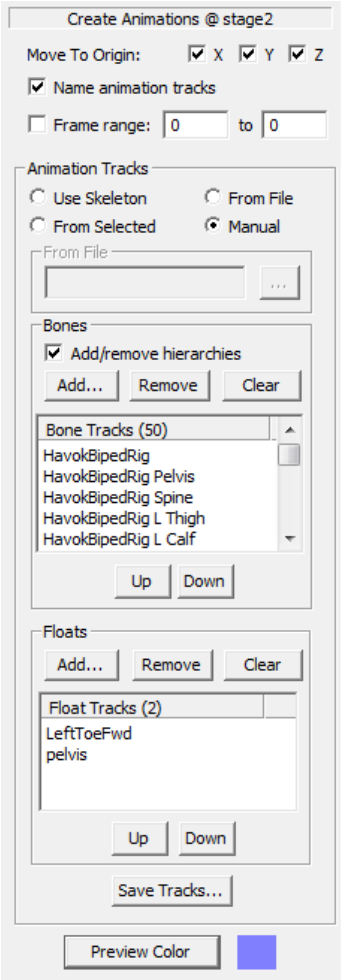

The Create Named Bone Sets dialog allows the user to create the individual partitions of the skeleton. It enforces that the partitions be contiguous by continuing to prompt the user for each subsequent partition.

**Note:**
When setting up partitions, it is necessary to have every bone in one partition. No bones can be outside of a partition and no bones can be in more than one partition.

#### 4.3.5.2 Create Animations

**Components Required** : Havok Animation

Given a skeleton (created by the Create Skeleton filter), this filter extracts the keyframes stored in the nodes of the scene associated with the skeleton bones, and creates a skeletal animation object from them (an `hkaInterleavedUncompressedAnimation` to be exact, which contains uncompressed animation data–both transforms and floats).




If the X, Y or Z **Move To Origin** boxes are checked, the entire animation will be translated so that the corresponding root bone component starts at the origin on the first frame. This may be useful for animations authored away from the origin, but note that in this process the corresponding offset is lost. See also Repositioning and Reorienting Animations in the Common Concepts section of this document. Note that these settings may override any transforming done by the Align Scene To Node filter; a warning will be issued if both features are used in the same configuration.

To use a subset of the valid frame range click the **Frame Range** check box and specify a valid start and end frame number. Both the start and end frame will be included in the animation. A warning is issued if the start frame number does not match the frame number of any Align Scene To Node filter instances in the same stack (this is not an error, merely a suggestion to avoid unwanted behavior in the most typical cases).

If the **Name Animation Tracks** box is checked, the name of each bone track will be exported with the animation. This information can be used, for example, in order to bind animations to skeletons at runtime. If you don't need this information, you can reduce the size of the exported animation by unchecking this box.

By default, animations are created from all of the bones and float slots in the skeleton, but it is also possible to animate a subset of the skeleton's bones or float slots, such as the legs or the upper body.

There are four available options when choosing what to animate:

- **Use Skeleton** : This is the default option. Animations are created from all the bones and float slots in the skeleton.
- **From Selected** : Only those bones that are selected at export time are animated. The list of selected bones is displayed in the list at the bottom.
- **From File** : The file loaded is a text file containing the names of the bones that should be contained in the rig. Each line of the file corresponds to a different rig bone or float slot. *The path for the filename is always stored relative to the original asset.* With this option selected, the control allowing you to specify the filename becomes enabled. To create a new rig file from a skeleton which has set up, use the **Save Tracks...** button.
- **Manual** : This allows you to construct the rig by directly specifying nodes in the scene. With this option selected, the controls allowing manual selection become enabled. To add a node click on the **Add...** button. This will bring up the *Select Node* dialog which allows you to choose a transform node (for Bones) or an animated float node (for Floats). For bones, if the **Add/remove hierarchies** check box is checked then all children of this node will be added to the list of bones. To remove a bone/float simply highlight it in the list at any time and click **Remove** ; if the **Add/remove hierarchies** check box is checked then all children are also removed. To empty the list, press **Clear** .

Also, when using the **Manual** option, you can change the order of the tracks inside the animation by selecting a bone or float track and using the **Up** and **Down** buttons.

Note that no animations will be created unless a skeleton is present in the stream (a Create Skeleton filter has been added).

**Note:**

The float tracks for the animation specified in the Create Animations filter by the 'Floats' list must correspond to the names of (some subset of) the animated float data present in the scene. While the animated *bone* data (as transforms associated with scene nodes) is always exported out of the modeler into the hkxScene object parsed by the filters, the user must explicitly request animated *float* data to be exported via one of two methods: Custom Attributes (see appropriate Max, Maya or XSI descriptions) or using Attribute Selection.
Using either of these two methods, the float data will be pulled out as Attributes of type `hkxAnimatedFloat` inside Attribute Groups associated with the corresponding scene nodes. You can then select these attributes by creating the rig with **Use Skeleton** , **Manual** or using **From File** .

**Note:**

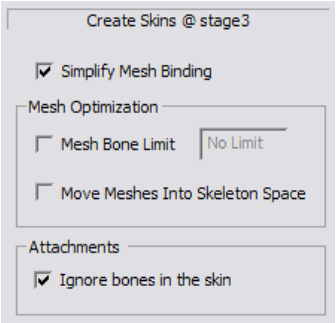
Preview of float tracks (using the Preview Tool Filter) can be done by enabling the 'Float Tracks' display under the 'View' menu. Float slot names and the current slot value will be displayed as text in the viewport.

Annotations are automatically created by the Create Animations filter. To learn more about annotations, refer to the Animation > Annotations section. Annotations may be removed by using the Prune Types filter.

#### 4.3.5.3 Create Skins

**Components Required** : Havok Animation

The Create Skins filter examines the skinned meshes in the scene (`hkxSkinBinding` objects) and creates a Havok mesh binding (`hkaMeshBinding`) object for each one of them. These mesh bindings are used to link the bones in the rig with the bone indices stored in the mesh. See the section on skinned meshes in the Animation chapter in this manual for more details on how this transformation is done.




Note that the mesh binding will be based on the `hkxSkinBinding` object exported from the modeler. You can control the number of bones associated with the skin by modifying the scene in the modeler or, in some cases, by changing related export options (for example, the XSI exporter's option, **Ignore Unused Bones in Skin Bindings** ).

The **Simplify Mesh Binding** option creates a trivial binding, whereby the mappings in the mesh binding are set to NULL. The bone indices in the original mesh are altered to map directly to bones in the skeleton on a one-to-one basis.

You can limit the maximum number of bone influences a mesh primitive can have to facilitate hardware skinning. The mesh will be broken up into sections such that each set of triangles has at most the specified limit of bones affecting it, and the remapping from the original skeleton bones to the reduced set of bones in each section will be stored and used at runtime.

**Note:**
When using the bone limit feature, DirectX must be selected and hardware skinning enabled in the Preview Tool filter in order to correctly display skinned meshes.

The **Move Meshes Into Skeleton Space** option is an optional optimization choice. It will transform the mesh vertices (and all associated data–normals, binormals, etc.) into the space of the root bone of the skeleton *before* the binding matrices are calculated. This will mean that the set of 'meshToBone' transforms will be identical for each mesh binding (or independent of the mesh) on export, in the case that you are exporting multiple mesh bindings. Thus at runtime the 'meshToWorld' transforms required for skinning will need to be computed only once for a given pose, and can be reused by all meshes, which may be more efficient.

**Note:**
Using this option will alter the mesh transforms and vertex data in-place hence also for all successive filters.

The filter will also create Attachment objects based on other meshes parented to the bones in the skeleton. Check the Animation chapter in this manual for details on attachments.

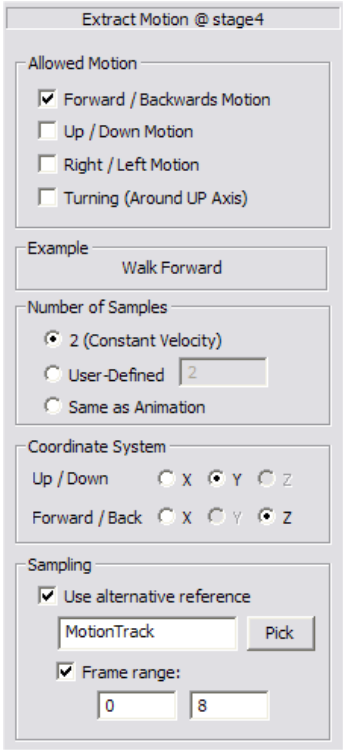
The **Ignore bones in the skin** option, when selected, will tell the filter not to create attachment objects from mesh nodes which are also bones of the skeleton.

Note that no skin will be created unless a skeleton is present in the stream (a Create Skeleton filter has been added).

#### 4.3.5.4 Extract Motion

**Components Required** : Havok Animation

The Extract Motion filter is used to sample the displacement (relative to the environment) from the animation. This movement can be reconstructed later and used to drive the world position of the animation during playback. See also Using Extracted Motion in the Common Concepts section of this document and the Motion Extraction section of the User Guide.




The **Allowed Motion** section specifies the type of movement to be extracted from the animation. Select any combination of these check boxes and the ' **Example** ' section displays the type of motion that will be extracted.

Use the **Number of Samples** section to specify how many samples to take when extracting the motion. This can be any number from 2 to n, where n is the number of frames in the animation.

Finally select the appropriate **Coordinate System** for performing the motion extraction. For most locomotions this corresponds to Z up in Max and Y up in Maya.

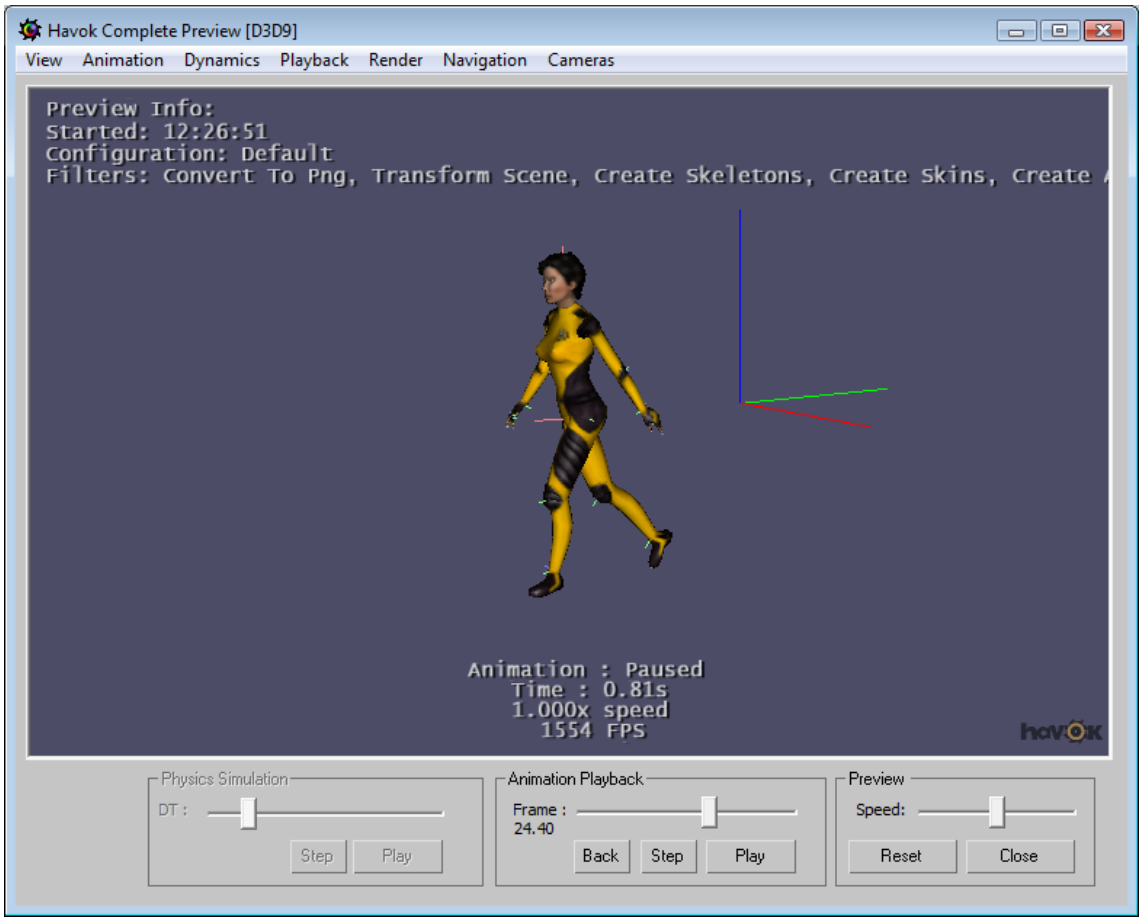
The **Sampling** > **Use Alternative Reference** option allows you to select an animation track (or potentially an entirely different object) other than the root animation track as a frame of reference for motion extraction. This option is switched off by default. When enabled the name of the node to use should be entered in the edit box. Alternatively the ' **Pick** ' button can be used to select a node.

A **Frame Range** can be specified for the alternative reference. This option should be used if the range of keyframes exported from the modeler doesn't match the length of keyframes in the animation (that is, if the **Frame Range** option was used in the Create Animations filter). In that case, select the start and end frames of the alternative reference frame so they match the start and end of the animation.

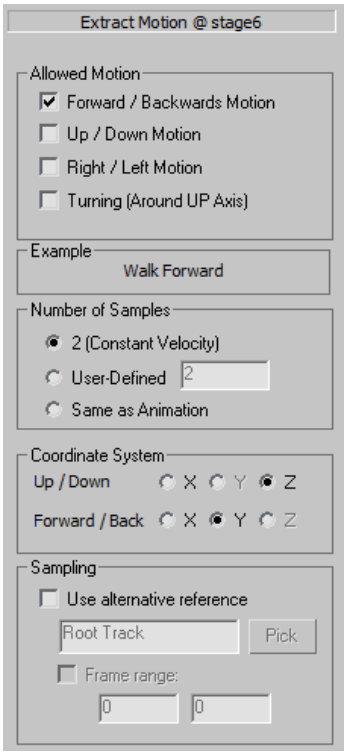
Float tracks will be ignored. See the section on Motion Extraction in the *Havok Animation* chapter for specific details on our motion extraction process.

Note that this filter expects to find at least one Interleaved Animation in the data stream (it usually follows a Create Animations filter).

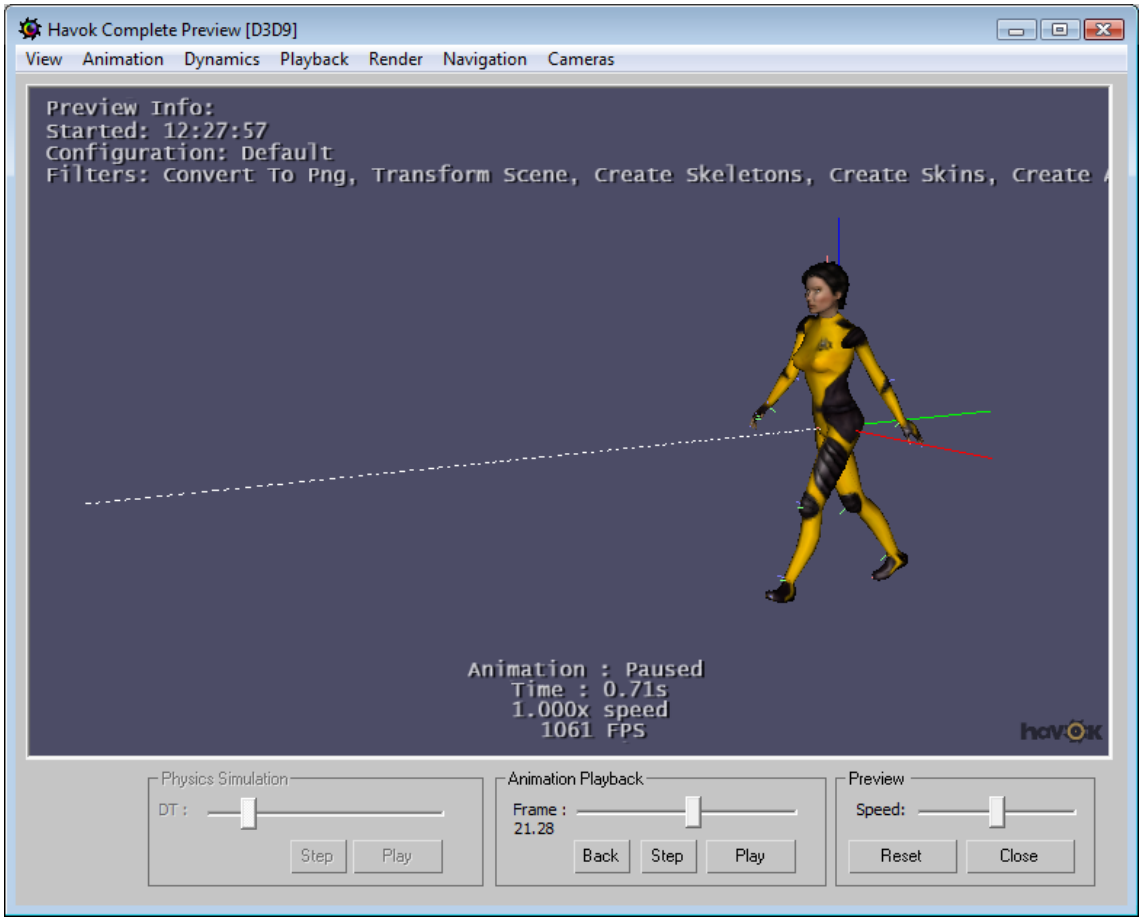
As an example, suppose that we wish to apply motion extraction to a walk animation which begins with the character's pelvis centered near the origin looks like this at a later time:




Thus we add an Extract Motion filter immediately after the Create Animation filter and customize it for our animation. We first determine the world axes which specify the components of motion by setting the **Coordinate System** . In this case it is clear that the up/down direction is the blue Z-axis, and that the forward/back direction is the green Y-axis (The left/right direction is then implicitly the remaining canonical axis, X). For this animation we will want to remove the forward motion, so we set this in the **Allowed Motion** section. The final filter settings are as follows:




The result can be seen in the Preview window by turning on the Extracted Motion display in the View menu. Note that the character remains in the position it was in on the first frame. The animated broken white line drawn ahead of the animation is the extracted motion. Accumulation of this motion during playback in the Preview window can be turned on/off under the Animation menu.




See also Using Extracted Motion in the Common Concepts section of this document and the Motion Extraction section of the User Guide.

#### 4.3.5.5 Footstep Analysis

**Components Required** : Havok Animation

The Footstep Analysis filter is used to analyze an input animation, such as a walk or run cycle, and automatically determine when the feet of the character are in contact with the ground. The result of the filter will be a series of annotations added to the animation which denote times at which feet change the state of contact with the ground.

Footstep analysis adds value to a variety of use cases. Example use cases include synchronizing animations on a particular footstep event, changing the state of an animated character at a specified moment in the footstep timeline, or applying inverse kinematics to a character's feet.

**Types of Contact**

The Footstep Analysis Filter is capable of detecting two different types of foot contact. These contact types are:

- **Strike** : The foot is considered to strike the moment *any* of the bones of the foot come into contact with the ground. The moment at which *all* bones leave contact with the ground is called a **Lift** .
- **Lock** : The foot is considered to lock the moment *all* of the bones of the foot come into contact with the ground. The moment at which *any* bone leaves contact with the ground is called an **Unlock** .

Additionally the Footstep Analysis Filter can compute the midpoints between these pairs of events, for a total of eight possible footstep events (see Footstep Events).

### Defining the Bones of the Feet

Footstep analysis begins by defining the bones of each foot. The user may define as many feet as they wish, each with an arbitrary number of bones. A typical biped foot consists of two bones, a heel and a toe, though a simplified model may only have one bone in the foot, and a complex model may have three or more bones. The bones of each foot are defined using the **Foot Definition** section of the Footstep Analysis Filter, shown below.

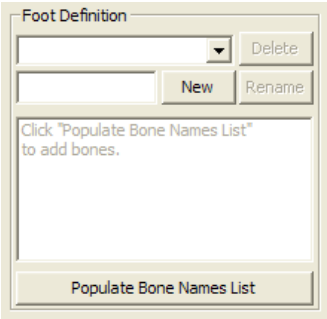



Figure 4.1: Foot Definition Interface

The first step in creating a foot definition, is to populate the list of bone names. To do this, first ensure that a valid skeleton exists in the scene; typically this is done by placing a Create Skeleton Filter instance in the filter pipeline prior to the Footstep Analysis Filter. Clicking the **Populate Bone Names List** button will fill out the names of all bones into the text list above the button.

To create a new foot definition, type a descriptive name into the text box to the left of the **New** button, then click the **New** button. A typical name might be simple, such as "Left" or "Right". Once the foot has been created, its name will appear in the menu to the left of the **Delete** button. Bones can be added to the foot by selecting them from the list of bone names. Holding the control key and clicking will add additional bones. An example foot definition is shown below.

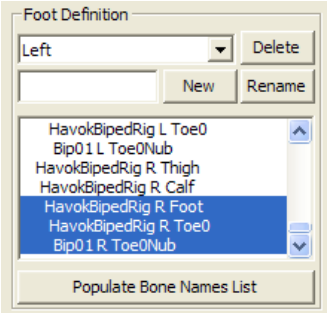



Figure 4.2: Example Foot Definition

An unlimited number of feet may be defined by the user. To create an additional foot, type a name into the text box to the left of the **New** button, then click the **New** button. Bone names may now be added to the new foot. To return to editing a previously defined foot, choose its name from the menu to the left of the **Delete** button.

Each foot definition within the same instance of the Footstep Analysis Filter will share the same settings for detecting foot events. If it is desired for different feet to have different tolerance settings, multiple instances of the Footstep Analysis Filter can be instanced in the filter pipeline.

### Analysis Settings

Three variables control how the Footstep Analysis Filter determines when footstep events occur in an input animation. As shown below, two of these settings are tolerance values, and the third defines the **Up** direction for the character.

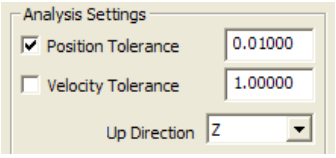



Figure 4.3: Footstep Analysis Settings

The Position Tolerance determines how close each bone must be to the lowest point which it achieves during the entire animation (the lowest point is considered to be contact with the ground). First the animation is scanned to find the lowest height of each bone throughout the animation. At each time, the height of each bone is compared to its lowest value. This setting is given in world units (typically meters).

The Velocity Tolerance determines the maximum speed at which a bone may be moving to be considered down. This setting is given in units per second (typically meters per second).

Position Tolerance or Velocity Tolerance may be activated by checking the check boxes to the left of the settings. If both are active, bones must meet both position and velocity tolerances to be considered in contact with the ground.

For visualization techniques and guidance in how to choose these settings, please see the Controlling Footstep Analysis section.

### Footstep Events

Eight possible footstep events can be detected, as shown below. Checking the check boxes next to the event type enables that event type for detection. Each event type can be given a unique name by the user. When the Footstep Analysis Filter is run, annotations will be created for each active event type, independently for each foot. The name of the annotation will be the name of the foot concatenated with the name of the event.

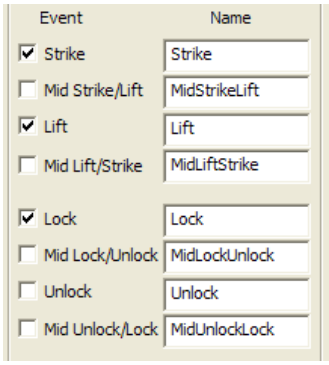



Figure 4.4: Footstep Event Definitions

Assuming a foot name of "Left" the example shown above would produce events named "LeftStrike", "LeftLift" and "LeftLock".

**Note:**

As the names of feet and annotations are concatenated together, it may be useful to begin event names with capital letters or an underscore to make the resulting annotation names more readable.

### Additional Options

Additional options are available to tune the analysis and provide diagnostic information to the user. These options are shown below.

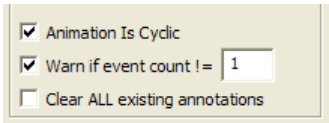



Figure 4.5: Additional Options

- **Animation Is Cyclic** : This box should be checked if the animation represents a continuous cycle. It is used for evaluating velocities at the boundaries of the animation.
- **Warn if event count != n** : Warns the user if the number of Strike or Lock events found is not equal to the user-specified value. Useful for debugging the analysis of noisy animations in which accidental footstep events are found.
- **Clear ALL existing annotations** : If checked, removes any pre-existing annotations from the animation. Otherwise existing annotations are left in place.

### Controlling Footstep Analysis

The Footstep Analysis Filter settings are best chosen by examining an animation asset. The more representative this asset is, the easier it will be to apply the same settings to other assets.

### Determining the Up Axis

The first step in addressing the filter settings is to determine the **Up** axis of the character. Add an instance of the Preview Tool Filter to the filter stack and view the Preview Window. An axis will be drawn at the center of the scene which can be used to determine the **Up** axis of the character as shown below.

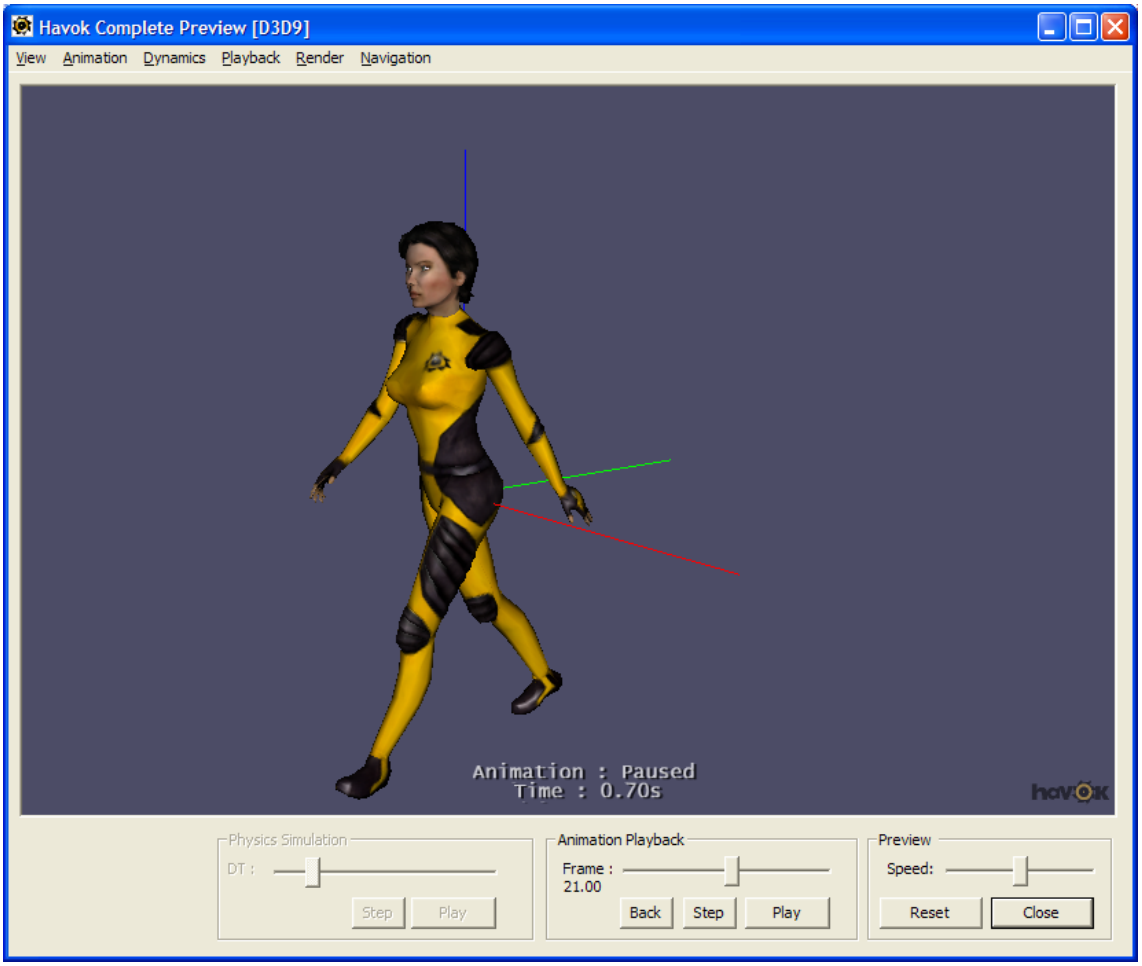



Figure 4.6: Determining the Up Axis

Note the axis at the center of the scene in the figure above. In this example the axis pointing upward is blue. Havok uses the convention of coloring the X, Y and Z axes Red, Green and Blue, respectively. In this example the Z axis is **Up** .

### Determining Position and Velocity Tolerances

Once the **Up** axis is correctly known, diagnostic information can be generated by the Footstep Analysis Filter to aid the user in choosing appropriate Position Tolerance and Velocity Tolerance settings. If the **Up** axis setting has changed, the filter stack must be re-run to generate correct diagnostic information.

The Position Tolerance is used to approximate the height of the foot above the ground. The Velocity Tolerance is used to approximate the speed of the foot. A character walking along a flat surface will benefit most from Position Tolerance, whereas a character walking along an uneven terrain is best modeled using the Velocity Tolerance. Both of these tolerances can be used simultaneously to properly detect times in which the characters feet are "sliding" (close to the ground but moving) or "hanging" (stopped in mid air). The Position Tolerance and Velocity Tolerance are defined in the Footstep Analysis Settings section.

The **Strike Position** and **Lock Position** graphs provide information for the user to determine an appropriate Position Tolerance setting. To view the **Strike Position** graph, enable the View > Footstep Display > Strike Position menu item of the Preview Tool Filter Window. A plot of the value used to determine a Strike is superimposed on the Preview Window, as shown below.

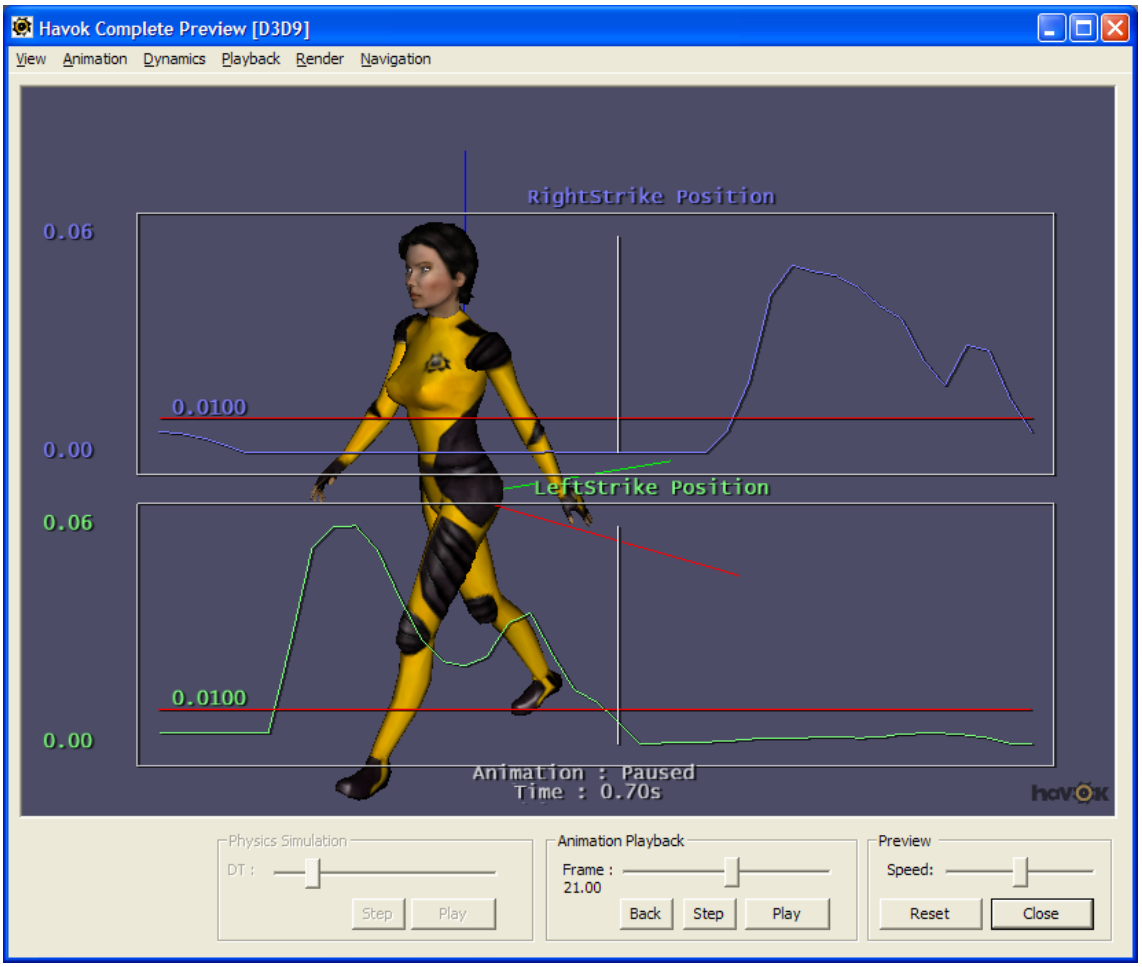



Figure 4.7: Strike Position Graph

Plots are drawn for each foot defined in the Footstep Analysis Filter. Each plot consists of a graph of the height used for Strike detection versus time. The graph is drawn in color and surrounded by a white frame. A vertical white cursor indicates the current time of the simulation. A red horizontal line indicates the current value of the Position Tolerance setting both graphically, by the position of the red line, and textually, by the value drawn at the left end. On the left of the white graph frame are drawn the maximum and minimum values of the graph. An appropriate setting for the Position Tolerance is one which is close to the minimum, but still allows for some minor noise in the signal. For the example shown above, the value of 0.0100 produced good results. A Strike event will occur each time the graph crosses the red tolerance line "downwards" (starting above and ending below) and a Lift event will occur each time the graph crosses the red Position Tolerance line "upwards" (starting below and ending above). The vertical white time cursor is positioned shortly after a Strike event has occurred for the Left foot.

To view the **Strike Velocity** graph, enable the View > Footstep Display > Strike Velocity menu item of the Preview Tool Filter Window. A plot of the value used to determine a Strike is superimposed on the Preview Window, as shown below.

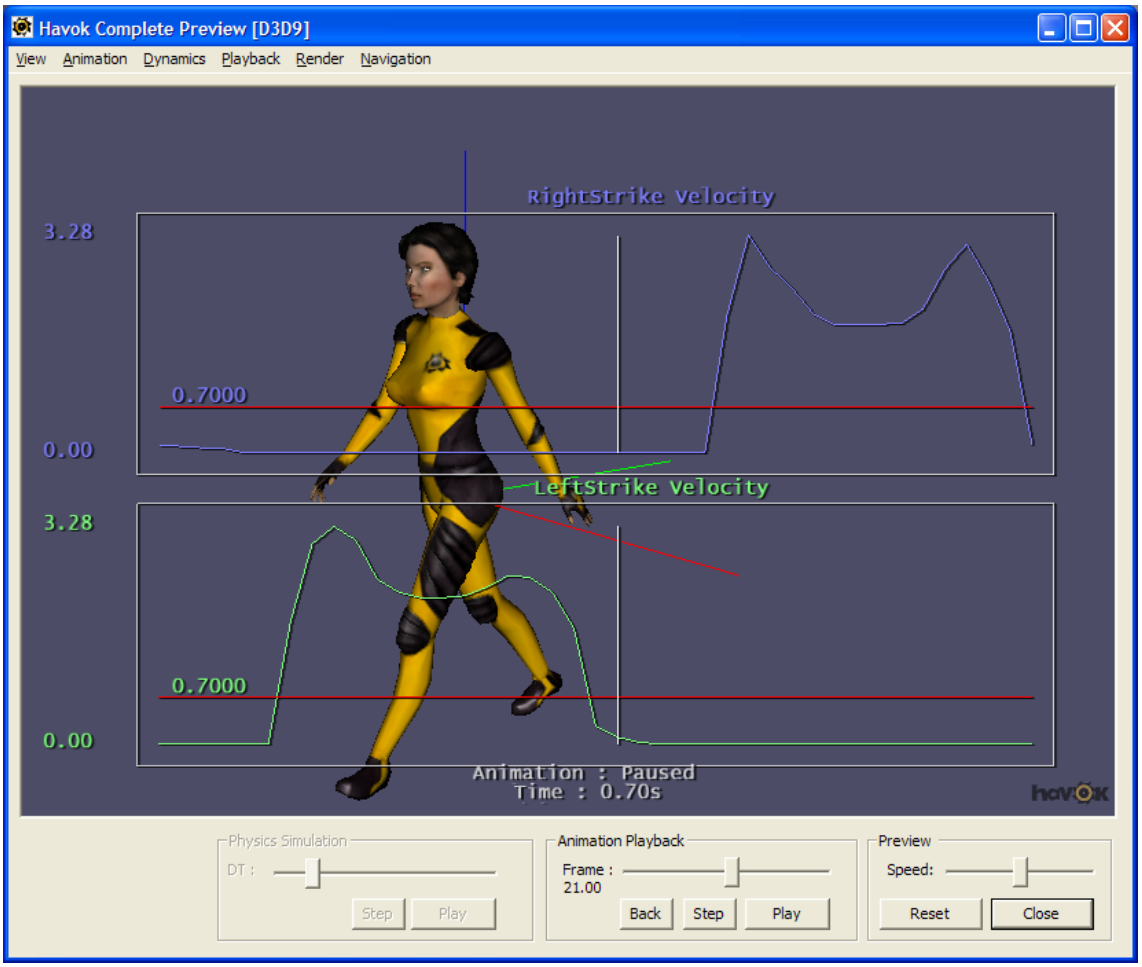



Figure 4.8: Strike Velocity Graph

The Strike Velocity graphs are drawn similarly to the Strike Position graphs.

If both Position Tolerance and Velocity Tolerance are active, a **Strike** event will occur the moment *both* the **Position** and the **Velocity** graphs fall below their respective tolerance values. A **Lift** event will occur the moment *either* the **Position** or **Velocity** value exceeds the respective tolerance value.

Enabling the View > Footstep Display > Strike menu item displays a plot of exactly when **Strike** and **Lift** events occur in the timeline, as shown in the example below. This graph shows a binary plot of when the foot is considered "down". At the moment each foot becomes "down" a **Strike** event will occur. Similarly at the moment each foot ceases to be "down" a **Lift** event will occur.

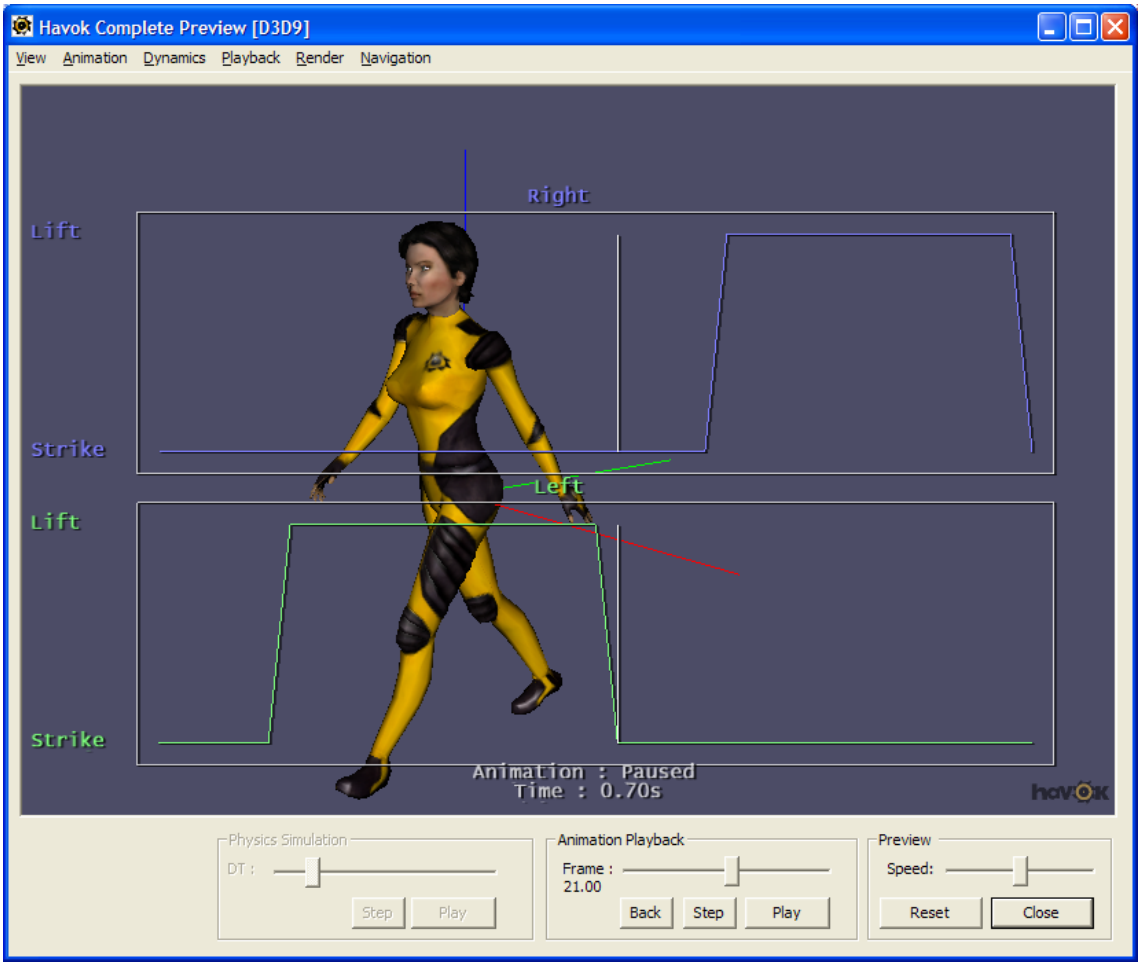



Figure 4.9: Strike Graph

The **Strike Position** graph plots the height of the *lowest* of the bones in each foot over time, whereas the **Lock Position** graph plots the height of the *highest* of the bones in each foot. Similarly the **Strike Velocity** graph plots the velocity of the *slowest* of the bones in each foot over time, whereas the **Lock Velocity** graph plots the speed of the *fastest* of the bones in each foot. The **Lock** , **Lock Position** and **Lock Velocity** graphs can be used to display corresponding information to their respective **Strike** counterparts.

### Previewing Footstep Events

The footstep events found by the Footstep Analysis Filter are added to animations in the form of annotations. These annotations can be seen in the Preview Tool Filter window as shown below. To view these annotations, ensure that the View > Footstep Display > Annotations menu is checked. As the animation plays annotations will appear on the screen at the time of the corresponding footstep events found. The playback of the animation can be controlled by using the animation playback controls shown at the bottom of the window below.

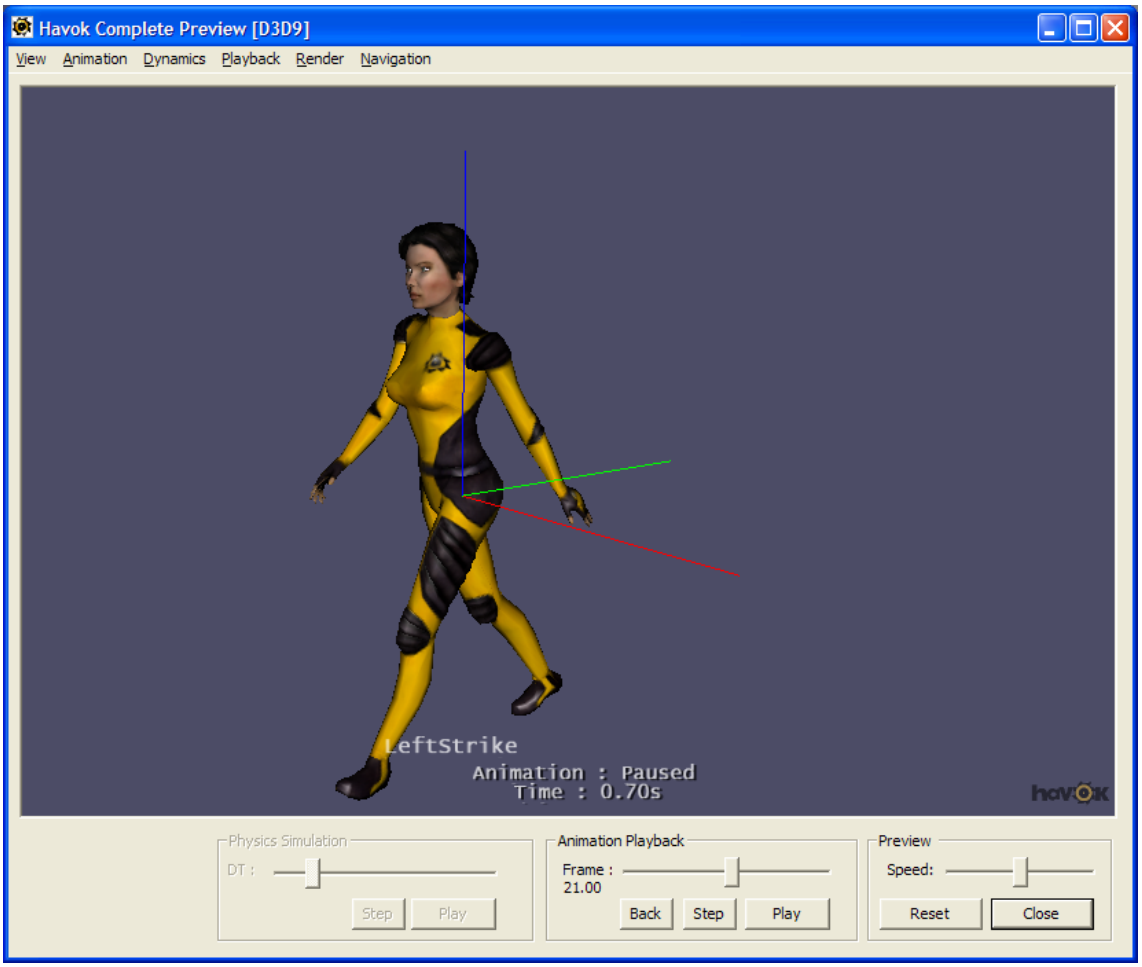
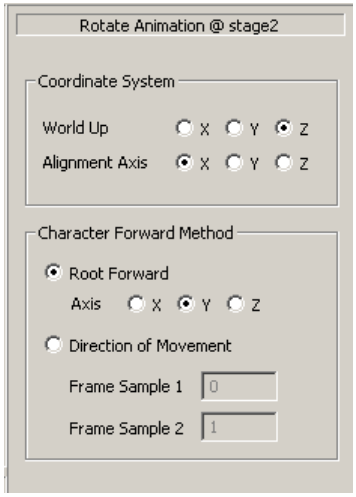



Figure 4.10: Event Annotations Preview

#### 4.3.5.6 Rotate Animation

**Components Required** : Havok Animation

The Rotate Animation filter rotates the animations so that the Root Forward axis of the character is aligned with one of the world coordinate axes. This filter should only be applied before the Extract Motion filter, if motion is to be extracted. After the animation has been processed through the Rotate Animation filter, the first frame of the animation will be aligned by rotation around the World Up Axis so that the character faces in the direction of the Alignment Axis. See also Repositioning and Reorienting Animations in the Common Concepts section of this document.



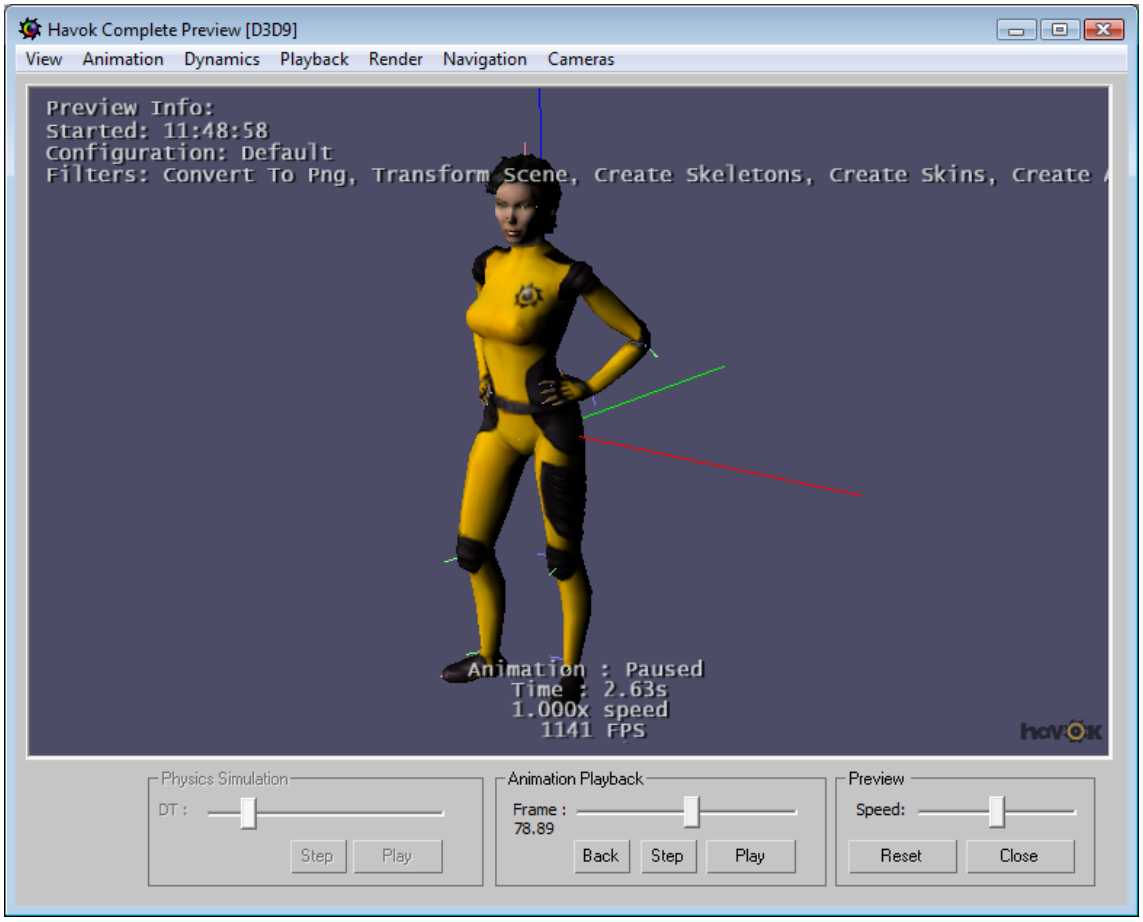

The **World Up Axis** is specified by selecting one of the three Cartesian coordinate axes.

The **Alignment Axis** defines the coordinate axis to which the characterÊijs forward facing direction will be aligned after the filter has processed the animation.

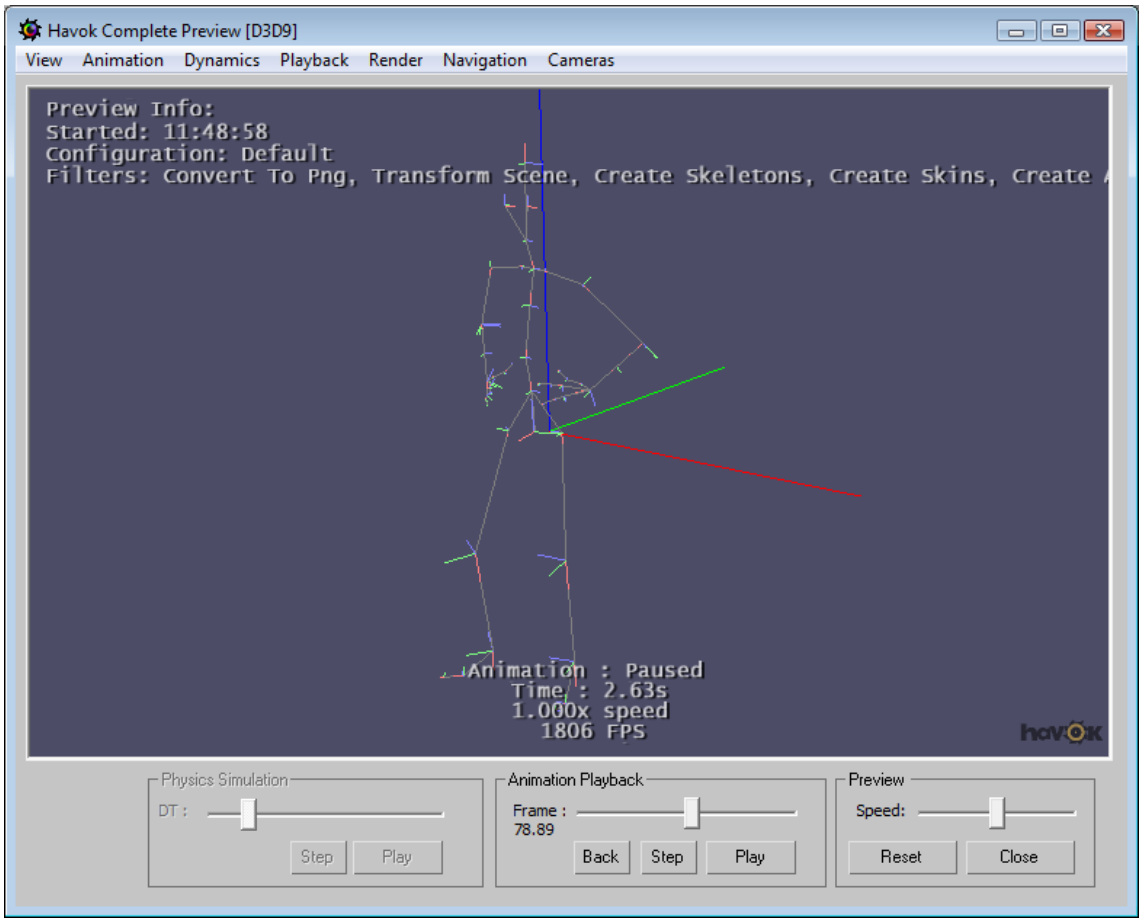
The **Character Forward** Method specifies how the characterÊijs forward facing direction is determined.

- Use the **Root Forward** method to specify one of the three Cartesian coordinate axes.
- Use the **Direction of Movement** method to specify two frames of the animation that will be used to determine a vector that will be used for the characterÊijs forward facing direction.

As an example, suppose that we wish to reorient the following idle animation where the character faces down the -Y axis, and we wish to reorient it so that it faces down the X-axis.

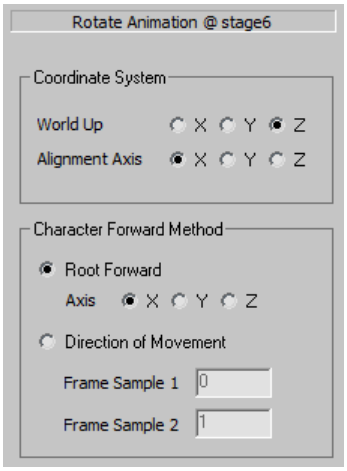



First we need to determine which axis of the root we wish to reorient. You can do this by examining the bone object in the modeler, but we can also view the skeleton only to see by pruning the mesh or by turning off 'Display Meshes' in the filter preview:

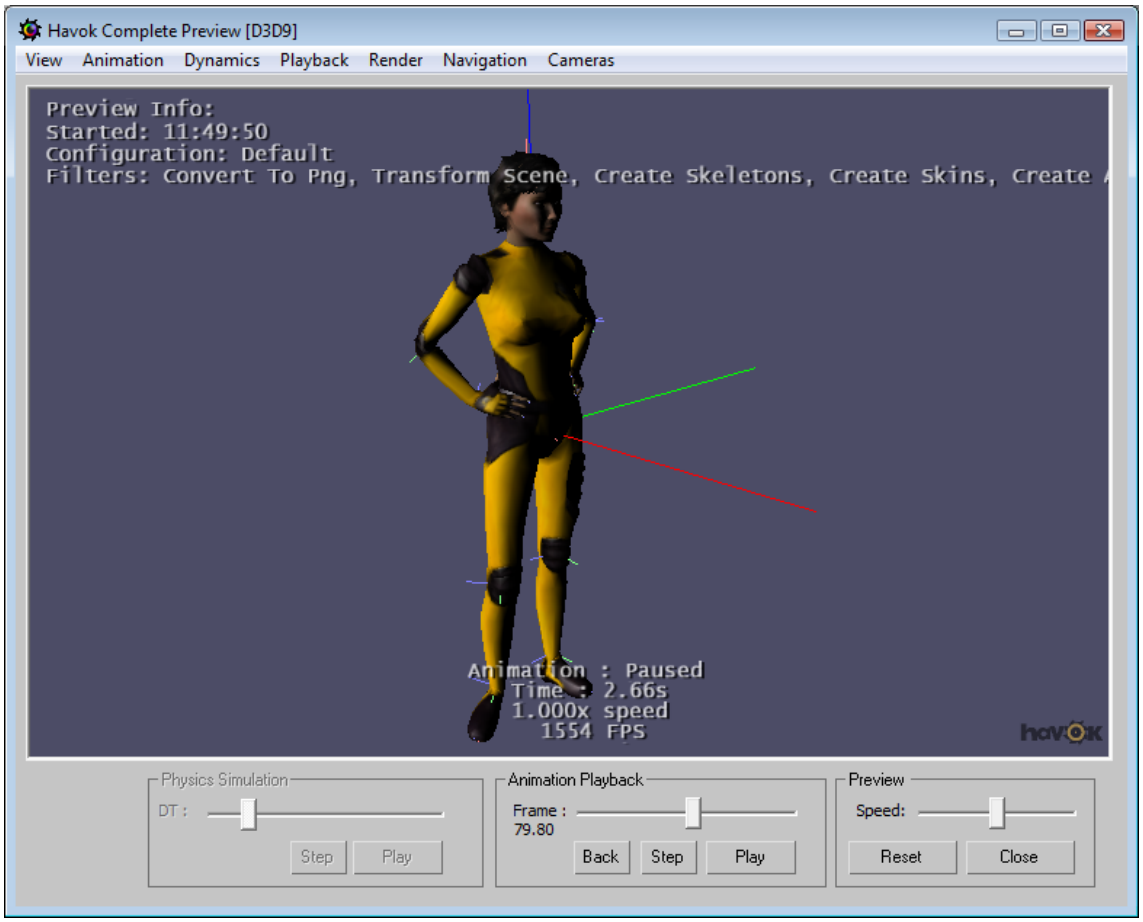



Now we can see that it is the red X-axis of the root (pelvis) which we wish to reorient, as this is the bone which is facing 'forward' in the space of the character, so this is our **Root Forward** . It is also clear that the (world) axis we want to rotate the root *around* is the blue Z-axis, so this is our **World Up Axis** . And our desired direction to align the root to is, as we said at the beginning, the X-axis so this is our **Alignment Axis** .

Thus we add a Rotate Animation filter immediately after the Create Animation filter with the setting as follows:



The result of this is as desired, with the character now having its forward direction facing down the world X-axis:




#### 4.3.5.7 Spline Compression

**Components Required** : Havok Animation

The Spline Compression filter provides an interactive interface for creating spline compressed animation data. The spline compression filter interface is shown in **??**. This section of the manual describes the Spline Compression filter interface. For detailed information about Havok's Spline Compression technology, please refer to the *Spline Compression* section of the Animation chapter.

Note that this filter expects to find at least one Interleaved Animation in the data stream (it usually follows a Create Animations filter).

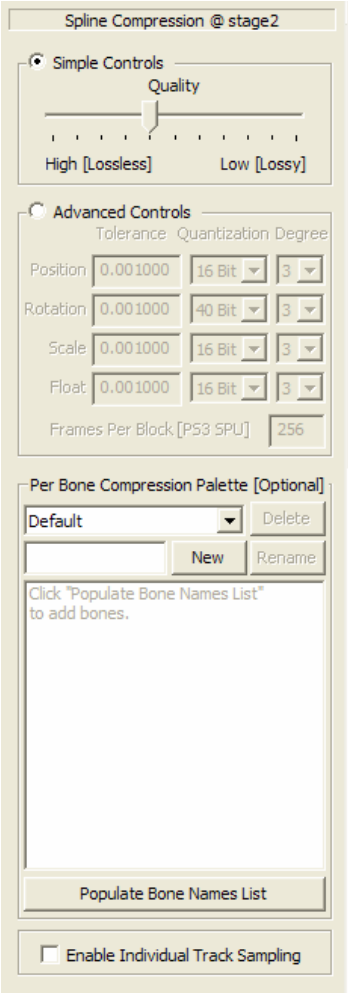



Figure 4.11: Spline Compression Filter Interface

The spline compression interface consists of four panes which are described in **??**, **??**,**??**, and **??** respectively. Each time the spline compression filter is run the size of the resulting compressed animation and the effective compression ratio will be displayed in the **Log Window** . Users of the PlayStation®3 SPU technology will additionally want to note the "Size of largest block" information.

### Simple Controls

The simple controls pane is shown in **??**. The slider bar provides a single control to increase or decrease the quality of the compressed animation. Users may try various quality settings and preview the results by using the Preview Tool Filter and clicking the **Run Configuration** button. The size of the resulting compressed animation is displayed in the **Log Window** .

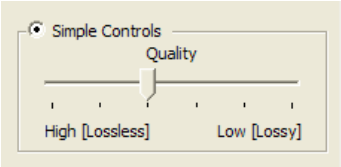



Figure 4.12: Simple Controls Pane

The simple control slider modifies the rotation tolerance of the compression settings. Each tick mark on the quality control slider corresponds to a 10x change in the rotation tolerance. The simple controls work best with animations which consist primarily of rotations. Users desiring more control over specific compression features may activate the Advanced Controls pane described in **??**.

### Advanced Controls

The advanced controls pane is shown in **??**. This pane allows users full control over all spline compression settings. The Tolerance value typically has the greatest impact on the quality of the animation and may be fine-tuned to the user's preference. The Tolerance setting should be the primary value used to govern compression settings. The Quantization settings may be changed in situations where higher compression ratios are desired and quality may be sacrificed. The Degree setting controls the smoothness of the result; degree 3 curves will tend to be more visually appealing than degree 1 curves at a modest cost in output file size. Each of these settings is described in more detail in the *Spline Compression Overview* section of the Animation chapter.

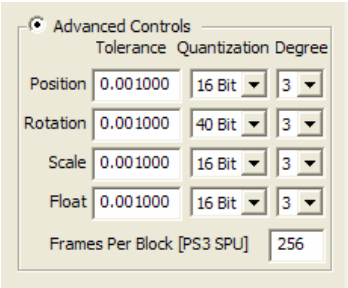



Figure 4.13: Advanced Controls Pane

Users of the PlayStation®3 SPU technology will additionally want to note the "Size of largest block" information printed to the **Log Window** . A warning will be issued if the animation is in danger of being too large to fit within the SPU memory requirements. If a warning is found, the size may be reduced by lowering the "Frames Per Block" setting at the bottom of the Advanced Controls pane. Reducing this value may provide a modest speed increase at the expense of additional memory usage.

Users desiring a simplified settings interface may activate the Simple Controls pane described in **??**.

### Speed Considerations

Spline decompression is optimized for speed for several platforms (refer to the console-specific documentation for quantitative timings). Decompression speed is largely unaffected by tolerance settings, although large values may increase speed as larger number of tracks will be marked static, rather than being represented as NURBS. The rotation quantization type may also have a modest impact, though this setting may have significant affects on decompression quality. Choosing a lower degree may also provide a modest speed boost with some affect on quality.

### Per Bone Compression Settings

The Per Bone Compression Palette pane, shown in **??**, allows users to tailor the compression settings individually for each bone in the skeleton. Per bone compression settings may be used any time the required accuracy of bones varies throughout the skeleton. The default setting is to apply compression settings uniformly for each bone.

Judiciously setting compression parameters per bone can result in a significant reduction in output data size. One such use case is to create level of detail compression by grouping bones roughly by depth in

the hierarchy; tight tolerances could be used for "major bones" such as spine and arm bones while more relaxed tolerances for "minor bones" such as hands and fingers could be used to gain additional savings. A similar use case might involve grouping bones as "upper body" and "lower body". Extremities such as hands often contain a significant number of bones though their visual impact may be limited.

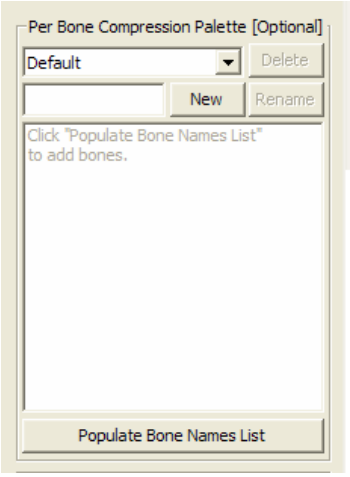



Figure 4.14: Per Bone Compression Palette Pane

The use cases described in the previous paragraph motivated the design of a compression settings palette. A small number of *per bone compression sets* may be created, and multiple bones may be added to each set. Any bones which are not explicitly added to a set are assumed to belong to the Default set.

### Creating a Per Bone Compression Set

The first step to create a new per bone compression set is to populate the list of bone names by clicking the **Populate Bone Names List** button. Note that this step expects at least one skeleton to be present in the data stream (typically created by an earlier Create Skeletons filter). Once the list of bone names has been populated, a per bone compression set may be created. Typing a name in the text box to the left of the **New** button followed by clicking the **New** button will create a new per bone compression set. If the text box has been left empty, the per-bone compression set will be created with a default name. A per bone compression set may be renamed at any time by typing a new name and clicking the **Rename** button. A per bone compression set may be removed at any time by clicking the **Delete** button.

Once a per bone compression set has been created users may add bones by clicking on the bone names in the list. Pressing the Control key while clicking allows multiple bones to be added to the set as shown in **??**. In this example, bones belonging to a character's hand have been added to a per bone compression set named "Minor Bones".

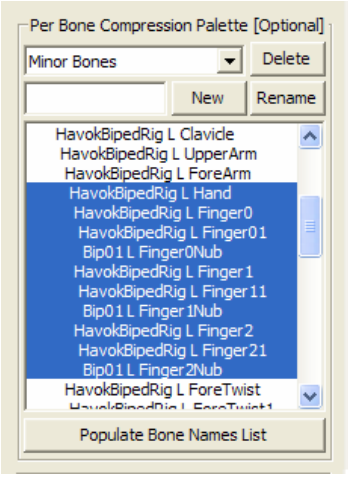



Figure 4.15: Per Bone Compression Palette Showing Added Bone Names

The compression settings for the current set may be adjusted using the controls described in **??** and **??**. These compression settings will be applied to each bone in the current set. Multiple per bone compression sets may be created, each with unique compression settings. Bones may belong to only one set. If a bone name is added to a set, it is automatically removed from all others. Bones which do not belong to any set are considered to belong to the Default set, and the compression settings of the Default set are applied. The Default set compression settings can be changed like any other set. The Default set cannot be renamed or deleted.

**Individual Track Sampling**

The check box shown in **??** allows individual tracks (and bones) of the spline compressed animation to be sampled at runtime. This feature is disabled by default as it requires a modest amount of data to be stored in the compressed animation which indicates the start location in memory for each track. This check box must be checked at the time of content creation in order to use the per bone or per track sampling features of the animation SDK. The affect of choosing this option on the size of the output may be determined by running the filter and checking the **Log Window** .

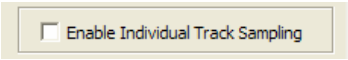



Figure 4.16: Individual Track Sampling Pane

#### 4.3.5.8 Quantized Compression

**Components Required** : Havok Animation

The Quantized Compression filter provides an interactive interface for creating Quantized compressed animation data. The Quantized compression filter interface is shown in **??**. This section of the manual describes the Quantized Compression filter interface. For detailed information about Havok's Quantized Compression technology, please refer to the *Quantized Compression* section of the Animation chapter.

Note that this filter expects to find at least one Interleaved Animation in the data stream (it usually follows a Create Animations filter).

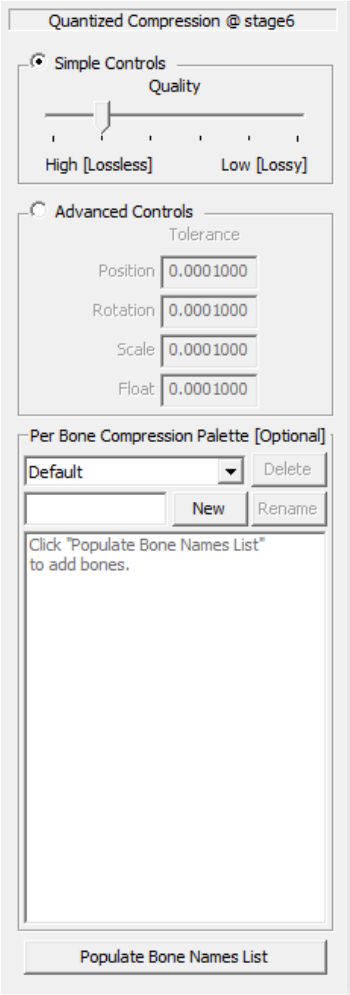



Figure 4.17: Quantized Compression Filter Interface

The Quantized compression interface consists of three panes which are described in **??**, **??**, and **??**, respectively. Each time the Quantized compression filter is run the size of the resulting compressed animation and the effective compression ratio will be displayed in the **Log Window** .

### Simple Controls

The simple controls pane is shown in **??**. The slider bar provides a single control to increase or decrease the quality of the compressed animation. You may try various quality settings and preview the results by using the Preview Tool Filter and clicking the **Run Configuration** button. The size of the resulting compressed animation is displayed in the **Log Window** .

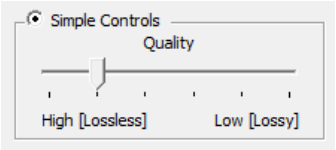



Figure 4.18: Simple Controls Pane

The simple control slider modifies the rotation tolerance of the compression settings. Each tick mark

on the quality control slider corresponds to a 10x change in the rotation tolerance. The simple controls work best with animations which consist primarily of rotations. Users desiring more control over specific compression features may activate the Advanced Controls pane described in **??**.

### Advanced Controls

The advanced controls pane is shown in **??**. This pane allows users independent control over all tolerance settings. These settings are described in more detail in the *Quantized Compression Overview* section of the Animation chapter.

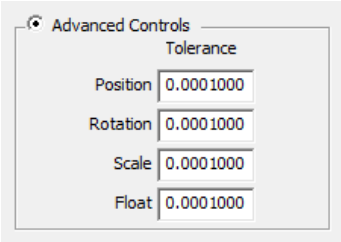



Figure 4.19: Advanced Controls Pane

Users desiring a simplified settings interface may activate the Simple Controls pane described in **??**.

### Speed Considerations

Quantized decompression performance is not very dependent on the Tolerance settings. But larger tolerances may result in some speed improvement as more bones have the potential to be labeled identical to the reference pose.

### Per Bone Compression Settings

The Per Bone Compression Palette pane, shown in **??**, allows users to tailor the compression settings individually for each bone in the skeleton. Per bone compression settings may be used any time the required accuracy of bones varies throughout the skeleton. The default setting is to apply compression settings uniformly for each bone.

Judiciously setting compression parameters per bone can result in a significant reduction in output data size. One such use case is to create level of detail compression by grouping bones roughly by depth in the hierarchy; tight tolerances could be used for "major bones" such as spine and arm bones while more relaxed tolerances for "minor bones" such as hands and fingers could be used to gain additional savings. A similar use case might involve grouping bones as "upper body" and "lower body". Extremities such as hands often contain a significant number of bones though their visual impact may be limited.

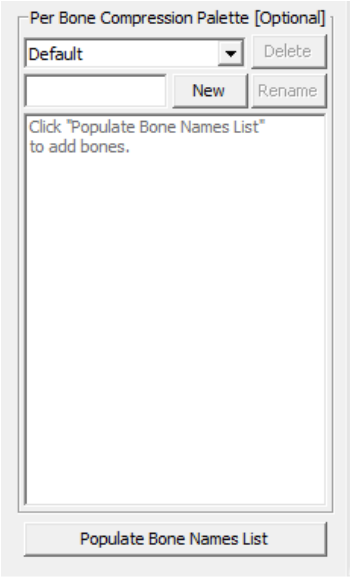



Figure 4.20: Per Bone Compression Palette Pane

The use cases described in the previous paragraph motivated the design of a compression settings palette. A small number of *per bone compression sets* may be created, and multiple bones may be added to each set. Any bones which are not explicitly added to a set are assumed to belong to the Default set.

**Creating a Per Bone Compression Set**

The first step to create a new per bone compression set is to populate the list of bone names by clicking the **Populate Bone Names List** button. Note that this step expects at least one skeleton to be present in the data stream (typically created by an earlier Create Skeletons filter). Once the list of bone names has been populated, a per bone compression set may be created. Typing a name in the text box to the left of the **New** button followed by clicking the **New** button will create a new per bone compression set. If the text box has been left empty, the per-bone compression set will be created with a default name. A per bone compression set may be renamed at any time by typing a new name and clicking the **Rename** button. A per bone compression set may be removed at any time by clicking the **Delete** button.

Once a per bone compression set has been created users may add bones by clicking on the bone names in the list. Pressing the Control key while clicking allows multiple bones to be added to the set as shown in **??**. In this example, bones belonging to a character's hand have been added to a per bone compression set named "Minor Bones".

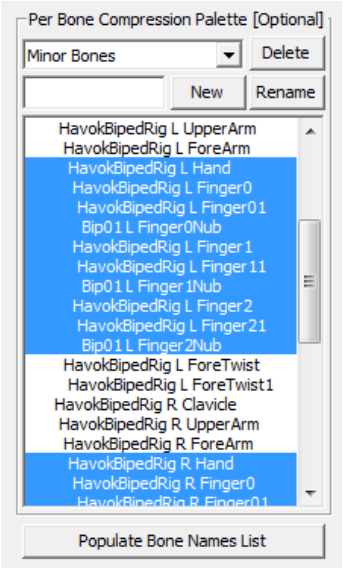



Figure 4.21: Per Bone Compression Palette Showing Added Bone Names

The compression settings for the current set may be adjusted using the controls described in **??** and **??**. These compression settings will be applied to each bone in the current set. Multiple per bone compression sets may be created, each with unique compression settings. Bones may belong to only one set. If a bone name is added to a set, it is automatically removed from all others. Bones which do not belong to any set are considered to belong to the Default set, and the compression settings of the Default set are applied. The Default set compression settings can be changed like any other set. The Default set cannot be renamed or deleted.

#### 4.3.5.9 Predictive Compression

**Components Required** : Havok Animation

The Predictive Compression filter provides an interactive interface for creating Predictive compressed animation data. The Predictive compression filter interface is shown in **??**. This section of the manual describes the Predictive Compression filter interface. For detailed information about Havok's Predictive Compression technology, please refer to the *Predictive Compression* section of the Animation chapter.

Note that this filter expects to find at least one Interleaved Animation in the data stream (it usually follows a Create Animations filter).

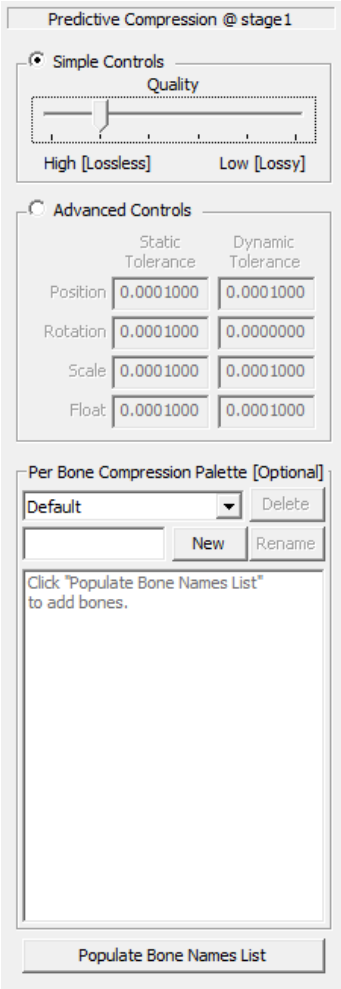



Figure 4.22: Predictive Compression Filter Interface

The Predictive compression interface consists of three panes which are described in **??**, **??**, and **??**, respectively. Each time the Predictive compression filter is run the size of the resulting compressed animation and the effective compression ratio will be displayed in the **Log Window** .

### Simple Controls

The simple controls pane is shown in **??**. The slider bar provides a single control to increase or decrease the quality of the compressed animation. You may try various quality settings and preview the results by using the Preview Tool Filter and clicking the **Run Configuration** button. The size of the resulting compressed animation is displayed in the **Log Window** .

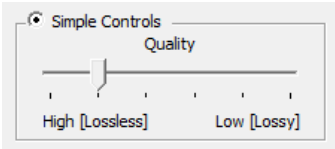



Figure 4.23: Simple Controls Pane

The simple control slider modifies the rotation tolerance of the compression settings. Each tick mark

on the quality control slider corresponds to a 10x change in the rotation tolerance. The simple controls work best with animations which consist primarily of rotations. Users desiring more control over specific compression features may activate the Advanced Controls pane described in **??**.

#### Advanced Controls

The advanced controls pane is shown in **??**. This pane allows users independent control over all tolerance settings. These settings are described in more detail in the *Predictive Compression* section of the Animation chapter.

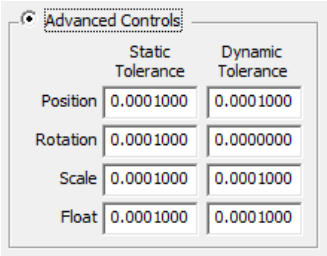



Figure 4.24: Advanced Controls Pane

Users desiring a simplified settings interface may activate the Simple Controls pane described in **??**.

**Note:**
The Dynamic Tolerance for Rotation should usually be kept at zero because it produces more compact animations with maximum quality. Please see the *Predictive Compression* section of the Animation chapter for more information.

#### Speed Considerations

The settings of the Dynamic Tolerances have no impact on decompression performance. Setting the Static Tolerances to higher values may result in some speed improvement as more bones have the potential to be considered to be static.

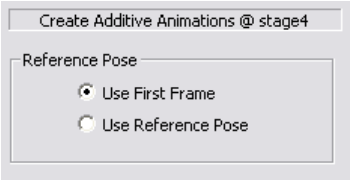
#### Per Bone Compression Settings

The Per Bone Compression Palette pane is identical to the one used for Quantized compression. Please see **??** for information on setting up per-bone compression.

### 4.3.5.10 Create Additive Animations

**Components Required** : Havok Animation

Additive animations allow you to create animation which represent fine character detail. These animation can be reused in many situations and layered on existing animations. The Create Additive Animations filter creates additive animations from all raw interleaved animation found in the data stream. To create this additive animation you can specify some options for which base pose to use when determining the detail in the animation.




There are two available options:

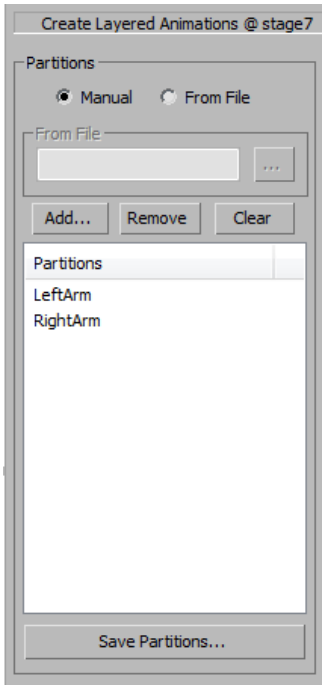
- **First Frame** : This is the default. The first frame of the animation is subtracted from each frame of the entire animation.
- **Reference Pose** : The reference pose of the animation (of the first rig in the scene) is subtracted from each frame of the entire animation.

Note that no additive animations will be created unless an interleaved animation has already been added to the stream (using the Create Animation filter). If you are using the **Reference Pose** option you must also have at least one skeleton there too (using the Create Skeleton filter).

#### 4.3.5.11 Create Partitioned Animations

**Components Required** : Havok Animation

Partitioned Animations allow you to create partial animations that only contain transforms for bones in predefined skeleton partitions (created in the Create Skeleton filter). The Create Partitioned Animations filter only works if partitions are first setup in the skeleton. The input to this filter must be an uncompressed animation data, meaning that this filter must appear before any compression filters.



The filter allows for two modes of setting up the partitions for the animations:

- **Manual** : This allows you to construct the partition list by directly specifying partitions from the skeleton. With this option selected, the controls allowing manual selection become enabled. To add a node click on the corresponding **Add...** button. This will bring up the *Select Partition* dialog which allows you to choose a partition node from the skeleton. To remove a partition simply highlight it in the list at any time and click **Remove** . To empty the list, press **Clear** .
- **From File** : The file loaded is a text file containing the names of the partitions that should be contained in the rig. Each line of the file corresponds to a different partition. *The path for the filename is always stored relative to the original asset.* With this option selected, the control allowing you to specify the filename becomes enabled. To create a new partition file, use the **Save Partitions...** button.

#### 4.3.5.12 Loop Animations

**Components Required** : Havok Animation

This filter forces an animation to loop by cross fading the start and the end of the animation over a number of frames. This number is specified by the filter options.

The cross fade happens for the specified number of frames at both the start and the end of the animation.

Usually this filter is applied after a Create Animations filter.

#### 4.3.5.13 Create Mapping

**Components Required** : Havok Animation

Sometimes we want to play an animation (or, in more general terms, convert a pose) on either a different character, or on different skeletal representations of the same character. The Create Mapping filter allows users to set up mappings for each of these use cases. Transferring a pose from one resolution to another of the same character is referred to as "Ragdoll Mapping" after the most common use case. Transferring a pose and extracted motion from one character to a different character is referred to as "Animation Retargeting".

For further information on create a mapping appropriate for ragdolls, or for different resolutions of the same character, please refer to **??**. For further information on create a mapping appropriate for retargeting animations between different characters, please refer to **??**.

**Creating Ragdoll Mappings**

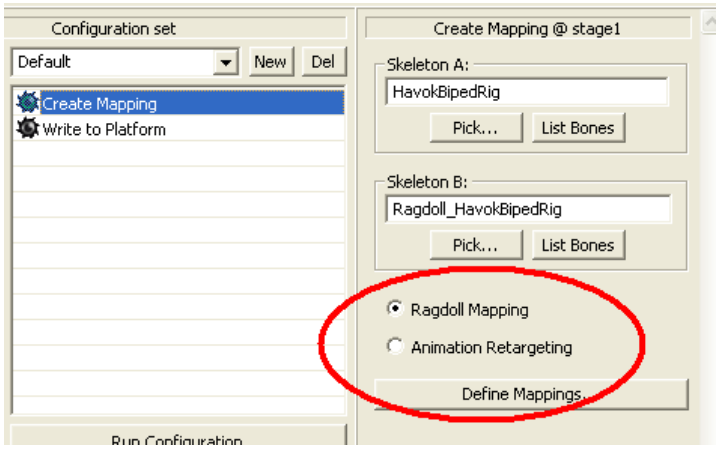
For Ragdoll Mapping, the most common use case is the transformation between a high-res (high bone count) representation of a character to a low-res (low bone count) version (a ragdoll for example), and

vice versa. In the first case, we may want to transform a high-res pose into a ragdoll pose in order to drive the ragdoll; in the second case, we may want to convert a ragdoll pose into a high-res pose in order to skin the ragdoll (or blend its behavior). In most cases, we will want to do both conversions multiple times.

A skeleton mapper allows us to do these conversions. The Create Mapping filter sets up all the information required by the skeleton mapper (hkaSkeletonMapper) at run time by comparing and matching two skeletons (in their reference pose), finding pairs of bones and pairs of chain of bones.

**Note:**

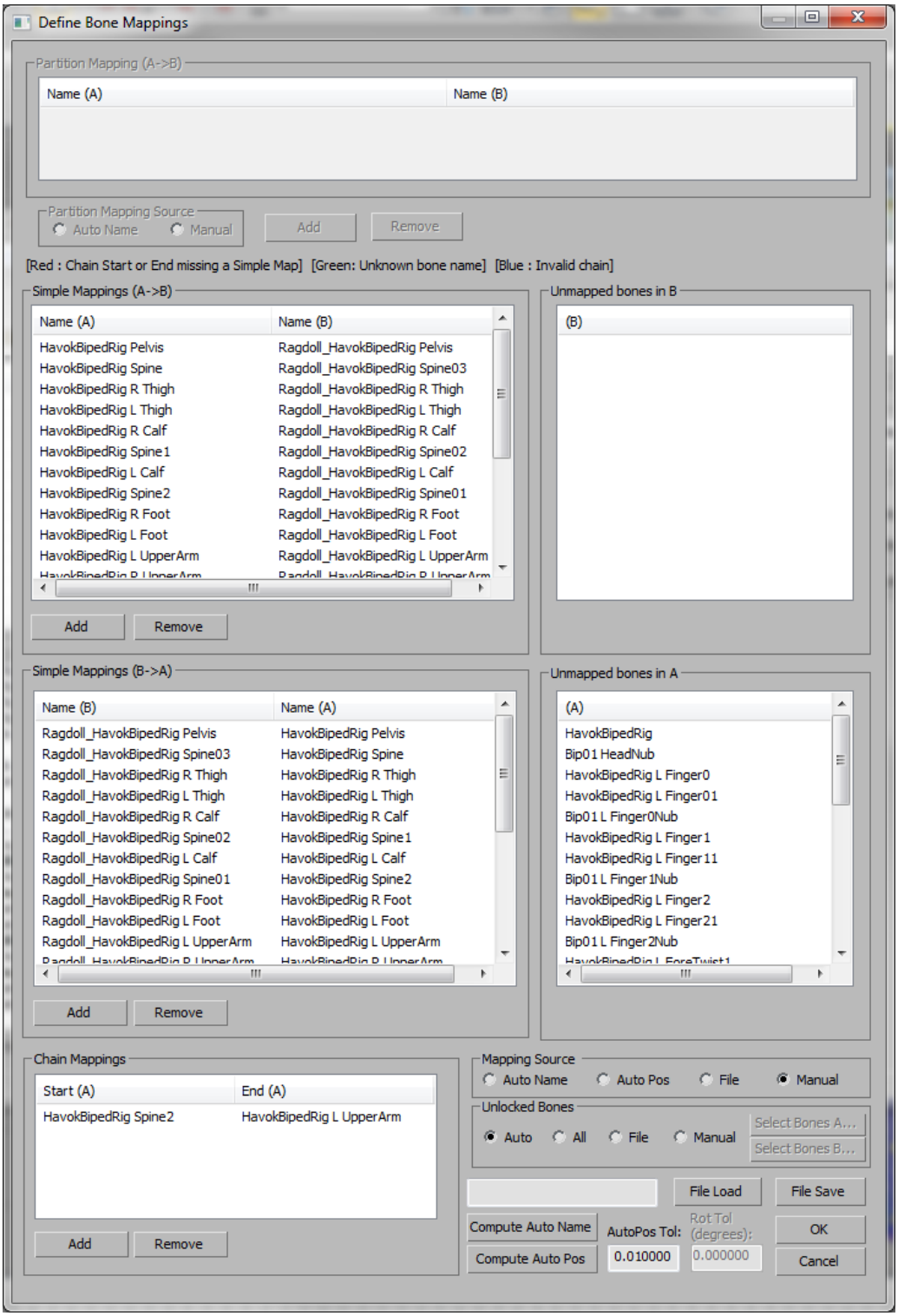
Be sure to choose "Ragdoll Mapping" mode in the Create Mapping Filter dialog to ensure that the result is appropriate for mapping ragdolls.




Thus, the minimal information a skeleton mapper requires is two skeletons, A and B. If not specified, the filter will pick A as the first skeleton, and B as the first skeleton associated with a ragdoll that are found. It is always best to manually specify both skeletons using the **Pick** buttons. Also, you won't be able to see/customize the mapping (use the **Define Mappings...** button) unless both skeletons have been specified.

In order for a skeleton mapper to operate properly at runtime, it requires knowledge about which bones/joints can translate and which can not. By selecting the **Lock Bone Translations Automatically** option, the filter will also flag all bones in both skeletons (except the roots) as rotation-only.

The **Define Mappings...** button opens a new window, which allows you to view and edit the mapping information, which is used at runtime in order to convert poses from A to B and from B to A.




### Simple Mappings

A "Simple Mapping" is a one-to-one association from a bone in one skeleton to a bone in the other skeleton. For example, a typical simple mapping from HighRes to RagDoll would be "HighRes Head" is mapped to "RagDoll Head". This means that, at runtime, the position and orientation of "HighRes

Head" will be used to place the "RagDoll Head".

Usually, for every Simple Mapping A->B there is also the symmetrical mapping B->A (in the example, "RagDoll Head"->"HighRes Head").

Simple mappings are listed in the first and second windows on the left side of the Define Bone Mappings window. In *Manual Mode*, you can add and remove simple mappings to either A->B or B->A by using the **Add** and **Remove** buttons in each list.

**Chain Mappings**

Chain mappings are atomic associations between 2 or more bones in one skeleton to 2 or more bones in the other skeleton. The best example (and the most common usage) is a spine: while a high-res skeleton may have 5 or more spine links, a ragdoll version of the same character may have just 1 or two links (rigid bodies).

A chain mapping drives a chain of bones by trying to match, as well as possible, the start and end bones of the chain to another chain. In the example of the spine, it may try to match the chain "spine1"-"spine2"-"spine3"-"spine4"-"spine5"-"head" ("spine1-head") to the chain "ragdollspine1"-"ragdollspine2"-"ragdollhead" ("ragdollspine1-ragdoll head").

A condition for the creation of a chain mapping is that both the start and the end of the chains have to be simple mappings; continuing with the spine example, spine1/ragdollspine1 and head/ragdoll head need to be simple mappings.

Chain Mappings are listed at the bottom of the Define Mappings dialog. They are listed with the names of the bones in skeleton A (the associated chain in B can be deducted by applying the A->B simple mappings). In *Manual Mode*, you can add and remove Chain Mappings by using the **Add** and **Remove** buttons.

**Partition Mappings**

Partition mappings are an optional mapping available only when there exist partitions within the skeleton (see Create Partitions Dialog) and only when using the Animation Retargeting Mappings. By creating a partition mapping it allows you to retarget a subset (or subsets) of bones in Skeleton A to a subset (or subsets) of bones in Skeleton B.

There are two ways to set up the partition mappings:

- **Auto Name** : Simple mappings A->B and B->A for partitions will be created by detecting partitions in both skeletons which have the same name. Note: The filter looks for an exact match on the partition names.
- **Manual** : In this mode you are able to modify the mapping information manually. This option is useful in the cases where mapping information calculated automatically (by the **Auto Name** option) may not be enough to provide a perfect mapping.

Bones that are mapped with simple mappings or chain mapping in a partition in Skeleton A must be in the corresponding mapped partition of Skeleton B.

**Unmapped Bones**

After applying all Simple and Chain Mappings from one skeleton to another, it is possible for many of

the bones in the destination skeleton not to have been modified. This is usually the case for the skeleton of higher resolution, and is uncommon for the low-res skeleton (ragdoll skeletons, for example).

When a bone is unmapped, its position and orientation is kept relative to (it moves with) its parent (i.e., its local transform is not modified).

Unmapped bones are listed on the right hand side of the Define Bone Mappings dialog. Any bones that are not part of a simple mapping or a chain mapping are listed there.

### Mapping Sources

There are different ways this mapping information can be set up: they are reflected in the "Mapping Sources" radio buttons:

- **Auto Name** : Simple mappings A->B and B->A will be created by detecting bones in both skeletons which have the same name (names are broken into words separated by spaces, underscores or periods. The last word is kept, though keywords such as "L", "R" "Left" and "Right" are also kept [case insensitive]). This option allows more control from the user (as you can carefully choose the names according to your desired mapping) but it involves extra work in order to name all the bones, and therefore is error prone. This mode is provided for backwards compatibility with older assets.

  In *Manual Mode*, you can still fill the mapping information in the dialog with the results of Auto Name by clicking on the **Compute Auto Name** button.

- **Auto Pos** : The most accurate and simple method to use is Auto Pos. This method will compare the position of the bones in each skeleton and create simple mappings A->B and B->A for each pair of bones whose position match within a tolerance (specified by the **AutoPos Tolerance** edit box).

  **Note:**
  Using this method requires that both skeletons (and their bones) are properly aligned (this alignment is also important for the proper behavior of the skeleton mapper).

  In *Manual Mode*, you can fill the mapping information in the dialog with the results of the Auto Pos calculations by clicking on the **Compute Auto Pos** button.

  Both Auto Name and Auto Pos methods first discover simple mappings. Chain Mappings are then constructed by detecting pairs of simple mappings where the bones in-between haven't been mapped (think of the "spine1" - "head" example).

- **File** : Mapping information can also be stored in a file and reused. This can be useful in order to protect the mapping information from being modified. The file in particular is specified in the edit box below, and its path is relative to the location of the asset. The **File Load** and **File Save** buttons allow you to save and load the current mapping information.

- **Manual** : In this mode you are able to modify the mapping information manually. This option is useful in the cases where mapping information calculated automatically (by the **Auto Pos** or **Auto Name** options, for example) may not be enough to provide a perfect mapping.

### Unlocked Bones

It is possible to unlock bones using the Create Mappings Filter. Locking bones is important for mapping ragdolls back to high resolution skeletons as it prevents unwanted stretching of bones that can sometimes occur due to the nature of mapping. Not all bones should be locked however. The most important bones to unlock are those that are responsible for movement of the character such as the root or pelvis node.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

Additionally bones that have siblings, typically the clavicles and hips, can also be unlocked to improve quality when mapping from a ragdoll skeleton to a high resolution skeleton. The define mappings filter allows several different ways to best specify which bones to unlock, as shown below.

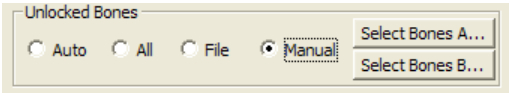



Figure 4.25: Unlocking Bones

Typically "Auto" is sufficient. Automatic mode will attempt to unlock any "mover" bones (such as pelvis or root bones) that are found. If finer control is required, either "File" or "Manual" mode may be used. An example of Manual selection is shown below.

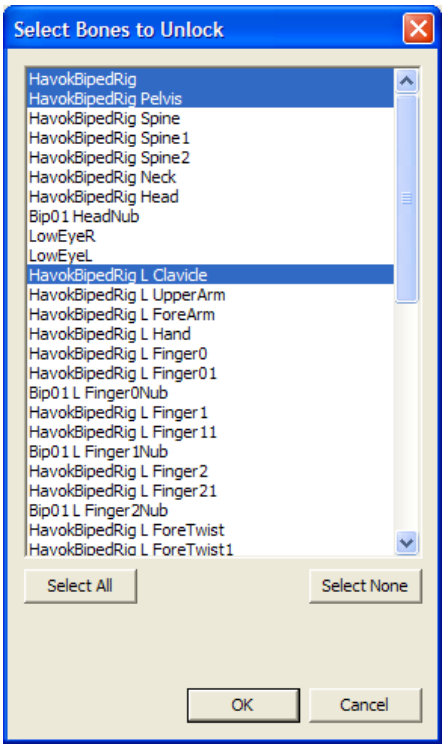



Figure 4.26: Unlocking Individual Bones

### Creating Animation Retargeting Mappings

The goal of using the Havok Content Tools for Animation Retargeting setup is to export `hkaSkeletonMapper` objects which can be used at runtime to perform animation retargeting. To create the `hkaSkeletonMapper` objects, two `hkaSkeleton` objects must be present in the scene, one for each character to be retargeted. When `hkaSkeleton` objects are created using the Create Skeletons Filter, it is important that each skeleton be given a unique name as shown below in **??**; otherwise both skeletons may have the same name and cannot be disambiguated by the Havok Content Tools.

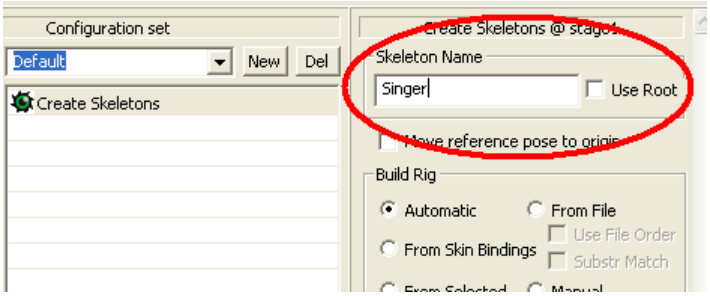



Figure 4.27: Creating Skeletons With Unique Names

**Note:**

The `hkaSkeleton` objects of each character must be present in the scene to create a retargeting mapping. This can be accomplished by either merging both characters into the same modeler scene, or alternately exporting each character's `hkaSkeleton` separately and loading both `hkaSkeleton` assets into the hctStandAloneFilterManager together. The latter technique avoids any ambiguity of having two characters with similar bone names loaded in the modeler at the same time. It also avoids the potential need to create N squared modeler files to retarget among N characters.

Retargeting `hkaSkeletonMapper` objects are created using the Create Mapping Filter. To use this filter, choose each of the `hkaSkeleton` objects corresponding to the characters to be retargeted; these will be the same ones created in the previous step. It does not matter which is chosen as "Skeleton A" and which is chosen as "Skeleton B", the Create Mapping Filter will create Retargeting `hkaSkeletonMapper` objects for retargeting in both directions. Be sure to choose the "Animation Retargeting" option as shown below in **??**.

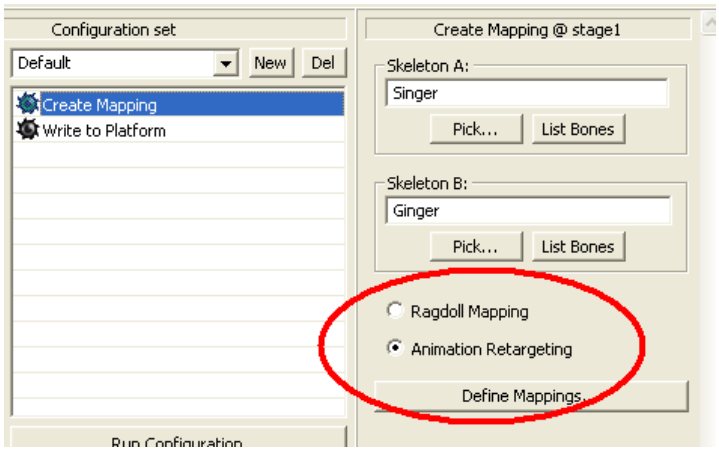



Figure 4.28: Creating a Retargeting Mapping

At the heart of the Retargeting setup process is defining the correspondence between bones of each skeleton. This is done by choosing the "Define Mappings..." button of the Create Mapping Filter. An example setup is shown below in **??**. The advance example below shows the mapping between two characters of different resolutions requiring a number of chain mappings.

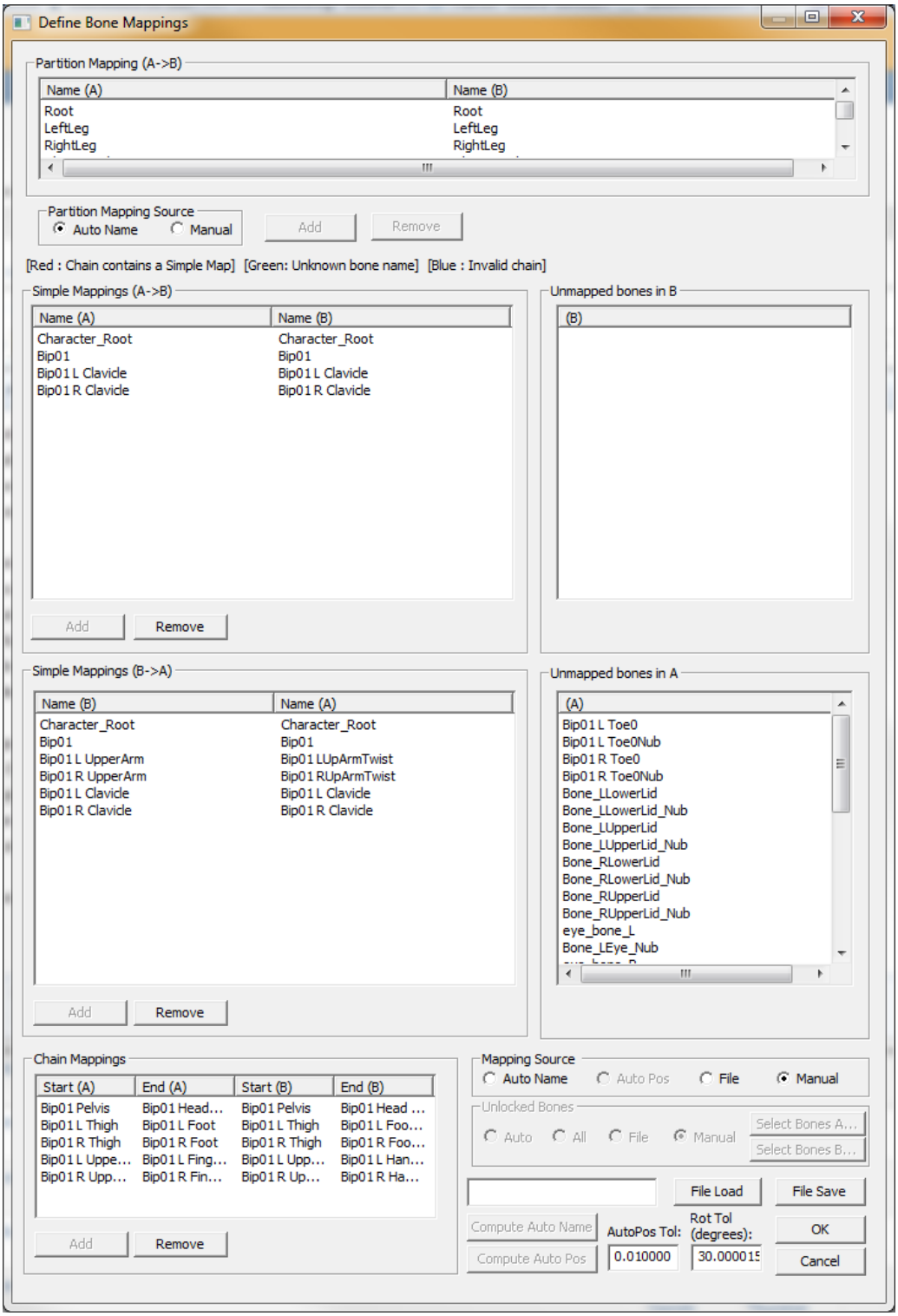



Figure 4.29: Creating a Retargeting Bone Correspondence

Two types of bone mappings may be created: Simple mappings and Chain mappings. Retargeting between characters of similar structure will typically involve a large number of simple mappings and few, if any, chain mappings. Structurally dissimilar characters, or characters of different resolutions may require one or more chain mappings.

Partition mappings are also optionally available for retargeting. Mapped partitions allow for partial animations containing only specified partitions for one skeleton to be retargeted to the corresponding partitions of the mapped skeleton.

Very short bones can cause quality issues when retargeting. The "AutoPos Tol" defines a length in world units below which bones are considered to have zero length. Any bones shorter than this value are considered to have the exact same length of zero. Additionally, bones of zero length should not be retargeted to bones of non-zero length (essentially this leads to a divide by zero in implementation). The `hkaSkeletonMapperUtils::createMapping()` function will warn if it finds two such mapped bones.

The quality of Retargeting also degrades as the difference in orientation between the bones of the original and retargeted characters grows large. Although there may be some retargetings in which large differences in angle are acceptable, typical corresponding bones should not differ in orientation by more than a few tens of degrees (30 degrees max is a good rule of thumb). The "Rot Tol (Degrees)" setting can be used to warn the user if two bones differ by more than a small change in angle. Additionally all retargeted characters should all have the same up and forward directions in the reference pose.

Please see the Animation Retargeting Section of the Havok Animation documentation for a full description of retargeting setup and the associated options.

### 4.3.6 Graphics Filters

Graphics filters are those which modify/visualize graphical elements of the asset. These can always be used regardless of which Havok Product is licensed for use at runtime.

#### 4.3.6.1 Preview Scene

**Components Required** : Common

The Preview Scene filter has been replaced with the Preview Tool filter.

#### 4.3.6.2 Convert to PNG

**Components Required** : Common

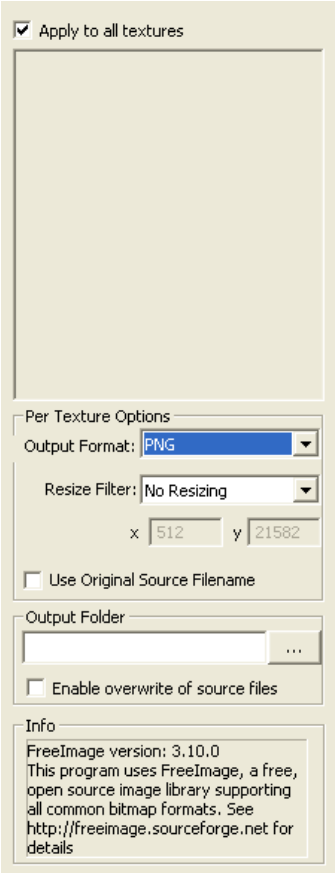
**Note:**
This filter is deprecated as it supports limited image conversion (just uses Windows GDI+) and does not expose enough functionality. It has been replaced by the GatherAndConvertTexture, CompressTexture, and EmbedTexture filters that have between them a lot more useful functions.

This filter will convert any texture in the scene pointing to an external file into an "inplace" texture, stored in PNG format. It uses GDI+ (a Windows module) to do the conversion so will only be able to convert BMP and say JPEG to PNG, for files that are PNGs already they will just be placed in the scene as is (inplace). Same for DDS (DirectDraw surfaces) and TGA files, they too will not be converted (as GDI+ does not deal with them) , just placed inline in the scene. DDS and TGA files will not be resized either. See the other texture manipulation filters for more advanced use. This filter is really just for our demos. If **Texture Resize** is on, the filter will also resize any converted texture to the given size.

#### 4.3.6.3 Gather And Convert Texture

**Components Required** : Common

This filter uses FreeImage (freeimage.sourceforge.net), a popular open source image conversion library, to convert any textures (inplace or file based) from one supported format to another, and also place that converted texture (if file based) into an output folder. If you want to re-compile this filter you will need to get the FreeImage SDK (ver 3.15 used in the Havok 2013.3 release, but any recent version should compile fine).



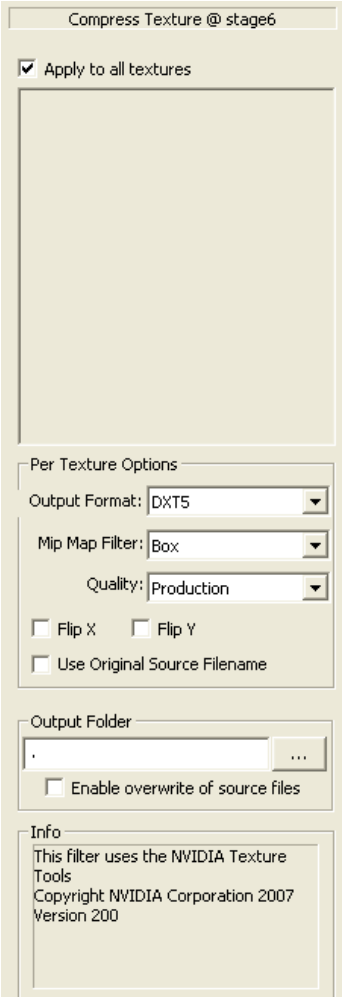

The current options expose the support formats as BMP, JPEG, PNG, TARGA (TGA), TIFF and HDR. This is really just to limit the options available, as FreeImage supports a lot more formats than this (feel free to recompile this filter and add some more if you need them). While converting the texture, the filter can apply a resize (different image resize filters provided by FreeImage are options). You can choose an output folder if you prefer (source file dir otherwise), and whether if the input and output paths match, whether to allow the original file to be overwritten.

#### 4.3.6.4 Compress Texture

**Components Required** : Common

A very common texture compression technique supported in games is S3 block texture compression

(S3TC, commonly referred to as DXT, en.wikipedia.org/wiki/S3TC ). Most image formats do not support it (such as PNG and so on) but the Direct Draw Surface (DDS) does (msdn.microsoft.com/en-us/library/bb943990(VS.85).aspx ) and is widely used in games not only for that but because it is a multiple layer format and thus can store mip-maps, 3D volumed textures, cube maps, and so on, all in various hardware orientated pixel formats. Thus it can be very useful to create these textures as part of the export pipeline for a given asset if not done already. We provide a simple filter to do this for you.

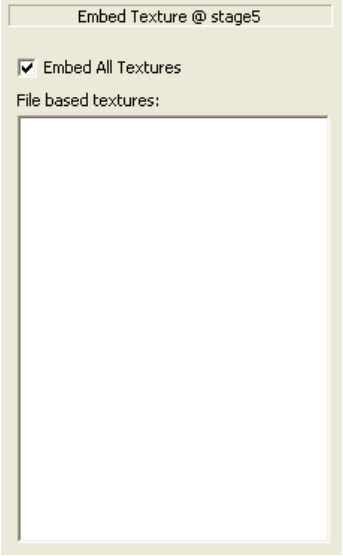



This filter uses the open source Nvidia Texture Tools (NVTT, code.google.com/p/nvidia-texture-tools/) which itself is based on libSqiush (code.google.com/p/libsquish/). It also uses FreeImage (just like the Gather and Convert Texture filter above). We use NVTT version 2.0 in the Havok 2013.3 build. These provide fast and stable conversion from any FreeImage support inplace (embedded) or file based texture to a DDS file, that can optionally be compressed, and also have its mip chain created as well. You can flip the texture vertically and/or horizontally too. Similar to the Gather and Convert texture filter, you can specify the output directory and whether overwriting the original is allowed or not (if file based). Note that DXT compression is a Lossy compression technique, so you can expect artifacts. Please refer to all the normal sources for further information on DXT compression and choosing the correct form for your usage.

#### 4.3.6.5 Embed Texture

**Components Required** : Common

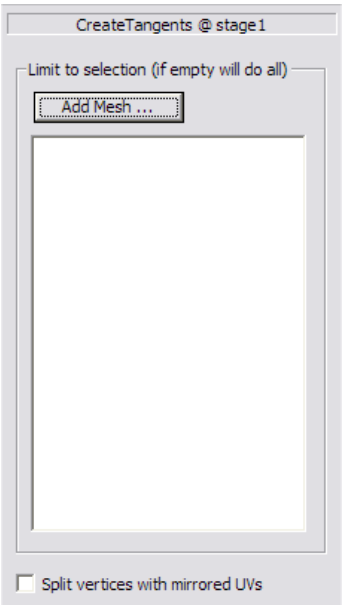
This texture filter simply takes all file based texture references (the default type for all file based textures from the Modelers) and embeds them directly in the HKX file as 'inplace' textures. This is commonly done to keep all assets self contained, but does bloat the HKX file and also any changes to the texture will require re-export. As such it is more a Havok Demo filter than a normal product filter. In the Havok demo framework embedded textures are not shared across assets, so also result in wasted video memory if the texture is used across multiple assets where they all embed their copy. There are not really any options for this filter other than what subset of textures to affect:




#### 4.3.6.6 Create Tangents

**Components Required** : Common

This filter will calculate tangents and binormals of the meshes in the scene by examining normals and texture UV coordinates.



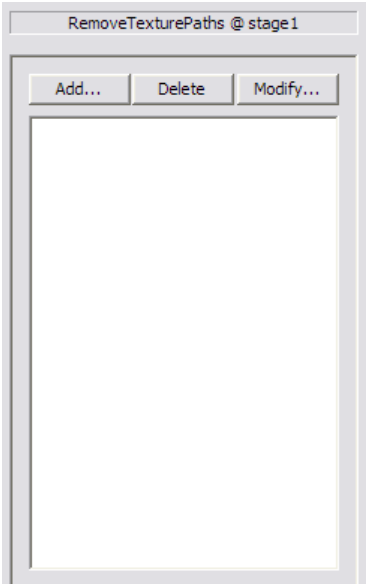

By default, the filter will operate on all meshes. Alternatively, you can use the filter options to specify which individual meshes to process.

When the **Split vertices with mirrored UVs** check box is selected, the filter will also search for vertices shared by triangles with mirrored UVs, and duplicate them accordingly (since tangent information will be different depending on the triangle).

#### 4.3.6.7 Remove Texture Paths

**Components Required** : Common

This filter can be used to convert absolute texture paths into relative ones.



The filter options allows you to define a list of texture folders. When the filter is executed, it will search through textures, and for any texture in the scene that points to one of the specified folders, it will convert the absolute path into a relative one (relative to that folder in particular).

## 4.4 3ds Max Tools

### 4.4.1 Introduction

In this chapter we present the 3ds Max-specific components of the Havok Content Tools:

- The 3ds Max Scene Exporter
- The 3ds Max Physics Tools
- The 3ds Max Animation Tools

We also present several tutorials:

- Export and Animation Basics: How to set up, export and process an animation.
- Physics Basics: How to set up, export and process a physics scene.
- More On Rigid Bodies: A few more concepts about rigid body setup.
- Rag Doll Setup: How to set up, export and process a ragdoll and its association with a high-res skeleton.

Check also:

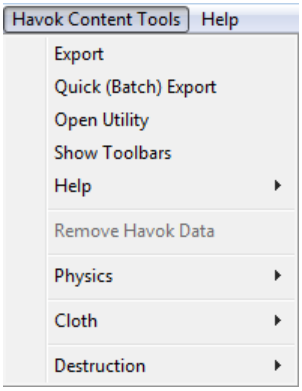
- Common Concepts: For common concepts useful for all Havok Tools (regardless of modeler).
- The Havok Filter Pipeline: For details on how the content created by the 3ds Max tools can be processed.
- Extending the 3ds Max Tools: For information about exporting custom data and MAXScript/C++ access.

### 4.4.2 3ds Max: Scene Exporter

Havok's 3ds Max Scene Exporter is responsible for exploring 3ds Max's scene graph, collecting scene information, and passing it to the filter pipeline for processing.

#### 4.4.2.1 Accessing the Scene Exporter

The Scene Exporter is implemented as a 3ds Max Utility object. The best way to access this utility and many of its options is through the **Havok Content Tools** menu and toolbar. If the toolbar is not visible then you should select the **Havok Content Tools** > **Show Toolbars** menu option.

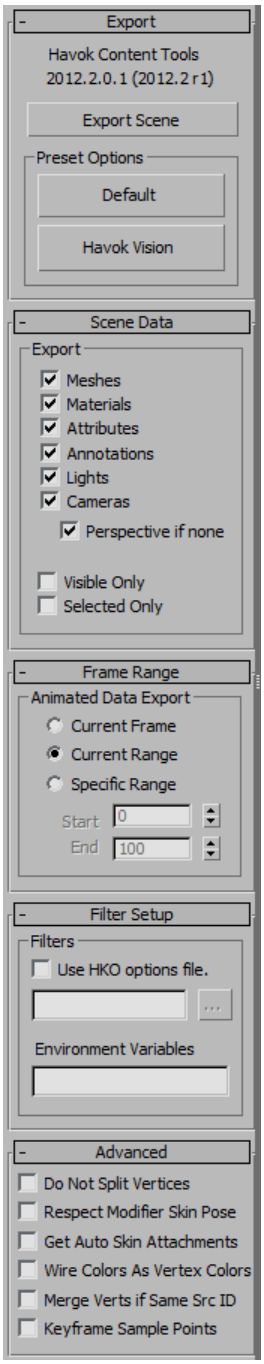


Most of the items in the menu and toolbar relate to the Physics 2012 and Cloth Tools and are described in their own section. Let's focus only on the options related to the Scene Exporter:

- **Export** : This invokes the Scene Exporter, which will navigate the current scene graph extracting information about meshes, lights, nodes, attributes, etc (according to the current export options). After this is done, the Havok Filter Manager will be invoked in order to process this data.
- **Open Utility** : This command will open the Havok Scene Exporter Utility, where various export options can be defined. These options are described in the following section.
- **Quick (Batch) Export** : This invokes the filter manager in batch mode, where the filter configurations in the filter manager will be executed without opening the filter manager UI. This can be very useful when you just want to reexport an asset (either to a file or to a remote preview) once the filter pipeline has already been setup

#### 4.4.2.2 Export Options

The export options for the 3ds Max Scene Exporter are exposed through the utility interface. The export options UI is displayed by pressing the button or selecting ' **Havok Content Tools** > **Open Utility** '.




- **Export Scene** : This button performs the same operation as the **Export** option in the **Havok Content Tools** menu or the button in the toolbar.
- **Preset Options** : This group can set the options to be compatible with **Havok Vision** , or the **Default** Havok settings. This will only change the options in the **Advanced** rollout.

The **Scene Data** rollout controls which parts of the scene will be exported or not:

- **Meshes** : Controls whether meshes are exported or not. Enabled by default. Note that the mesh

will not be exported if the "Renderable" attribute is unchecked in the object properties.

- **Materials** : Controls whether materials and textures are exported or not. Enabled by default.
- **Attributes** : Controls whether custom Havok attributes are exported or not. Enabled by default.
- **Annotations** : Controls whether annotations are exported or not. Enabled by default.
- **Lights** : Controls whether lights are exported or not. Enabled by default.
- **Cameras** : Controls whether cameras are exported or not. Enabled by default.
  - **Perspective if none** : If this is enabled and no cameras are available to export then a default camera, based on the perspective viewport, will be exported with the scene. Enabled by default.
- **Visible Only** : Only objects which are currently visible in 3ds Max will be exported. Disabled by default.
- **Selected Only** : Only objects which are currently selected in 3ds Max (along with all their ancestors) will be exported. Disabled by default.

The **Frame Range** rollout controls how animated data is handled by the exporter:

- **Current Frame** : The exporter will only sample the scene at the current frame (so no animation data will be exported). This option can considerably speed up the export process.
- **Current Range** : Animated data (node transforms, annotations, attributes) will be exported; the animation will be sampled through the current animation range in 3ds Max. This is the default option.
- **Specific Range** : This allows you to specify an animation range (through the **Start** and **End** edit boxes) for the scene exporter to sample animation. This option is useful if you want to control the range of the animation export regardless of the current range in 3ds Max.

The **Filter Setup** rollout provides the ability to customize some elements of the filter pipeline:

- **Use HKO Options File** : Enabling this check box allows you to choose an HKO file that contains the desired filter configurations to use for processing the asset. Options (filter configurations) are saved with the asset (the max scene), but sometimes it is useful to specify a custom set of options explicitly (for example, during batch processing, or to avoid having to build a filter set up from scratch in new assets).
- **Environment Variables** : Here you can specify pairs of (variable, value) by concatenating strings of the form: **VARIABLE1=VALUE1; VARIABLE2=VALUE2;** etc. This data is not usually stored with the final asset but can be used by a (custom) filter during process. For example, the Write to Platform filter will replace any environment variables specified in its **Filename** field by their values. For more, see the section on environment variables.

The **Advanced** rollout contains other miscellaneous, less-used options:

- **Do Not Split Vertices** :

  By default, the 3ds Max Scene Exporter will export as many mesh vertices as required in order to store texture and normals. So, for example, while for a cube only 8 geometric vertices would be required, in order to store per-face normal and texture coordinates, 24 vertices need to be exported.

  Selecting this option will override the default behaviour, and multiple vertices that have the same position will be exported as one, regardless of their normal or UV coordinates. This may reduce the size of the exported mesh, but due to the lack of accurate normal and texture coordinates the mesh may not display as it originally did in the modeler.

- **Respect Modifier Skin Pose** :

  Toggle the bind pose of child bones as this also influences the unskinned mesh's transform. This is required if using the Process Vision Data filter to export to Havok Vision.

- **Get Auto Skin Attachments** :

  By default, mesh nodes parented to bones in a bone hierarchy (not skinned to bones) will not be exported; setting 'Get Auto Skin Attachments' will skin such attachments to their parents. Must be selected if using the Process Vision Data filter to export Vision animations.

- **Wire Colors As Vertex Colors** :

  If selected meshes with no material will use the wire colors as the corresponding vertex colors during export.

- **Merge Verts if Same Src ID** :

  Selecting this option will increase the amount of welding that is done. It is preferred in Havok Vision that only the vertices that have the same source ID are able to be merged.

- **Keyframe Sample Points** :

  Selecting this option will process the animation in a format that is understandable by Havok Vision, by including keyframe data in the export pipeline that Havok Vision uses to generate animation files. Must be selected if using the Process Vision Data filter to export animations.

### 4.4.3 3ds Max: Physics Tools

#### 4.4.3.1 Introduction

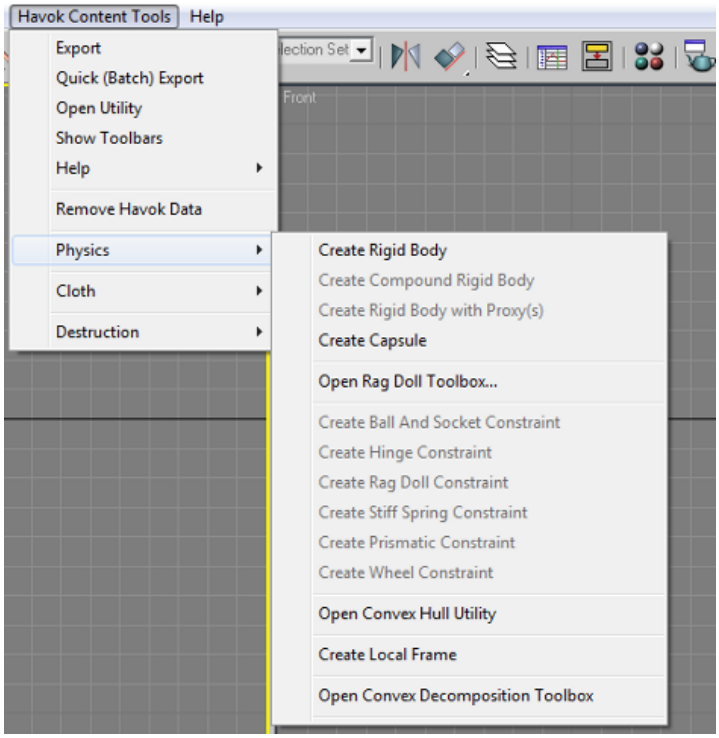
While all modelers, including 3ds Max, have the ability to specify scene and animation data that has a close relationship to the equivalent runtime objects in Havok (and provide many sophisticated tools to manipulate that data), the tools available for physical set up in the modeler is limited and do not necessarily match the features in the runtime. Moreover, since constructing complex sets of physical systems (such as ragdolls) can be a daunting task, providing tools that automate and facilitate these and another operations becomes more and more important.

It is for those reasons that the Havok Content Tools include special 3ds Max tools (modifiers, scripts, etc) for setting up physical information in the 3ds Max scene. In the following sections we will explore those tools.

Advanced users should first read the Common Concepts chapter for in-depth details on some of the rigid body and constraint concepts.

**Accessing the Physics Tools**

The Havok Physics Tools for 3ds Max can be accessed through the **Havok Content Tools** menu and toolbar. If the toolbar is not visible then you should select the **Havok Content Tools** > **Show Toolbar** menu option.




**Havok Modifiers in 3ds Max**

Most of the Havok physics data added to a scene is done so through the use of special modifiers. While it is possible to add those modifiers to an object by hand (through the modify panel, via scripting, etc.), it is highly recommended that you use the **Havok Content Tools** menus, scripts and toolbar to do so as doing otherwise may leave the scene in an inconsistent state.

For reference, the modifiers used by the Havok Physics 2012 Tools for 3ds Max are:

- Rigid Body Modifier: Adds rigid body information to an object.
- Shape Modifier: Adds shape (collision detection) information to an object.
- Constraint Modifiers (Ball and Socket, Hinge, Ragdoll, etc.): Add constraint information to a rigid body.

You can remove all Havok modifiers from selected objects at any time by choosing the **Physics** > **Remove Havok Modifiers** option in the **Havok Content Tools** menu.

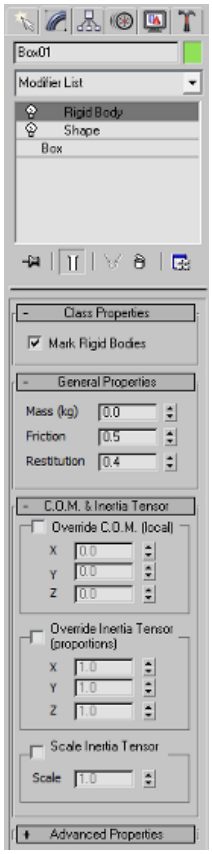
#### 4.4.3.2 Rigid Bodies

In this section we will describe how to create and manipulate rigid bodies and shapes in 3ds Max. For information about concepts and properties associated with rigid bodies and shapes, please check the

Rigid Body Concepts section.

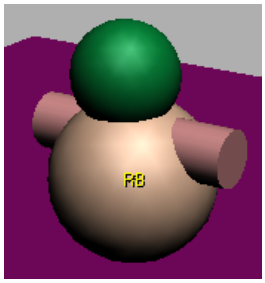
### The Rigid Body Modifier

A Havok rigid body modifier defines the root of a rigid body and contains information on the dynamics of the body. Any object which contains a rigid body modifier will be converted to a runtime rigid body by the Create Bodies Filter (or the Create Rigid Bodies filter in Physics 2012), provided there is at least one associated shape modifier.

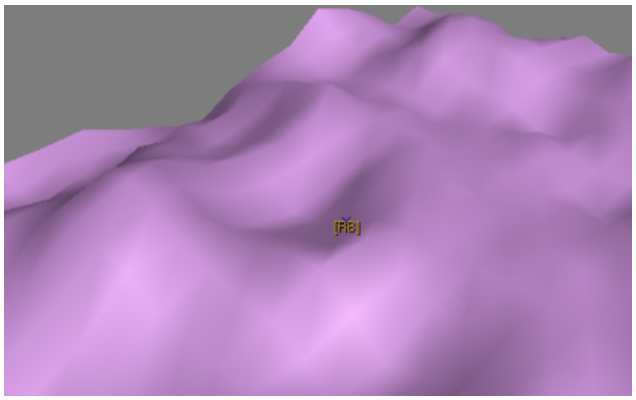



### Class Properties

**Mark Rigid Bodies** is a class property (it applies to all rigid body modifiers). When enabled, rigid bodies (objects with the rigid body modifier) will display a " **RB** " label in the viewports (so they are easy to identify).



If the rigid body is fixed (i.e. its mass is zero), the label will surrounded by square brackets (" **[RB]** ") and will be displayed in a different color:




**Note:**
The colors used for these labels, and many other Havok elements, are customizable through 3ds Max's 'Customize User Interface...' window.

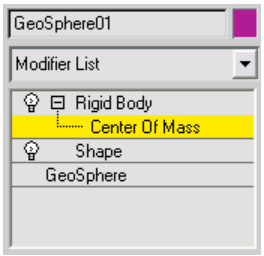
### General Properties

Check the common Rigid Body Properties section for details on the properties listed here.

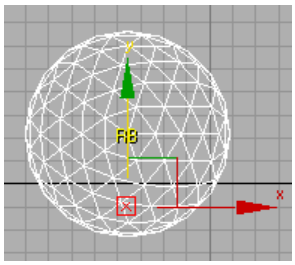
### C.O.M. & Inertia Tensor

By default, the center of mass and inertia tensor of the rigid body will be calculated by the Create Bodies filter based on the rigid body geometry and mass distribution. You can however override this calculation and specify the values within the modeler instead: either by typing the values directly; or by manipulating them in the viewports.

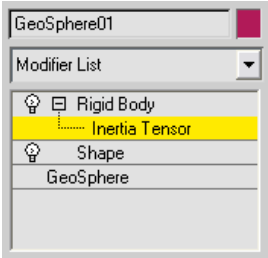
Selecting the **Override C.O.M.** check box enables manual specification of the center of mass. It also adds a new " **Center of Mass** " subobject to the Rigid Body Modifier:



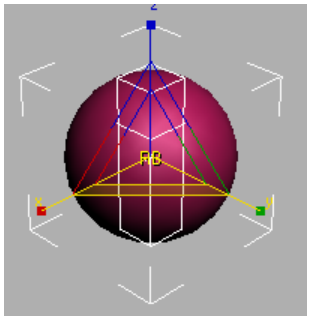

You can select this subobject mode and use the Move tool in order to place the center of mass of the rigid body (displayed as a little crossed square):




Similarly, selecting the **Override Inertia Tensor Proportions** enables manual specification of the inertia tensor proportions. It also adds an " **Inertia Tensor** " subobject to the Rigid Body Modifier:

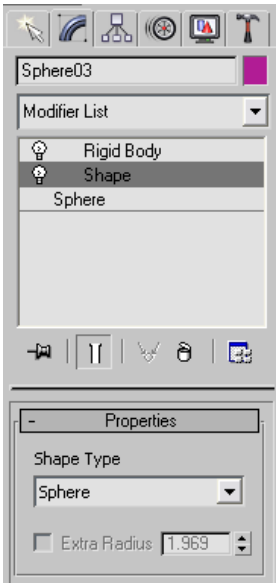



You can select this subobject mode and use the Scale Tool in order to change the proportions of the inertia tensor (displayed as a 3D box):




### The Shape Modifier

A Havok shape modifier contains information about how an object's mesh will contribute to the collision detection setup of a rigid body. One or more shape modifiers must be associated with a rigid body modifier in order for the Create Bodies filter to produce a runtime physics body.




Check the common Shape Properties section for details on the properties listed here.

Shape modifiers currently have no representation in the viewports.

### Creating a Simple Rigid Body

Most of the rigid bodies you will ever want to create will be simple rigid bodies, where both the rigid body and shape information is associated with the same object.

You can create a simple rigid body by selecting an object in 3ds Max and clicking on the **Create Rigid Body** menu option or toolbar button .

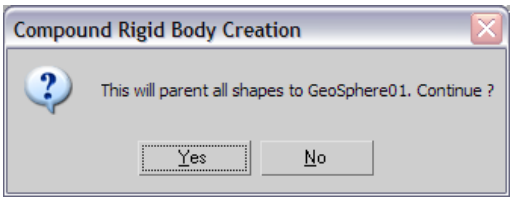
This will add both a rigid body modifier and a shape modifier to the selected object.

### Creating a Compound Rigid Body

A compound rigid body is a rigid body with more than one shape (more than one collision detection primitive). In a compound rigid body one object has both rigid body and shape information while one or more other child objects add extra shape information.

You can create a compound rigid body by selecting two or more objects in 3ds Max and clicking on the **Create Compound Rigid Body** menu option or toolbar button .

If the selected objects are parented to each other, the topmost object in the hierarchy will automatically hold the rigid body and shape modifiers while the rest of objects will each hold a shape modifier. If the objects are not already parented, they will be parented automatically (after user confirmation), by considering the last object selected as the parent of the new hierarchy:

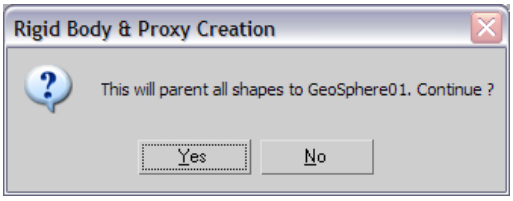



**Creating Rigid Body with Proxy(s)**

A rigid body with proxy(s) is a rigid body where the node that contains the rigid body modifier doesn't contain any shape modifier; the shape modifiers are stored in one or more child nodes (the proxies). In that sense, a rigid body with proxy(s) is similar to a compound rigid body, with the difference being that the object which has the rigid body modifier has no shape modifier in this case.

You can create a rigid body with proxy(s) by selecting two or more objects in 3ds Max and clicking on the **Create Rigid Body with Proxy(s)** menu option.

If the selected objects are parented to each other, the topmost object in the hierarchy will automatically hold the rigid body modifier while the rest of objects will become the proxies and just hold shape modifiers. If the objects are not already parented, they will be parented automatically (after user confirmation), by considering the last object selected as the parent of the new hierarchy:




#### 4.4.3.3 Constraints

In this section we will present how constraints are created and manipulated in 3ds Max.

**Constraint Modifiers**

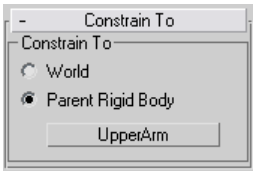
A Havok constraint modifier contains information which will be used to create a runtime constraint on export. Since a constraint must act on a rigid body, a constraint modifier is useful only if the object also contains a rigid body modifier. If a pair of rigid bodies are to be constrained, a constraint modifier is attached to only one of the bodies (the *child*), and it maintains a reference to the other body (the *parent*). Again, please consult the common Constraint Concepts section of this manual for more details on child and parent spaces.

There are currently three types of Havok constraint parameter sets, the specifics of which are described in the common Constraint Types section of this manual. Each of these types provides the same basic constraint parameters, which allow you to specify the child and parent spaces. For example, a hinge constraint is shown:

### Constrain To Rollout

In this rollout you can choose which other rigid body (if any) the constraint is attached to:

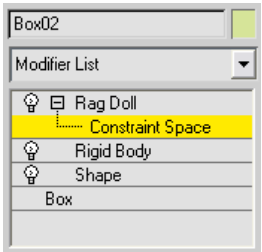
- **World** : If this option is selected, the constraint will attach the child rigid body to a fixed location in space.
- **Parent Rigid Body** : If this option is selected, the constraint will attach the child rigid body to the rigid body specified in the pick node button below (that rigid body will become the parent rigid body).



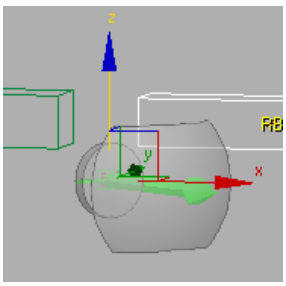

### Constraint Spaces Rollout

Constraints can be considered as having two distinct spaces: the *child* (in the local space) and the *parent* (in the parent object's space). See the common Constraint Spaces section for more details.

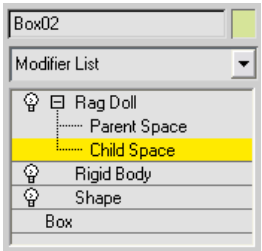
Depending on the state of the **Lock** buttons in the **Constraint Spaces** rollout, various subobjects may be available in the constraint modifier for manipulation. If the translation and rotation of the parent space are locked, then a single subobject appears:



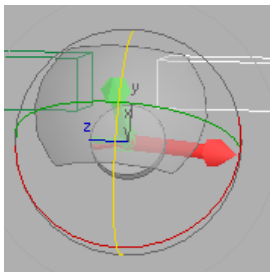

- **Constraint Space Subobject** : In this subobject you can manipulate both child and parent spaces as one, using the move or rotate tool (depending on the type of constraint and the state of the **Lock** buttons).




If the parent space translation or rotation is unlocked, two subobjects may appear:




- **Parent Space Subobject** : In this subobject mode you can manipulate the position and/or orientation of the parent space of the constraint.



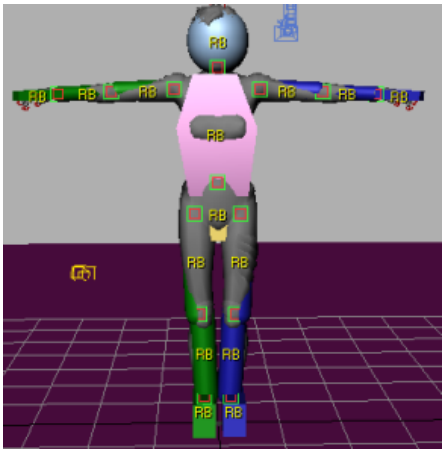
- **Child Space Subobject** : In this subobject mode you can manipulate the position and/or orientation of the child space of the constraint.

### Properties Rollout

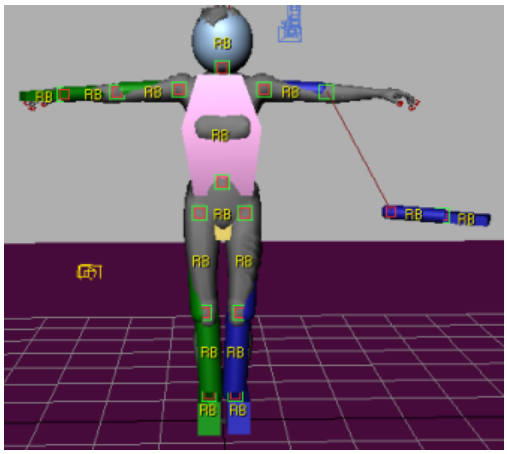
The properties here are specific to each type of constraint. Check the common Constraint Types sections for details on each constraint type.

### Display

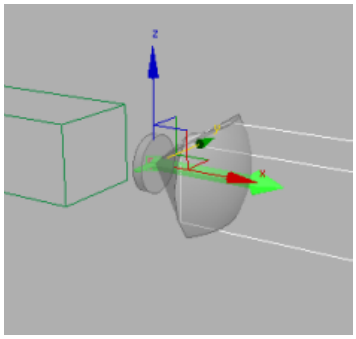
Constraints are displayed in the viewports as two markers, representing the location of the child and parent spaces of the constraint. In most cases, those two spaces should be aligned, so both markers are drawn at the same location:



If the spaces are misaligned, a dotted line is also drawn between them to display the offset:

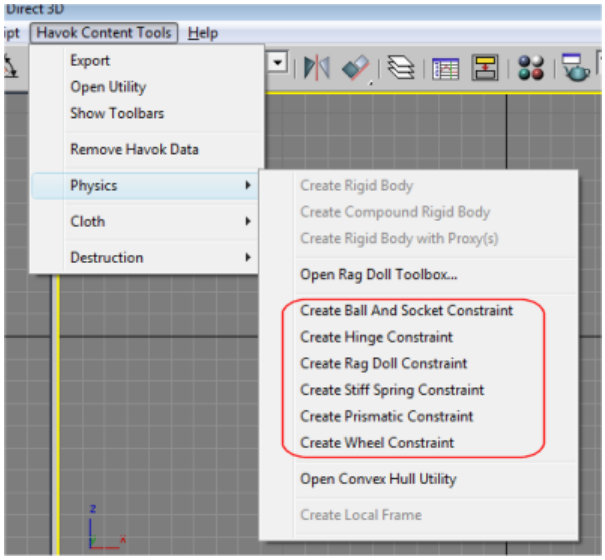


Whenever a constraint is selected in the Modify panel , any 3D representation it may have will be displayed in the viewports, based on its constraint space and parameter values:



### Creating a Constraint

You can create a constraint by using either the **Havok Content Tools** menu or toolbar:



You can create a constraint by either having one or two rigid bodies selected:

- If one rigid body is selected, this rigid body will be the child rigid body of the constraint (and the constraint modifier will be applied to it). If it has a parent node and that parent node is also a rigid body, the constraint's parent rigid body will be set to that node; otherwise the rigid body will be constrained to the world.
- If two rigid bodies are selected, and one is a descendant of the other, the descendant will be the child rigid body of the constraint (the modifier will be applied to it) and the ancestor will be the parent rigid body. If they are not explicitly parented, the last rigid body selected will be the parent and the first rigid body will be the child.

A single rigid body can have any number of constraints applied to it.

#### 4.4.3.4 The Rag Doll Toolbox

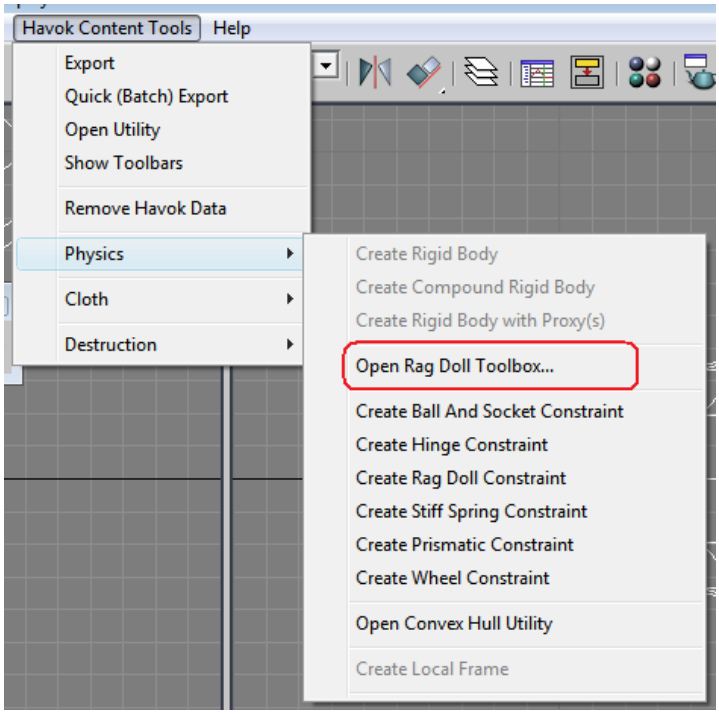
**Introduction**

One of the most common usages of rigid body simulation is the creation of articulated systems of rigid bodies representing characters (ragdolls). This usually involves the creation and alignment of rigid body proxies for the bones, as well as the setup and alignment of constraints that represent the joints in the character. The Rag Doll Toolbox provides a set of tools to facilitate those operations. It is built as a MAXScript tool that works on top of the lower-level physics tools.

See also: Customizing the 3ds Max Rag Doll Toolbox

**Opening the Rag Doll Toolbox**

You can open the Rag Doll Toolbox by using either the **Havok Content Tools** menu or toolbar:



The Rag Doll Toolbox is a MAXScript floating window. It can also be docked to either side of the 3ds Max UI:

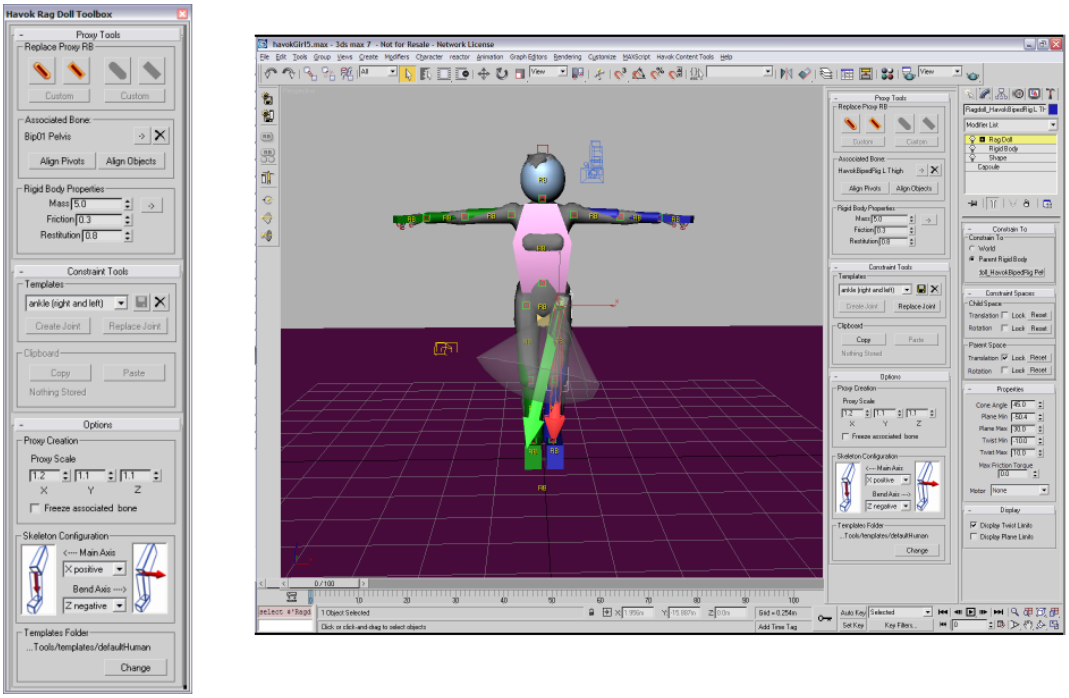



Figure 4.30: The 3ds Max Rag Doll Toolbox UI

### Proxy Tools

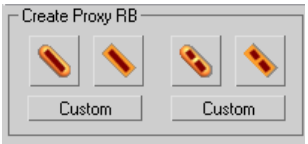
The first task when creating a ragdoll representation of a character is to construct a set of rigid bodies which represent a selection of the character's bones. These rigid bodies are usually collision primitives like capsules and boxes, which are very efficient to simulate by the physics runtime (capsules in particular).

When creating these rigid bodies, it is very important that their pivots (the location of their transforms) are aligned to the bones they represent (since, at runtime, movements on the bones will have to be converted to movement on the rigid bodies and vice versa). Thus, it's very important to keep track of the association between bones and the proxies that represent them.

The **Proxy Tools** rollout allows you to:

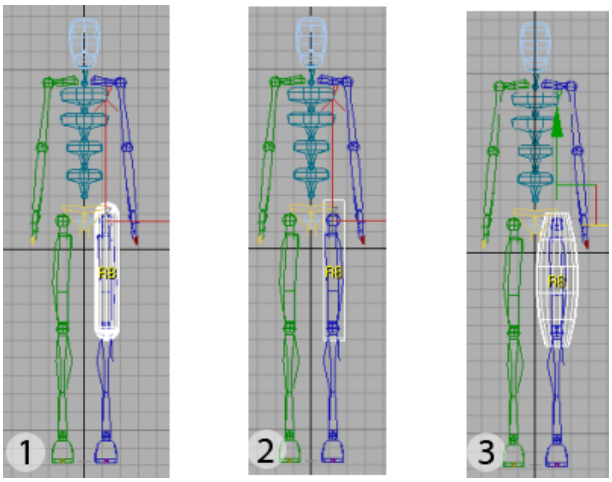
- Given a bone, create a proxy representation (a box, a capsule, or a custom rigid body) of it and align and associate the proxy with the original bone.
- Given a chain of bones, create a single proxy representation of that chain and align and associate that proxy with the chain of bones.
- Given a proxy rigid body (associated with a bone or chain of bones), replace it with another proxy.
- Keep track of the association between rigid body proxies and bones, and easily realign them.
- Quickly access and modify the basic properties of one or multiple rigid body proxies (mass, restitution and friction).

### Create/Replace Proxy RB




Buttons in this group box allow you to create proxies and associate them with bones.

The first set of buttons (on the left) create *simple proxies*–rigid body proxies that are associated with a single bone:



1. **Create Capsule Proxy(s)**

   With one or more bones selected, this will create a capsule rigid body bounding each of the selected bones. It will also associate and align each capsule with each bone.

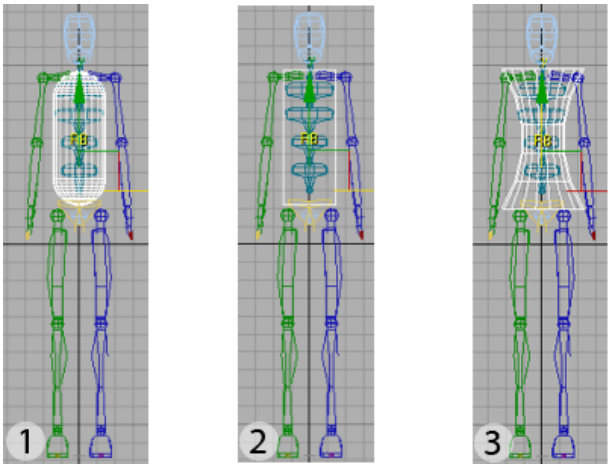
2. **Create Box Proxy(s)**

   With one or more bones selected, this will create a box rigid body bounding each of the selected bones. It will also associate and align each capsule with each bone.

3. **Associate Custom Proxy**


   With a single bone selected, this will open a " **Select by Name** " dialog where you can select any mesh object to become the proxy rigid body for the selected bone. It will then associate and align that rigid body with the selected bone.

The second set of buttons (on the right) create what we call *chain proxies*–rigid body proxies that are associated to a chain of bones (a set of bones in hierarchical order). Chain proxies are useful in order to create single proxies that represent multiple bones, like a rigid body bounding multiple spine links:




1. **Create Capsule Chain Proxy**

   With a chain of bones selected, this will create a capsule rigid body bounding all of them. It will also associate and align the capsule with the chain of bones.

2. **Create Box Chain Proxy**

   With a chain of bones selected, this will create a box rigid body bounding all of them. It will also associate and align the box with the chain of bones.

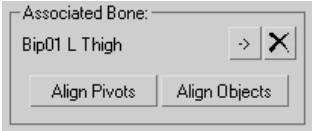
3. **Associate Custom Chain Proxy**

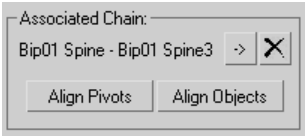

   With a chain of bones selected, this will open a "Select by Name" dialog where you can select any mesh object to become the proxy rigid body for the chain of bones. It will then associate and align that rigid body with the chain.

When a rigid body proxy (either single or chain proxy) is selected, these "Create Proxy" buttons become " **Replace Proxy** " buttons; the proxy they create will replace the selected proxy (and the association with the original bone will be updated).

**Bone Associations**

Controls in this box display and operate on associations between rigid body proxies and bones. Thus, they are only enabled when one or more rigid body proxies (rigid bodies created by one of the "Create Proxy" options above) are selected.



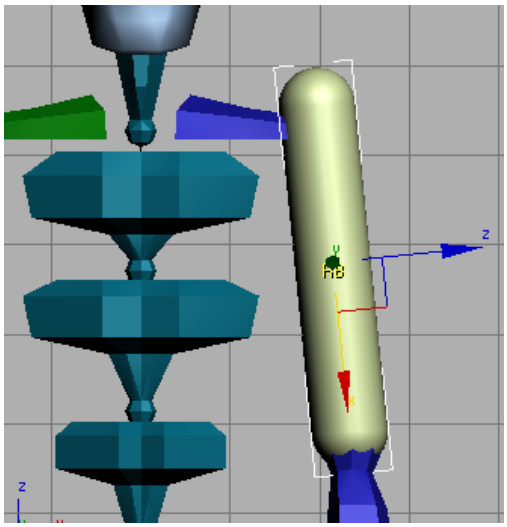

If a single proxy is selected, the name of the associated bone (or chain of bones, if it's a chain proxy) will be displayed below the **Associated Bone** / **Associated Chain** box.

Pressing the **Select Associated Bone(s)** button will select the bone or chain of bones associated with the selected proxy.

The **Remove Association** button will remove the current association between the selected proxy and the bone or chain of bones. The proxy will remain as a rigid body, but its association with the original bone(s) will be lost.

The **Align Pivots** button will ensure that the pivot of the proxy matches the pivot of the associated bone (or the first bone of the associated chain) by changing the local pivot of the proxy (without actually moving the proxy). Use this button whenever you have modified a proxy and want to make sure that the pivots are aligned (without displacing the proxy)

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 



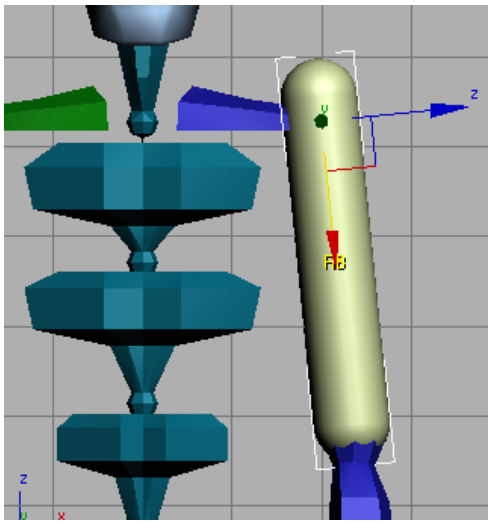
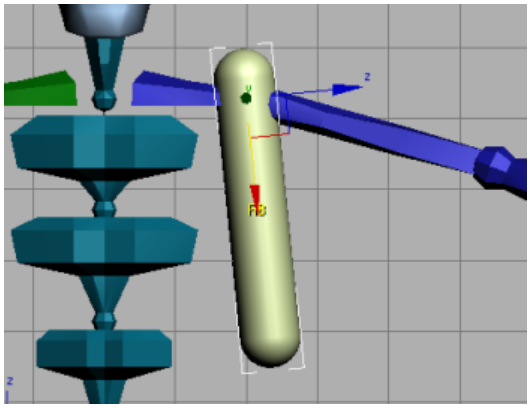


Table 4.1: Aligning Pivots

The **Align Objects** button will ensure that the pivot of the proxy matches the pivot of the associated bone by moving the proxy rigid body. Use this button whenever you move or rotate a bone and want to make sure that the associated proxy follows that movement.



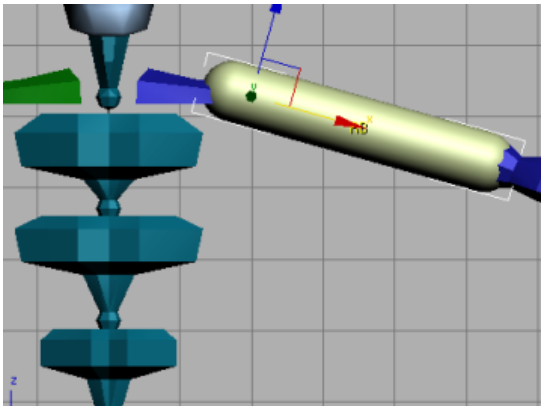
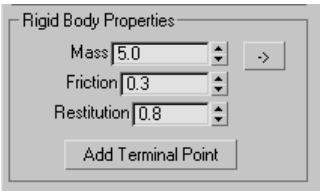


Table 4.2: Aligning Objects

### Rigid Body Properties

The controls in the **Rigid Body Properties** box are enabled whenever one or multiple rigid bodies (proxies or not) are selected.



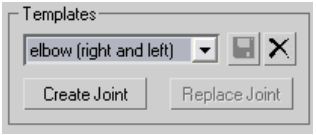
They allow you to modify the basic rigid body properties of the selected objects easily; they are particularly handy when you want to modify multiple rigid bodies at the same time.

The **Open RB Modifier** button will open the rigid body modifier associated with the selected rigid body in the Modify panel (so it can be edited there as well).

The **Add Terminal Point** button is used to add extra leaf bones to skeletons to be used for raycasting, specifically to prevent ragdoll penetration with landscapes using the hkaDetectRagdollPenetration utility. Selecting a ragdoll proxy and clicking the add terminal point button creates a point helper as the child of the proxy. The helper is located on the local bounding box of the proxy at the center of the face furthest from the proxy pivot.

**Constraint Tools**

The **Constraint Tools** section of the Rag Doll Toolbox gives you the ability of reusing the setup of constraints by using *templates*. A template is a representation of Havok constraint which is stored as a text file. The file includes the type of constraint, plus any parameter values which the original constraint had when the template file was created.




The **Templates** dropdown list show all the available templates. These list is built by looking at all templates stored in the current *templates folder* , the location of which can be changed in the Options rollout of the Rag Doll Toolbox.

The **Save Template As...** button is enabled only when a rigid body with a single constraint is selected. It allows you to save the parameters of that constraint as a template. Templates are saved as `*.txt` files.

The **Remove Template** button will permanently remove the selected template (i.e., it will delete the template file from the templates folder).

The **Create Joint** button is only enabled when two rigid bodies are selected. It will create a constraint based on the currently selected template. If the rigid bodies are parented, it will apply the constraint to the child rigid body. If they are not parented, but they are proxies of bones that are parented, the rigid bodies will be first parented accordingly, and then the constraint will be applied to the child. Otherwise, the user is required to manually parent the rigid bodies before creating the joint.

The **Replace Joint** button is only enabled when a rigid body with a single constraint is selected. It will replace that constraint with a new one based on the selected template. The rigid body will be constrained to the same parent it was originally constrained to.

**Clipboard**

The Constraint Tools also allow you to cut & paste constraints without explicitly storing them as templates:

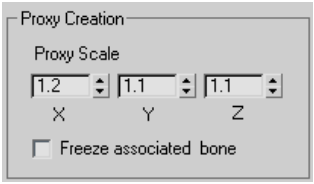
The **Copy** button is enabled only when a rigid body with a single constraint is selected. It will store the parameters of that constraint in memory (the type of constraint stored is displayed)

The **Paste** button operates as the **Create Joint** and **Replace Joint** buttons (depending on the selection), with the difference that it applies the template stored in the clipboard rather than the template selected in the drop down list.

**Rag Doll Toolbox Options**

This rollout contains general options that apply to both the Proxy Tools and Constraint Tools. These options are saved with the scene, and are presented in three sets:

**Proxy Creation Options**




The **Proxy Scale X, Y and Z** values are used during the creation of bounding proxies (capsule, box) for a bone. The size of the box/capsule will be calculated by taking the bounding box of the bone and then scaling it by the given values.

This is particularly useful when the size of the bones may not properly represent the full volume of the character part / limb they represent.

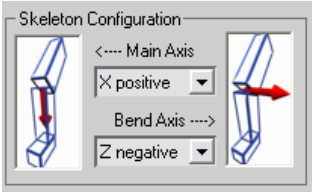
Also, it is usually desirable to have some overlap between bones, so there are no gaps at the joints. This is why it is useful to have some extra scale in the direction of the main axis of the bone (by default X).

When the **Freeze Associated Bone** option is on, creating a proxy from a bone will automatically freeze the original bone. This can be very useful in order to reduce the amount of selectable objects in the viewport during proxy creation operations (so bones that already have proxies are no longer selectable in the viewports).

**Skeleton Configuration Options**

When storing and applying constraint templates (check the Constraint Tools section for details), it is important to have a convention for the local axis of the bones.

Most 3ds Max tools, including Bones and Character Studio, follow some convention on the local orientation of their bones. The default convention used by the Rag Doll Toolbox is that of Character Studio, but should your tool of choice have a different convention, the options here allow you to define it:

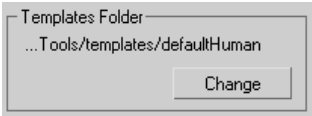



The **Main Axis** option should specify the name and sign of the axis (in the local space of the bone) that points towards the length of the bone (to the next bone), outwards. The default is X positive.

The **Bend Axis** option should specify the name and sign of the axis (in the local space of the bone) around which a positive (counter clockwise) rotation bends the joint inwards. The default is Z negative.

**Templates Folder Options**

Templates available to the Constraint Tools are collected from the list of available template files in an specific folder. This folder is defined in the **Templates Folder** options box.

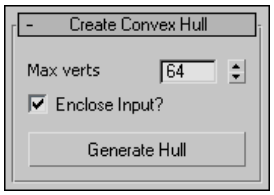



You can change the location of the current templates folder by clicking on the **Change** button and selecting a different folder. This allows you to have different templates folder for different types of characters.

#### 4.4.3.5 The Convex Hull Tool

In this section we describe a simple tool for generation of convex hulls from selected meshes. This tool generates an approximate convex hull from the currently selected objects (either meshes or nurbs) with a user-specified maximum number of vertices **Max verts** . The hull is approximated if the requested number of vertices is less than the actual number of vertices present in the complete convex hull. This is useful in a number of contexts, such as generation of assets for platforms which require <64 vertices per convex body. If the specified maximum exceeds the number of vertices in the true hull, the tool just generates the true hull. Note that if multiple meshes/nurbs are selected, the hull tool generates a single hull from all of their vertices combined.

When **Enclose input** is checked (default), the convex hull when approximated using **Max verts** , is forced to enclosed the input(s)




The convex hull tool utility consists of fields for the two parameters **Max verts** and **Enclose input** described above, and a **Generate Hull** button to generate the hull using these parameters. Generated

hulls are placed in the scene root with the name "foo_Hull01" where "foo" is the name of the first object in the selection. Subsequently generated hulls accumulate in the scene root but are given unique names.

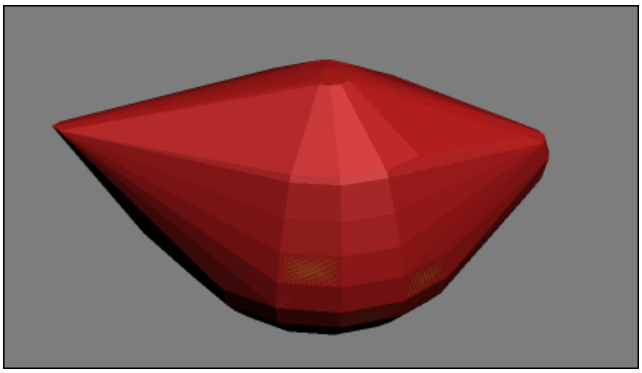


Figure 4.31: Full convex hull of a teapot

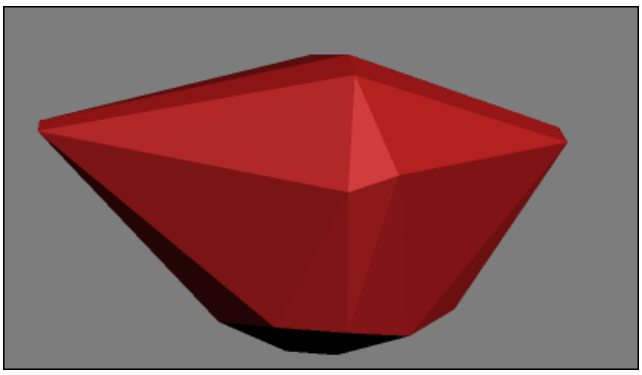


Figure 4.32: Approximated convex hull of a teapot with **Enclose input** checked

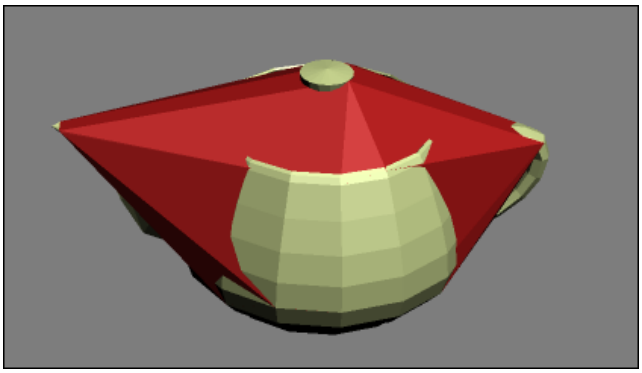


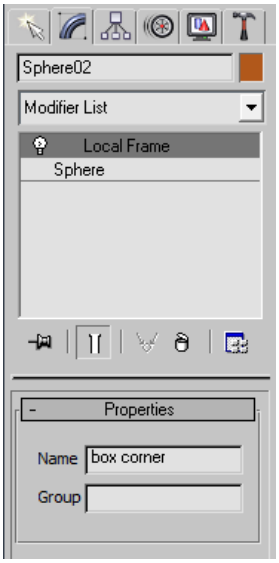
Figure 4.33: Approximated convex hull of a teapot with **Enclose input** unchecked

#### 4.4.3.6 Local Frames

This section describes how to create **local frames** in 3ds Max. To learn more about the concept of a local frame please see the Local Frame Concepts section.

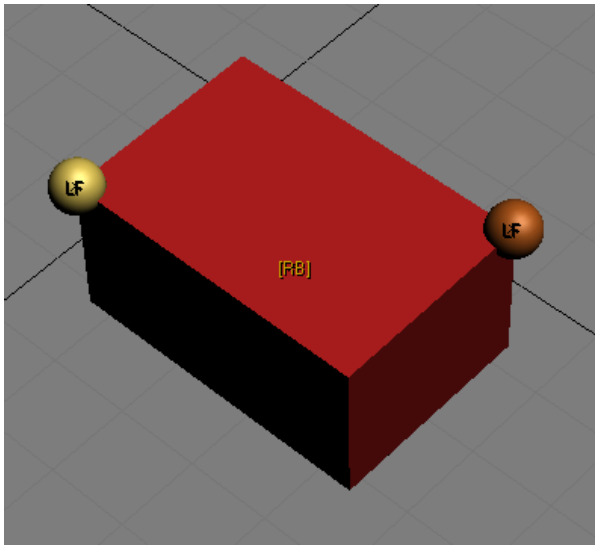
You can create a local frame by selecting an object in 3ds Max and clicking on the **Create Local Frame**

menu option or toolbar button . This will add a **local frame modifier** to the selected object as shown here:




The modifier has a **Name** property in which you can name the local frame and a **Group** property in which you can name the group to which this local frame belongs.

An object that has a local frame modifier will be displayed with the label **"LF"** in the viewports. The following image shows a rigid body box with two local frames parented to it displayed in 3ds Max:




Note that local frames are not yet supported by the new Havok Physics Create Bodies Filter in the 2013.1 release (they still work with Physics 2012). Support will be available in a future release.

In the Physics 2012 filters, any object with a local frame modifier will be converted to a runtime local frame in the Create Rigid Bodies filter (if parented to a rigid body) or the Create Skeletons filter (if parented to a skeletal bone).

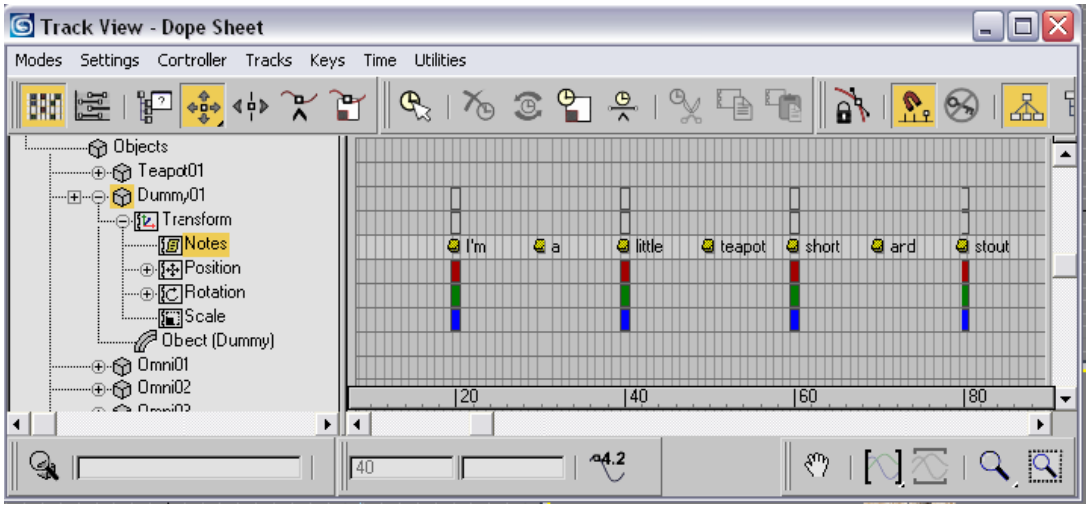
### 4.4.4 3ds Max: Animation Tools

3ds Max already provides a wealth of tools for creating content for animation playback. The Havok Content Tools use these features provided by 3ds Max without the need to provide custom extensions (as it does for the physics). Thus concepts like skeletons, bones, skins and animations in 3ds Max transfer to the same concepts/objects in the Havok SDK. The following section outlines one of the less obvious connections between 3ds Max and the Havok Animation SDK.

#### 4.4.4.1 Annotations in 3ds Max

Please check the common Annotations section for details on annotations in the Havok Animation SDK.

3ds Max has built-in support for what is called "note tracks", which work in very similar fashion to annotations. Therefore, in order to create annotations in 3ds Max, simply attach note tracks to the desired bone's transform; the string values will be exported as annotations. The 3ds Max user reference manual contains a detailed section on setting up note tracks.

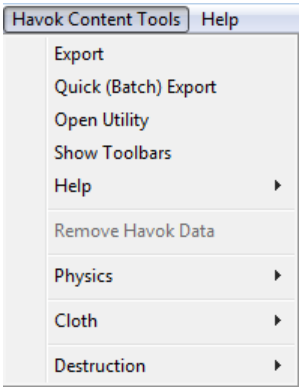



### 4.4.5 Tutorial: Export and Animation Basics

In this tutorial we are going to explore the basic operations involved in exporting and processing an asset. We are going to use the example of exporting skeletons, animations, etc., but the principles regarding how assets are exported, the filter pipeline and the use of the filter manager and individual filters for processing applies to all Havok products.

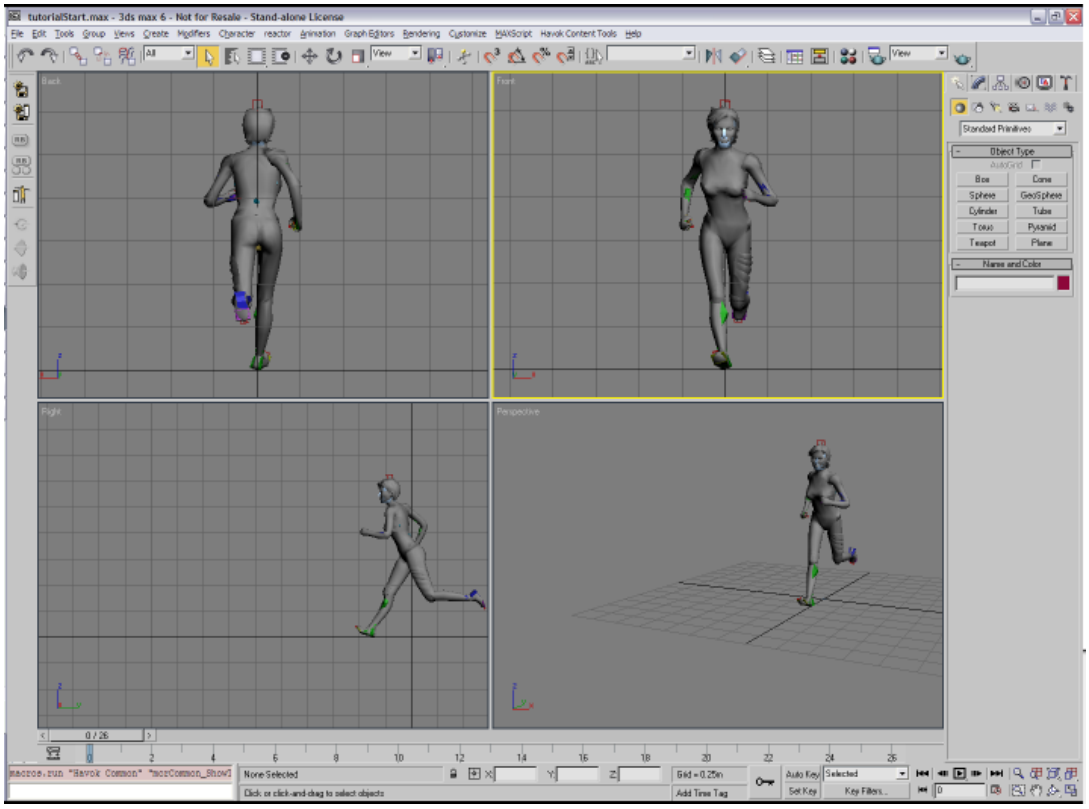
#### 4.4.5.1 Getting Started

Ensure that you have installed the Havok Content Tools for your version of 3ds Max. You should see a **Havok Content Tools** menu in the 3ds Max menu bar:




The Havok Content Tools toolbar will be useful for this and the next tutorials, so it is useful to have it in the UI. If it is not present, click on **Show Toolbars** in the **Havok Content Tools** menu. You can dock the toolbar to one of the sides, or leave it floating.

Open the file `"scenes/havokContentTools/tutorials/exportBasics/tutorialStart.max"`:




The scene contains the animation of a girl running (a single cycle). If you examine the scene, you will notice that the animation is constructed by using a set of animated Character Studio bones which modify a mesh skinned with the Physique modifier.

We want to export this information (bones, animation, skin weights, etc) from 3ds Max into objects that the Havok Animation/Complete SDK can interpret.

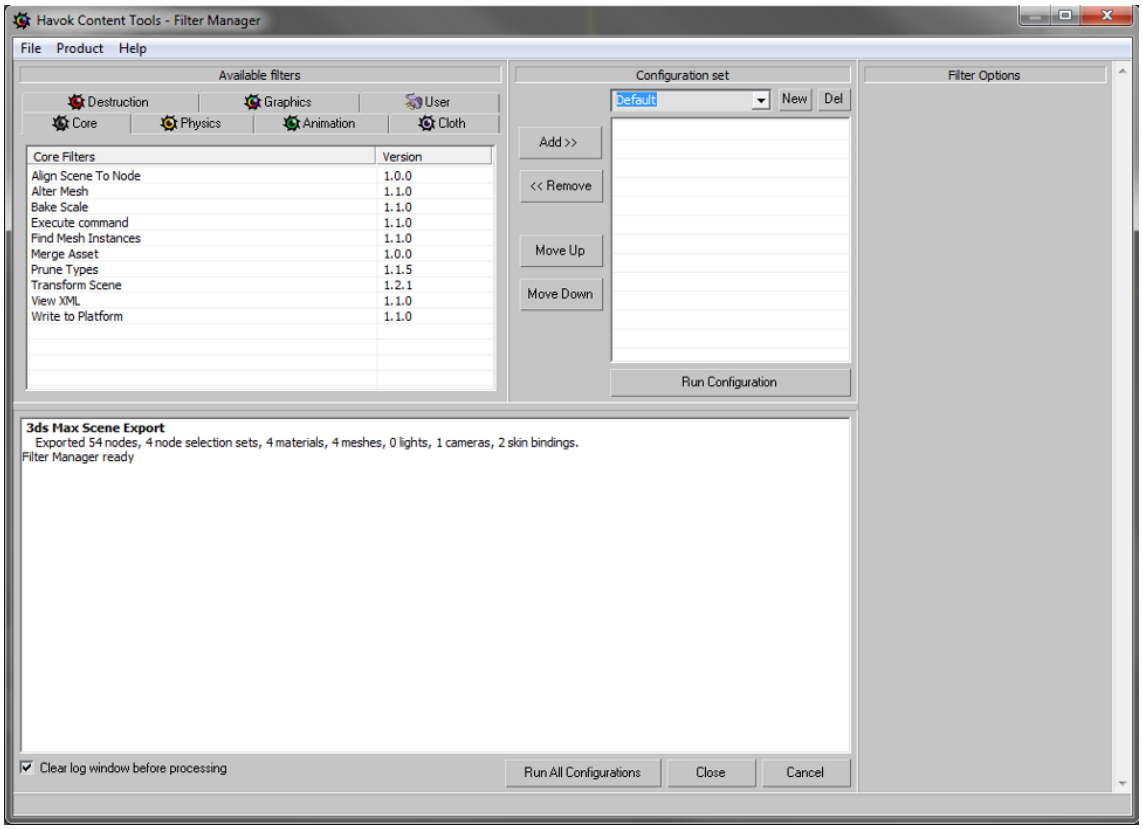
**Note:**

Even if you are not a user of Havok Animation or Havok Complete, you can still follow this tutorial as the Havok Content Tools packages contain filters for all products. This tutorial focuses on introducing the basic concepts behind asset export and processing so it is relevant for all Havok products.

#### 4.4.5.2 Exporting the Scene

Exporting the current scene is as simple as to click on the **Export** button in the **Havok Content Tools** menu or toolbar.

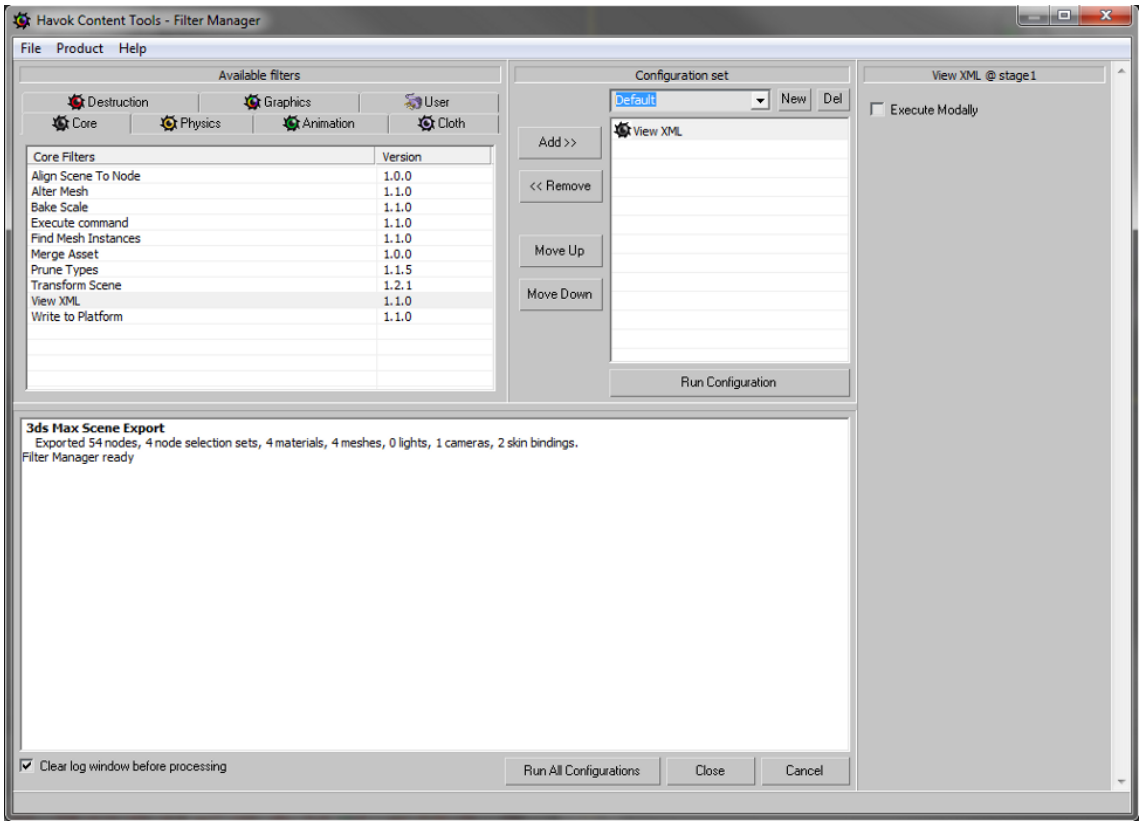
By doing this the 3ds Max Scene Exporter is invoked: the scene is navigated and relevant information is pulled from the 3ds Max nodes, modifiers, meshes, etc. and converted into a modeler-independant format. After that, that content is passed to the filter manager; its UI should appear after a short time:




The **Configuration Set** window (in the middle of the dialog) lists the filters that will be applied to the content we just exported. It is now empty, which means that nothing will happen to the content. You can add filters by selecting them from the **Available Filters** tabs and clicking on the **Add** >> button.

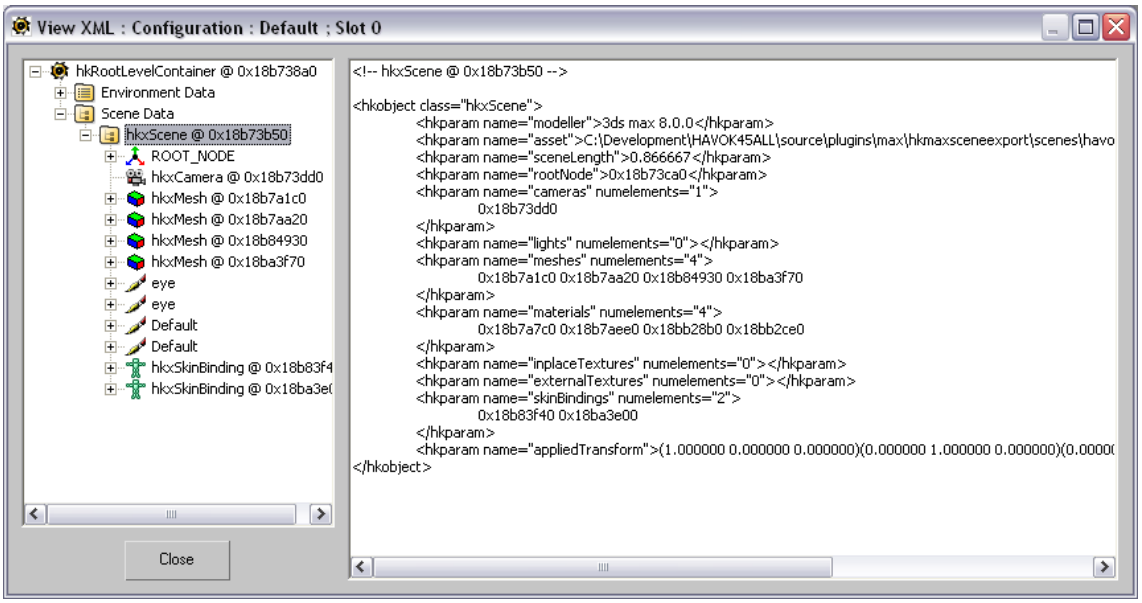
Let's start by adding a very useful filter for debugging purposes: the View XML filter.

- On the **Available Filters** window, click on the **Core** tab (if it's not selected already)
- Double-click on the **ViewXml** filter (or select it and press **Add** >> )
- That filter should now appear at the top of the **Current Filters** list




Filters are executed in the order they appear in the **Current Filters** list; if a filter modifies the content, that modified content is what is used by the next filter (check the filter pipeline documentation for details).

The View XML filter doesn't modify any content, though. It just opens a window that presents the current content in a human-readable XML format. Let's check what we exported from 3ds Max by clicking on the **Run Configuration** button.




Notice that what was exported is an "`hkxScene`" object, which contains information about nodes, meshes, materials and skin bindings. This is what the 3ds Max Scene Exporter extracted from our current scene, and is the raw information that we are going to process further until we can create a useful set of objects ready to be used by our runtime.

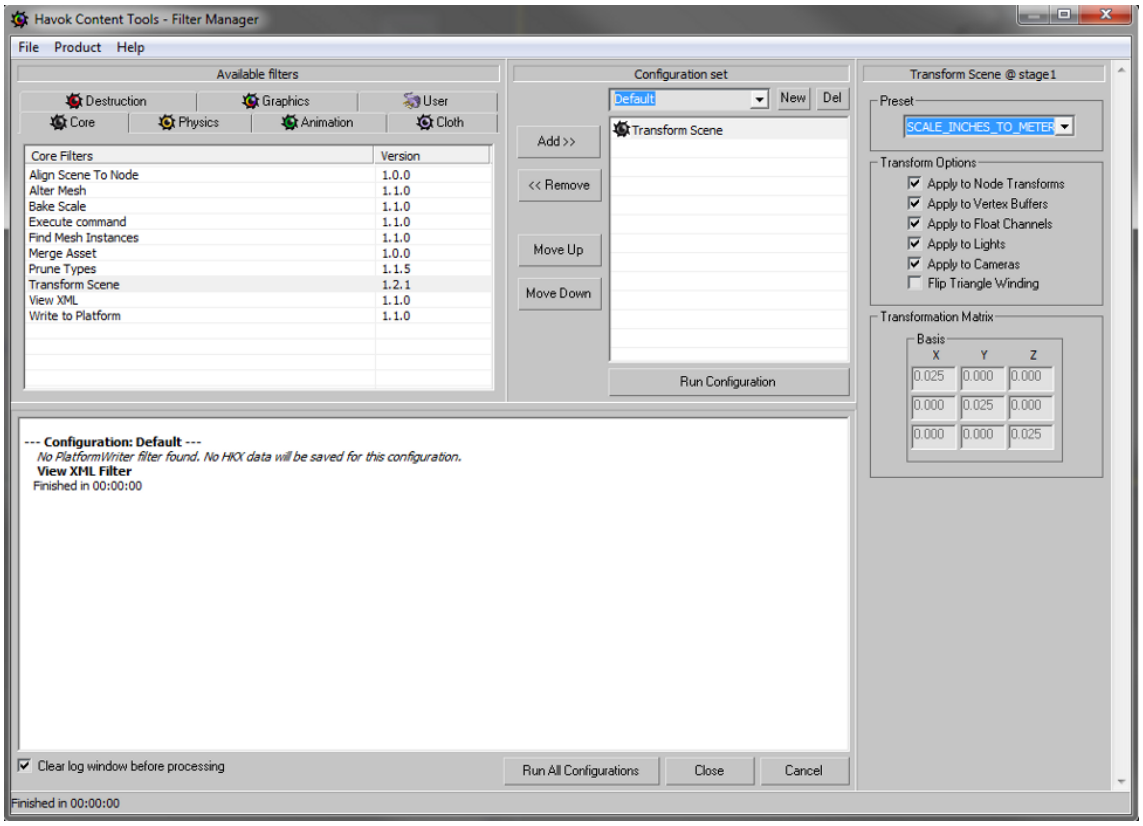
Close the XML window and get back to the filter manager. Remove the View XML filter by selecting it and clicking on << **Remove** or hitting **DEL** .

**Tip:**

Feel free to add the View XML filter at different locations during this and other tutorials in order to examine how processing is done by different filters. This filter is a very useful debugging tool.

#### 4.4.5.3 Processing the Scene

Let's now add some useful filters for processing the scene. The first filter that we'll add will be the Transform Scene filter (from the **Core** tab):



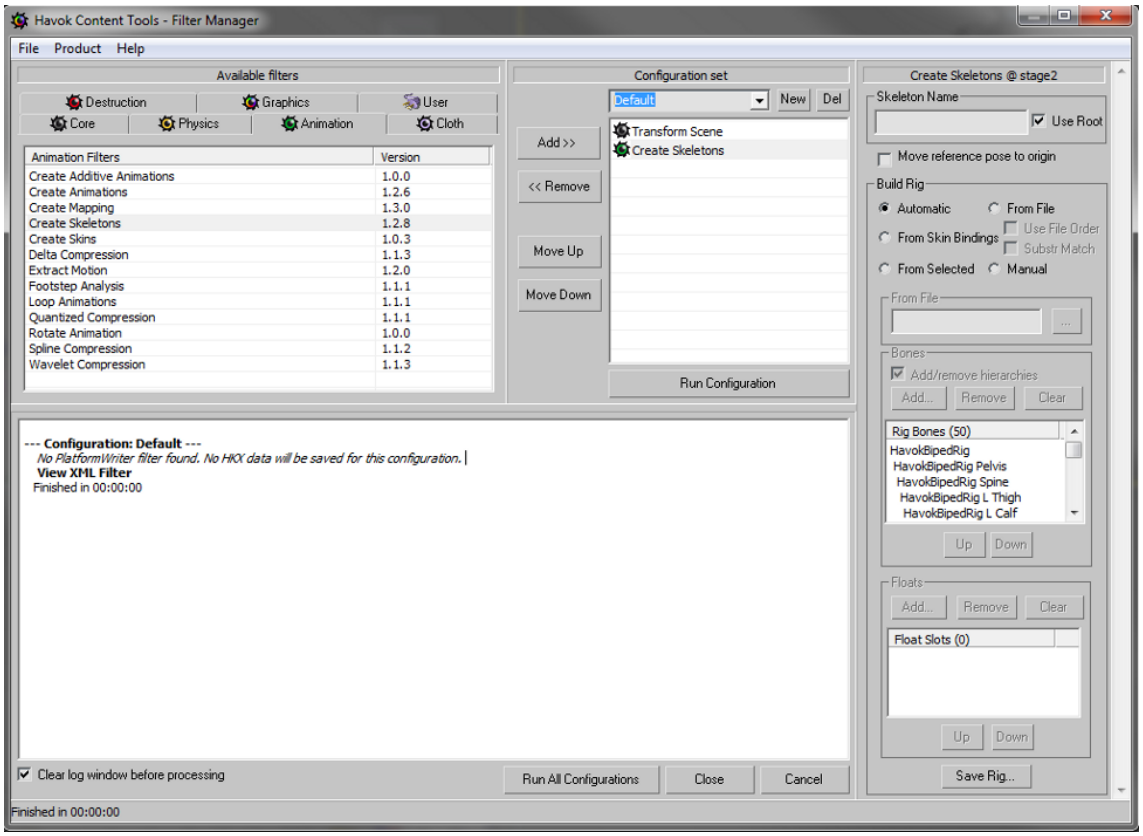

This filter is very useful and, in the majority of cases, it will be the first filter that you'll add to any filter configuration. This filter applies an arbitrary transform to all the objects (nodes, meshes, etc) in the scene. In this case, we want to apply it because in 3ds Max the default units are inches, but in our runtime units are meters. Hence we use the `SCALE_INCHES_TO_METERS` preset in the options of the filter (the right panel in the filter manager).

The next filter we will apply is the Create Skeleton filter (from the **Animation** category). This filter takes a look at the nodes in the scene and creates an `hkaSkeleton` object (used by the Havok Animation/Complete SDK).

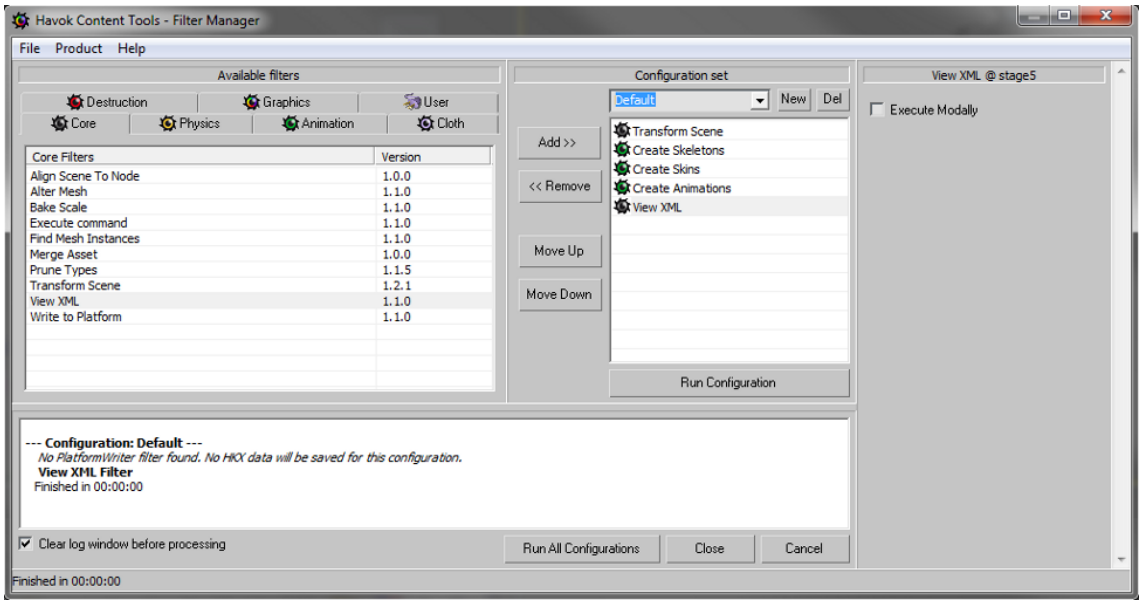
**Note:**

If your current Product Selection is set to Havok Physics 2012, some of the filters will appear in red, and when run, warnings will be generated. You can either ignore those warnings, or change your Product Selection temporarily to Havok Complete.

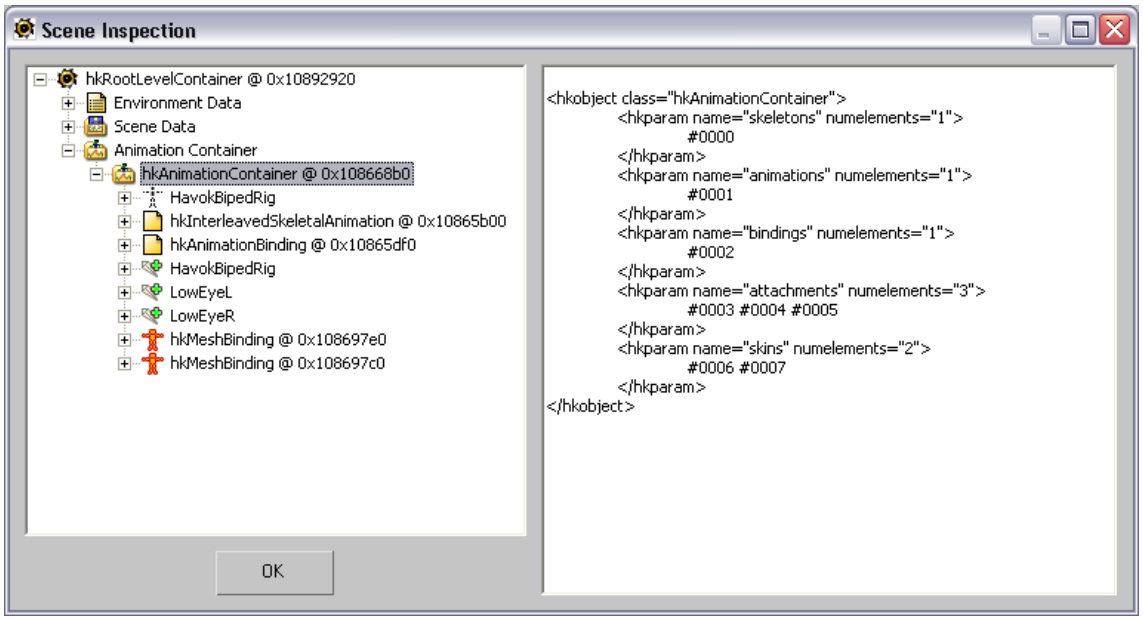
As you can see, this filter also has options associated with it (they appear in the right panel).




We will leave the default options as they suit us fine. You can check the documentation for this and other filters if you want to learn more about how they work. After this filter, add the Create Skin and Create Animations filters (also in the **Animation** category); leave the default options as well. Finally, place the View XML filter again at the bottom of the filter list.



If you run (press the **Run Configuration** button) the filter processing, the View XML filter will show you the effect of all the processing:

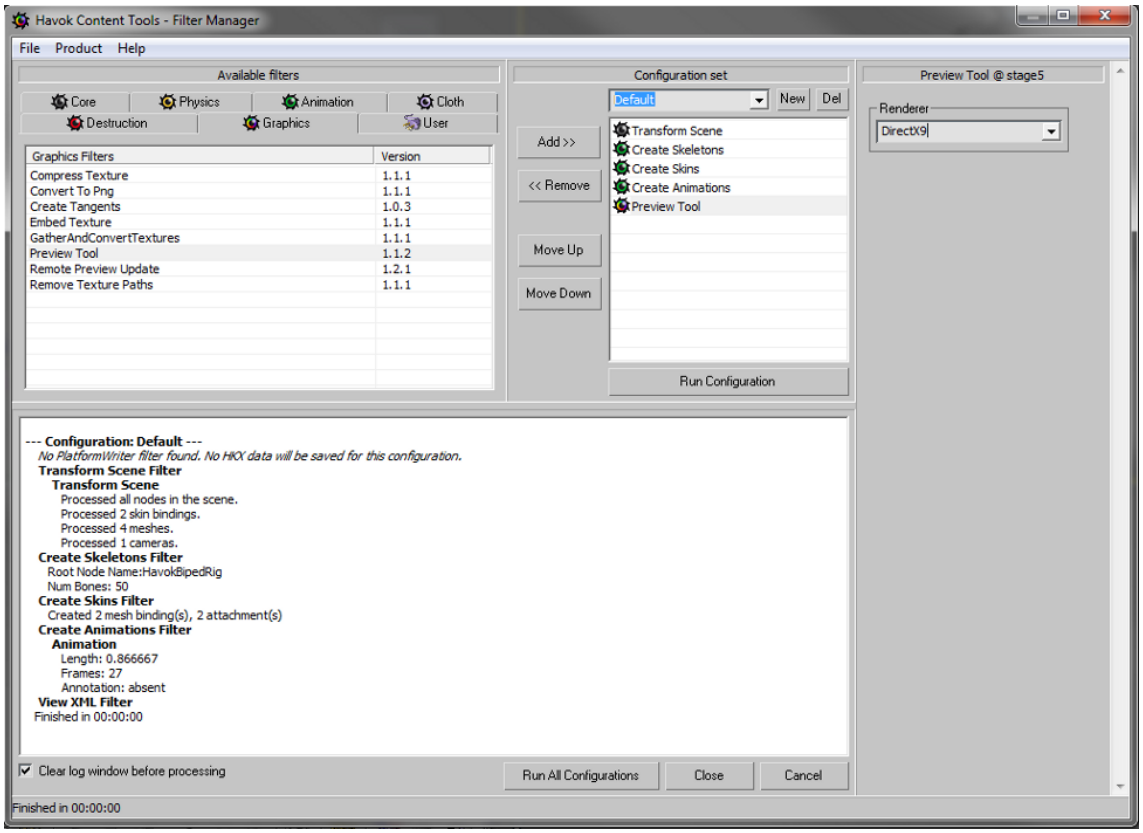



Notice how a new root-level object (an animation container) is now part of the content, and how it contains Havok Animation/Complete objects like `hkaInterleavedSkeletalAnimaton`, `hkaSkeleton` (HavokBipedRig), `hkaMeshBinding`, `hkaBoneAttachment`, etc. that we should be able to use at runtime.

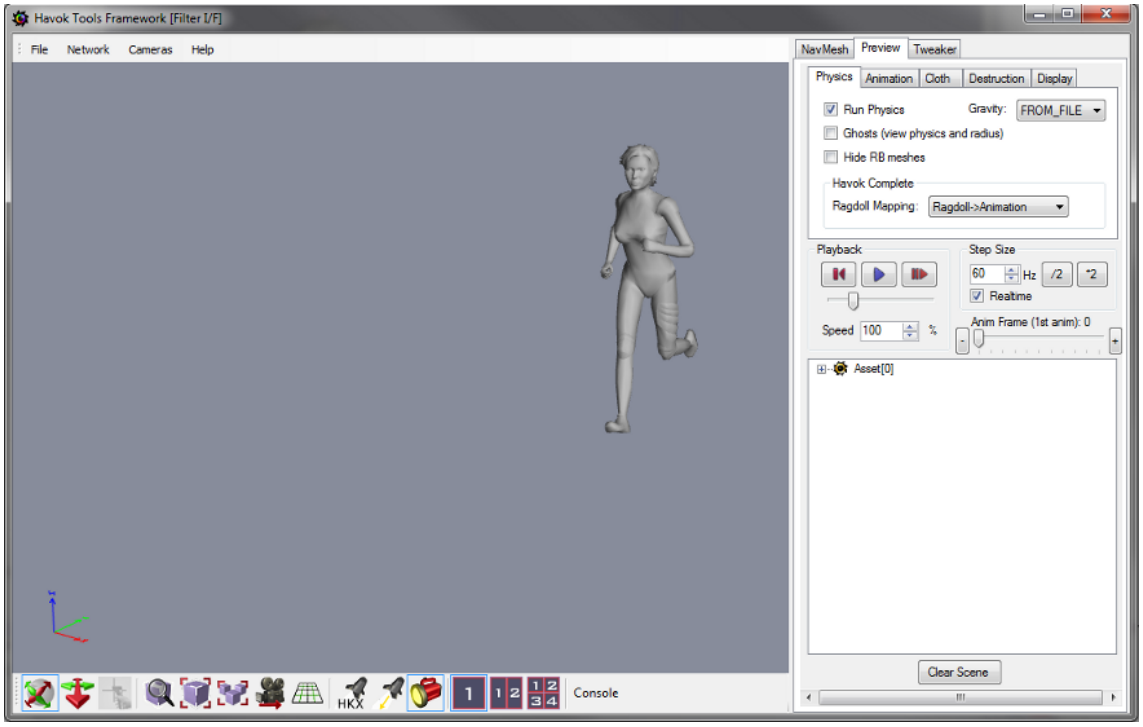
#### 4.4.5.4 Previewing the Scene

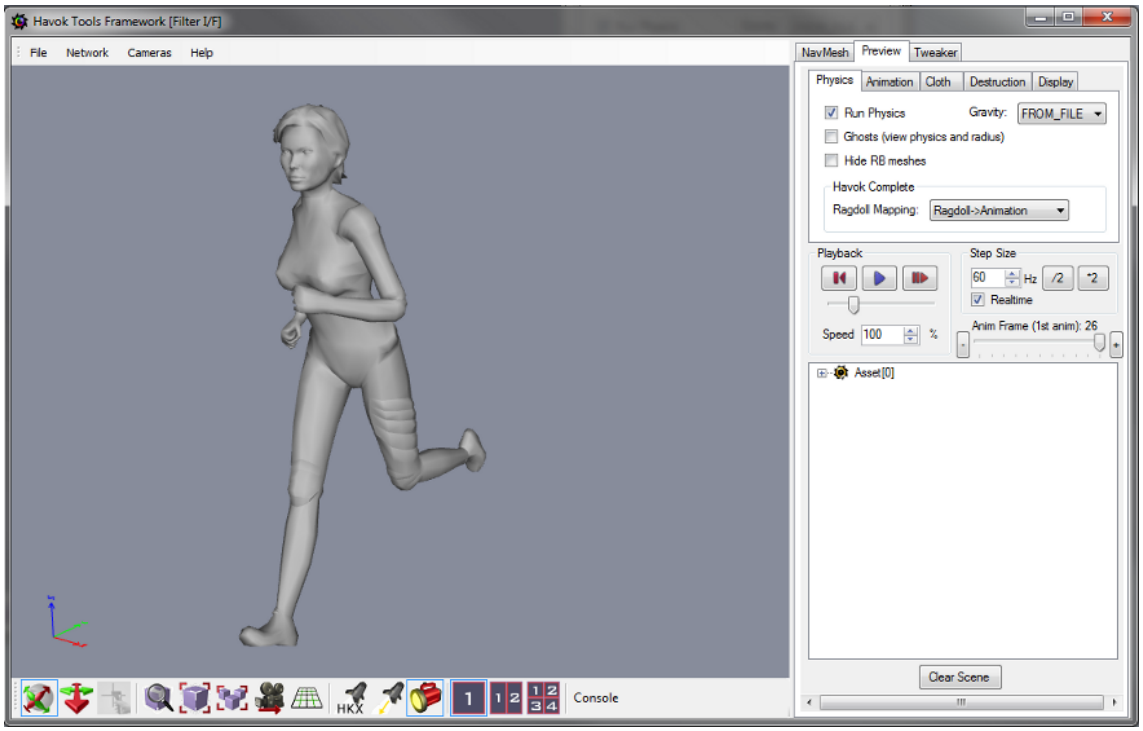
Although examining the contents by viewing the XML output give us an indication on how our processing went, in most cases visual inspection is the only way to ensure that the processed content matches our expectations. Havok provides the Preview Tool filter (in the **Graphics** category) just for that: this filter is built with the Havok runtime, including some display libraries, and will play back any Havok content it finds.

Replace the View XML filter at the end of the filter list with a Preview Tool filter, and click again on **Run Configuration** :




A modeless window will appear with our animation and skinning running in real-time:



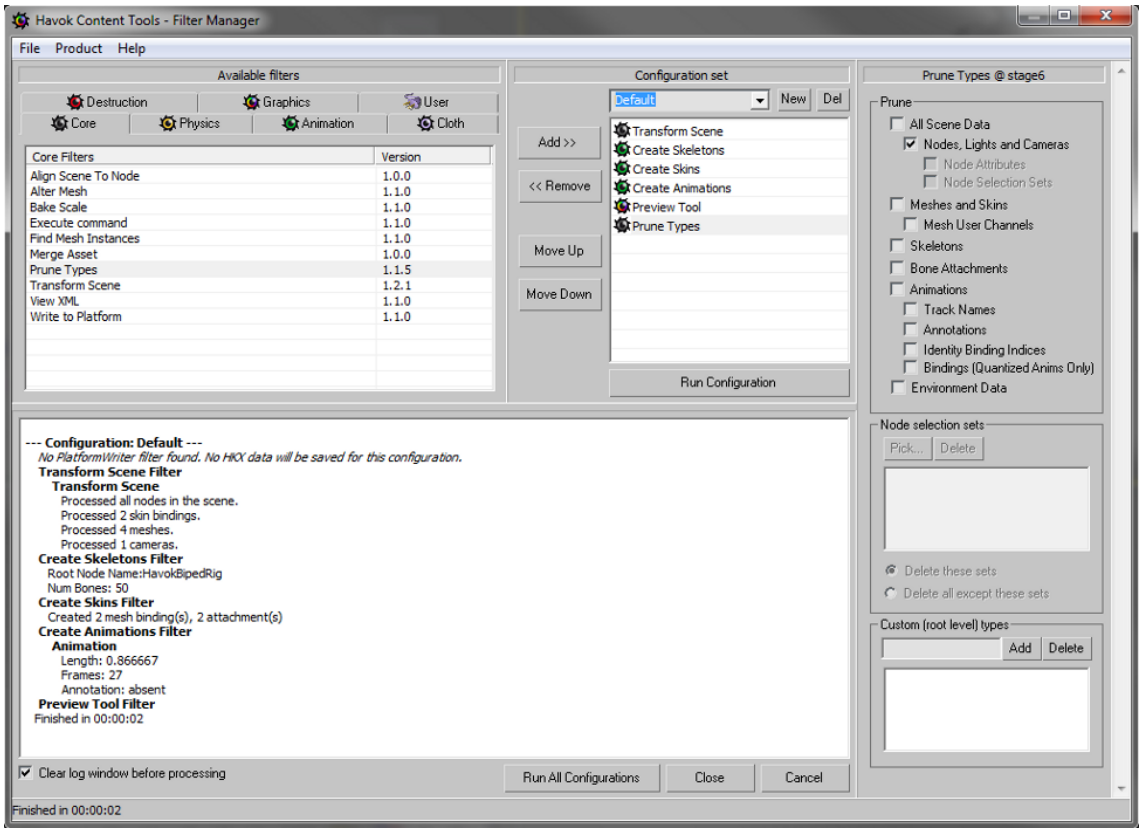



The Preview window gives you multiple options to change the speed of playback, the current frame in the animation, etc. Check the Preview Tool filter documentation for more details.

#### 4.4.5.5 Pruning Some Data

Most of the filters either modify (e.g. the Transform Scene filter) or add (e.g. the Create Skeleton filter) information to the asset. Sometimes it is useful to remove some of the data in the assets that is no longer relevant (particularly at the end of the processing). For example, once we've created our skeletons and animations, the original data regarding nodes and their keyframes is probably no longer needed (since it has been transformed into other classes).

We can add a Prune Types filter (in the **Core** category) after the Preview Tool filter. Leave the default options as they are; this will remove nodes (and their keyframes), lights and camera information.




**Note:**

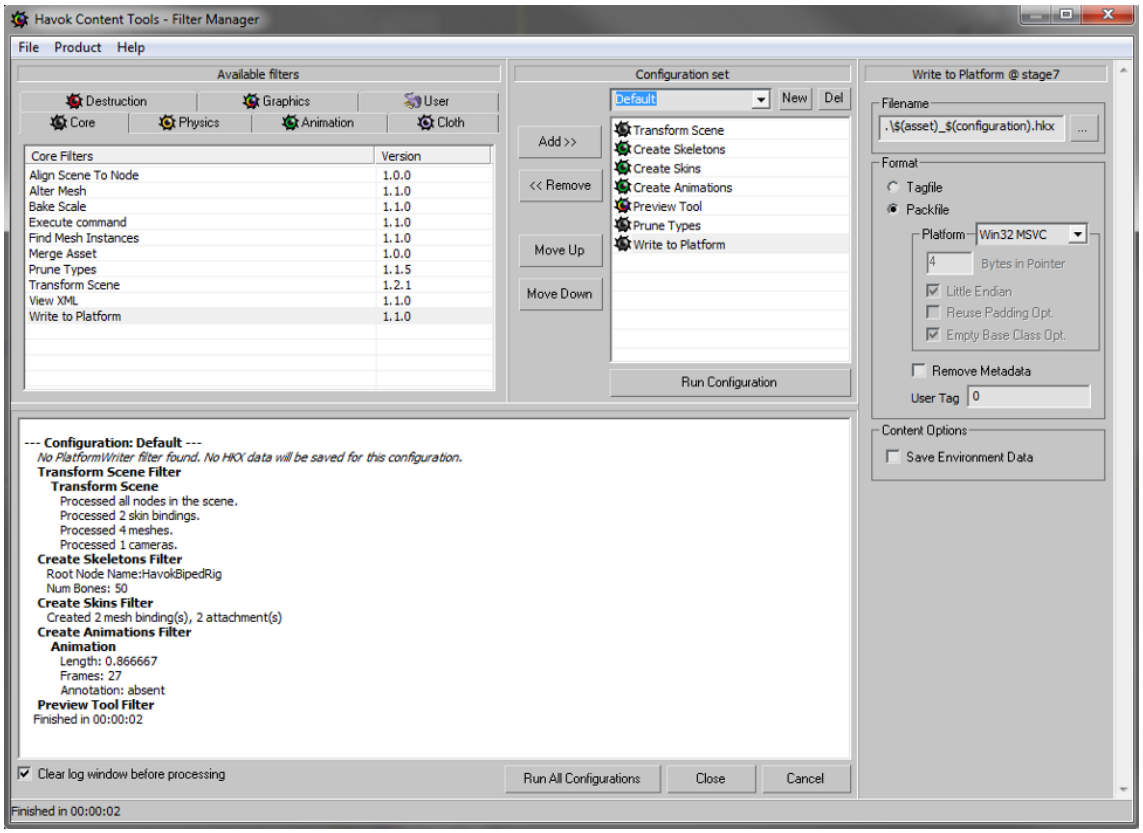
You could also add this filter before the Preview Tool filter, but since the Prune Types filter will remove camera information, the preview will have to use a default camera (rather than reflect the camera in the original scene).

#### 4.4.5.6 Writing the Processed Contents

So far, all the processing done by the filters has happened in memory, and hasn't been written to any file.

The Write to Platform filter (in the **Core** category) will serialize the processed content to a file. It can write either XML format or binary format for many different platforms. Add the Platform Writer filter at the end of the filter configuration. By default it will save a binary `.hkx` file for the Win32 platform in the same location as the asset.

If you now process the asset again (click on **Run Configuration** ), you will observe in the log window that a file has been written:




The processed asset is now saved alongside our 3ds Max file, ready to be loaded into the Havok SDK.

This finalizes this first tutorial: you can find the final 3ds Max scene in the "`tutorialEnd.max`" file.

We'd like to encourage you to experiment with different filters and filter options. By checking the results in the filter processing log, as well as the output of the View XML filter, you should be able to get a good idea of what each filter does.

If you are a Havok Animation/Complete user, the following sections go a little further in processing this scene by doing some motion extraction. If you are a Havok Physics 2012 user, skip the following section and go to the Physics 2012 Basics Tutorial.

#### 4.4.5.7 Appendix (Animation/Complete Only): Extracting Motion

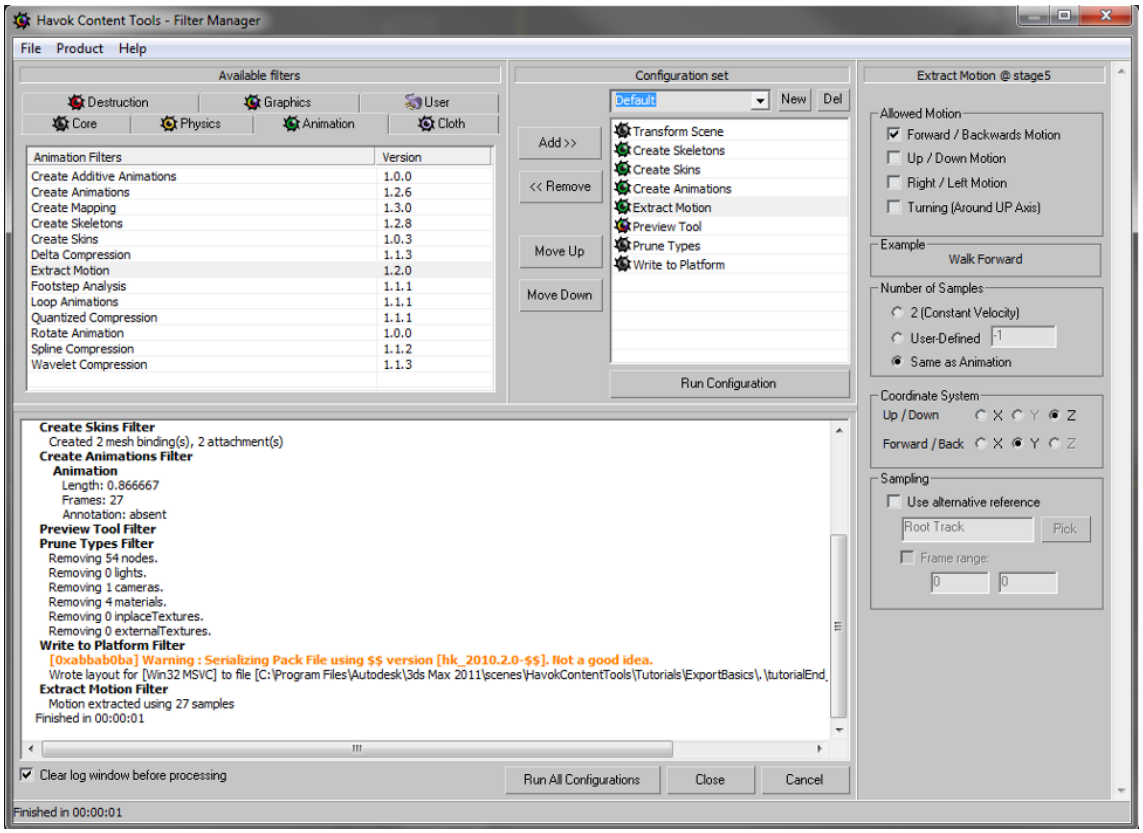
The animation we just exported has some *motion* on it: in between the first and last frame of the animation there has been some displacement on the character (it has moved forward).

Motion extraction, explained in detail in the *Havok Animation* section of this manual, is the process by which we examine an animation and divide it into two: a motion/displacement component (the moving forward during running) and a local animation component (the movement of arms, legs and body during the run cycle).

After motion extraction, thus, our running animation should become a run-on-the-spot animation, complemented by some information on how the character should displace when that animation is applied (the extracted motion).

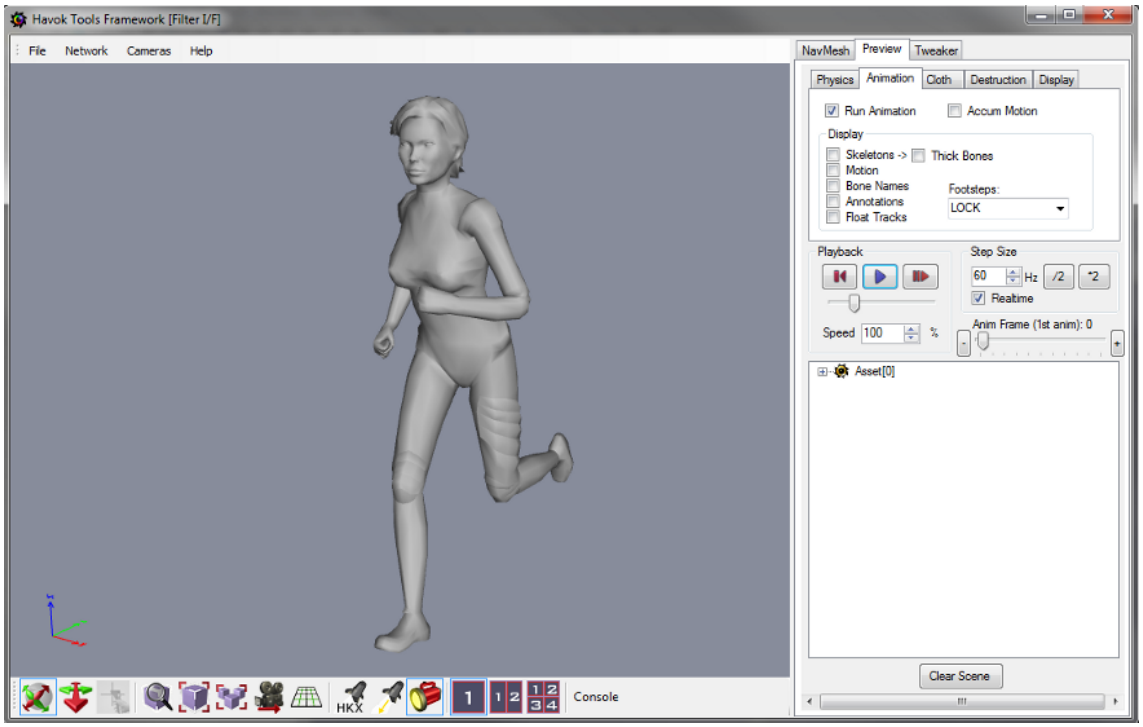
In this appendix to the tutorial we are going to do motion extraction to our run animation.

- Open the Filter Manager again (if you have closed it) by clicking on **Export** .
- Add the Extract Motion filter (in the **Animation** category) to the filter list, just after the Create Animations filter (use the **Move Up** and **Move Down** buttons to place the filter in the right place).

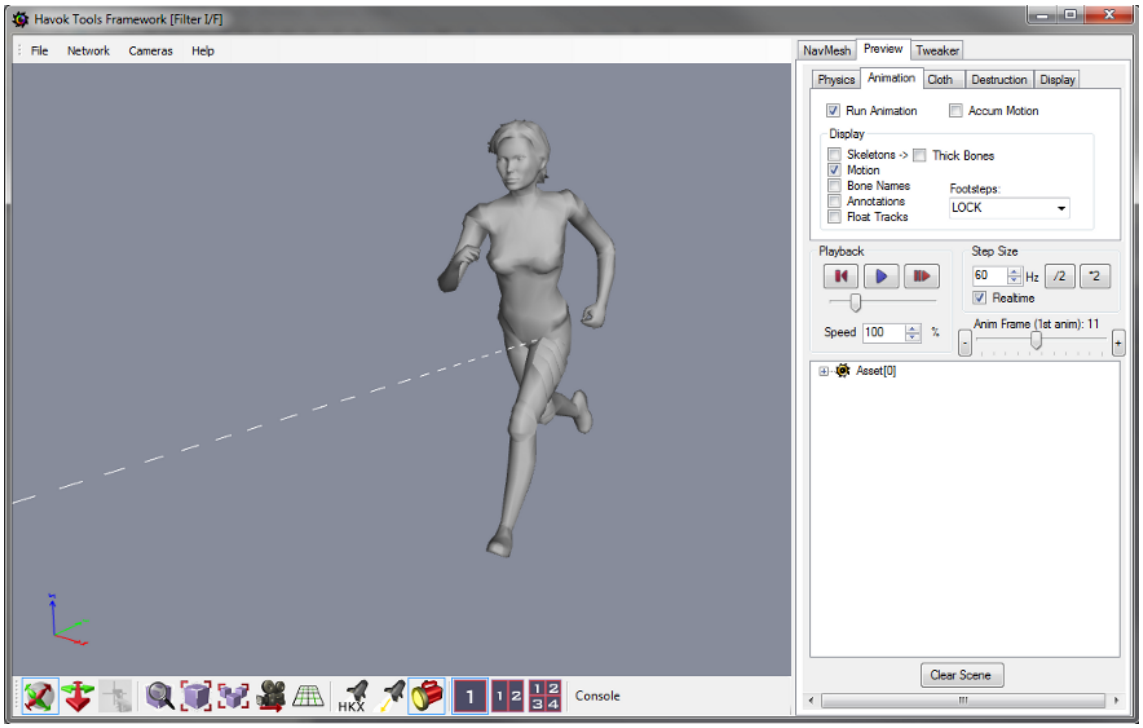



The filter options allow you to specify what kind of motion you want to extract from the animation. In our case the motion is a forward motion so we'll leave the **Forward/Backwards** check box on. But, since in our animation the forward direction is Y (the character is running in that direction) we need to specify that in the **Coordinate System** options (the default was X).

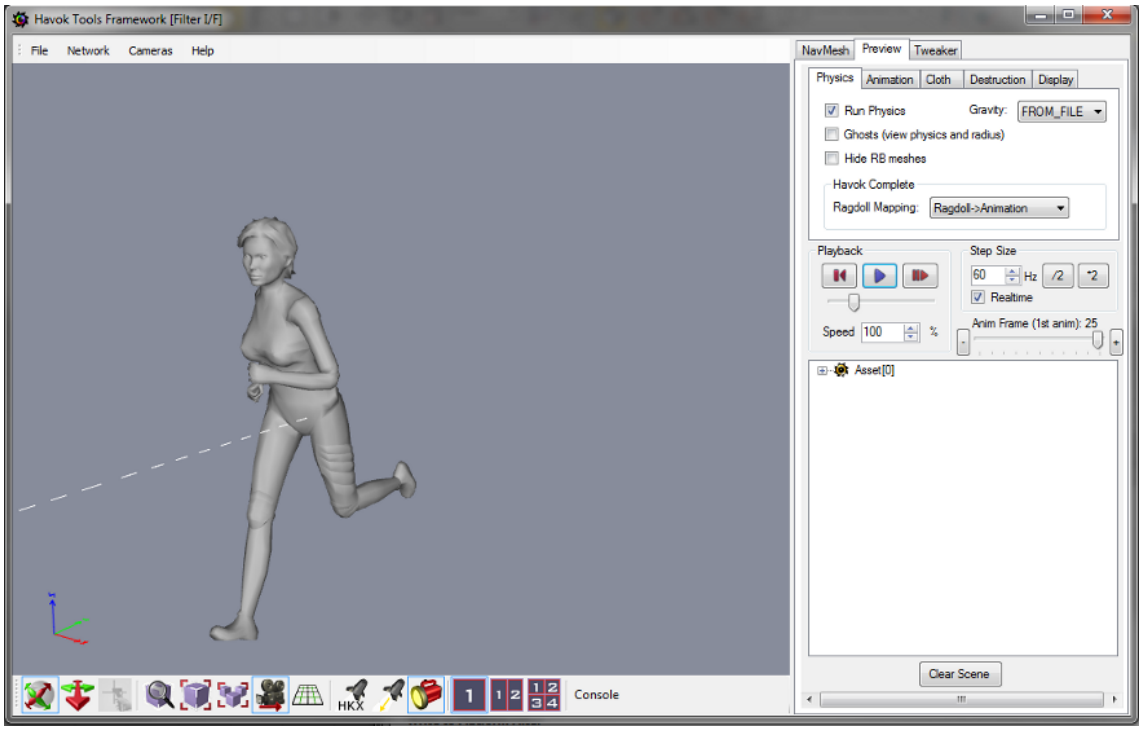
If we now run our filters, it will open the Preview dialog; you can see how our animation is now a run-on-the-spot animation:




The Preview Tool filter knows about motion and it allows you to visualize it. In the preview dialog, click on **Display** > **Motion** :




A dashed line is displayed showing the motion extracted from the animation. The preview can also apply and accumulate the extracted motion. In the **Animation** menu, click on **Accumulate Motion** :




Notice how the character now moves according to the motion that was extracted; notice also how that motion accumulates: the character moves forward continuosly over animation loops (instead of warping back to the starting position at the beginning of each loop).

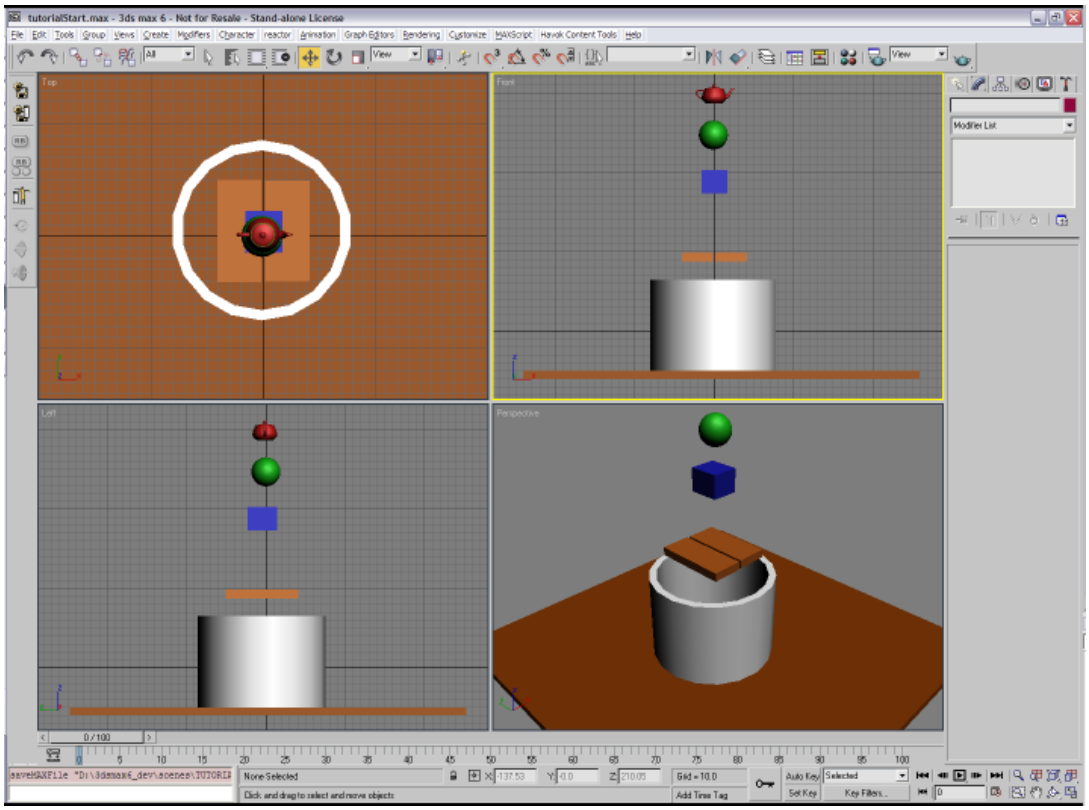
You can find the tutorial scene with motion extraction applied in the file "`tutorialEnd_MotionExtracted.max`".

### 4.4.6 Tutorial: Physics 2012 Basics

In this tutorial we are going to explore the basic operations for setting up rigid bodies and constraints in 3ds Max, and processing them through the filter pipeline. This tutorial assumes that you are familliar with the basics of exporting and processing scenes, covered by the Export And Animation Tutorial. We also recommend that you take a look at the rigid body and constraint concepts sections of this documentation.

#### 4.4.6.1 Getting Started

Start by loading the scene "`tutorialStart.max`", located in the "`scenes/havokContentTools/tutorials/physicsBasics`" folder.

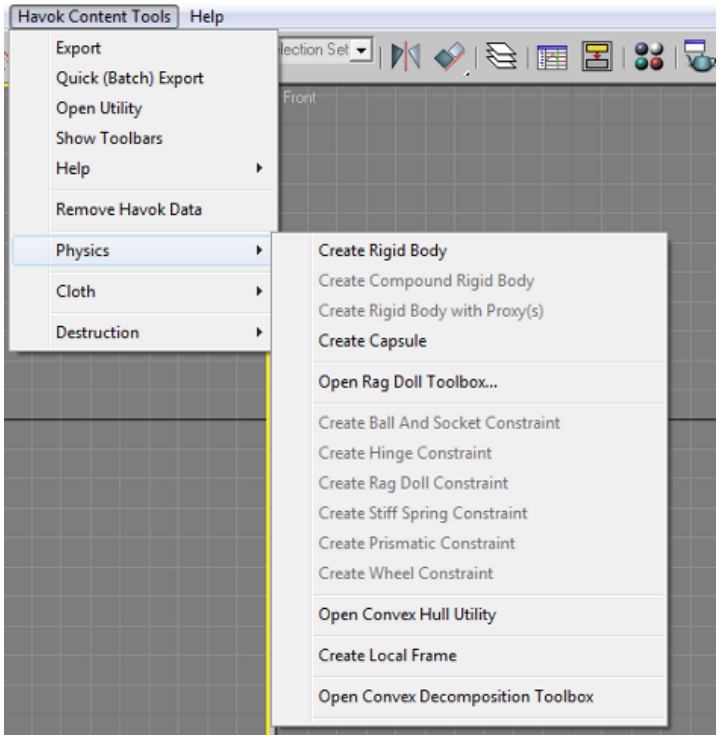



In this scene we have a set of objects: a teapot, a sphere, a box, a trap door, a can and a ground. We will attach rigid body properties to them so they behave as they would in real life.

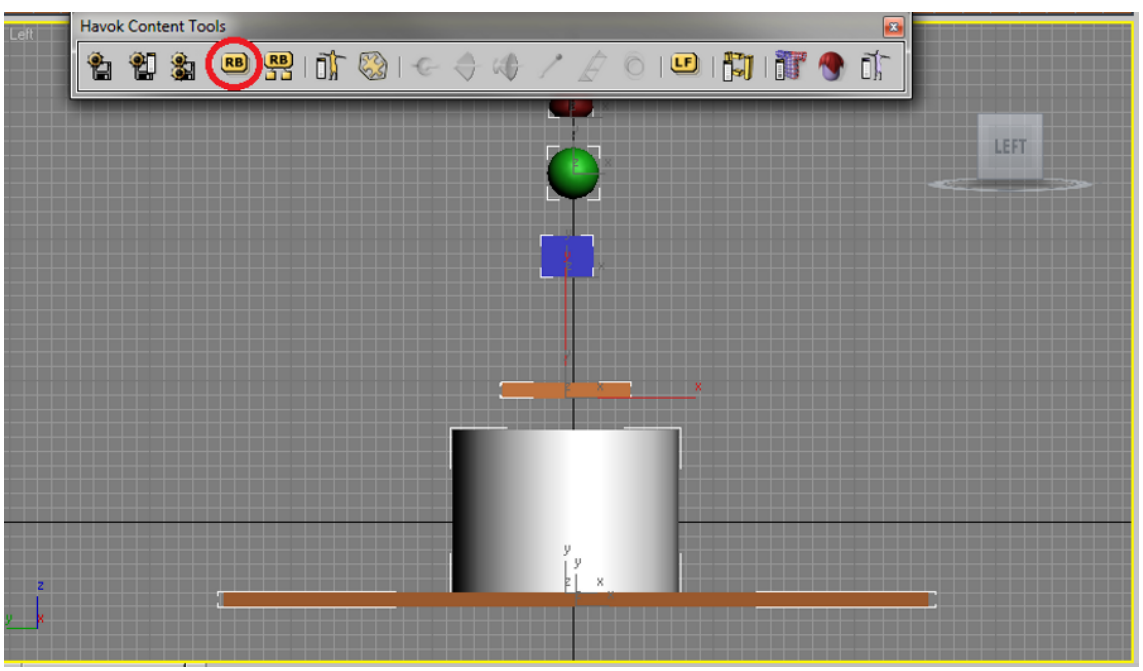
#### 4.4.6.2 Setting Up Rigid Bodies

To start with, we need to make these objects become rigid bodies, i.e., attach rigid body information to them so we can later on create rigid bodies and simulate them using the Havok SDK.

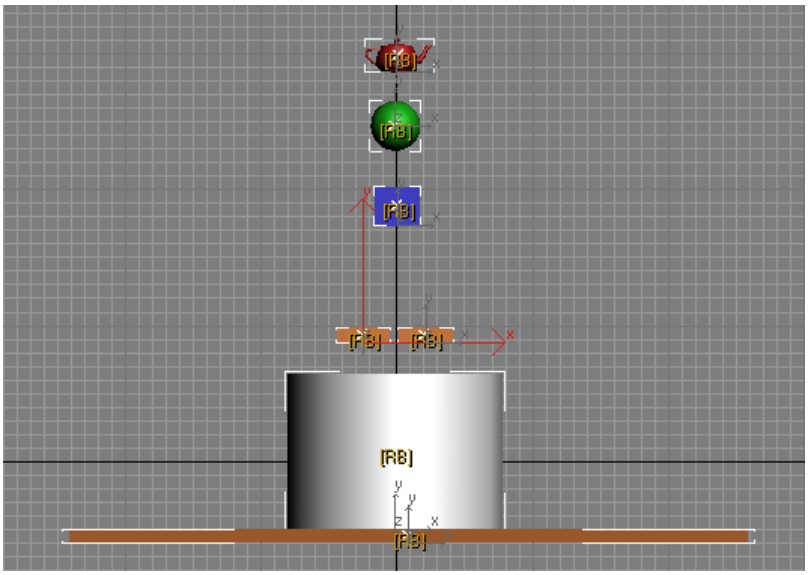
The Havok Content Tools provide a set of physics tools in 3ds Max in order to do so easily. These tools are available through the **Havok Content Tools** menu and toolbar:



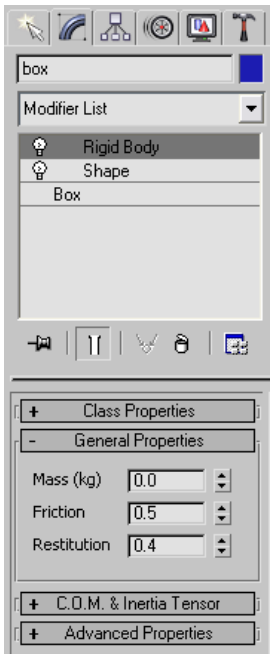
In our case, we want to create a rigid body out of every single object, so let's start by doing so. Select all the objects in the scene, and click on the **Create Rigid Body** button or menu option:



Notice how all the objects in the scene become labeled with the text " **[RB]** ":



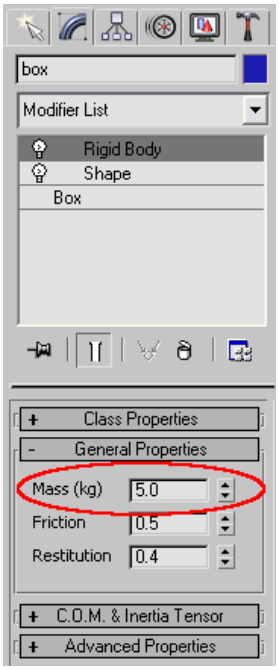

This means that those objects have now rigid body properties (and are currently fixed bodies). In 3ds Max, rigid body properties are stored in modifiers. For example, if you select the blue box and then go to 3ds Max's Modify panel , you should see two modifiers applied to the box:




The rigid body modifier contains information about the dynamic behavior of the object: mass, friction and restitution are specified here. The shape modifier contains information about the representation of the object for collision detection.

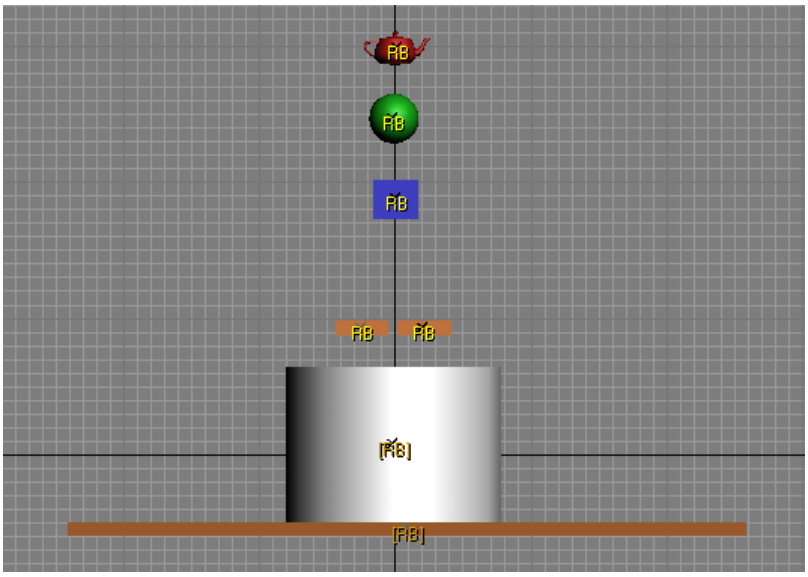
We will be exploring these modifiers a little bit more later on. For now, notice how the mass of the box is currently zero; this is the default mass for new rigid bodies. Since Havok will consider a mass of zero to be an indication of the rigid body being fixed in space (not moving), we want to make sure that we specify a mass for all the dynamic objects in our scene; that would be the teapot, the sphere, the box and the trap doors.

Set the **Mass** property to a value different than zero (5 kg for example):




Then, without leaving the Modify panel , do the same for the sphere, the teapot and each trap door (select each one of them individually and set the mass to 5).

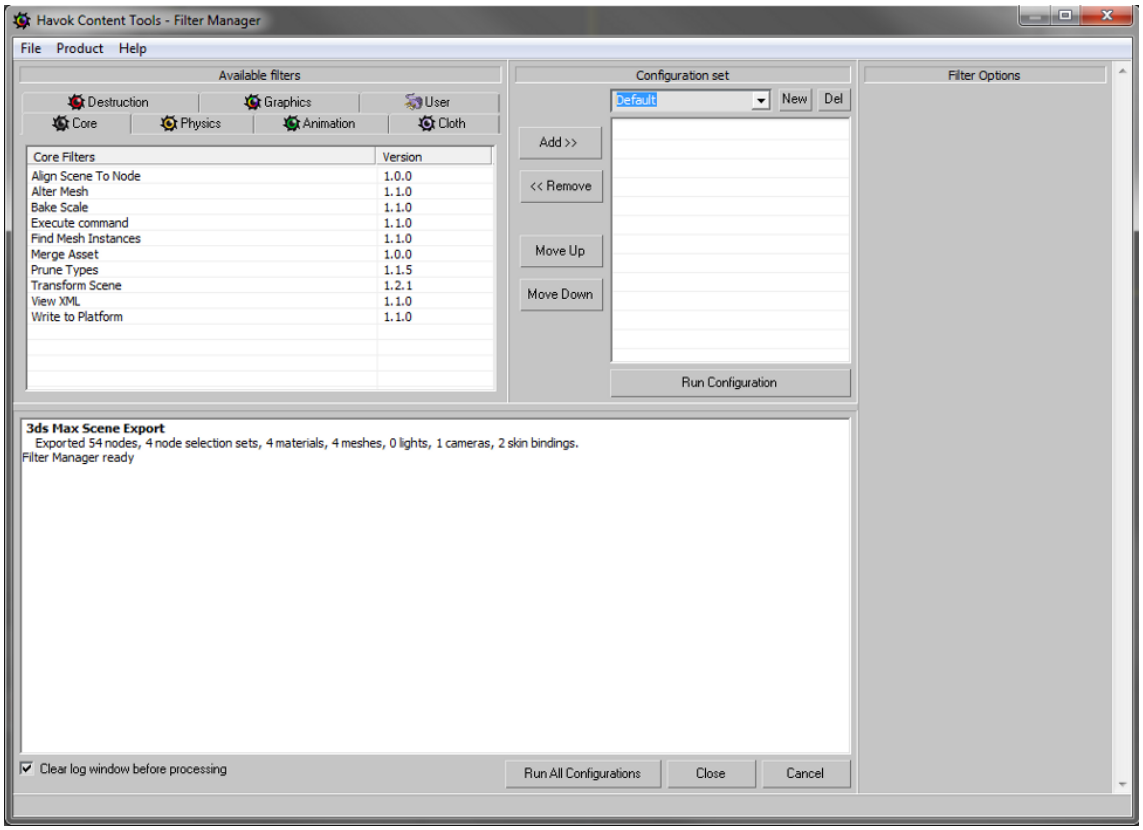
After doing this, we have a scene setup with 7 rigid bodies, 5 of them dynamic/movable and 2 of them (the can and the ground) fixed:




#### 4.4.6.3 Previewing the Scene

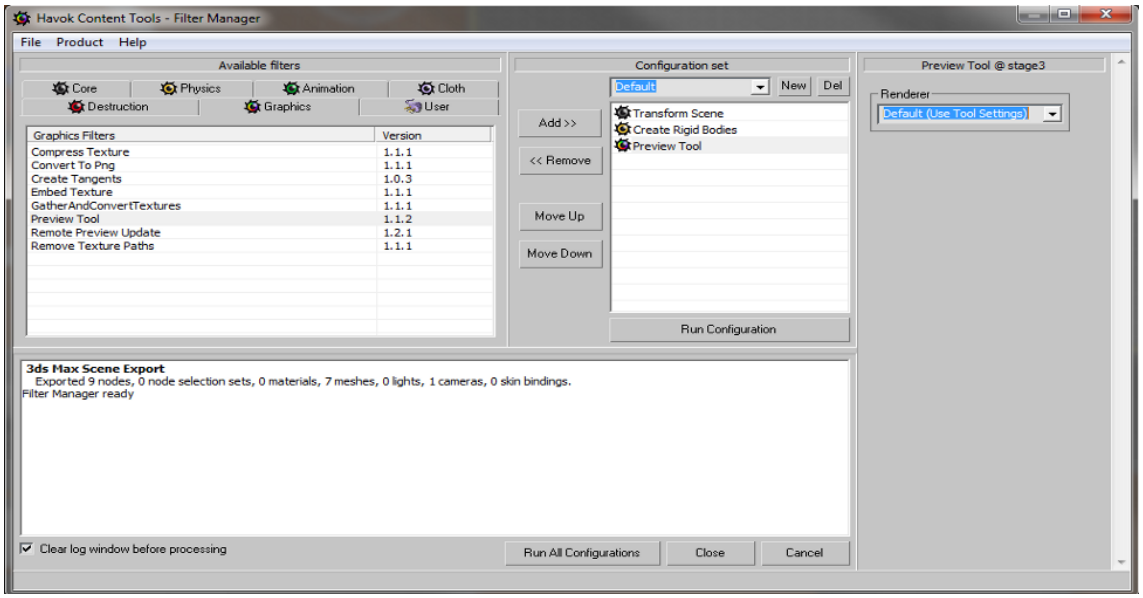
Now that we have some content set up, we'd like to preview it and check that everything behaves as expected. In the previous tutorial we saw how to export, process and preview assets using the 3ds Max scene exporter and the filter pipeline. We are now going to do the same, but this time using some processing specific to physics assets.

Start the export and processing by clicking on the **Export** button or menu option. This should open the filter manager, with an empty filter setup:

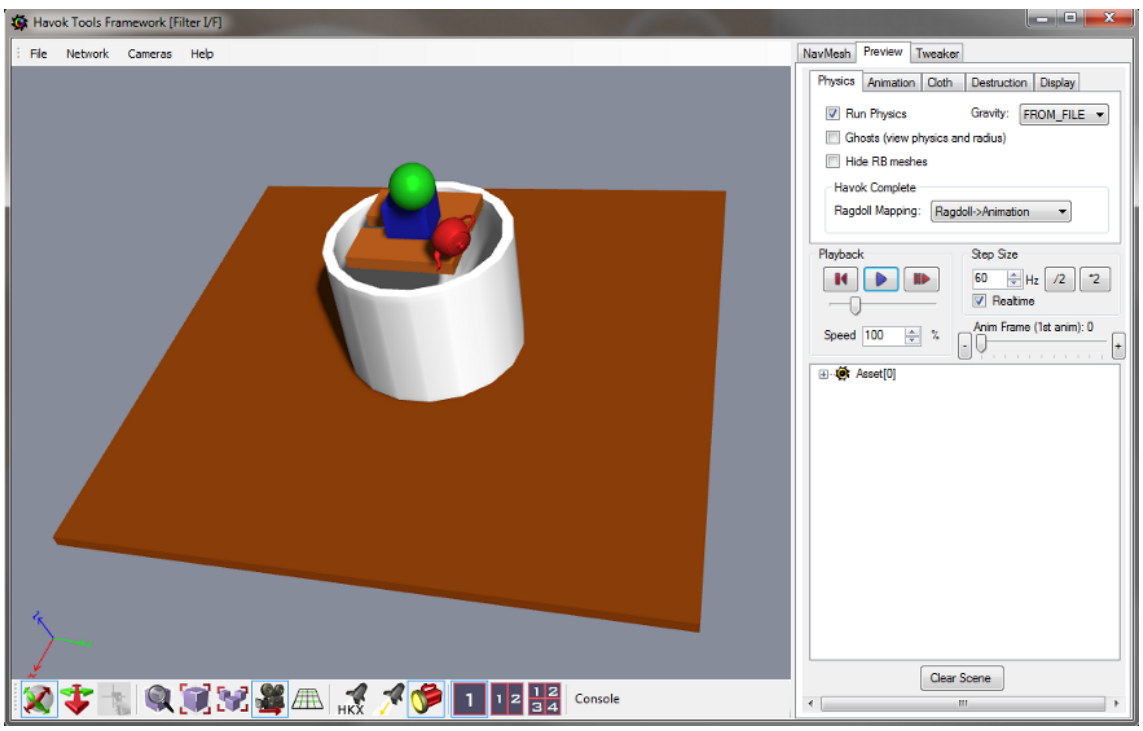



Insert, in this order, the following filters:

1. Transform Scene ( **Core** category): Since 3ds Max works in inches and we work in meters, set the **Preset** parameter to `SCALE_INCHES_TO_METERS`.
2. Create Rigid Bodies ( **Physics 2012** category): This filter will detect the rigid body information we assigned to our objects and will create rigid bodies based on it.
3. Preview Tool ( **Graphics** category): We want to visually inspect the behaviour of our objects, so we will use this filter to render and simulate them.




If we now process our asset (press the **Run Configuration** button), a preview window should appear with our scene




Press the **Play/Stop** button to start the simulation.

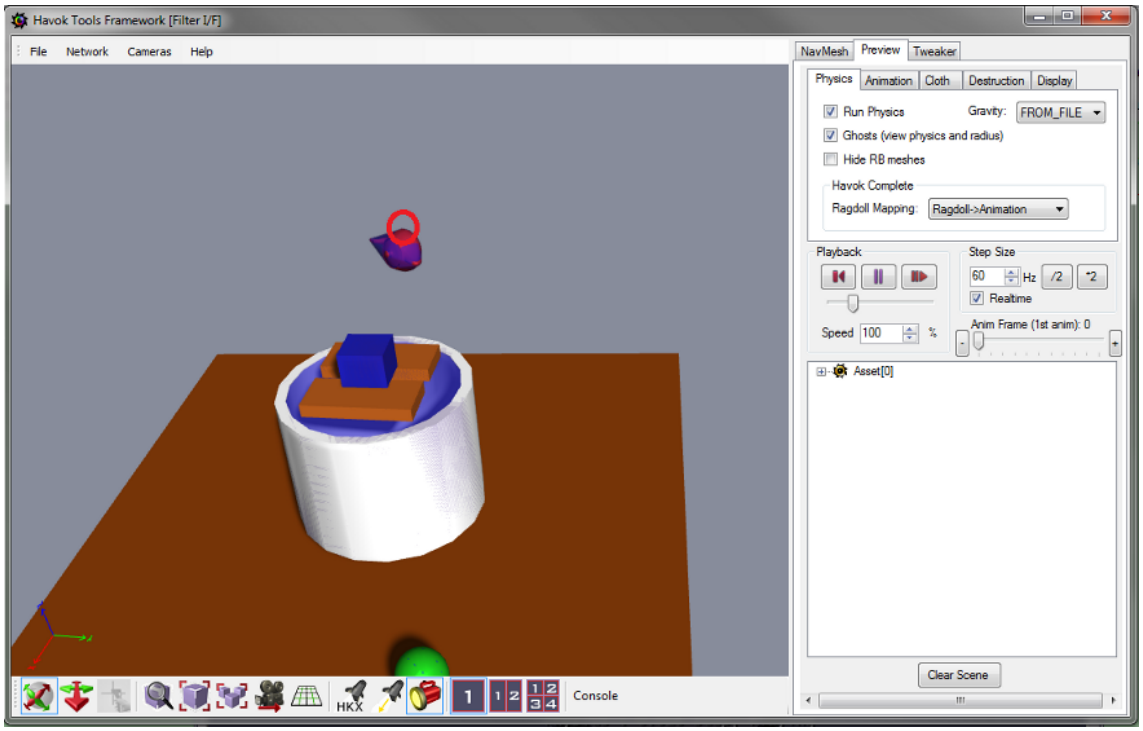
You will notice that all the objects fall as expected, but there are a couple of things that are not quite right:

- None of the objects fall inside the can–it is like there is an invisible lid on it. If you enable the menu

option **View** > **Physics 2012 Ghosts** you should see a ghost representation of the rigid bodies displayed in the preview (in alpha blended blue). That shows that there is indeed a lid on the can. It looks like the can is being simulated as a closed volume!

- The trap doors do not behave as such, and are just falling as any other object. This is because we haven't setup any constraint to specify that they should behave as hinged doors. We will take a look at this later in this tutorial.

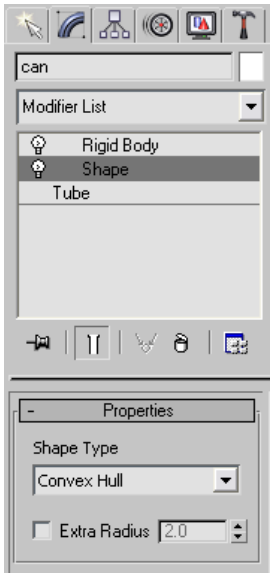
**Mouse Picking**
You can interact with the objects in the scene by using *mouse picking*: with the mouse cursor over one of the movable objects, press the **SPACE** bar and, without releasing it, move the cursor around. The rigid body will behave as if a spring was attached between itself and the mouse pointer. Release the **SPACE** bar to release the object.




Close the preview and the filter manager and return to our 3ds Max scene.

#### 4.4.6.4 Shape Properties

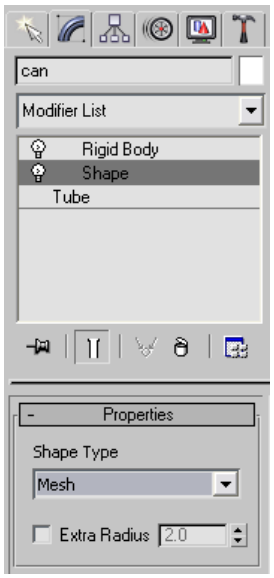
Let's take a look at the can. Since our problems are related to collision detection, we'll take a look at the shape information associated with it. Select the can object in the viewports and, in the Modify panel, select the Shape modifier:




Notice how the **Shape Type** parameter is set to Convex Hull.

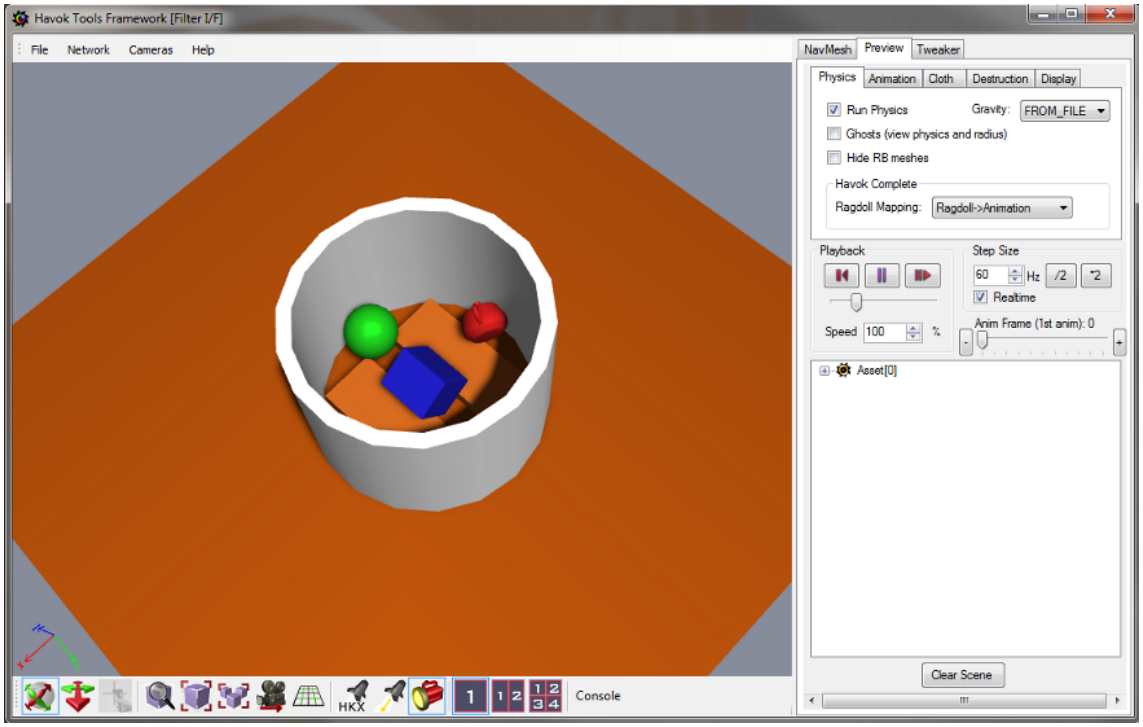
When creating a rigid body from a mesh, the default value for this Shape Type parameter is taken by examining the object type in 3ds Max. Thus, the default Shape Type for boxes is (Bounding) Box, the default Shape Type for spheres and geospheres is (Bounding) Sphere. For other arbitrary objects (like our can or the teapot), the default Shape Type is Convex Hull[1]. The Convex Hull of an object is the tightest convex volume that includes all the vertices of the object. The best way to visualize a convex hull is by imagining shrink-wrapping the object. In our case, using the convex hull of the can results on simulating a closed cylinder, rather than an open one, leading to the behaviour we saw in the preview.

In our case we want to use the exact mesh of the object. Change the **Shape Type** parameter to **Mesh** :




[1] The reason why the default Shape Type is Convex Hull and not Mesh is that convex objects are much faster to simulate than concave meshes and, in many cases (the teapot for example), simulating a mesh as its convex hull gives very good results. Also, movable (not fixed) concave objects (objects with Shape Type set as Mesh) are not really suitable for interactive simulations (a warning will be generated if used).

If you now **Export** the scene again and process the asset (press **Run Configuration** ), you will see that the objects now properly fall inside the can during the simulation:

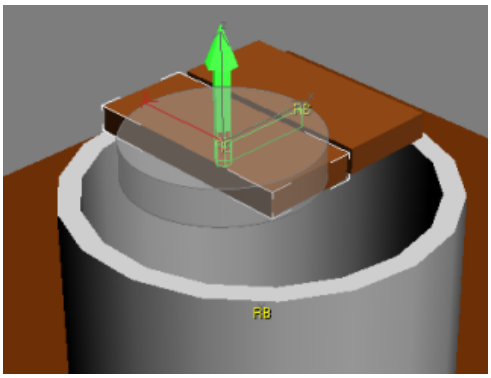



Close the preview and the filter manager and return to our 3ds Max scene.

#### 4.4.6.5 Setting Up Constraints

The last thing we want to improve in our scene is the behavior of the trap doors. Currently they behave as the other objects (they just fall). We want them to behave as if they were constrained to their location by hinges. The Havok Content Tools for 3ds Max give you the ability to set up and visualize constraints in 3ds Max. In our case, we want to create a hinge constraint–a constraint that will limit the movement of the trap doors to a rotation around a single axis.

Select one of the trap doors and click on the **Create Hinge Constraint** button or menu option:



This adds a **Hinge** constraint modifier to the object. First of all, notice how the **Constrain To** parameter is set to **World** ; this means that the constraint will only affect our rigid body, and its position, orientation and limits will be absolute rather than relative to another (parent) rigid body. This is what we want.

This modifier, when selected in the Modify panel, displays a representation of the hinge: an axis and a volume of allowed rotation (with another axis specifying where zero-rotation is).

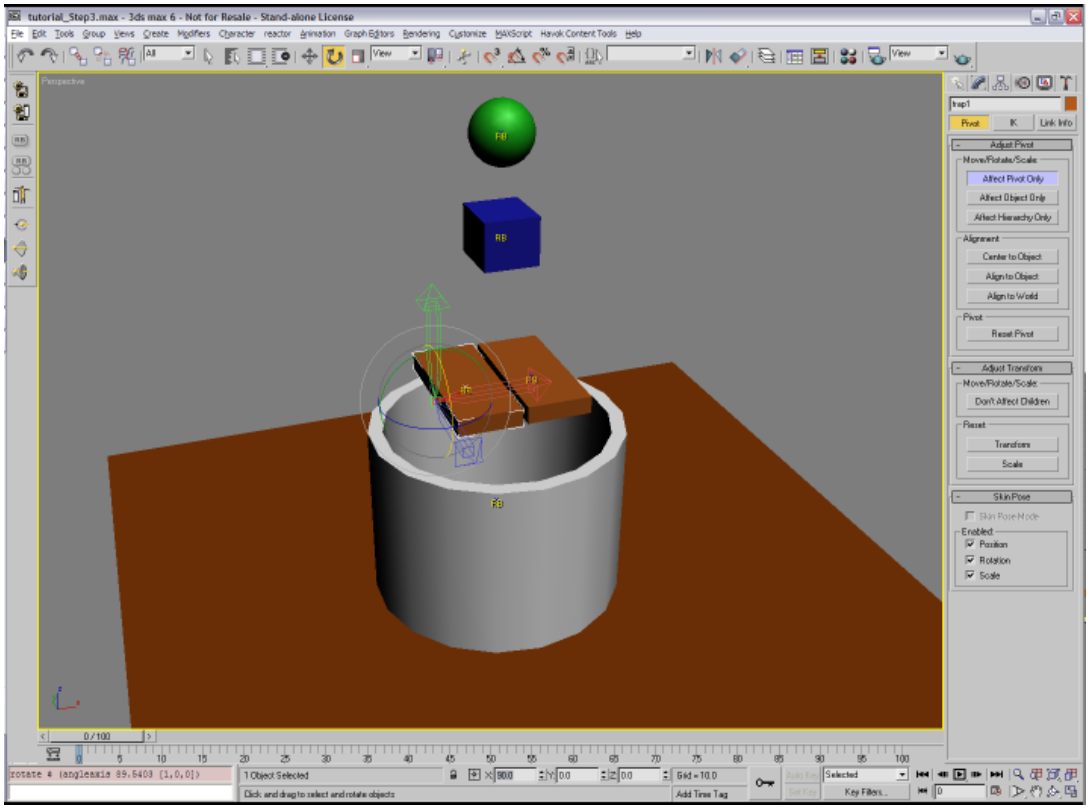
By default these axes are aligned to the object's Z and X axis, and placed at the object's pivot point. In our case, however, we'd like to place the constraint on the side of the trapdoor and orientate it so that the trap door would move up and down rather than sideways.

We can change the orientation of the constraint limits and axes in two different ways: by modifying the object's pivot or by manipulating constraint spaces.

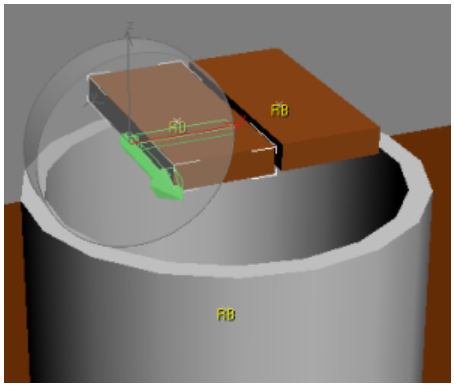
**Orientating the constraint by modifying the pivot**

With the trap door selected, go to the **Hierarchy** panel and click on the **Affect Pivot Only** button. Then, with standard Move and Rotate tools, move the pivot of the trap door to the side, and

orientate the pivot so the Z axis (in blue) points alongside the length of the door:

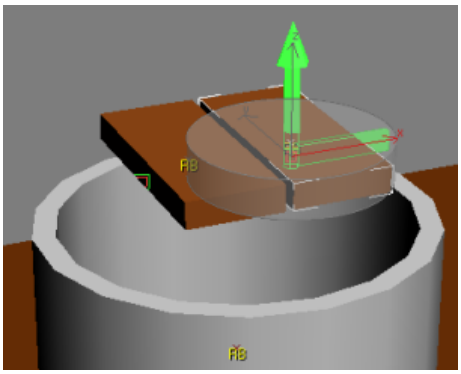



If you now go back to the Modify panel, you will see how the constraint is placed and orientated following the new setup of the pivot:



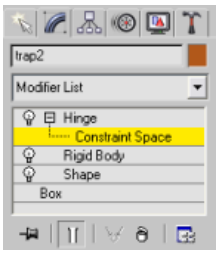

### Orientating the constraint by manipulating constraint spaces

Let's now set up a hinge constraint for the other trap door. Select the other trap door and click on **Create Hinge Constraint** :

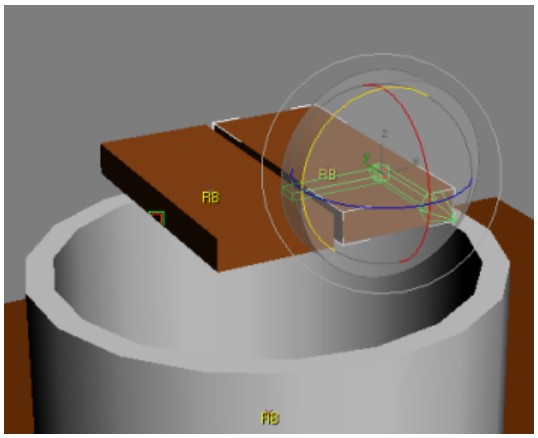



Again, the constraint's default orientation is not what we want. This time, rather than modifying the pivot of the object, let's manipulate the constraint spaces associated with the hinge.

In the **Modify** panel, open the sub-objects in the **Hinge** modifier and select the **Constraint Space** sub-object:



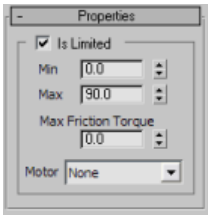

In this sub-object mode you can use the standard move and rotate 3ds Max tools to manipulate the location of the constraint. Move the constraint so that it is placed on the side of the trap door, and rotate it so the main axis points alongside the length of the door and the zero-rotation axis (the one inside the volume) points towards the other trap door:

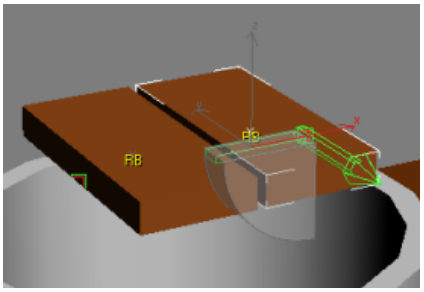



**Limiting the hinges**

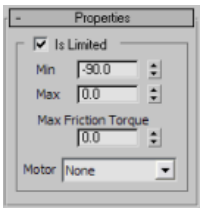
By default, hinge constraints allow full freedom of rotation around the main axis. It is possible, however, to limit that amount of rotation to a specific range.

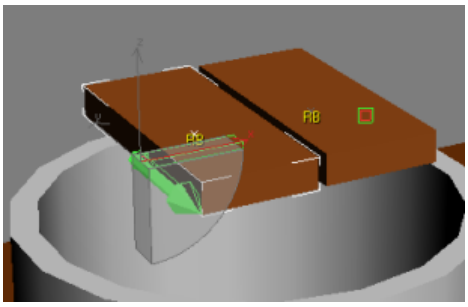
Select the right trap door, and in the **Modify** panel, check the **Is Limited** box. Notice how the **Min** and **Max** parameters are now enabled, and in the viewports the volume (displayed in transparent grey) reflects those two values (the limits are specified counterclockwise from the zero-rotation axis). Change the **Min** limit to 0. This, as reflected in the viewport, should enforce that the trapdoor doesn't rotate upwards:




Now do the same steps for the left trap door; in this case, change the **Max** limit to 0:



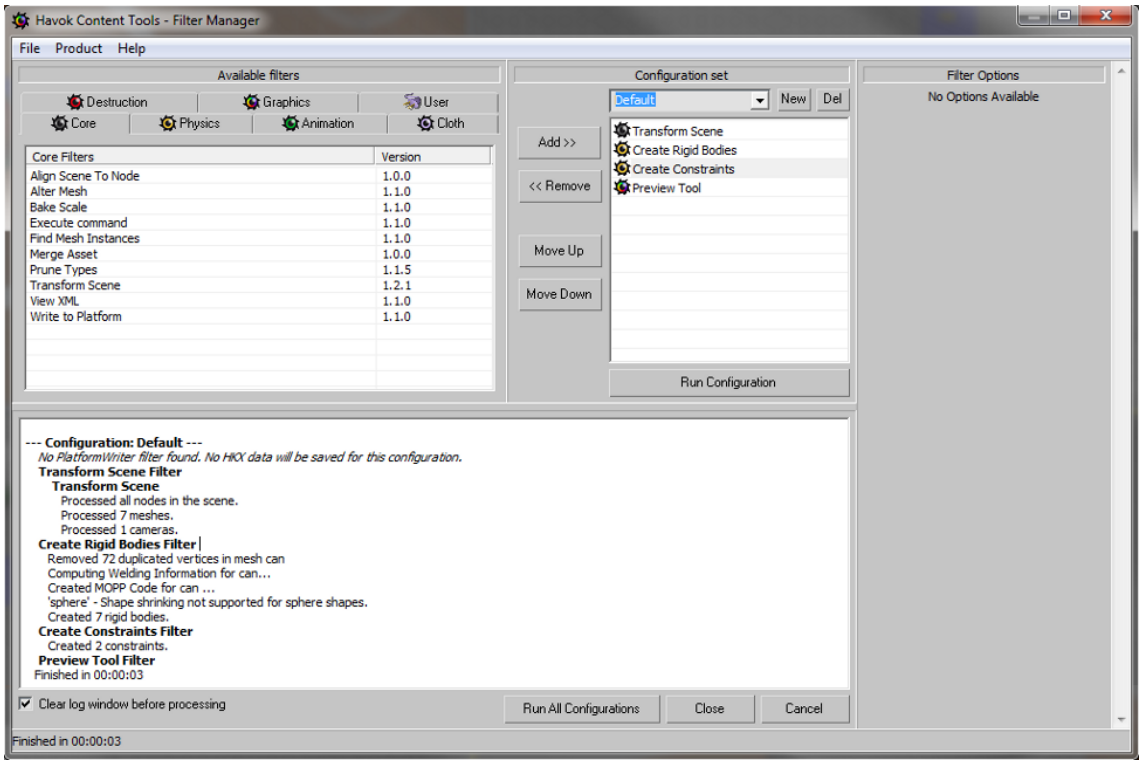




#### 4.4.6.6 Previewing the Scene

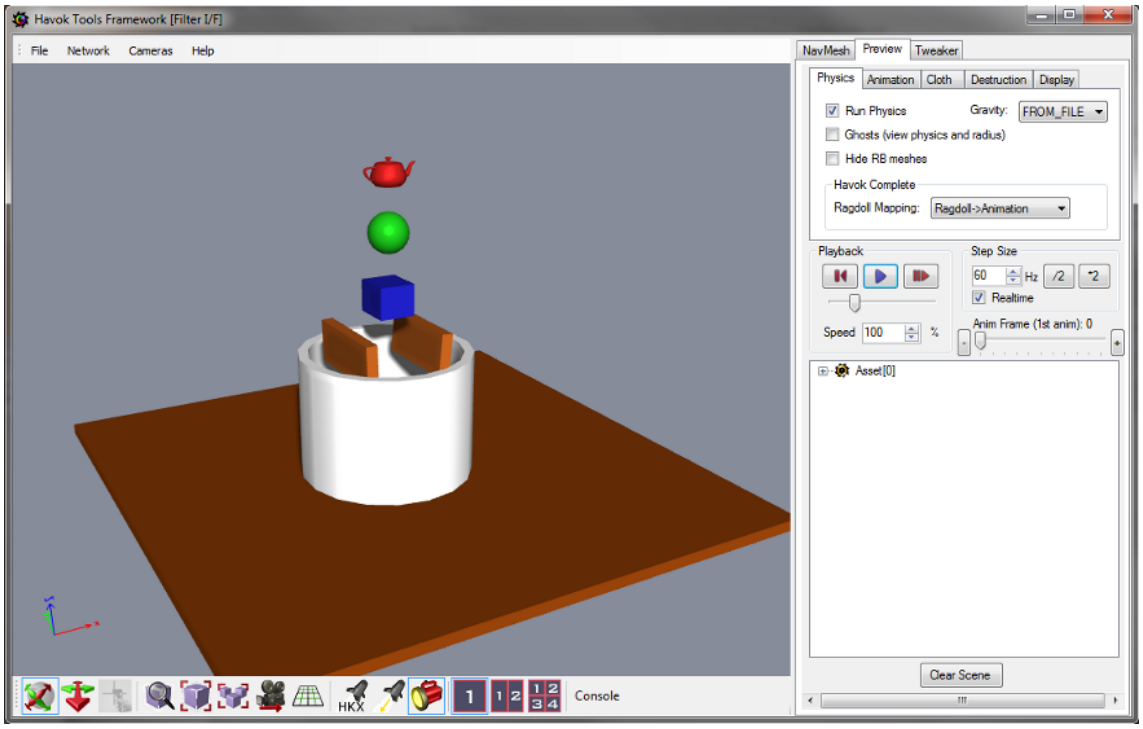
Now that we have set up our constraints, let's see what effect they have in the simulation.

Open the filter manager again by exporting the scene .

We now need to add a Create Constraints filter (in the **Physics 2012** category), in order to create constraint objects from the data we set up in 3ds Max. Place it after the Create Rigid Bodies filter, and before the Preview Tool filter.




If you now execute the processing by pressing **Run Configuration** , you will see the constraints in action:



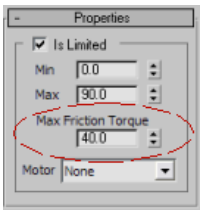

You can check that the limits we applied to the hinges are working by using mouse picking; you will notice that the doors won't move outside the limits we specified.

Close the preview window and the filter manager, and return to 3ds Max.

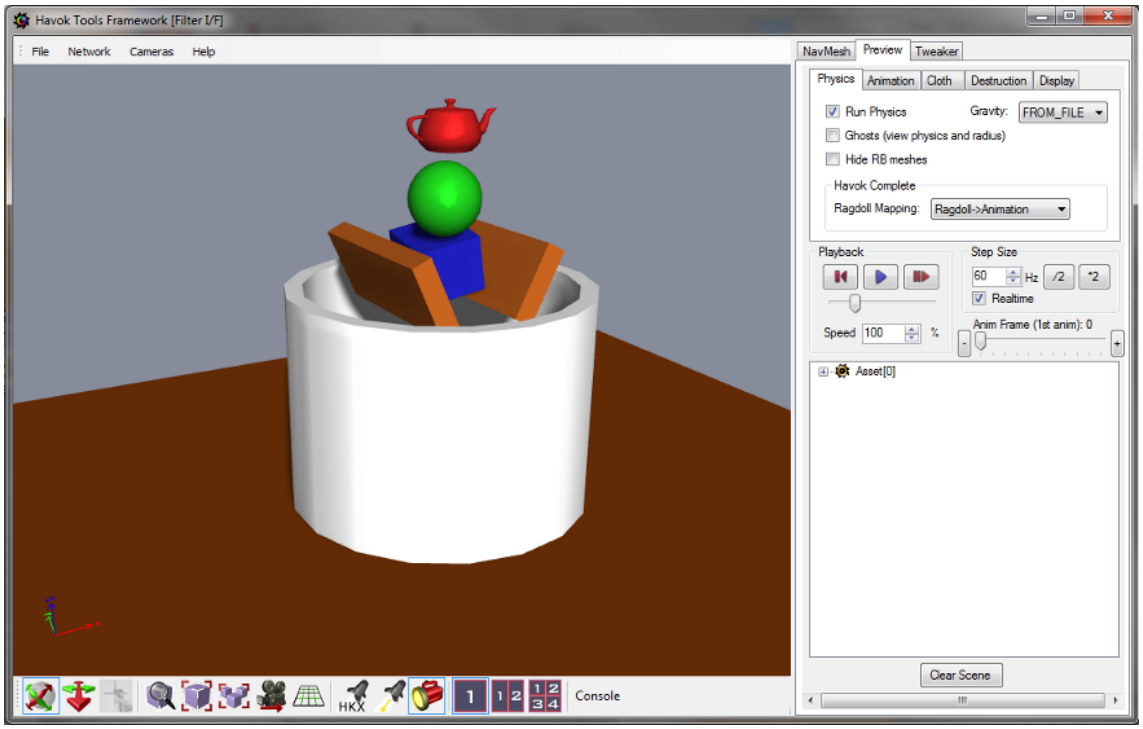
#### 4.4.6.7 Adding friction to constraints

Our setup is almost finished. The last thing we'd like to fix is the fact that the trap doors open even before the objects hit them (due to gravity). We'd like to add some friction to the hinges so the doors only move when pushed by the objects falling on them.

Select one the trap doors, and in the **Modify** panel, change the **Max Friction Torque** parameter to 40. This parameter specifies how much torque (angular force) is absorbed by the hinge's friction; only torques above this value will cause the hinge to move. Repeat for the other trap door.




If you now export and preview the scene, you will notice how the trap doors remain closed until the objects fall on top of them.



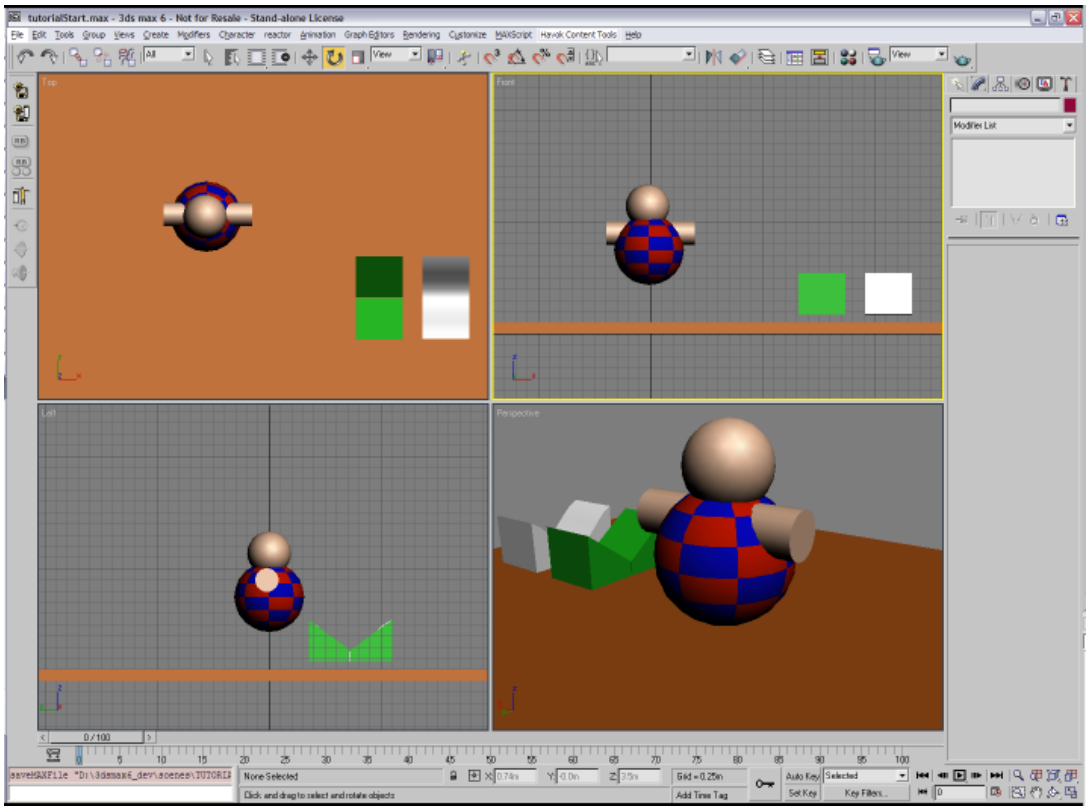

And this finishes our tutorial. You can find the fully completed tutorial in the scene file "`tutorialEnd.max`".

### 4.4.7 Tutorial: More on Rigid Bodies

In this tutorial we will explore some more concepts regarding rigid body setup, like creating compound rigid bodies and changing the center of mass. The tutorial assumes that you are acquainted with the basics of exporting and processing assets and working with rigid bodies. We recommend you follow the first two tutorials (Export and Animation Basics and Physics Basics) before proceeding with this one.

#### 4.4.7.1 Getting Started

Start by loading the scene "`tutorialStart.max`", located in the "`scenes/havokContentTools/tutorials/moreOnRigidBodies`" folder.

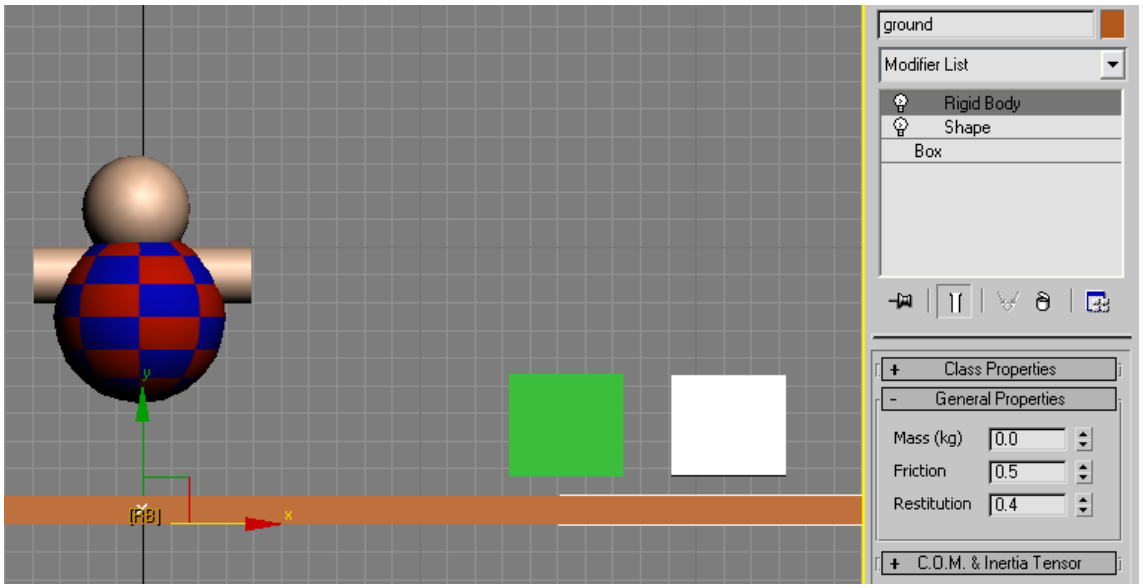



In the scene you should find the following:

- Two spheres and a cylinder, making up the shape of a little toy man
- A ground box
- A concave object (a little ramp)
- Two convex objects that together make a shape similar to the ramp

#### 4.4.7.2 Creating a Fixed Rigid Body

Let's start by making a rigid body out of the ground box. Select it and press the **Create Rigid Body** button or menu option. Since the default mass is zero, the ground box will be fixed in space. That's exactly what we want.

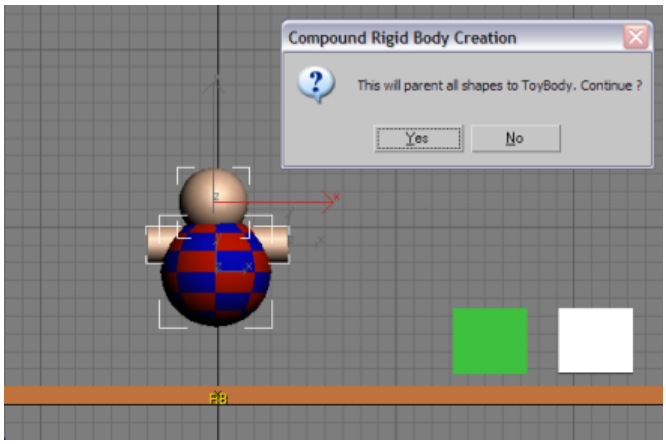



#### 4.4.7.3 Creating a Compound Rigid Body

For the toy man we want to create a rigid body from the two spheres and the cylinder. When we create a rigid body out of multiple objects, we are creating a compound rigid body.

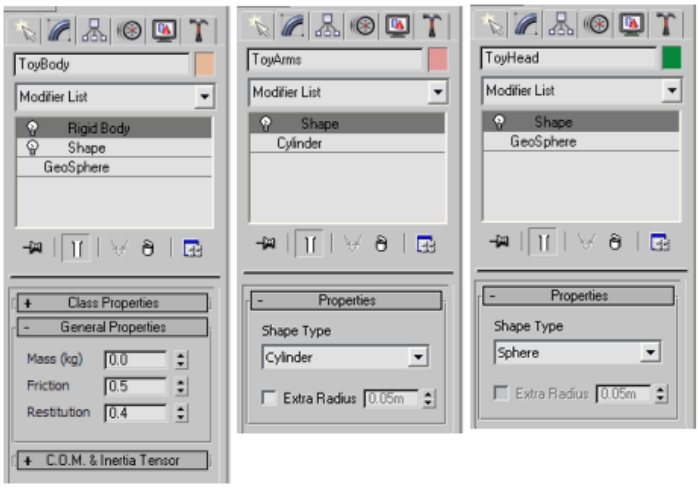
When creating a compound rigid body, one of the objects in it needs to take the role of rigid body; this object will hold the rigid body information, and the rigid body will be named after it. The other objects need to be parented to that rigid body object. If the objects are not parented already, the last object selected will be considered to be the rigid body object.

Holding ' **ctrl** ', select in turn the head, the arms and finally the body of the wobbly man. Then click on the **Create Compound Rigid Body** button or menu option:

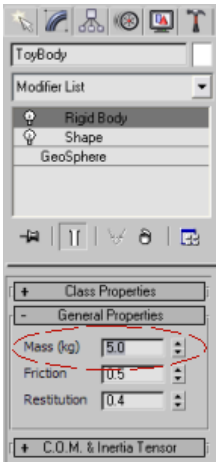



A confirmation dialog appears. This is because, as we mentioned, compound rigid bodies need to form a hierarchy. The Havok Physics 2012 Tools have assumed that the last one selected (`ToyBody`) is the parent (the one that will hold the rigid body information), but is asking us for confirmation before proceeding to reparent the arms and the head. Click **Yes** to continue.

After this, `ToyArms` and `ToyHead` will be parented to `ToyBody`. Shape information (a Havok Shape Modifier) will be added to each of the three objects (as the three of them are used for collision detection). Rigid Body information (the Rigid Body modifier) will only be added to `ToyBody`. You can verify this by looking at the modifier stack of each object:



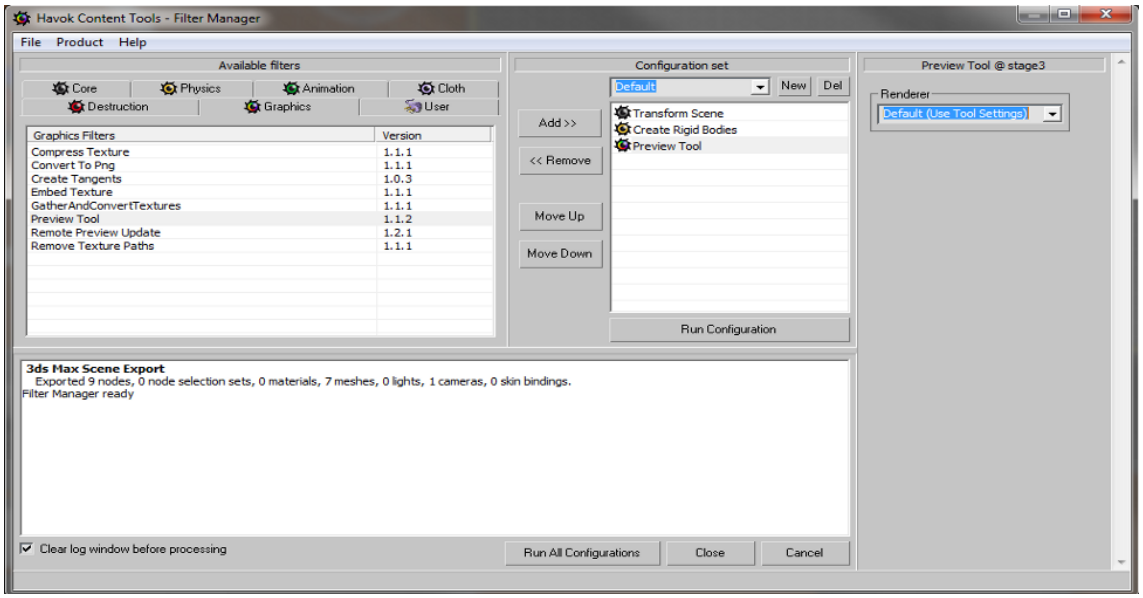

Notice that the rigid body has a default mass of zero. Since we want it to move, we need to set the mass to a different value, let's say 5 kg.



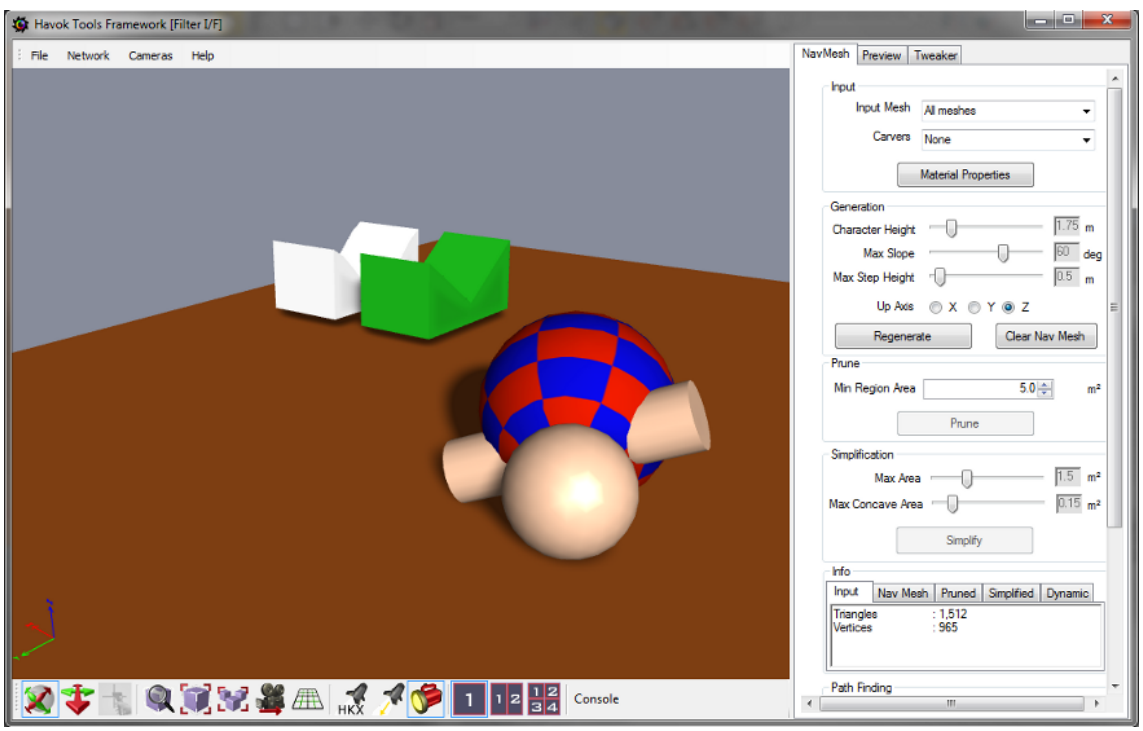

Also notice how the arms of the toy have the Shape Type parameter already set to " **Cylinder** " and the head to " **Sphere** "; the Havok Content Tools will automatically detect some 3ds Max primitives and set the right **Shape Type** for them when creating rigid bodies. It is still useful always to verify the shape type.

### Previewing the Scene

Let's now preview the behaviour of our compound rigid body. Click on **Export** and, in the filter manager, add the Transform Scene (using the `SCALE_INCHES_TO_METERS` preset), the Create Rigid Bodies and the Preview Tool filters (as we did for the previous tutorial):




Click the **Run Configuration** button; this will process the asset and open a preview window:




As you can see (using mouse picking to interact with the toy man) the three objects behave as a single rigid body.

Close the preview and the filter manager and return to 3ds Max.

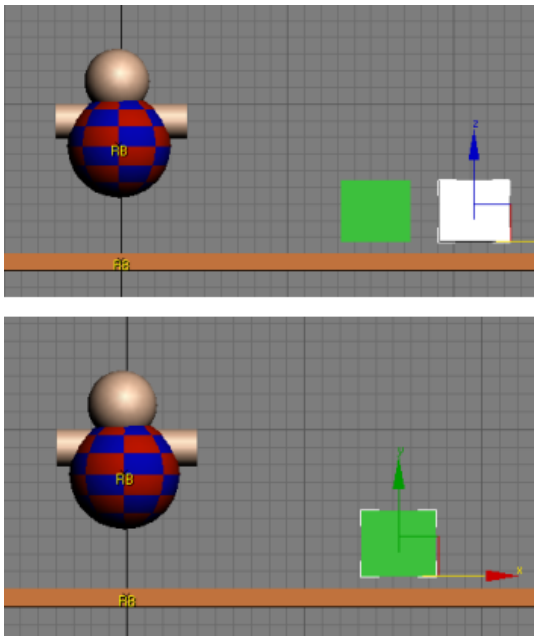
#### 4.4.7.4 Creating a Rigid Body with Proxy Shapes

Sometimes we may want to simulate an object as a rigid body using another object or set of objects for the collision detection rather than the mesh of the object itself. For example, we may want to simulate a concave object as the combination of multiple convex pieces.

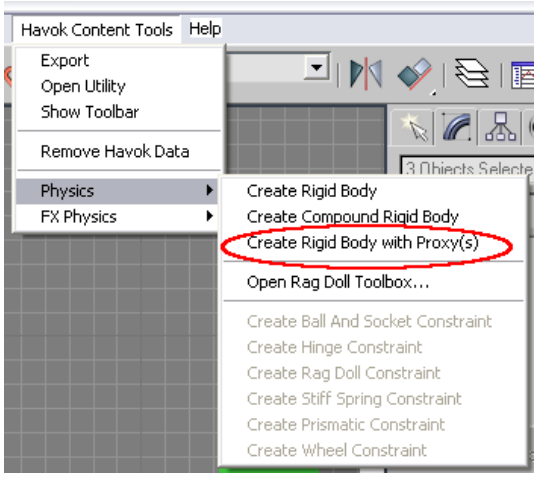
Creating a rigid body with proxy shapes is very similar to creating a compound rigid body. The only difference is that, in this case, the object containing the rigid body information (the rigid body modifier) does not contain shape information (the shape modifier) since its mesh is not used for collision detection. Shape information is instead taken only from its descendants.

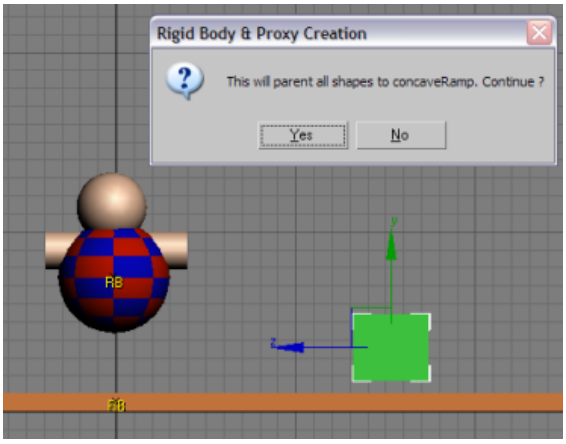
Let's see this in action in this scene. Notice how the white ramp is a concave object. Ideally, we'd like to simulate it using two convex objects (since simulating convex objects is far more efficient than simulating concave objects).

We will use the two green pieces to create a proxy shape representation of the ramp. Let's start by moving the objects so the two pieces and the concave ramp match their locations in the scene:



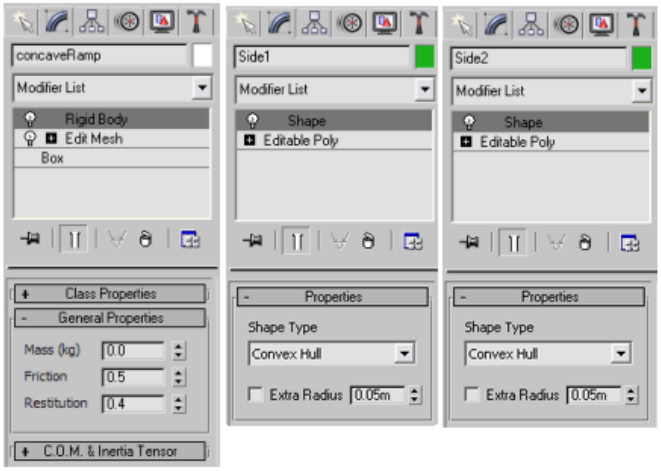

We now want to convert the white ramp to a rigid body, but using the two convex pieces as the shapes. Hence, select the two green convex shapes first and then the white concave ramp, and click on the **Physics 2012** > **Create Rigid Body with Proxy(s)** menu option.



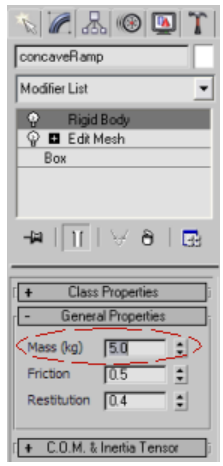




If you selected the objects in the right order, the confirmation dialog above will appear. Click on **Yes** to confirm. Alternatively, start by first parenting the two convex pieces to the ramp manually, and then with the three objects selected click on **Create Rigid Body with Proxy(s)** .

Notice how the `concaveRamp` object now has only a rigid body modifier added, while the two convex pieces (which are now parented to it) each have only a shape modifier added:

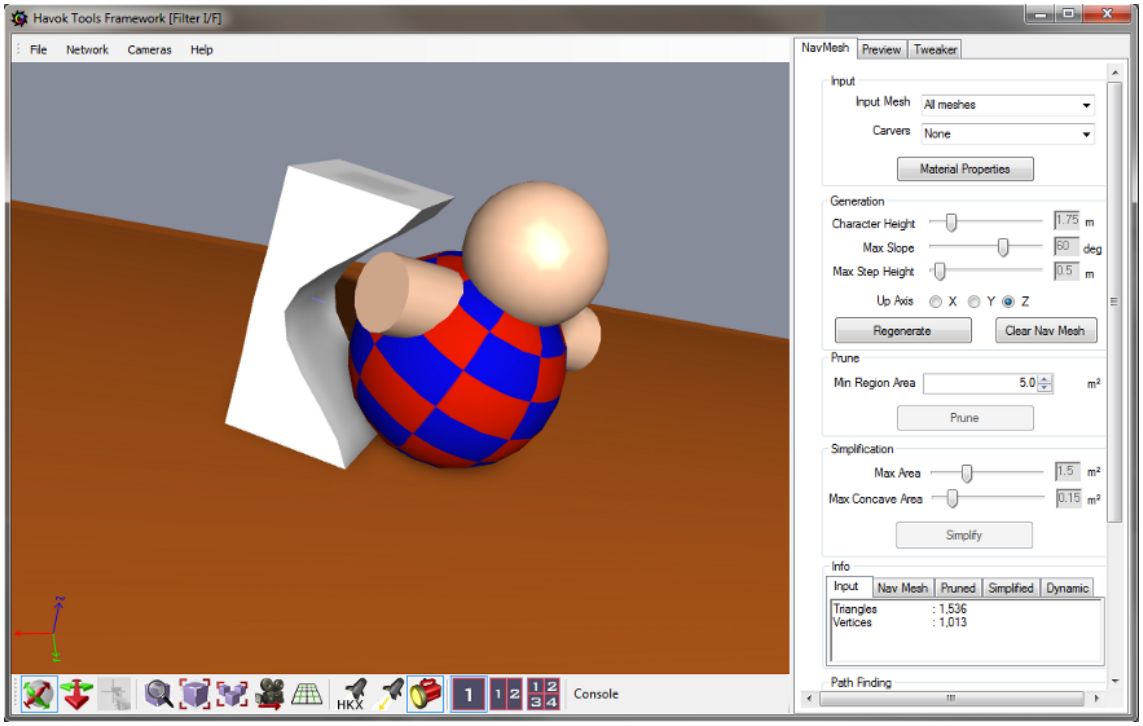


Notice how since the two convex pieces are not boxes, spheres, capsules or cylinders, their **Shape Type** is set by default to be " **Convex Hull** ". Also notice how, again, the default mass of our rigid body is zero. Since we want it to be dynamic (movable), we'll set it to something different (5 kg for example):

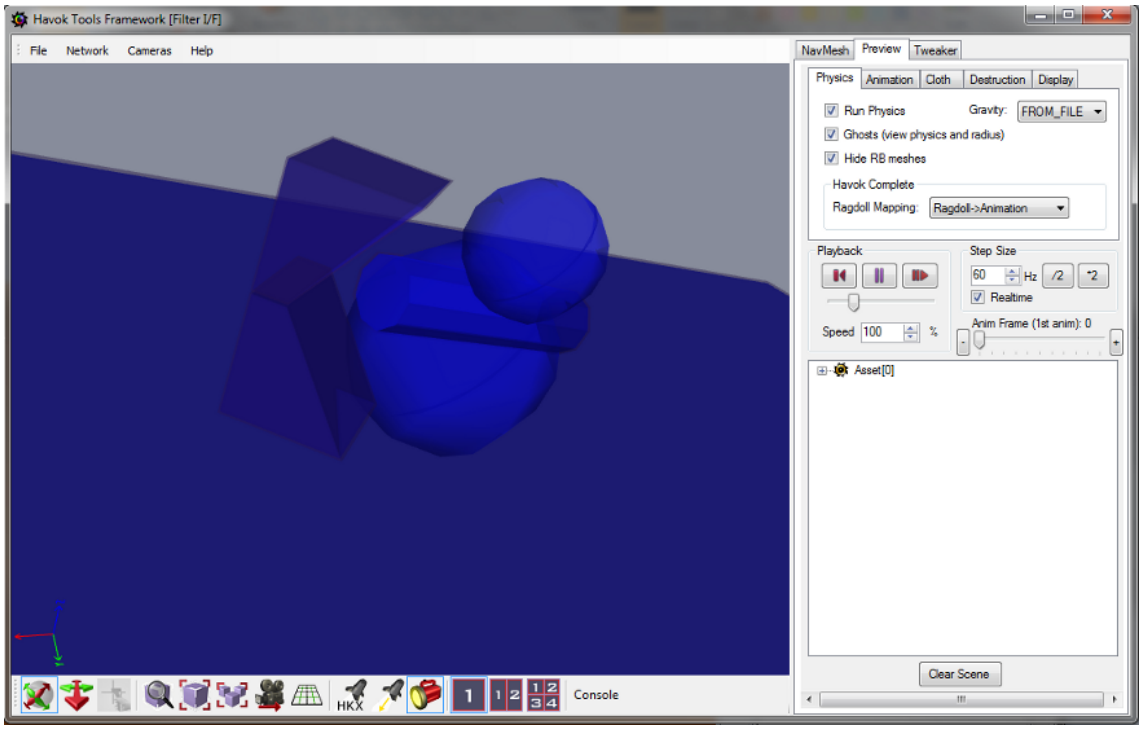



### Previewing the Scene

Let's see how the ramp behaves in our preview. Export the scene and press on **Run Configuration** to execute the preview. You should see how the ramp is simulated using the convex pieces:




To view exactly what is simulated, switch off **Display Meshes** and switch on **Physics 2012 Ghosts** in the **View** menu of the preview:




Notice how the two convex pieces, and not the concave mesh, is being used to simulate the ramp.

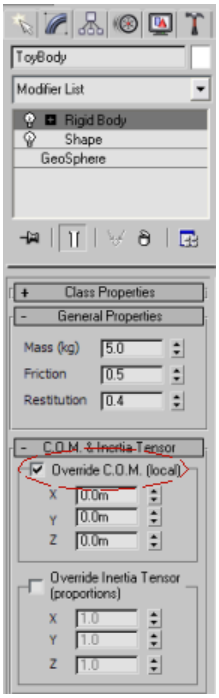
**Note:**

Unfortunately the Preview Tool filter currently displays the proxies along with the actual rigid body mesh. For this reason it may be difficult to make out the curved ramp mesh, with the proxy meshes obscuring it. Enabling wireframe mode may help here.

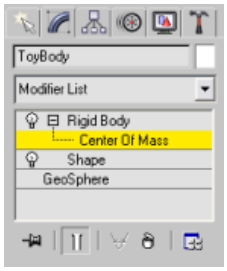
#### 4.4.7.5 Changing the Center of Mass

We'd like out little man to behave like a tumbler, one of those toys that keep themselves straight. Those toys work by having a very low center of mass (usually by having a lead weight inside them). The Havok Content Tools allow you to change the center of mass of rigid bodies easily.

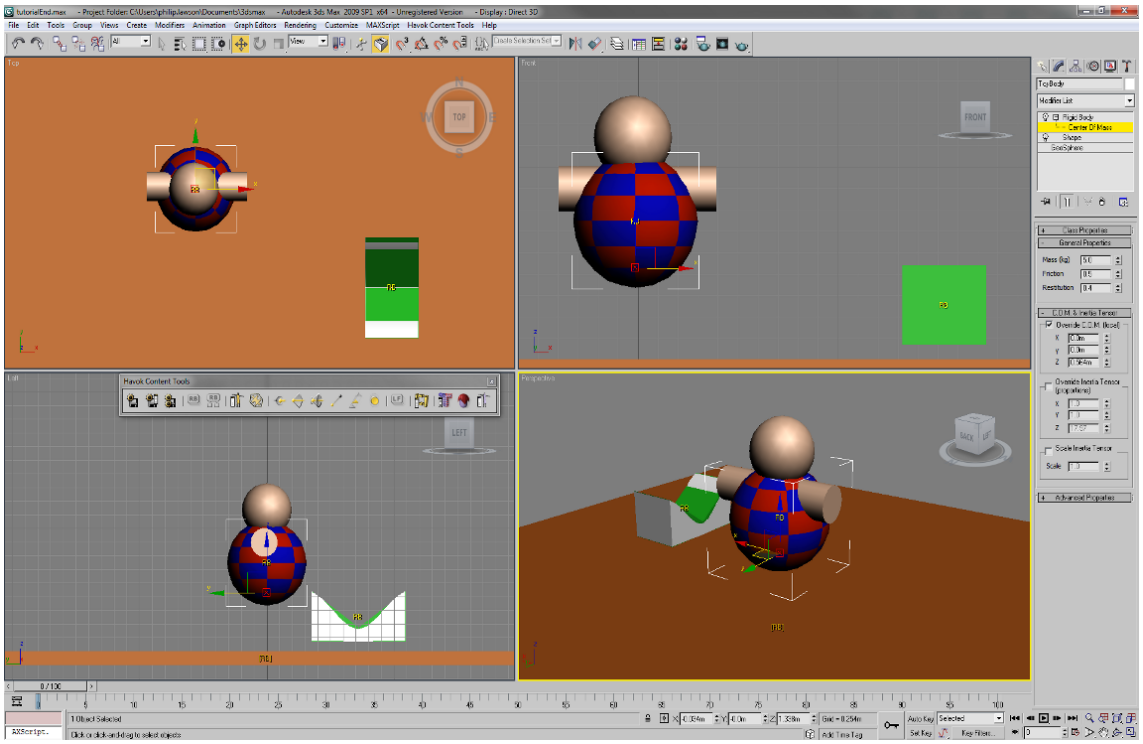
Back in 3ds Max, select the `ToyBody` rigid body modifier, and in the **COM and Inertia Tensor** rollout, check the **Override COM (local)** check box.



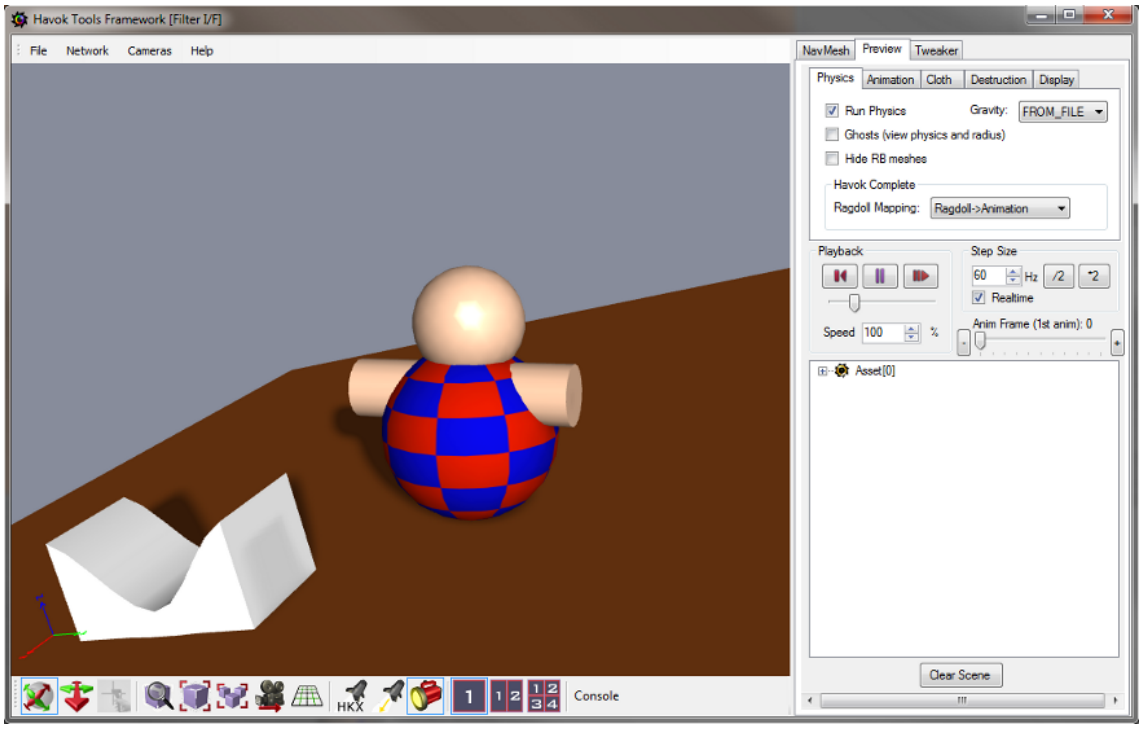

A new " **Center of Mass** " sub-object appears in the rigid body modifier:




This sub-object allows you to change the location of the Center of Mass (displayed as a red crossed box) visually in the viewports (using the standard move tool). In our case we want to lower it:



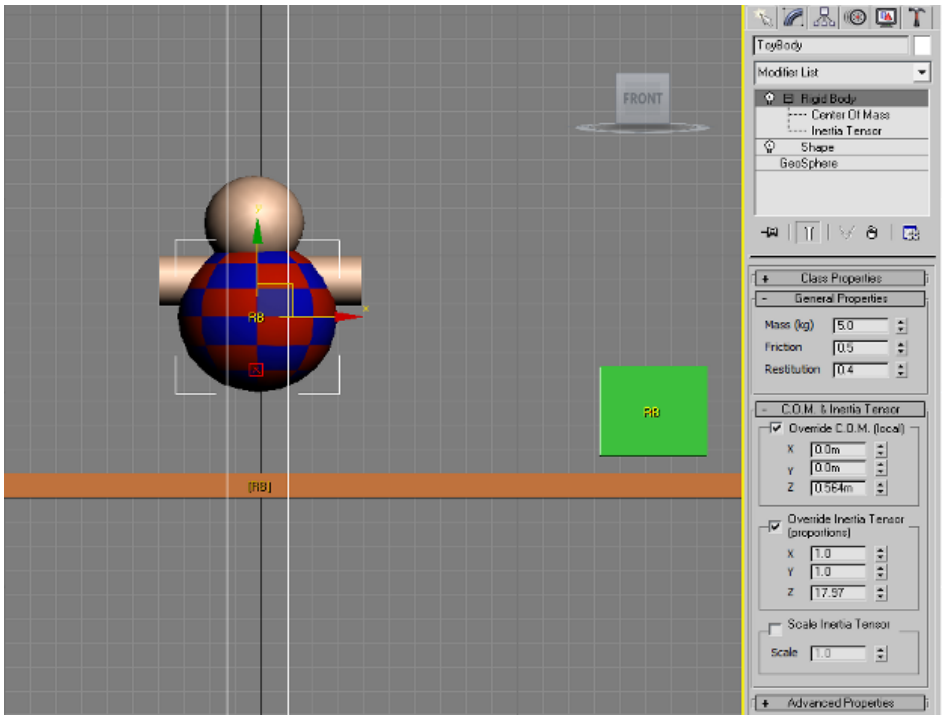
If you execute the preview again (export and press **Run Configuration** ), you see how now our toy keeps himself straight. Use mouse picking to interact with the toy:



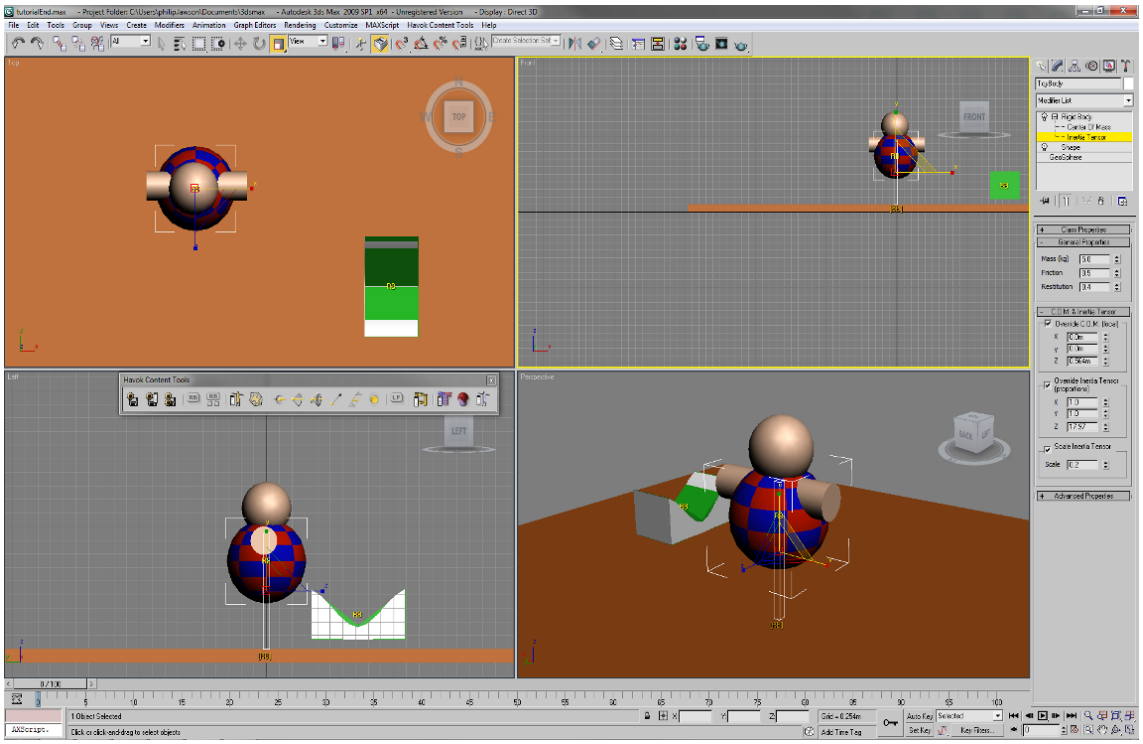
#### 4.4.7.6 Changing the Inertia Tensor

We'd now like to customize the motion of our little man even more by changing the proportions of its inertia tensor.

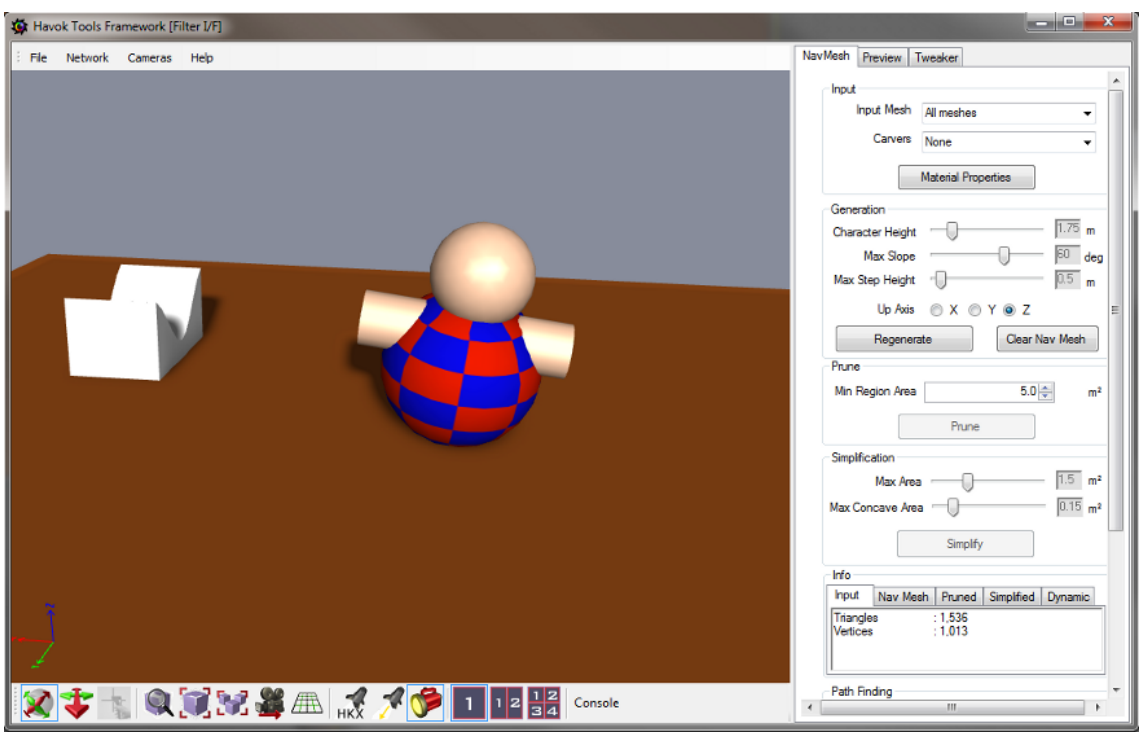
Select its rigid body modifier again, and check the box **Override Inertia Tensor (proportions)** . This will add another sub-object, **Inertia Tensor** , to the rigid body modifier, and will also show a visual representation of the inertia tensor in the viewports (a unit cube in this case):




You can visually change the proportions of the inertia tensor by going into the **Inertia Tensor** sub-object and using 3ds Max Scale tool. In our case (but feel free to experiment with different settings) we'd like an inertia tensor which is biased in the vertical direction. So we scale that direction up:



If you now preview again, you will see that our little man will tend to wobble more around this vertical direction, while it won't lean as much as before:




And this finishes our tutorial. You can find the fully completed tutorial in the scene file `"tutorialEnd.max"`.

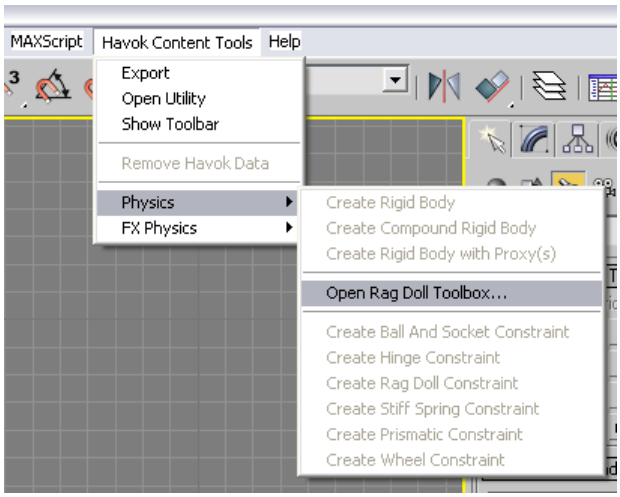
### 4.4.8 Tutorial: Rag Doll Toolbox

The "Rag Doll Toolbox" is a set of tools which facilitates the creation of proxies and the reuse of constraint setups. In this tutorial we will use the Rag Doll Toolbox to set up a (low res) hierarchy of constrained proxy rigid bodies to represent a (high res) hierarchy of bones.

This tutorial consists of three distinct parts, in which we will:

1. Create and associate rigid bodies with an existing skeleton using the Rag Doll Toolbox.
2. Create appropriate constraints between the rigid bodies using the Rag Doll Toolbox.
3. Setup a mapping between the rigid body skeleton (ragdoll) and the original high-res skeleton

To begin, open and examine the Rag Doll Toolbox, via the main **Havok Content Tools** menu or the toolbar icon:



The Rag Doll Toolbox appears as a MAXScript window consisting of three rollouts:

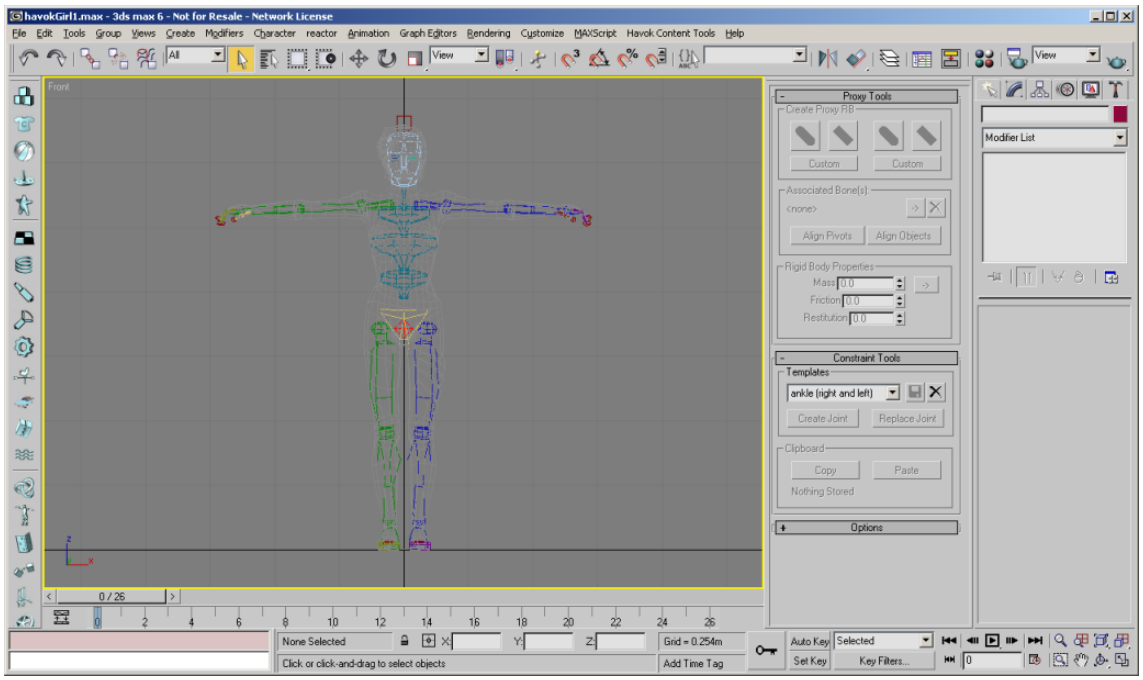
- **Proxy Tools** : Utilities to create proxy (rigid body) representation of animation bones, and associate one to each other.
- **Constraint Tools** : Utilities to create constraints between proxies and reuse constraint data between characters.
- **Options** : General options regarding both Proxy and Constraint Tools.

As well as being a floating window, the Rag Doll Toolbox can also be docked to the sides of the 3ds Max interface. Since you will usually work with one rollout at time (you will usually start by creating all of your proxies, then you will create your constraints), you can also close any rollout to reduce the size of the window.

#### 4.4.8.1 Creating Rigid Bodies

Sample scenes are provided with the Havok Content Tools for this tutorial. These files can be found in "`scenes/havokContentTools/tutorials/ragDollToolbox/`".

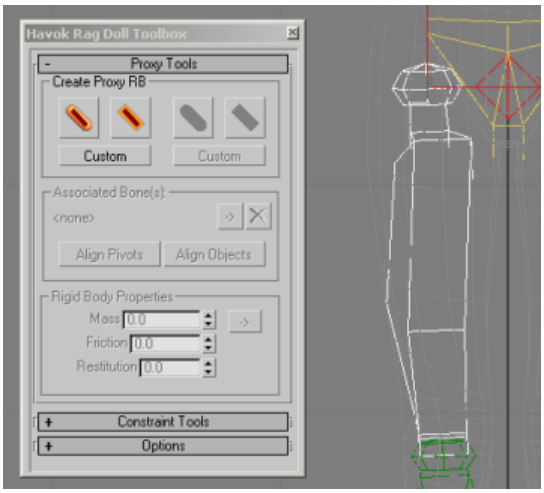
Let's start by loading the scene " **havokGirl1.max** "; this scene contains a biped structure and a mesh which is attached to the biped through the use of a *Physique* modifier:



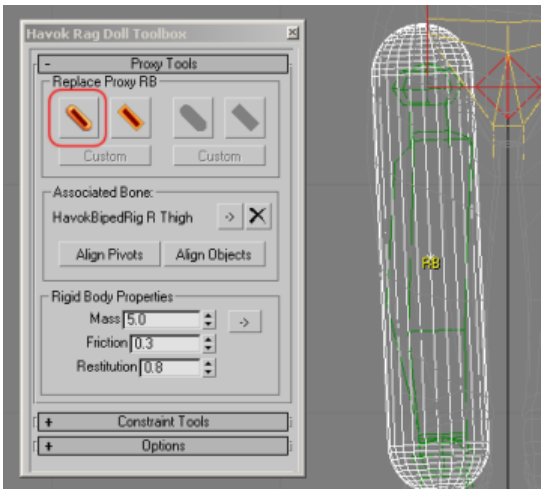

In this section we will focus on creating a rigid body representation of this character (a ragdoll) by creating and associating proxies to the bones of the biped. The **Proxy Tools** rollout of the Rag Doll Toolbox provides the tools to achieve this.

### Simple Proxies

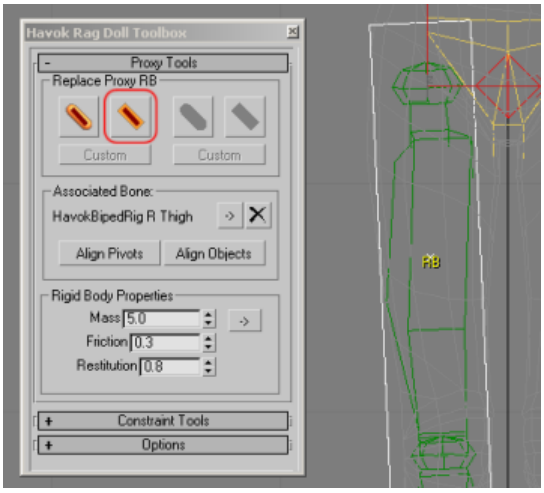
Start, for example, by selecting one of the thigh bones of the character. Notice how two of the buttons, **Create Bounding Capsule** and **Create Bounding Box** , now become enabled:



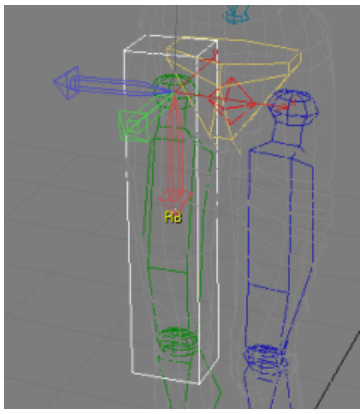

Click the first button to create a new capsule proxy for the thigh bone. A new capsule shape will be created to fit around the bone, with Havok rigid body and shape modifiers already attached, and initialized to a set of default values:




If you would prefer to use a box proxy instead, simply select the newly created capsule proxy and click the second of the two enabled buttons. This will replace the proxy with a box version, while retaining any rigid body properties you may have changed:



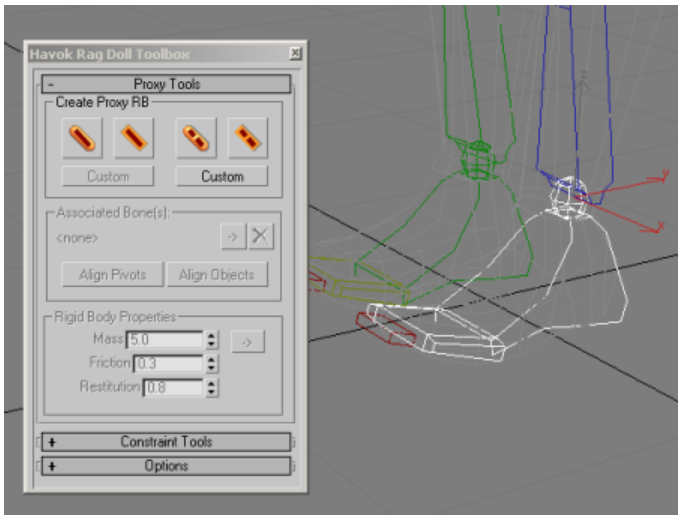

Observe that in each case above, the new proxy is created unparented (so as not to pollute the skeleton hierarchy). Observe also that the rotate pivot point of a new proxy automatically becomes aligned to that of the bone:



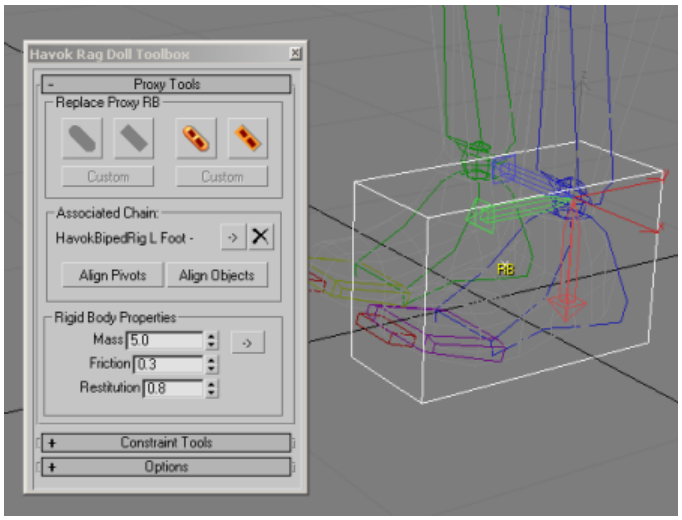

Continue creating simple proxies for each simple bone in the biped: thighs, calves, upper arm, hand, pelvis, head.

### Chain Proxies

Suppose that you would like to create a single proxy to represent a chain of bones, for example the feet in this example. Select the two bones representing a foot (holding **CTRL** for multiple selection). When a valid chain of bones is selected, the second set of buttons in the **Create Proxy RB** section of the toolbox becomes enabled, allowing chain proxies to be created:




Click one of the buttons to create a capsule/box proxy for the selected bones:



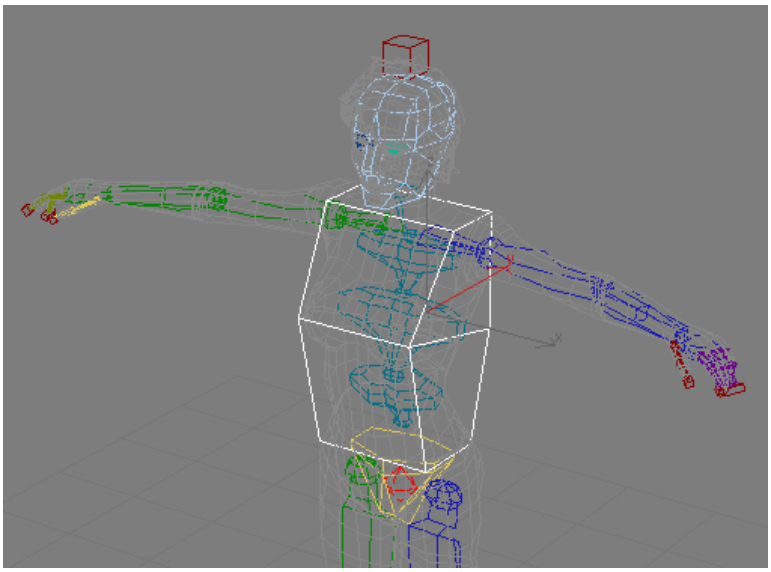

The only different between proxies created from a chain and proxies created from a single bone is that chain proxies keep a reference to both the start and end of the chain (while single proxies keep a single reference to a bone). Pivots of chain proxies are aligned to the start of the chain (i.e. the ankle in this case). For any other purposes, chain proxies operate the same way as single proxies.

Continue creating chain proxies for the other foot, and for the two forearm bones in each arm.

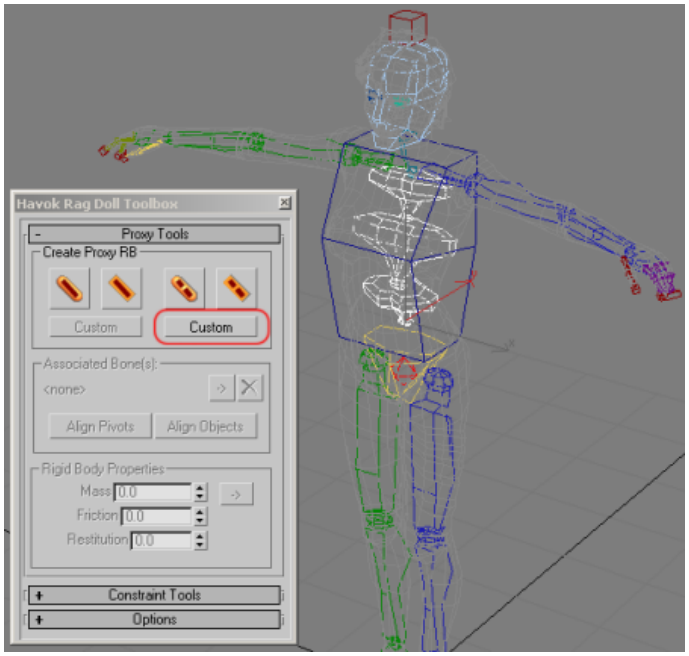
**Custom Proxies**

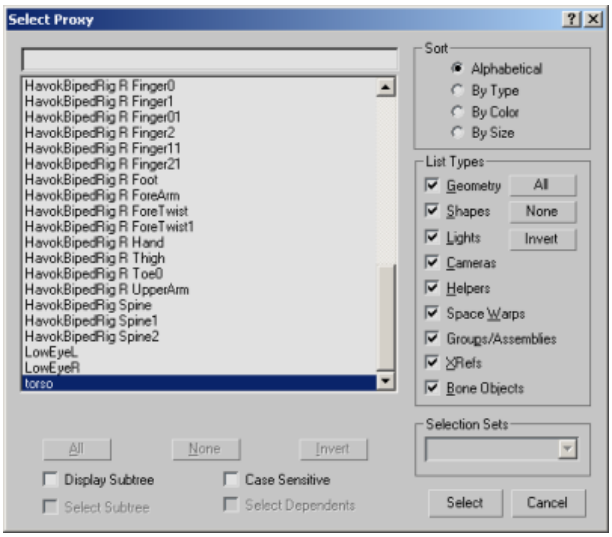
Sometimes a box or a capsule is not enough to accurately represent the volume around a bone or a chain of bones. For example, take the torso region of this model. In this case, instead of creating a box/capsule using the toolbox, we will first create a custom mesh and then associate it with the spine bones.

Create a new (convex) polygon mesh in world space to represent the character's torso, as shown, using any of the standard 3ds Max tools:



To associate this mesh as a chain proxy for the spine bones, select the three spine bones (holding 'ctrl' for multiple selection), then click the **Custom** button in the toolbox. This will open a **Select By Name** dialog where you can pick the existing mesh to use as a proxy. Select your custom mesh and click **Select** :




Once the custom mesh has been selected, it becomes associated as a proxy for the set of spine bones. From this point it behaves like any other chain proxy, i.e. it inherits default Havok properties, it can be replaced by clicking the other chain proxy buttons, its pivot is aligned to the most senior of the associated bones, etc.

Note that the "Shape Type" property of the new proxy's shape modifier should be set to ' **Convex Hull** ', to accurately represent a mesh such as this. Verify that this is the case.

**Reshaping/resizing Proxies**

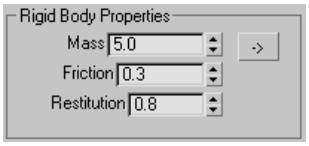
After a proxy has been created, it may not accurately represent the skin volume. To correct this, the proxy can be moved/rotated/etc. using the standard tools to achieve the desired fit, with one caveat: **non-uniform scale should be avoided** . The reason for this is that the proxy meshes used in the modeler act as guides for creating the rigid body shapes (based on the 'Shape Type' attributes). If any non-uniform scale is introduced into the proxy meshes then the exported rigid bodies may not accurately match the proxies displayed in the modeler. This is true for capsules in particular.

Therefore the preferred method of editing a proxy is to use the modify panel to change the construction attributes (width/height/etc.) and use the translate/rotate tools to position the proxy.

After moving or rotating a proxy, the **Align Pivots** button (in the 'Associated bone' section of the toolbox) **should always be pressed** to ensure that the proxy pivot matches the joint position. This helps to create accurately mapped ragdolls during the filter process.

**Setting Rigid Body Properties**

The **Rigid Body Properties** section of the toolbox allows quick access to rigid body properties of selected proxies. As you create proxies, think about these properties and try to set appropriate values for each proxy:




These properties can also be applied in bulk, for example to set the friction property for all of your proxies you can select them all together and input some value into the **Friction** field above.

**Finishing with rigid bodies**

Using the techniques above, you should have created *simple* proxies for the following bones:

- Head
- Upper Arms
- Hands
- Pelvis
- Thighs
- Calves

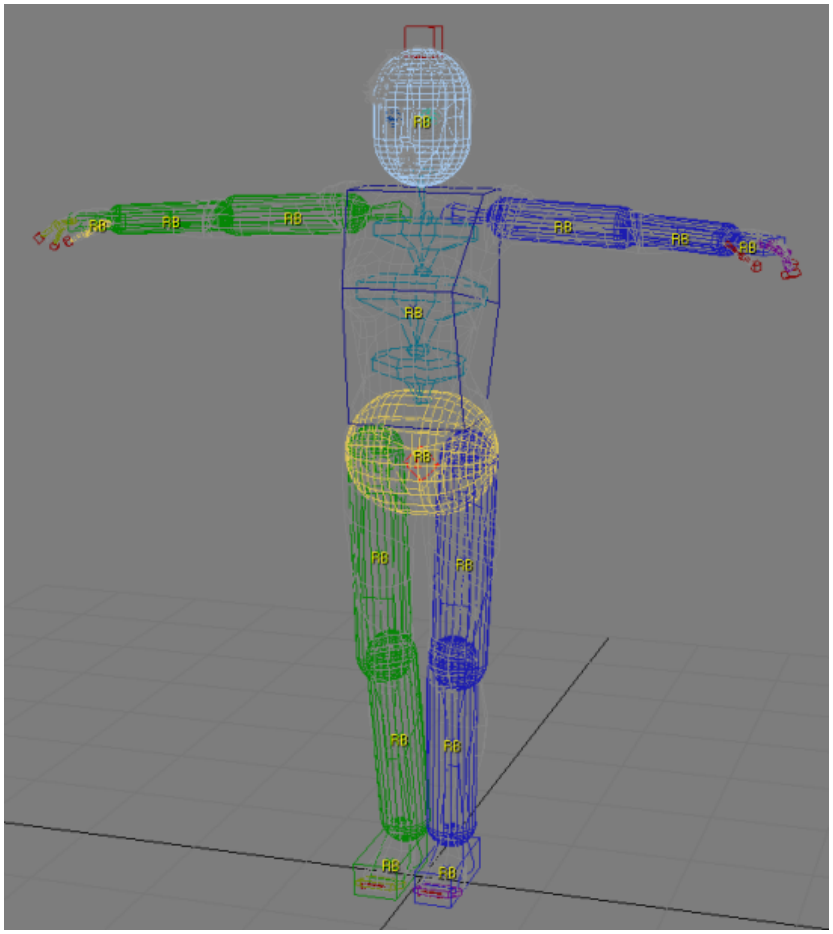
and you should have created *chain* proxies for the following bones:

- Feet
- Forearms

as well as a *custom* proxy for the torso.

The pivot of each proxy should be aligned to the associated bone(s), and appropriate rigid body properties should be specified.

You may wish to open the sample "`havokGirl2.max`" file; this contains the original skinned biped with a suitable set of rigid bodies created and associated to the bones:



#### 4.4.8.2 Creating Constraints

After creating the proxies, we are left with a flat collection of rigid bodies (they are not parented to each other, or connected in any way). To finish setting up the ragdoll, we need to organize the rigid bodies into a constrained hierarchy. The **Constraint Tools** rollout in the Rag Doll Toolbox gives us the tools to achieve this.

**Templates**

Templates are representations of Havok constraints, saved as text files. Each represents a particular type of constraint, plus information about its spaces, limits, etc. so that it can be reused between different characters. For example, an "elbow" template might describe a hinge constraint with certain limits and spatial orientation.

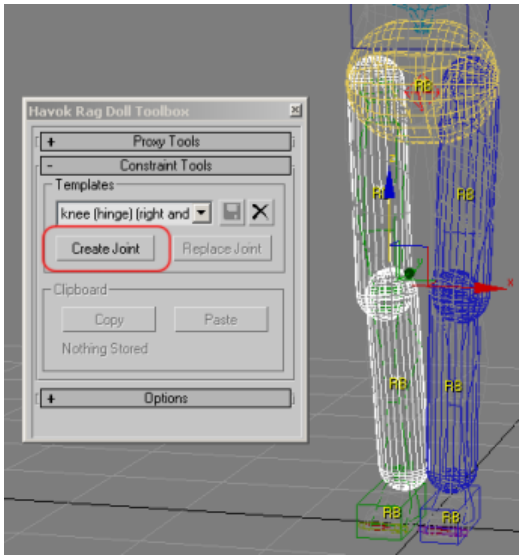
Templates are stored together in an specific directory. Although a default set of templates (`defaultHuman`) is provided, in most cases you will be creating your own templates to match the structure of your characters.

You can change the current location of the templates folder at any time from within the **Options** rollout of the Rag Doll Toolbox.

When working with templates, some assumptions need to be made regarding the local axis of the bones. In particular, the main axis (an axis pointing towards the length of the bone, in the "outside" direction) and the bend axis (an axis around which a positive rotation (using the right hand rule) causes a bending of the joint) need to be defined. By default these axis are defined as +X and –Z, since this is the convention used by Character Studio. You can change the convention used by your characters in the **Options** rollout if necessary.

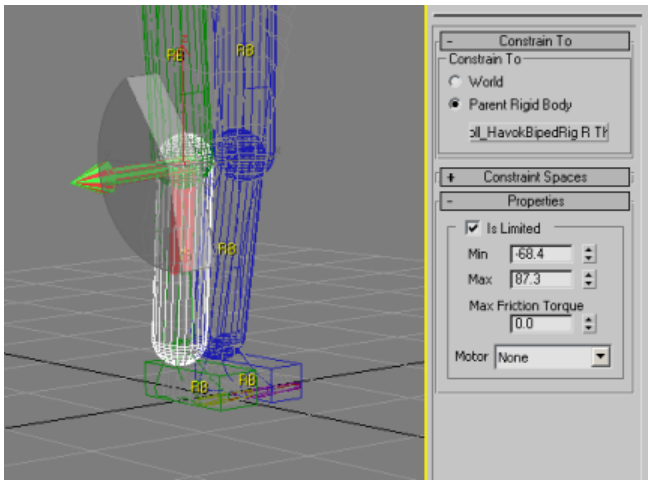
### Applying Templates

To apply a template, start by selecting two proxies in the viewports. For example, select the left thigh and calf. In the templates combo box, select "`Knee (Hinge) (Right and Left)`" and then click on " **Create Joint** ". (You may prefer to use a 'Ragdoll' version of the constraint, which is also provided):




Several things happen at this stage:

- The two proxies become parented to each other; in this case the calf is parented to the thigh
- A constraint (a hinge constraint modifier) is added to the child proxy (the calf)
- The properties of the constraint are set from those stored in the template




### Modifing Constraints

Once a template is applied, you can always modify its properties by changing the constraint parameters through the Modify panel (as shown above). You can then save (and possibly replace) the constraint as a new template by clicking on the " **Save** " icon beside the template combo box.

### Replacing Constraints

After a template has been applied, it is possible to replace the applied constraint with a new template. In this case however, you must only select the child object (the one with the constraint modifier), instead of the pair, and then press the " **Replace Joint** " button. No re-parenting of proxies is done in this case.

### Copy and Paste

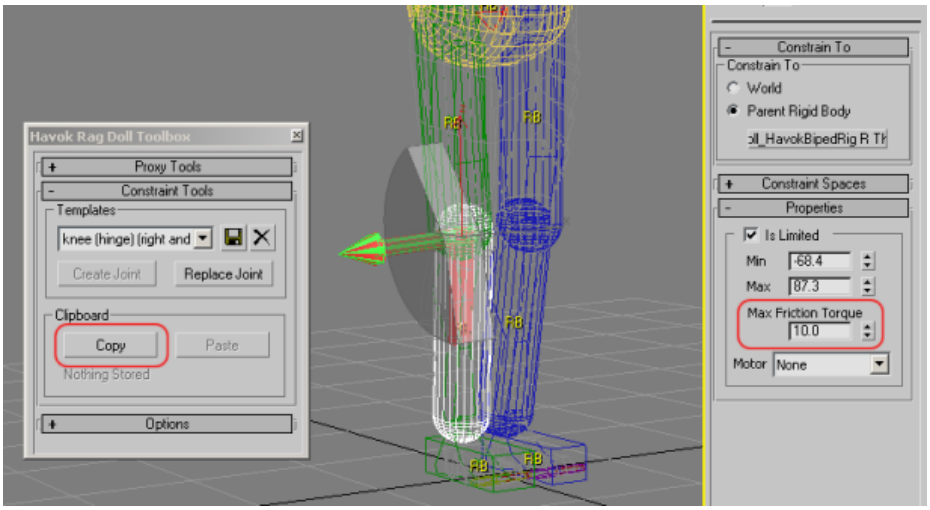
Symmetric characters usually have similar constraint properties on their left and right side. The copy and paste mechanism of the toolbox is useful in this case.

With an already constrained proxy selected, press the " **Copy** " button to store its constraint as a template in memory. The constraint can then be applied to:

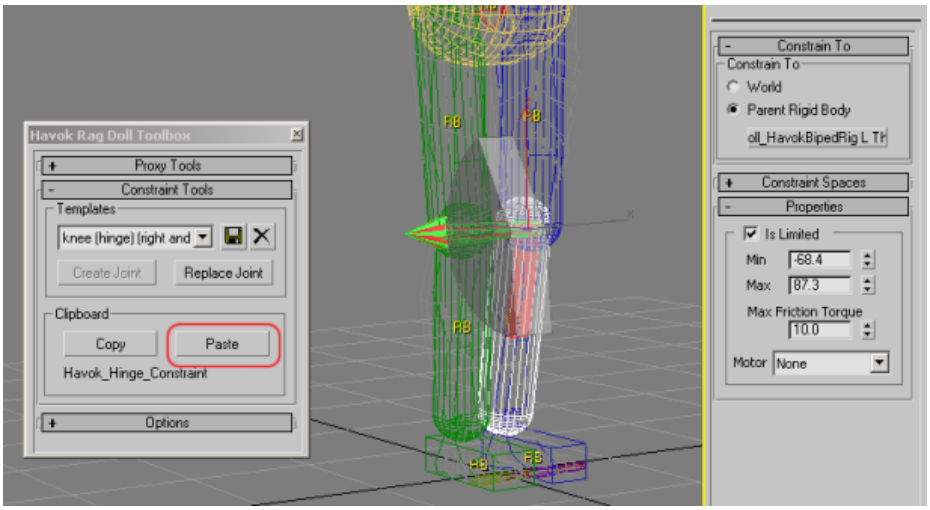
- A pair of unconstrained proxies, by selecting the pair and pasting. This behaves in the same way as creating a new joint.
- An already constrained proxy, by selecting the proxy and pasting. This behaves in the same way as replacing an existing joint.

For example, let's say we have modified the left knee friction torque, and we now want to apply the same constraint with the same values to the other knee:

- Select the calf, and change the **Max Friction Torque** value of the constraint to 10.
- Click on **Copy** in the Constraint Tools Clipboard.

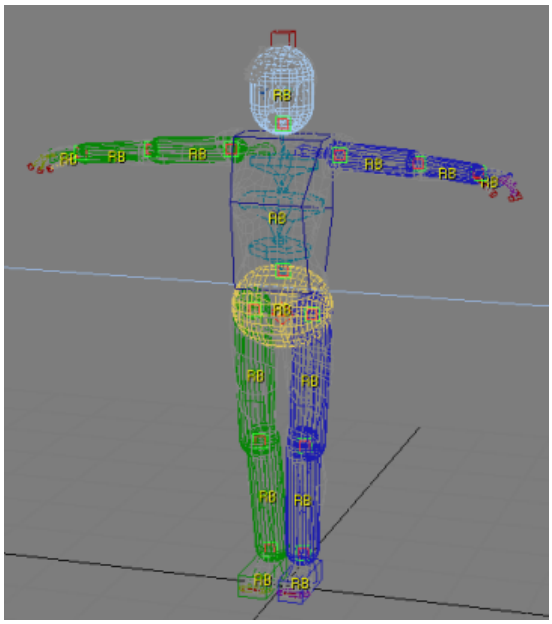



- Select the opposite thigh and calf together in the viewport.
- Click on **Paste** in the Constraint Tools Clipboard.




### Finishing with Constraints

Continue applying templates to each pair of proxies, using the default templates provided (feel free to modify any parameters). You may wish to open the sample "`havokGirl3.max`" file; this contains the final skin, skeleton, and the constrained proxy hierarchy.

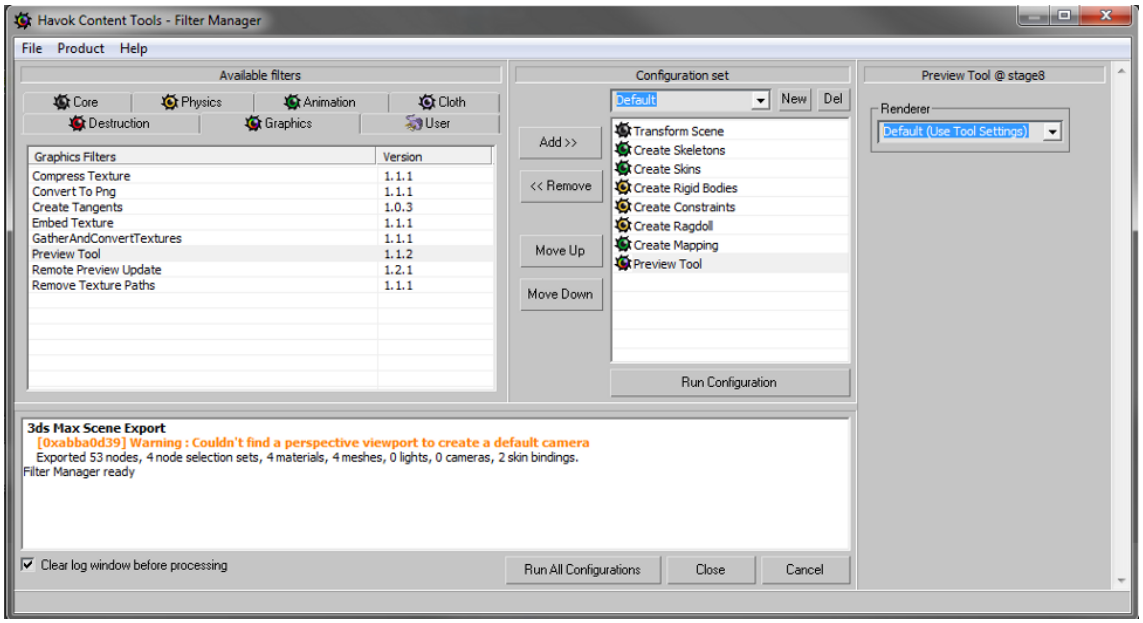


#### 4.4.8.3 Processing the Scene

Now that we have the ragdoll set up to match the biped in the scene, we want to process so that:

- The original skeleton and skin are generated.
- The rigid bodies and constraints for the ragdoll are generated.
- A "ragdoll instance" and skeleton for the ragdoll are generated.
- A pose mapper between ragdoll skeleton and original skeleton is generated (so the ragdoll can eventually drive the skin).
- We can preview it.

Export the scene to the filter manager using the Havok exporter. The following filter setup is required to process the ragdoll (this setup is included with the "`havokGirl3.max`" file):




- **Transform Scene**

  3ds Max stores the scene internally in inches (by default) and Havok works in meters. Using the `SCALE_INCHES_TO_METERS` preset performs the necessary scaling.

- **Create Skeleton**

  We want to create a skeleton for the bones of the character (so we can skin it). The Create Skeleton filter will detect the skinned skeleton and, with the " **From Skin Bindings** " option, will create a (36-bone) skeleton for us. Switch off the " **Move Reference pose to Origin** " as we want the skeleton to remain where it is in the scene.

- **Create Skin**

  This simple filter will create a skin object and associate it with the skeleton we just created.

- **Create Rigid Bodies**

  This filter has no options; it will detect the rigid bodies and shapes in the scene and create the actual rigid body objects for the simulation.

- **Create Constraints**

  As with the Create Rigid Bodies filter this has no options; it will create actual physical constraints based on the setup we defined in the scene.

- **Create Rag Doll**

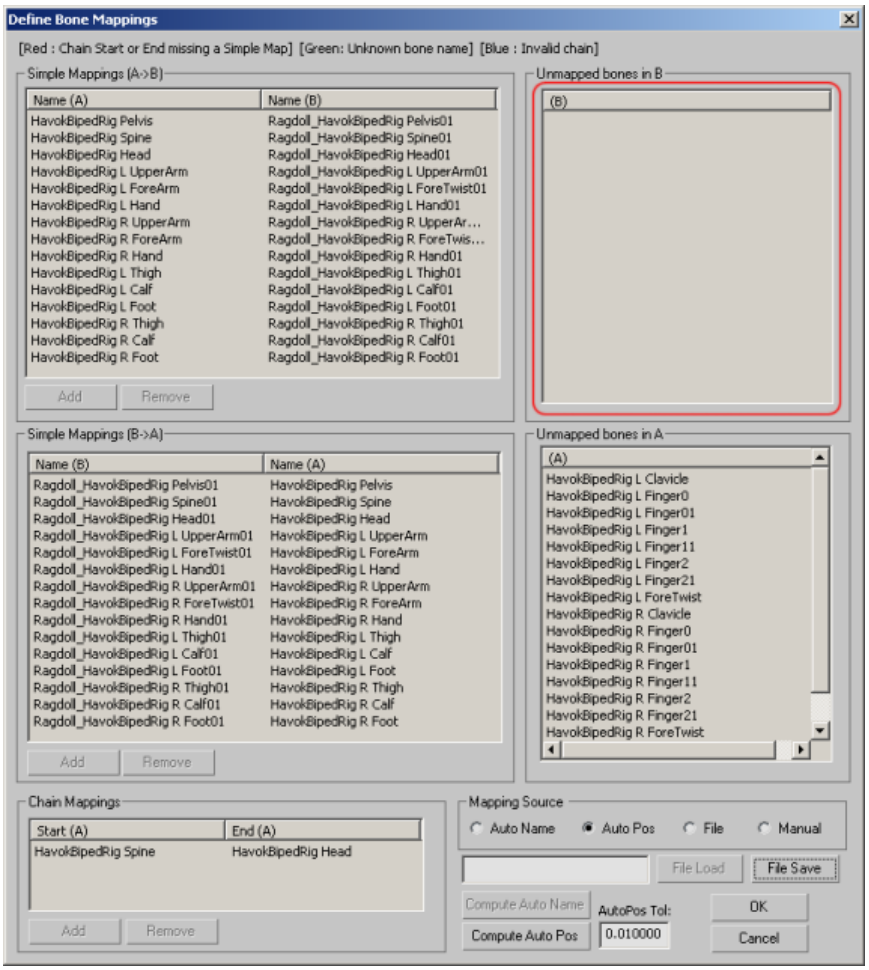
  This filter creates a Ragdoll runtime object (an `hkRagdoll` instance), which contains the rigid bodies, constraints and skeleton associated with a ragdoll. Choose ' **Automatically** ' in the ' **Detect Rag Dolls** ' section, which works fine in this case as there is only one ragdoll in the scene.

- **Create Mapping**

  This filters sets up skeleton mapper information that allows the transformation of poses between two different skeletons (in our case, the original biped skeleton and the proxy rigid body skeleton).

  For ' **Skeleton A** ', pick 'HavokBipedRig Pelvis' (the skeleton root). For ' **Skeleton B** ', pick 'Ragdoll_HavokBipedRig Pelvis01' (the ragdoll root).

  Click on ' **Define Mappings** ' to verify the mapping:




For a successful mapping, all of the bones in one skeleton should be mapped to (some of) the bones in the other skeleton. In this case we can see that there are no unmapped bones in the 'B' skeleton (the ragdoll) so this is a successful mapping.

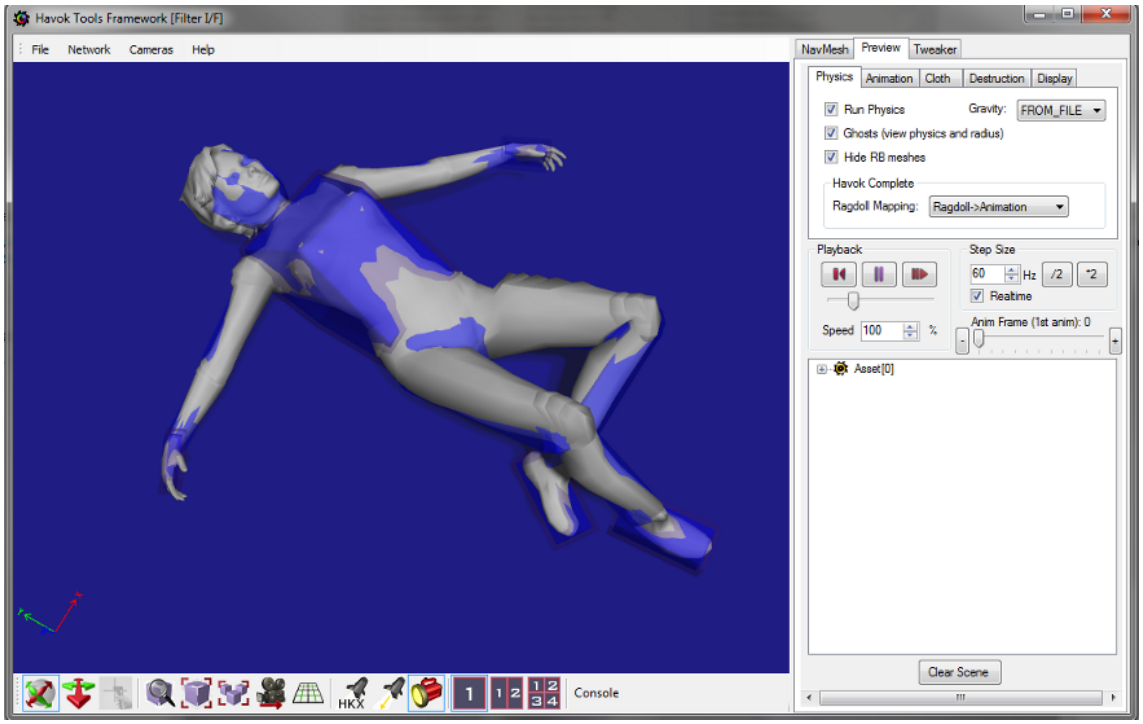
The default behaviour of the mapping filter is to match the skeletons by comparing the positions of the bones in each. This is why it was important to ensure that the pivot points of the rigid bodies were aligned to the pivot points of the joints. If you have any unmapped bones in your skeleton, return to the modeler and ensure that each proxy's pivot is aligned to that of its associated bone(s). Also ensure that ' **Auto Pos** ' is selected as the **Mapping Source** in the **Define Mappings** dialog.

Click ' **OK** '.

- **Preview Tool**

  So that we can interactively test the ragdoll behavior.

Run the filter setup above. When the preview window appears, play the physics and interact with the ragdoll using the space bar to pick and drag the rigid body shapes:




If the ragdoll does not behave as expected, close the filter manager and review the rigid bodies and constraints. Tweak the attributes if necessary and continue to test using the filter manager. Try appling ragdoll constraints instead of hinge constraints for the elbows/knees. Also try varying the rigid body masses, the constraint's **Max Friction Torque** parameter and the angular limits of the constraints and observe the results in the previewer.

## 4.5 Maya Tools

### 4.5.1 Introduction

In this chapter we are going to present the Maya specific components of the Havok Content Tools. These include:

- The Maya Scene Exporter
- The Maya Physics Tools
- The Maya Animation Tools

We also present several tutorials:

- Export and Animation Basics: How to set up, export and process an animation.
- Physics Basics: How to set up, export and process a physics scene.
- More On Rigid Bodies: A few more concepts about rigid body setup.

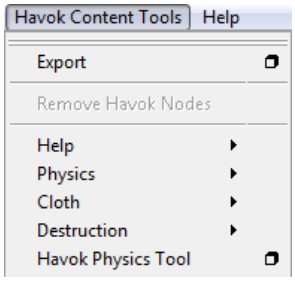
- Rag Doll Setup: How to set up, export and process a rag doll and its association with a high-res skeleton.

Check also:

- Extending the Maya Tools: For information about exporting custom data and MEL/C++ access.
- Common Concepts: For common concepts useful for all Havok Tools (regardless of modeler).
- The Havok Filter Pipeline: For common concepts used in processing exported data (regardless of modeler).

#### 4.5.1.1 Loading the Plugins

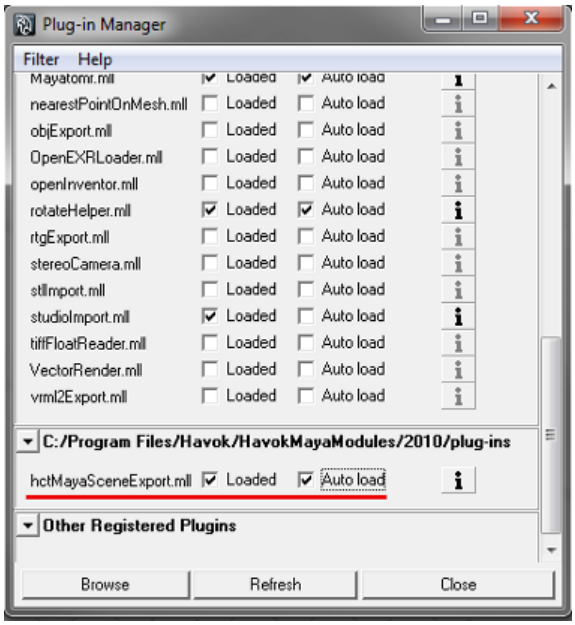
Currently, the Havok Content Tools for Maya are provided in the form of two plugins - one for the Scene Exporter and one for the Physics, Cloth and Destruction tools. Both of these plugins need to be loaded for the Havok Tools to be useful. If the plugins are loaded then both a **Havok Content Tools** menu and a Havok shelf should be present:



If any of these elements are missing from the Maya UI then it is likely that either one or both of the plugins are not loaded correctly. If they are present then you may skip the rest of this section.

**Note:**
Depending on the order in which the exporter plugin and the physics plugin are loaded, the order of items in the main menu and the Havok shelf tab may appear different to the images above and throughout this documentation. There is no requirement for either plugin to be loaded before the other, just that both are loaded.

The Havok Content Tools installer installs the Maya plugins, scripts and other files to a 'module' folder, normally under "`%ProgramFiles%\Havok\HavokMayaModules\<x.x>`". It also instructs Maya to look for plugins under this path, by creating a special 'pointer' file at "`%CommonProgramFiles%\Alias Shared\modules\maya\<x.x>\HavokContentTools.txt`": this is a simple text file which specifies the path to the module folder. If these files are set up correctly, then the plugin ("`hctMayaSceneExport.mll`") will appear in Maya's **Windows** > **Settting/Preferences** > **Plugin Manager...** :




If the plugins appear in the plugin mananger window, ensure that the ‘ **loaded** ’ checkbox is checked for each Havok plugin, and also that the ‘ **auto load’** checkbox is checked to avoid having to load the plugins manually in the future.

If the plugins do not appear in this window then exit Maya, re-install the Havok Content Tools, and re-open Maya.

If they still do not appear, Maya is probably not finding the installed module ‘pointer’ file. Run the following MEL command within Maya: "`getenv MAYA_MODULE_PATH`" - this gives a list of paths where Maya is looking for module files. Locate the "`HavokContentTools.txt`" file, and copy it to the one of these module paths. Restart Maya again and look for the plugins.

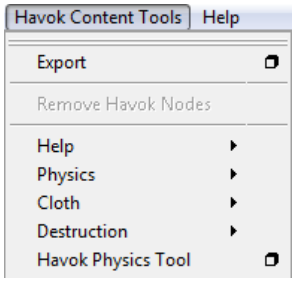
If still not found, then refer to your Maya documentation and environment variables to figure out where the module should be placed.

### 4.5.2 Maya: Scene Exporter

The Maya Scene Exporter is responsible for collecting scene information from Maya’s scene graph and passing it to the filter pipeline for processing.

#### 4.5.2.1 Accessing the Scene Exporter

The Scene Exporter is implemented as a custom Maya command, with a set of optional arguments to configure its behaviour. The best way to access this command and its options is through the **Havok Content Tools** menu and shelf:



Most of the items in the menu and shelf relate to the Physics Tools and are described in their own section. Let's focus only on the options related to the Scene Exporter:

- **Export**

  When this command is invoked the Scene Exporter will navigate the current scene graph, extracting information about meshes, lights, nodes, attributes, etc., according to the current export options. Once this is complete, the Havok Filter Manager will be invoked so that this data can be processed further.

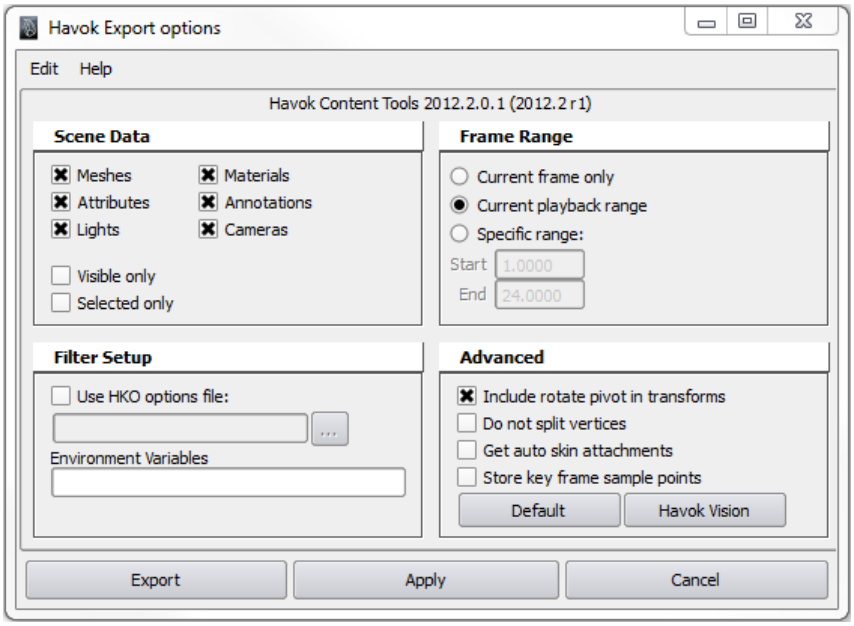
- **Export Options** (the little box beside the 'Export' menu item)

  This command will open the Havok Export Options window, where various export options can be defined. These options are described next.

- **Quick (Batch) Export** : This invokes the filter manager in batch mode - the filter configurations in the filer manager will be executed without opening the filter manager UI. This can be very useful when you just want to reexport an asset (either to a file or to a remote preview) once the filter pipeline has already been setup

#### 4.5.2.2 Export Options

The export options UI is displayed by pressing the little box to the right of the **Havok Content Tools -> Export** menu item, or the toolbar button.




The export options are divided into several sections (with default values indicated in the screenshot above):

- **Scene Data**

  This section defines which nodes in the Maya scene should be considered in the export process:

  - **Meshes** : Controls whether meshes are exported
  - **Materials** : Controls whether materials and textures are exported
  - **Attributes** : Controls whether custom Havok attributes are exported - note that some Physics Filters depend on these attributes
  - **Annotations** : Controls whether annotations are exported
  - **Lights** : Controls whether lights are exported
  - **Cameras** : Controls whether cameras are exported

  - **Visible only** : If enabled, only nodes which are currently visible in Maya are exported
  - **Selected only** : If enabled, only nodes which are currently selected in Maya (along with all their ancestors) are exported

- **Frame Range**

  This section defines the range over which animated data (node transforms, attributes, annotations) is sampled:

  - **Current Frame Only** : The exporter will sample the scene at the current frame only (so no animation data will be exported). This option can considerably speed up the export process if no animation is required.
  - **Current Playback Range** : Animated data will be exported, with the data sampled over the current playback range in Maya. This is the default option.
  - **Specific Range** : Allows you to specify an frame range (with the **Start** and **End** edit boxes) over which the exporter should sample animated data. This is useful if you want to control the range of the animation export regardless of the current range in Maya.

- **Filter Set**

  This section provides the ability to customize some elements of the filter manager:

  - **Use HKO file** : It is possible to specify an options (HKO) file for the filter manager to use (check the Configurations section for details). Filter configurations are always saved with the asset (the Maya scene), but sometimes it is useful to specify a custom set of options explicitly (for example, during batch processing, or to avoid having to build a filter set up from scratch in new assets). Enabling the check box allows you to choose an HKO file which contains your desired filter configuration(s).
  - **Environment Variables** : Here you can specify pairs of (variable, value) by concatenating strings of the form: **VARIABLE1=VALUE1; VARIABLE2=VALUE2;** etc. This data is not usually stored with the final asset but can be used by a (custom) filter during process. For example, the Write to Platform filter will replace any environment variables specified in its **Filename** field by their values. For more, see the section on environment variables.

- **Advanced**

  This section provides options which should not normally be changed:

  - **Include Rotate Pivot in Transforms**

    Transforms in Maya do not include the pivot point as part of the transform. However it is desirable for the Havok tools and exporter to do so. With this option enabled, exported nodes will use transforms which include the **rotate pivot** [2] of the Maya nodes.

    The Havok Physics Tools for Maya also use this option. In particular: with this option enabled constraint spaces are considered to be relative to the rotate pivot points of their transform nodes; with this option disabled, constraint spaces are considered relative to the usual Maya transform (normally the center an object).

    This option is enabled by default - this is the recommended setting.
  - **Do Not Split Vertices**

    By default, the Maya Scene Exporter will export as many mesh vertices as required in order to store texture and normals. So, for example, while for a cube only 8 geometric vertices would be required, in order to store per-face normal and texture coordinates, 24 vertices need to be exported.

    Selecting this option will override the default behaviour, and multiple vertices that have the same position will be exported as one, regardless of their normal or UV coordinates. This may reduce the size of the exported mesh - but due to the lack of accurate normal and texture coordinates, the mesh may not display as it originally did in the modeler.
  - **Get Auto Skin Attachments**

    By default, skin attachments are not applied to meshes that do not have any bones in there parent tree, and they are instead exported as plain meshes. Selecting this option forces meshes to get auto skin attachments if there parent tree contains any bones at all. Must be selected if using the Process Vision Data filter to export animations.
  - **Keyframe Sample Points**

    Selecting this option will process the animation in a format that is understandable by Havok Vision, by including keyframe data in the export pipeline that Havok Vision uses to generate animation files. Must be selected if using the Process Vision Data filter to export animations.
  - **Preset Options**

    This control can set the advanced export options to be compatible with **Havok Vision** , or the **Default** Havok settings.

[2] Objects in Maya have both a scale pivot and a rotate pivot, which are normally located in the same place. We only consider the rotate pivot.

The export options window also provides several buttons. ‘ **Export** ’ begins the export process using the current options. ‘ **Apply** ’ closes the options window and saves any changes made. ‘ **Cancel** ’ closes the options window, discarding any changes.

### 4.5.3 Maya: Physics Tools

#### 4.5.3.1 Introduction

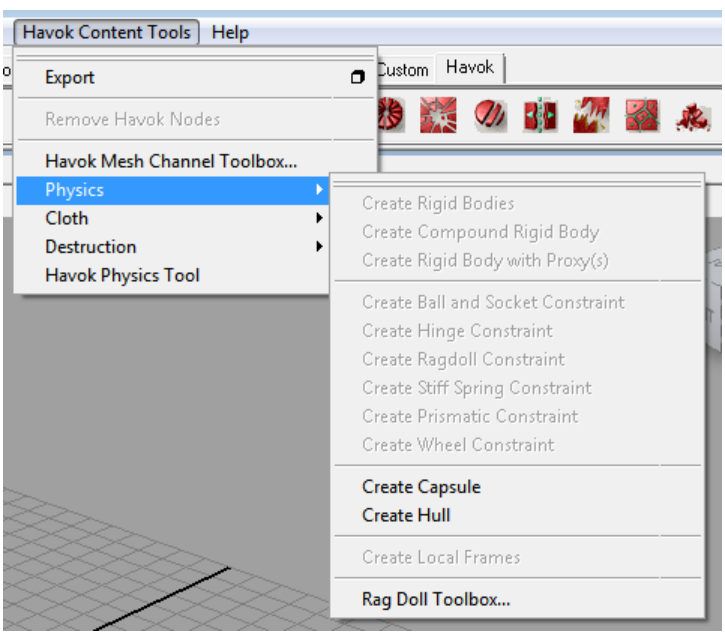
All modeling packages, including Maya, have the ability to specify scene and animation data that has a close relationship to the equivalent runtime objects in Havok (and provide many sophisticated tools to manipulate that data). However, for physical set up the tools available in the modeler are limited and do not necessarily match the Havok runtime features. Moreover, as the construction of complex physical systems (such as rag dolls) can be a complex task, providing tools that automate and facilitate these and another operations becomes more and more important.

It is for these reasons that the Havok Content Tools include special Maya tools (nodes, scripts, etc.) for setting up physical information in the Maya scene. In the following sections we will explore those tools.

Advanced users should first read the Common Concepts chapter for in-depth details on some of the rigid body and constraint concepts.

**Accessing the Physics Tools**

The physics tools for Maya can be accessed through the **Havok Content Tools** > **Physics** menu or the Havok shelf:



**Havok Nodes**

Most of the Havok physics data in a scene is added through the use of custom nodes, attached as children of an object's transform node. These custom node types are registered with Maya by the Havok Physics plugin. For reference, the main nodes used by the Havok Physics Tools for Maya are:

- Rigid Body Node: Adds rigid body (dynamics) information to an object
- Shape Node: Adds shape (collision detection) information to an object
- FX Rigid Body Node: Adds FX rigid body (dynamics) information to an object
- Constraint Nodes (Ball and Socket, Hinge, Ragdoll): Add constraint information to an object

The Havok Content Tools also provide some other custom nodes, generally for storing options and some internal data. These node types are recognizable by the common '`hkNode`' prefix.

While it is possible to manually add those nodes to an object (through scripting, via the outliner panel, etc.), it is highly recommended that you use the Havok Content Tools menus, scripts and toolbar to do so as doing otherwise may leave your scene in an inconsistent state.

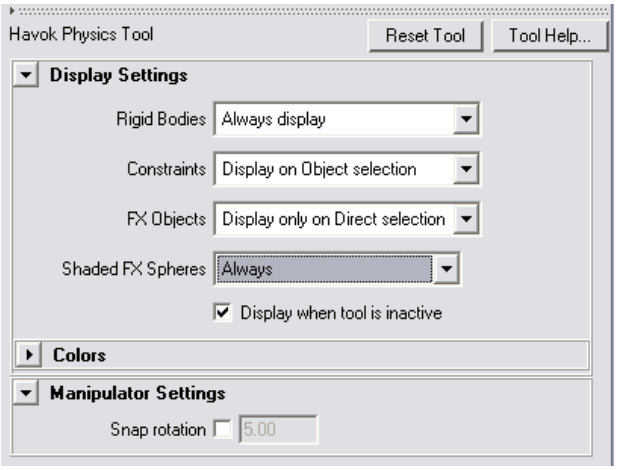
You may remove any Havok Physics nodes from selected objects at any time by choosing the **Remove Havok Nodes** menu option.

### The Havok Physics Tool

Some Havok nodes represent themselves in the viewports in various ways. The Havok Physics Tool provides the ability to view and manipulate these Havok nodes.

The tool can be activated by clicking on the **Havok Physics Tool** menu option or shelf button . It is activated automatically whenever a new Havok rigid body / constraint is created, so that the new object appears in the viewports.

While the tool is active, its icon will appear in Maya's toolbox, alongside the usual (select, move, rotate, scale, manipulate) tools. The tool also becomes mapped to a shortcut key (' **Y** ' by default). You can access the tool properties by double-clicking on the Havok Physics Tool icon, or by clicking the little box beside the Havok Physics Tool menu item. This shows the following UI:



Rigid bodies and constraints are usually only drawn while this tool is active[3]. The **Display Settings** rollout provides further display options for any Havok rigid bodies and constraints in the scene:

- **None** : Do not represent or manipulate such objects in the viewports.
- **Selection** : Represent all such objects in a basic way, and fully represent and allow manipulation of selected objects only (default).
- **All** : Represent all such objects fully in the viewports, and allow manipulation of selected objects.

The **Colors** rollout allows the user to customize the colors used to draw the various elements of rigid bodies and constraint. These settings are stored with the application.

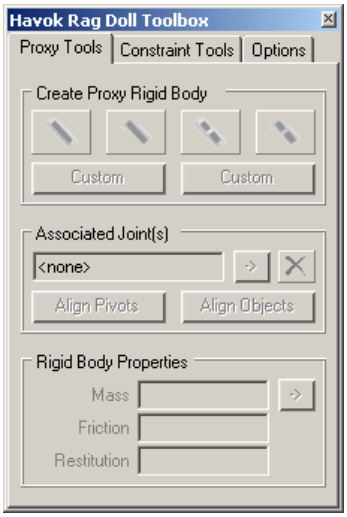
The **Manipulator Settings** rollout allows the user to enable rotation snapping. This setting affects any rotation manipulators in the viewports while this tool is active. This may be useful if you want to set constraint limits to specific angular increments for example.

### Constraint display size

Havok constraints in a scene are drawn in the same way, and to a similar scale, as Maya's manipulators - they have a *screen size* as opposed to a world size. This means that Havok constraints will always appear the same size on-screen regardless of the camera viewpoint. **The global display size (for both manipulators and constraints) can easily be changed at any time using the +/- keys on the keyboard.**

### MEL scripts/procedures

Maya's scripting language, MEL, is used for several components of the Havok Content Tools for Maya. Firstly, it handles much of the UI (menus, toolbars, attribute layouts, etc.). Secondly, it provides the Rag Doll Toolbox. This is a set of scripted controls which is layered upon the core physics tools. The purpose of the toolbox is to simplify the creation of rag doll representations of skeletal structures - it takes care of common tasks such a fitting rigid bodies to bones, aligning pivots, setting up hierarchies, creating constraints, etc.



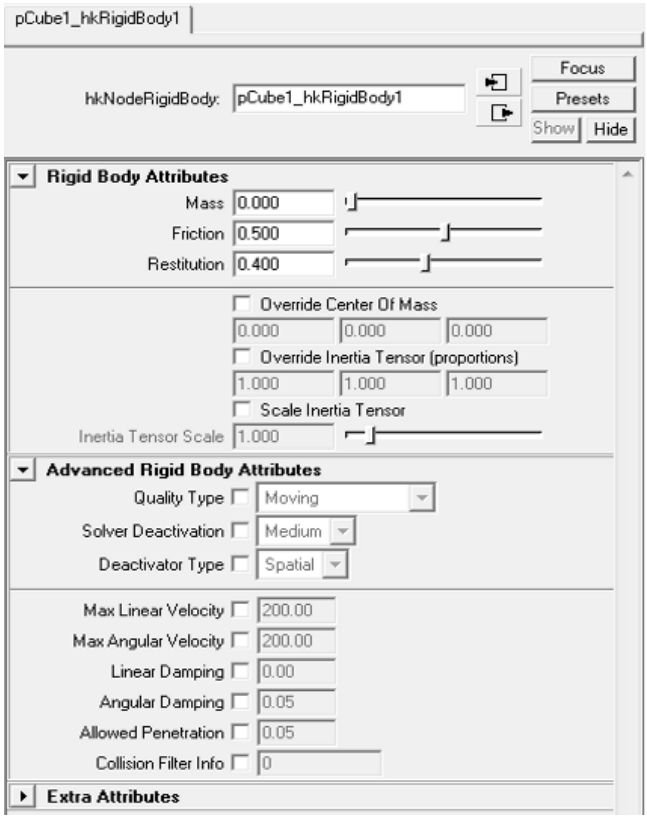

[3] The exception being that Havok nodes are always drawn when directly selected, regardless of whether the Havok Physics Tool is active or not.

#### 4.5.3.2 Rigid Bodies

In this section we are going to describe how to create and manipulate rigid bodies and shapes in Maya. For information about concepts and properties associated with rigid bodies and shapes, please check the Rigid Body Concepts section.

**Rigid Body Node**

A Havok rigid body node defines the root of a rigid body and contains information on the dynamics of the body. Any object which contains a rigid body node will be converted to a runtime rigid body by the Create Bodies filter (or the by the Create Rigid Bodies filter for Physics 2012), provided there is at least one associated shape node.



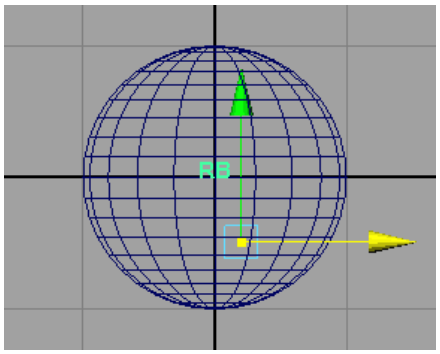

Check the common Rigid Body Properties section for specific details on the attributes listed here.

Note that rigid bodies are represent by ' **RB** ' labels in the viewport whenever the Havok Physics Tool is active. If a rigid body has zero mass, its label changes to ' **[RB]'** to indicate that the body is fixed.

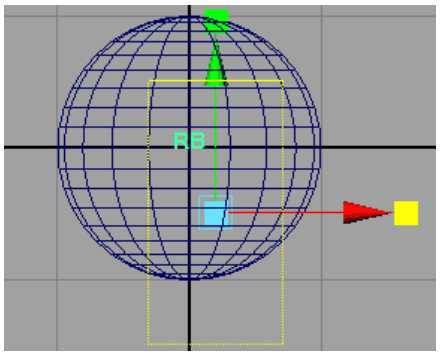
**Overriding the Center of Mass and Inertia Tensor**

By default, the center of mass and inertia tensor of the rigid body will be automatically calculated by the Create Bodies filter based on the rigid body geometry and mass distribution. You can, however, override those values and specify them, either by inputting values directly into the attribute editor, or by manipulating them in the viewports.

Selecting the **Override Center Of Mass** check box will display a point in the viewport, which can be manipulated by activating the Havok Physics Tool and dragging the triad manipulator handles:

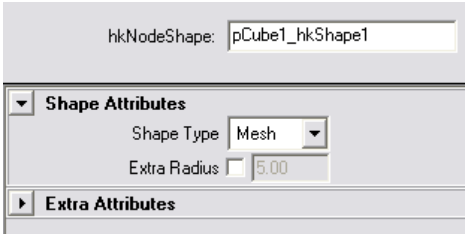



Similarly, selecting the **Override Inertia Tensor Proportions** check box will display a box in the viewport, which can also be manipulated by activating the Havok Physics Tool, in this case by dragging the scale manipulator handles:




### Shape Node

A Havok shape node contains information about how an object's mesh will contribute to the collision detection setup of a rigid body. One or more shape nodes must be associated with a rigid body node in order for the Create Bodies filter to produce a runtime rigid body.




Check the common Shape Properties section for specific details on the attributes listed here.

Whenever a new Havok shape node is created by the Havok Content Tools, its *Shape Type* attribute is automatically determined based on the mesh which it applies to. However it is always useful to verify each shape type attribute to ensure that it matches your requirements.

Shape nodes currently have no representation in the viewports.

**Creating a Simple Rigid Body**

Most of the rigid bodies you will ever want to create will be simple rigid bodies - where both the rigid body and shape information is associated with the same object.

You can create a simple rigid body by selecting an object in Maya and clicking on the **Create Rigid Body** menu option or shelf button.

This will add both a rigid body node and a shape node to the selected object.

**Creating a Compound Rigid Body**

A compound rigid body is a rigid body which has more than one shape (more than one collision detection primitive). In a compound rigid body one object has both rigid body and shape information while one or more other child objects add additional shape information.

You can create a compound rigid body by selecting two or more objects in Maya and clicking on the **Create Compound Rigid Body** menu option or shelf button.

If the selected objects are parented to each other, the topmost object in the hierarchy will automatically hold the rigid body and shape nodes while each of the other objects will hold a shape node. If the objects are not already parented, they will be parented automatically (after user confirmation), by considering the last object selected as the parent of the new hierarchy.

**Creating a Rigid Body with Proxy(s)**

A rigid body with proxy(s) is a rigid body where the object which contains the rigid body node doesn't contain any shape node - the shape nodes are stored in one or more child nodes (the proxies). In that sense, a rigid body with proxy(s) is similar to a compound rigid body, with the difference being that the object which has the rigid body node has no shape node in this case.

You can create a rigid body with proxy(s) by selecting two or more objects in Maya and clicking on the **Create Rigid Body with Proxy(s)** menu option.

If the selected objects are parented to each other, the topmost object in the hierarchy will automatically hold the rigid body node while the rest of the objects will become the proxies and just hold shape nodes. If the objects are not already parented, they will be parented automatically (after user confirmation), by considering the last object selected as the parent of the new hierarchy.

#### 4.5.3.3 Constraints

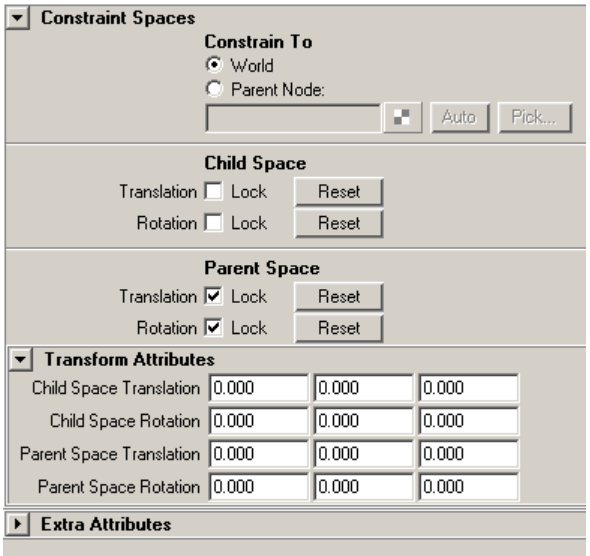
In this section we will present how Havok constraints are created and manipulated in Maya.

**Constraint Nodes**

A Havok constraint node contains information which will be used to create a runtime constraint on export. Since a constraint must act on a rigid body, a constraint node is useful only the object which contains it also contains a rigid body node. If a pair of rigid bodies are to be constrained, a constraint

node is attached to only one of the bodies (the *child*), and it maintains a reference to the other body (the *parent*). Again, please consult the common Constraint Concepts section of this manual for more details on the child and parent spaces.

There are currently three types of Havok constraint node - the specifics of which are described in the common Constraint Types section of this manual. Each of these types provides the same basic constraint attributes, which allow you to specify the child and parent spaces (each constraint type also provides further attributes relevant to that type only):




### 'Constraint To' controls

These controls allow you to choose which other rigid body (if any) is the constraint is attached to:

- **World** : The constraint will attach the child rigid body to a fixed location in space.
- **Parent Rigid Body** : The constraint will attach this (child) rigid body to the (parent) rigid body specified in the text field. The parent can be chosen using either of the following methods:
  - **Auto** : This will set the hierarchical parent of the constraint's transform node to be the constraint parent. This is usually the desired setup.
  - **Pick** : This allows you to choose any suitable rigid body in the scene as the constraint parent, via a popup window.

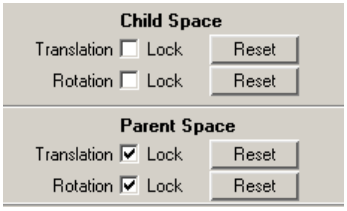
**Note:**
The parent rigid body specified here is in fact the transform node of a rigid body node, not the rigid body node itself. This is to be consistent with Maya's representation of objects as a transform node with of a set of child leaf nodes (mesh / rigid body / shape / etc.).

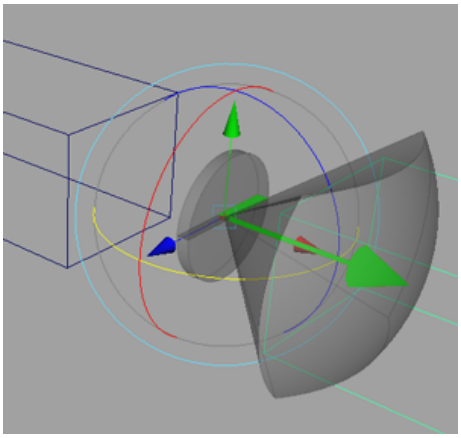
### 'Constraint Space' controls

Constraints can be considered as having two distinct spaces: the child (in the local space) and the parent (in the parent object's space). See the common Constraint Spaces section for more details.

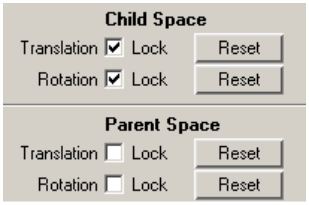
The individual [X,Y,Z] components of each constraint space are always editable, via the 'Transform Attributes' rollout. However it is often easier and more intuitive to manipulate the spaces in the viewport using the Havok Physics Tool.

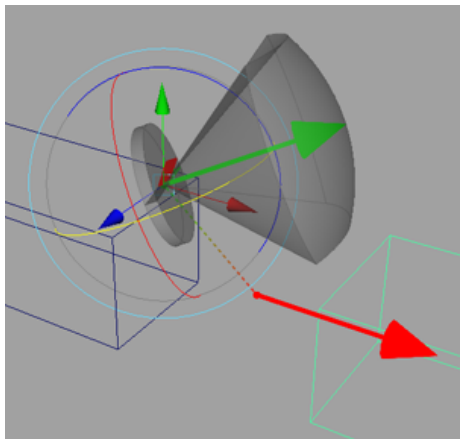
Depending on the state of the **Lock** buttons in the **Constraint Spaces** rollout, various methods of viewport manipulation become available. If the parent space attributes are locked and either of the child space translation or rotation attributes is unlocked, the constraint spaces can be manipulated as a single entity - both the child and parent spaces move together:



If the child space attributes are locked and the either of the parent space translation or rotation attributes is unlocked, the constraint parent space can be manipulated independantly of the child space:





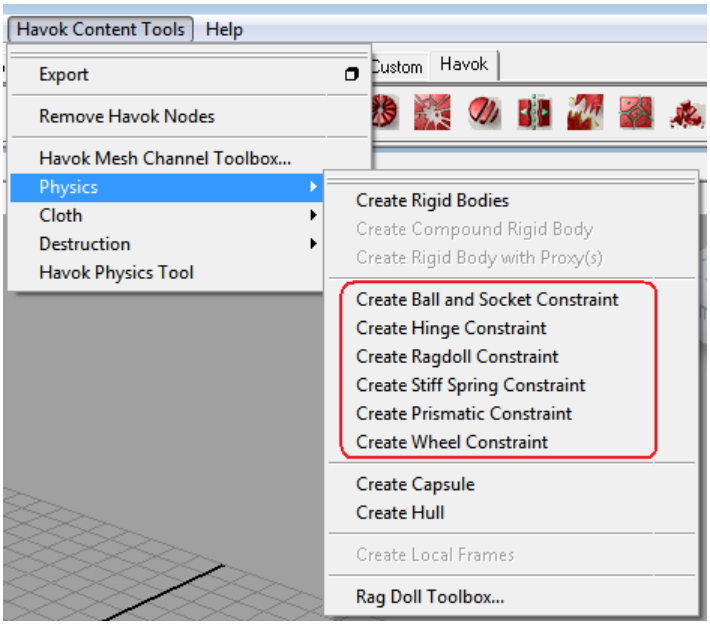
In all cases, the **Reset** buttons return a constraint space attribute to its initial setting, moving the constraint space to the pivot point of the appropriate object.

**Note:**

If all of the child and parent space attributes are unlocked, then it is possible to fully manipulate both spaces independantly and at the same time. However this can become difficult since you now have four manipulators which can be difficult to distinguish from each other. It is best to unlock a space only when you wish to alter it, and lock it again once you are satisfied with the change. Of course, if you want the parent space to move with the parent object in the modeler, then the parent space must always remain unlocked.

### Creating a Constraint

You can create a constraint by using either the **Havok Content Tools** menu or shelf buttons:



Either one or two rigid body objects must be selected before creating a constraint. Then, depending on

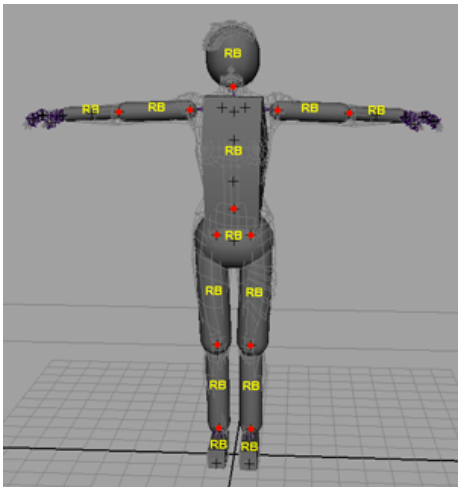
the number of objects selected:

- If one object is selected, a constraint node will be attached to that object. If the object's hierarchical parent is also a rigid body then that object will be set as the constraint parent; otherwise the rigid body will be constrained to the world.
- If two objects are selected and one is a descendant of the other, a constraint node will be attached to the descendant and the ancestor will set as the constraint parent. If the objects are not explicitly parented, a constraint node will be attached to the first rigid body and the second will be set as the constraint parent.

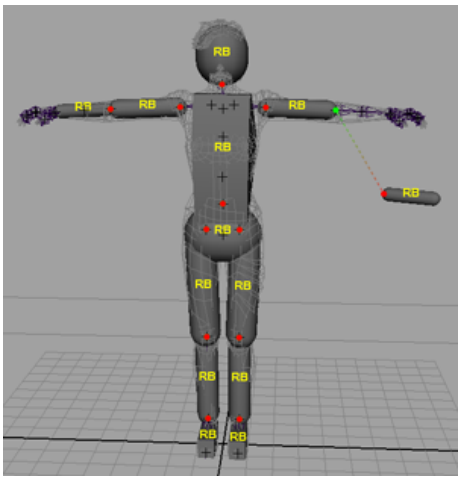
A single rigid body object can have any number of constraints applied to it.

**Constraint Display**

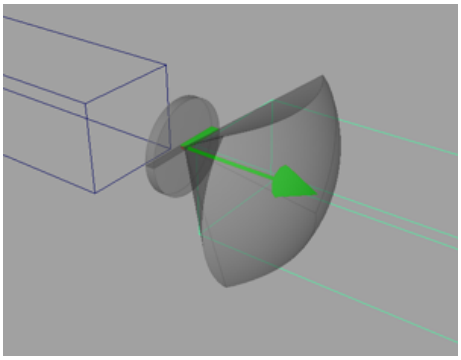
With the Havok Physics Tool active, constraints are displayed in the viewports as two dots representing the location of the child and parent spaces of the constraint. In most cases, these two spaces should be aligned, so both dots are drawn at the same location:




If the parent and child spaces are misaligned, a dotted line is drawn between them to indicate the offset:



When a constraint node is directly selected or the Havok Physics Tool is active with appropriate settings, a 3D representation of a constraint will be displayed to represent the constraint spaces and whatever other properties may be present (such as angular limits):



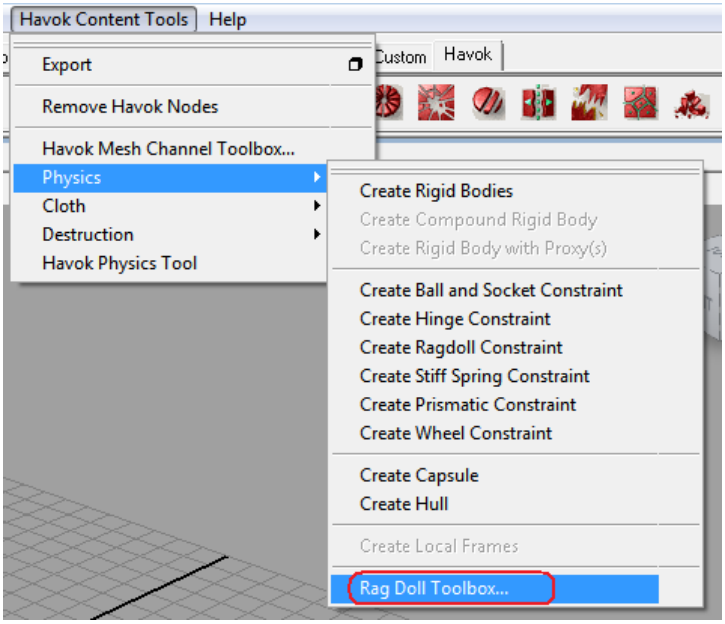
Constraints are displayed in the same way as manipulators, in that they have a *screen size* as opposed to a world size - constraints will always appear the same size on-screen regardless of the camera viewpoint or properties. **The global display size (for manipulators and constraints) can easily be changed at any time using the +/- keys on the keyboard.**

#### 4.5.3.4 The Rag Doll Toolbox

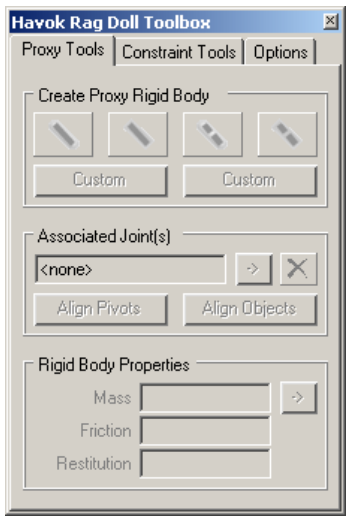
**Introduction**

One of the most common applications of rigid body simulation is the creation of articulated systems of rigid bodies representing characters (rag dolls). This usually involves the creation and alignment of rigid body proxies for the character's bones, as well as the setup and alignment of constraints that represent which character's joints. The Rag Doll Toolbox provides a set of tools to facilitate those operations. It is built entirely as a MEL tool that works on top of the lower-level physics tools.

You can open the Rag Doll Toolbox using either the **Havok Content Tools** menu or shelf:



The toolbox appears as a floating window with several tabs: the *Proxy Tools* are used to create and associate rigid bodies with bones; the *Constraint Tools* are used to organize and constrain these rigid bodies together:

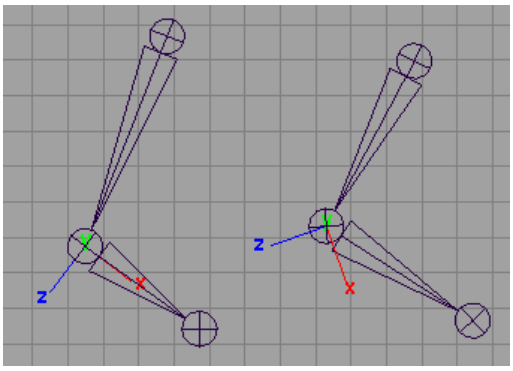



Several default properties of the Rag Doll Toolbox can be customized to suit your implementation and to ensure consistency across different installations of the Havok Content Tools - see Customizing the Maya Rag Doll Toolbox for more.

### Issues with Joint Spaces in Maya

Before describing the various Rag Doll tools, it is worth mentioning a problematic issue which is specific to Maya.

Maya works in terms of *joint's* rather than *bone's*. These joints may be assigned any orientation. For example, both of the following scenarios are valid within Maya:




In the first case, the X axis of the central joint points along the smaller bone. In the second case, there is no axis pointing along this bone. Therefore the local space of a joint in Maya does not necessarily correspond to the space of any bone which connects from the joint to a child joint. This indeterminism causes several problems for the Rag Doll Toolbox:

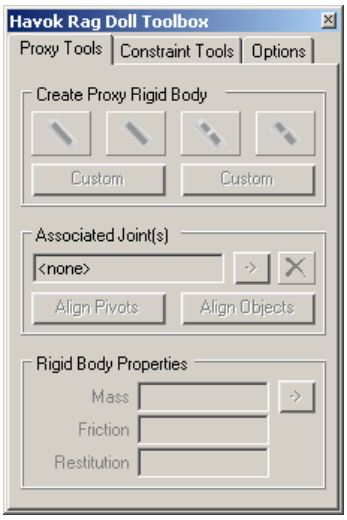
- When we create a new proxy associated to a bone, we need a 'bone basis' in which the proxy should be constructed. If we do not have a deteminate bone basis then it becomes difficult to create a proxy with the same basis as the bone.
- When we apply a templated constraint between proxies, that template would have been generated relative to some proxy/bone basis. Since proxy bases are indeterminate, the orientation of any applied template will often not end up as expected.

If the joint spaces in your rig do in fact consistently match those of the bones in you rig, then any such problems can be avoided. However, this is usually not the case. An ' **Override joint spaces** ' option is provided by the Rag Doll Toolbox which attempts to provide a workaround for this problem. If this checkbox is disabled, the Rag Doll Toolbox always takes the joint space to be that of the bone. If the checkbox is enabled, the toolbox attempts to determine a suitable bone space. To determine a bone space, the Rag Doll Toolbox looks for a single child joint of the joint we are interested in. If one exists, then we rotate the joint space so that its X axis points toward the child. If none exists, then we just use the joint space as-is.

If you are sure that your joint spaces are set up correctly, then it is safe to disable this option before working with the rag doll toolbox. Otherwise it is recommended that you leave this option enabled.

**Proxy Tools**

The first task when creating a rag doll representation of a character is to construct a set of rigid bodies to represent a selection of the bones of the character. These rigid bodies are usually collision primities such as capsules and boxes, which are very efficient to simulate by the physics runtime (capsules in particular).




The Proxy Tools provide the following functionality:

- Given a bone, create a proxy representation (a box, a capsule, or a custom rigid body) and align and associate the proxy with the original bone.
- Given a chain of bones, create a single proxy representation of that chain and align and associate that proxy with the chain of bones.
- Given a proxy rigid body (associated with a bone or chain of bones), replace it with another proxy.
- When creating proxies, it is very important that their pivots are aligned to those of the bones they represent, since, at runtime, movements on the bones will be converted to movement on the rigid bodies and vice versa. The proxy tools track this association and provide the ability to realign the proxy pivot/object at any time.
- Quickly access and modify the basic properties of one or multiple rigid bodies (mass, restitution and friction).

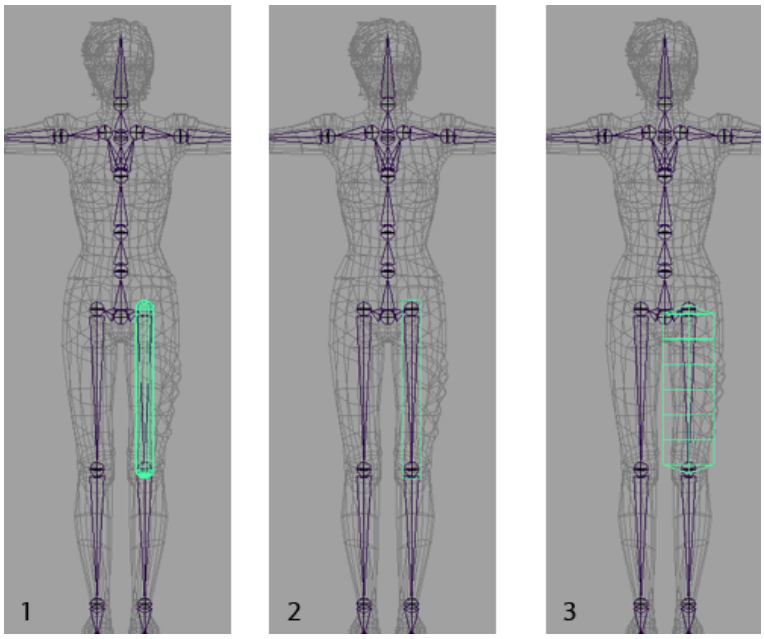
### Create/Replace Proxy Rigid Body

These buttons allow you to create proxies and associate them with bones:

When a rigid body proxy (either single or chain proxy) is selected, these "Create Proxy" buttons become " **Replace Proxy** " buttons - the proxy they create will replace the selected proxy, and the association with the original bone will be updated.

**Note:**
When proxies are created, they should have the same space as the bone with which they are associated. However this is not always possible in Maya. See Issues with Joint Spaces for more.

The first set of buttons (on the left) create *simple proxies* - where each rigid body is associated with a single bone.

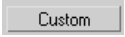



1. **Create Capsule Proxy(s)**

   With one or more bones selected, this will create a capsule rigid body bounding each bone selected. It will also associate and align each capsule with each bone.

2. **Create Box Proxy(s)**

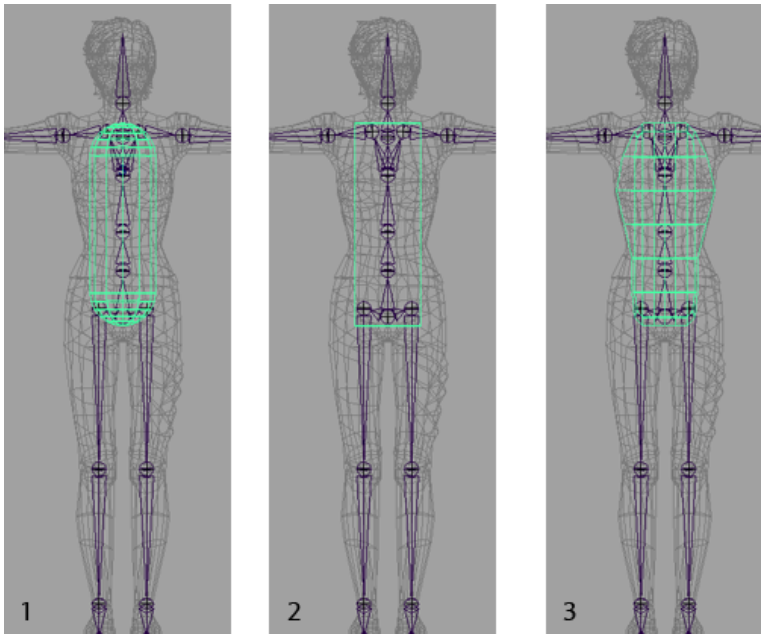
   With one or more bones selected, this will create a box rigid body bounding each bone selected. It will also associate and align each box with each bone.

3. **Associate Custom Proxy** 


   With a single bone selected, this will open a popup window where you can select any mesh object to become the proxy rigid body for the selected bone. It will then associate and align that rigid body with the selected bone.

The second set of buttons (on the right) create what we call *chain proxies* - rigid bodies which are associated to a set of bones in hierarchical order. Chain proxies are used in order to create single rigid bodies to represent multiple bones, such as a rigid body bounding multiple spine links.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 




1. **Create Capsule Proxy(s)**

   With a chain of bones selected, this will create a capsule rigid body bounding all of them. It will also associate and align the capsule with the chain of bones.

2. **Create Box Proxy(s)**

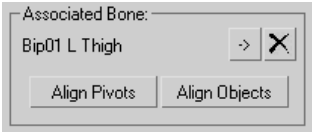
   With a chain of bones selected, this will create a box rigid body bounding all of them. It will also associate and align the box with the chain of bones.

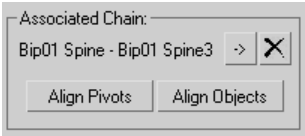
3. **Associate Custom Proxy**


   With a chain of bones selected, this will open a popup window where you can select any mesh object to become the proxy rigid body for the chain of bones. It will then associate and align that rigid body with the chain.

### Bone Associations

These controls are used to display and edit the associations between rigid body proxies and bones. Thus, they are only enabled when one or more rigid body proxies (rigid bodies created by one if the "Create Proxy" options above) are selected:



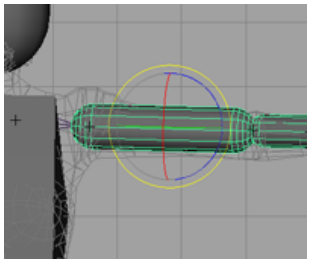



When a single proxy is selected, the name of the associated bone(s) is displayed below the **Associated Bone / Associated Chain** box.

The **Select Associated Bone(s)** button will select the bone(s) associated with the selected proxy.

The **Remove Association** button will remove the current association between the selected proxy and the bone or chain of bones. The proxy will remain as a rigid body, but the association with the original bone(s) will be lost.

The **Align Pivots** button will ensure that the rotate pivot of the proxy matches the pivot of the associated bone (or the first bone of the associated chain), without actually moving the proxy. Use this button whenever you have modified a proxy and want to ensure that the pivot is aligned:



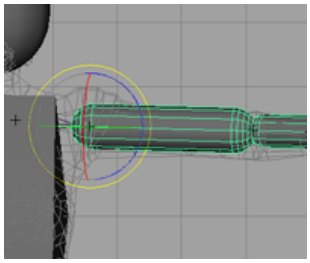
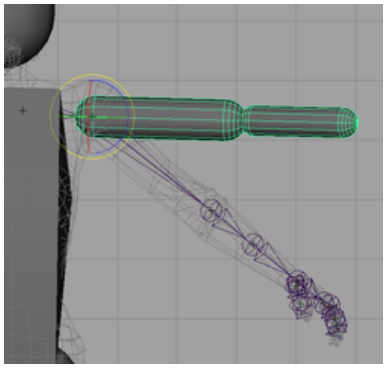


Table 4.3: Aligning Pivots

The **Align Objects** button will ensure that the pivot of the proxy matches the pivot of the associated bone by moving and/or rotating the proxy rigid body. Use this button whenever you move or rotate a bone and want to update the associated proxy accordingly:



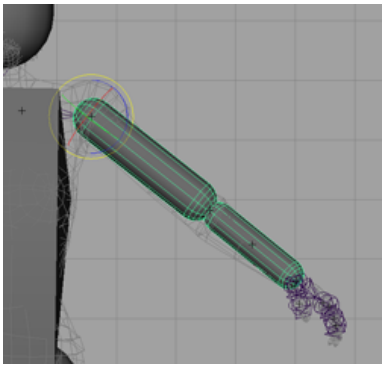
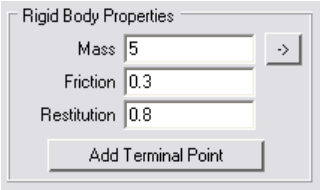


Table 4.4: Aligning Objects

### Rigid Body Properties

These controls are enabled whenever one or more rigid bodies (bone proxies or not) are selected. They allow you to modify the basic rigid body properties of the selected objects easily, and are particularly useful when you want to modify multiple rigid bodies at the same time.



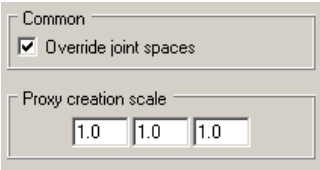

The **Open RB Node** button -> will open the rigid body node associated with the selected rigid body in the attribute editor (e.g. so that further properties can be edited).

The **Add Terminal Point** button is used to add extra leaf bones to skeletons to be used for raycasting, specifically to prevent ragdoll penetration with landscapes using the hkaDetectRagdollPenetration utility.

Selecting a ragdoll proxy and clicking the add terminal point button creates a locator node as the child of the proxy. The locator is positioned on the local bounding box of the proxy at the center of the face furthest from the proxy pivot.

### Proxy Creation Options

The 'options' tab of the toolbox provides some options which relate to proxy creation:

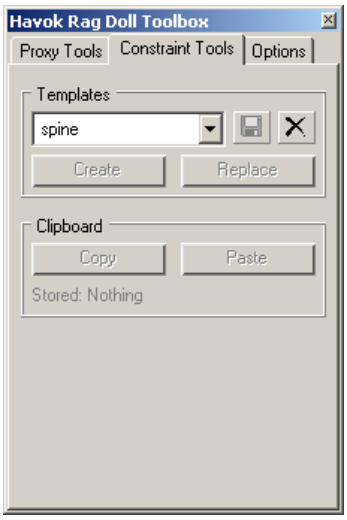



The ' **Override joint spaces** ' option has been explained already - see Issues with Joint Spaces for more

Whenever new proxies are created, they are automatically sized to fit the bones which they associated to. However, bones are often much narrower than the skin volume and so the proxies may not be wide enough. A ' **Proxy Creation Scale** ' option is provided for this reason. This is an additional scaling factor which is used during the creation of new capsule/box rigid body proxies - it can be though of as a local [X,Y,Z] scale applied to a bone space before a new proxy is created to fit around it. Normally, X points along the bone axis.

### Constraint Tools

Once the rigid body proxies have all been created, the next step in setting up a rag doll is to organize them into a constrained hierarchy to match that of the original skeleton.




The Constraint Tools provide the following functionality:

- Given a pair of proxies, reparent them in a way which reflects the original skeleton, and apply a constraint between them.
- Save and reuse constraint setups between characters, as *template* files.
- Easily replace a constraint with another templated version.
- Quickly copy and paste constraints within the same scene.

**Templates**

The Rag Doll Toolbox provides the ability to reuse constraint setups through the use of *templates*. These are text files which contain a type of constraint and its attribute values.

The dropdown list show all the available templates - this list is built by looking at all templates stored in the current *templates folder*, the location of which can be changed in the options section of the Rag Doll Toolbox.

The **Save Template As...** button is enabled only when a rigid body with a single constraint is selected. It allows you to save that constraint and its attribute values as a new template. Templates are saved as `*.txt` files.

The **Remove Template** button will permanently remove the selected template (i.e., it will delete the template file from the templates folder).

The **Create/Replace** buttons are those which do the main work, and are enabled only when two rag doll proxies are selected / an already constrained proxy is selected. When either of these buttons is pressed the following steps take place:

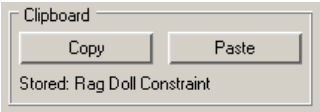
1. Ensure that hierarchy of the selected proxies is correct, by looking at the hierarchy of the associated bones. If not correct then reparent the proxies accordingly.
2. Create/replace a constraint node on the 'child' object - the one which is lower down in the hierarchy. The type of constraint which should be created is stored in the template file.
3. Set the constraint's parent object as the one higher up in the hierarchy.
4. Set the constraint's attribute values based on those stored in the template.

**Note:**

When working with templates, the orientation of applied templates is based on the relative space in which the template was created. Due to Issues with Maya's Joints, it is often necessary to rotate new constraints into the correct orientation after they have been applied from a template.

### Clipboard

While working on a symmetric character's constraints, it is often desirable to replicate constraints on each side of the character. The clipboard provides a quick way to create a temporary template which can then be applied throughout the rest of the character.



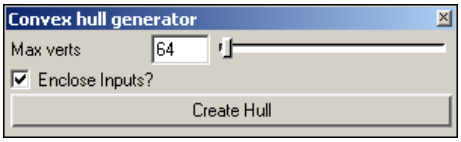

The **Copy** button is enabled only when a rigid body with a single constraint is selected. It will create a template of that constraint in memory, and display the type of the stored constraint for reference.

The **Paste** button operates as the **Create/ Replace** template buttons (depending on the selection), with the difference that it applies the template stored in the memory rather than the template selected in the drop down list.

#### 4.5.3.5 The Convex Hull Tool

In this section we describe a simple tool for generation of convex hulls from selected meshes. This tool generates an approximate convex hull from the currently selected objects (either meshes or nurbs) with a user-specified maximum number of vertices **Max verts** . The hull is approximated if the requested number of vertices is less than the actual number of vertices present in the complete convex hull. This is useful in a number of contexts, such as generation of assets for platforms which require <64 vertices per convex body. If the specified maximum exceeds the number of vertices in the true hull, the tool just generates the true hull. Note that if multiple meshes/nurbs are selected, the hull tool generates a single hull from all of their vertices combined.

When **Enclose input** is checked (default), the convex hull when approximated using **Max verts** , is forced to enclosed the input(s)



The convex hull tool utility consists of fields for the two parameters **Max verts** and **Enclose input** described above, and a **Generate Hull** button to generate the hull using these parameters. Generated hulls are placed in the scene root with the name "foo_Hull01" where "foo" is the name of the first object in the selection. Subsequently generated hulls accumulate in the scene root but are given unique names.

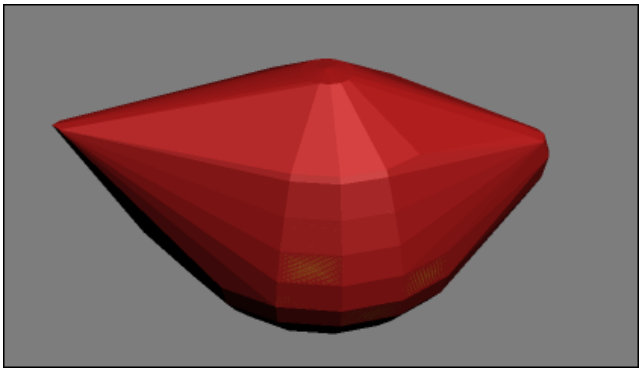


Figure 4.34: Full convex hull of a teapot

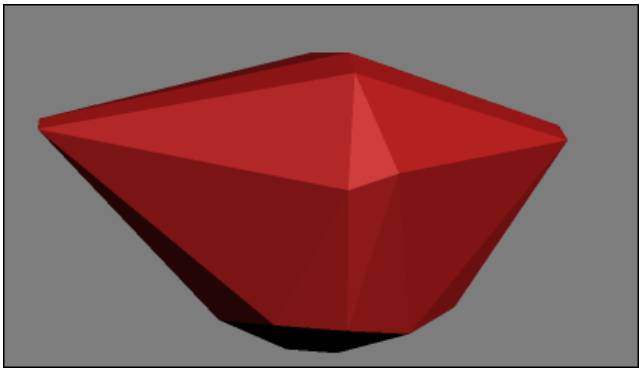


Figure 4.35: Approximated convex hull of a teapot with **Enclose input** checked

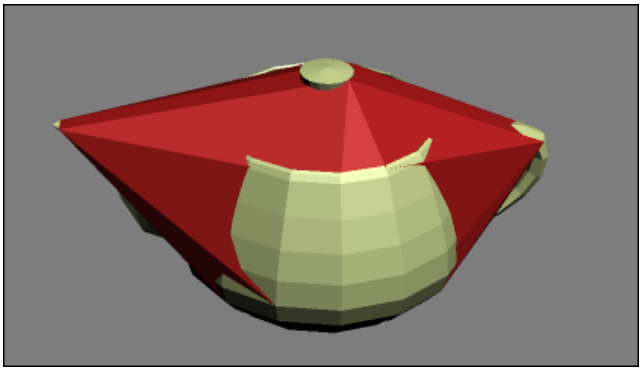


Figure 4.36: Approximated convex hull of a teapot with **Enclose input** unchecked

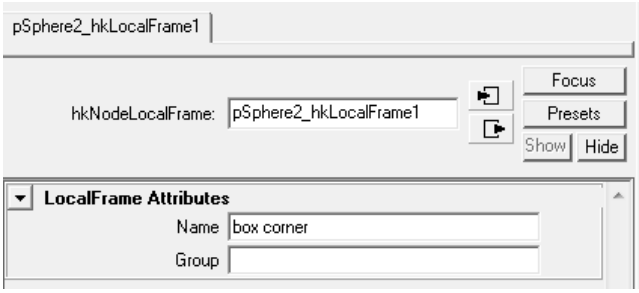
#### 4.5.3.6 Local Frames

Note that local frames are not yet supported by the new Havok Physics Create Bodies Filter in the 2013.1 release (They still work with Physics 2012). Support will be available in a future release

This section describes how to create **local frames** in Maya. To learn more about the concept of a local frame please see the Local Frame Concepts section.

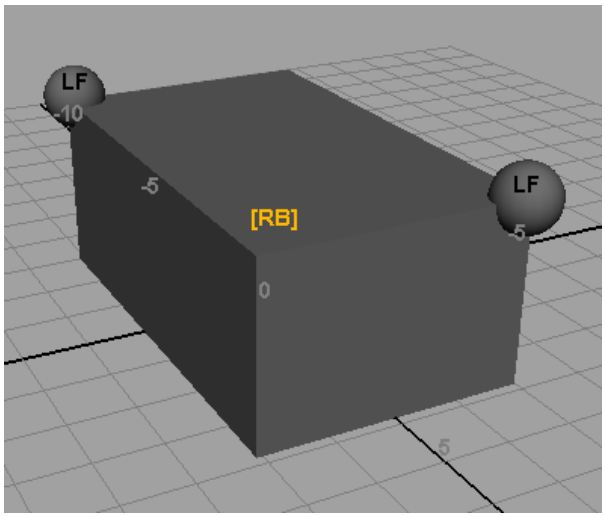
You can create a local frame by selecting an object in Maya and clicking on the **Create Local Frame** menu option or toolbar button .

This will add a local frame node to the selected object, which will appear in the Attribute Editor like this:




The node has a **Name** attribute in which you can name the local frame and a **Group** attribute in which you can name the group to which the local frame belongs.

An object that has a local frame node will be displayed with the label **"LF"** in the viewports. The following image shows a rigid body box with two local frames parented to it displayed in Maya:




Any object with a local frame node will be converted to a runtime local frame in the Create Rigid Bodies filter (if parented to a rigid body) or the Create Skeletons filter (if parented to a skeletal bone).

### 4.5.4 Maya: Animation Tools

Since Maya already provides a wealth of tools for creating content for animation playback, the Havok Content Tools use the features provided by Maya without the need of providing custom extensions (as it

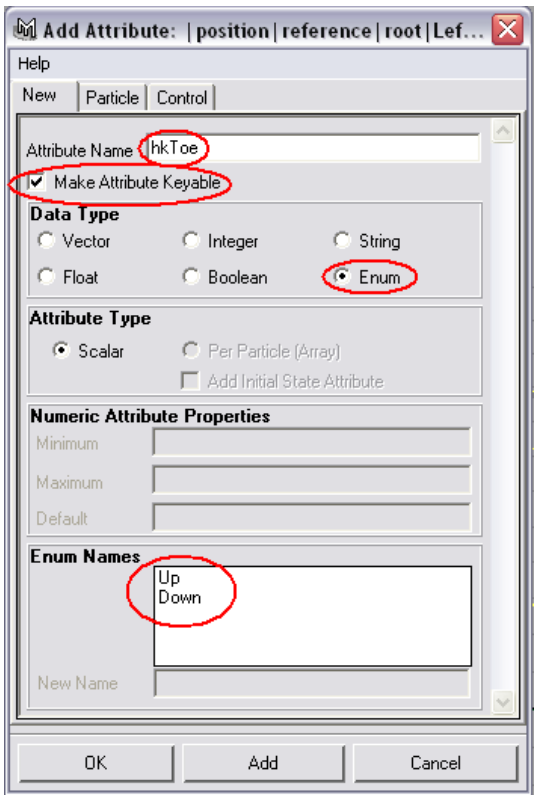
does for the physics). Concepts like skeletons, bones, skins and animations in Maya transfer to the same concepts/objects in the Havok SDK. The following section outlines one of the less obvious connections between Maya and the Havok Animation SDK.

#### 4.5.4.1 Annotations in Maya

Please check the common Annotations section for details on annotations in the Havok Animation SDK.

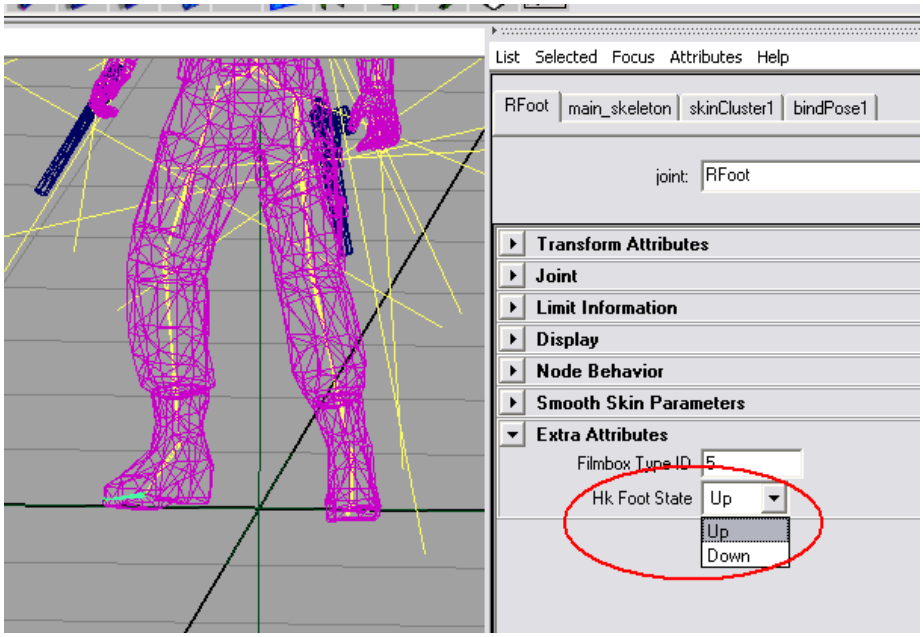
Maya does not explicitly support annotations, so dynamic attributes are used instead. Any compatible dynamic attributes owned by bones in the scene are converted to annotations at export time. The following steps explain how to create such attributes:

1. Select the bone to which you wish to add the annotation. Select **Modify** > **Add Attribute...** in the main Maya menu or **Attributes** > **Add Attribute...** in the Maya Attribute Editor to raise the following dialog:



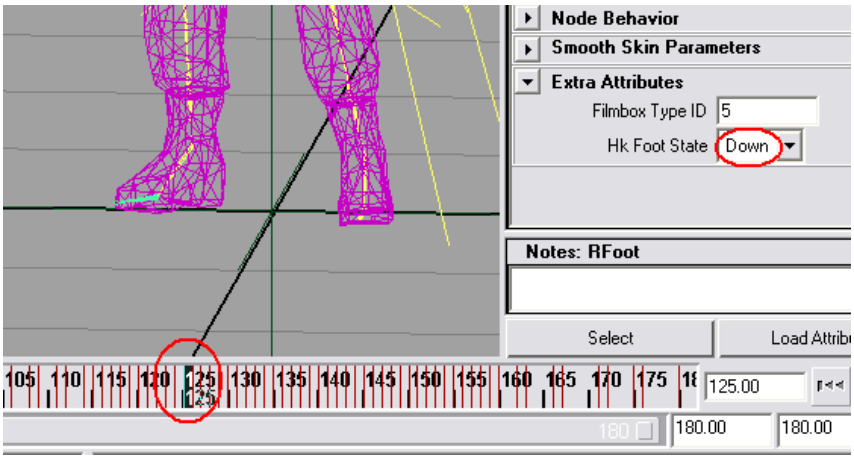

   Create an attribute whose name begins with '`hk`', and ensure it is both keyable and enumerated. Populate the **Enum Names** dialog with the string values you wish to annotate the bone with, and click the **Add** button.

2. Once you have added an attribute, it will be displayed in the attribute editor window, under **Extra Attributes** :




The dropdown displays the value of the new attribute at the current frame. Expanded, it displays the list of annotation values to choose from that you set up earlier.

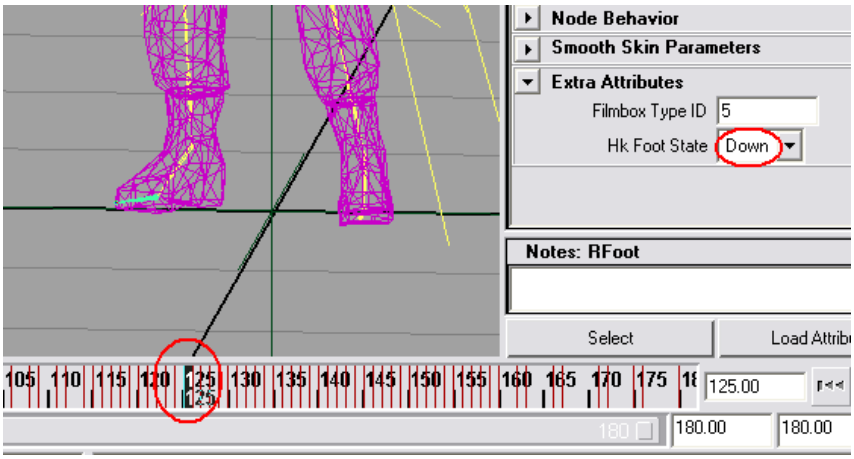
3. To create an annotation, you need to create a keyframe at the desired animation frame. To do this, first set the time slider to the desired frame and choose the desired annotation value:




To create a keyframe, reselect the bone in the model view and press the ‘ **S** ’ key. A vertical red line should appear in the time slider where the key has been created. Note that if you’ve changed the value of the enumerated attribute just before you press the ‘ **S** ’ key, the keyframe won’t be created because the attribute dropdown will have keyboard input focus. This is why you should reselect the bone in the model view before creating the keyframe.

4. Create as many keyframes as you like, changing the value of the enumerated custom attribute as you go. Note that the value of the custom attribute at the current frame is displayed in the attribute editor only when the animation is stopped and it doesn’t change during animation playback or when moving the frame slider by hand.

   You can view the annotations you’ve created in the Persp/Graph view. Simply click the Persp/Graph button on the left hand side of the screen, and select the custom attribute to see a graph of enumeration value against time.



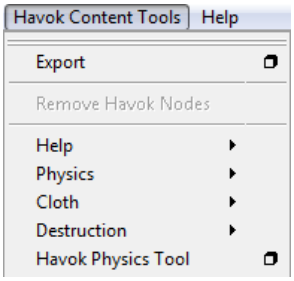

5. At export time, annotation strings are created by appending the name of the enumerated value to the name of the custom attribute. For example, a custom keyable enumerated attribute called '`hkFootState`' with enumerated values '`up`' and '`down`' will yield annotations called '`hkFootStateUp`' and '`hkFootStateDown`'.

### 4.5.5 Tutorial: Export and Animation Basics

In this tutorial we are going to explore the basic operations involved in exporting and processing an asset. We are going to use the example of exporting skeletons, animations, etc., but the principles regarding how assets are exported, the filter pipeline and the use of the filter manager and individual filters for processing applies to all Havok products.

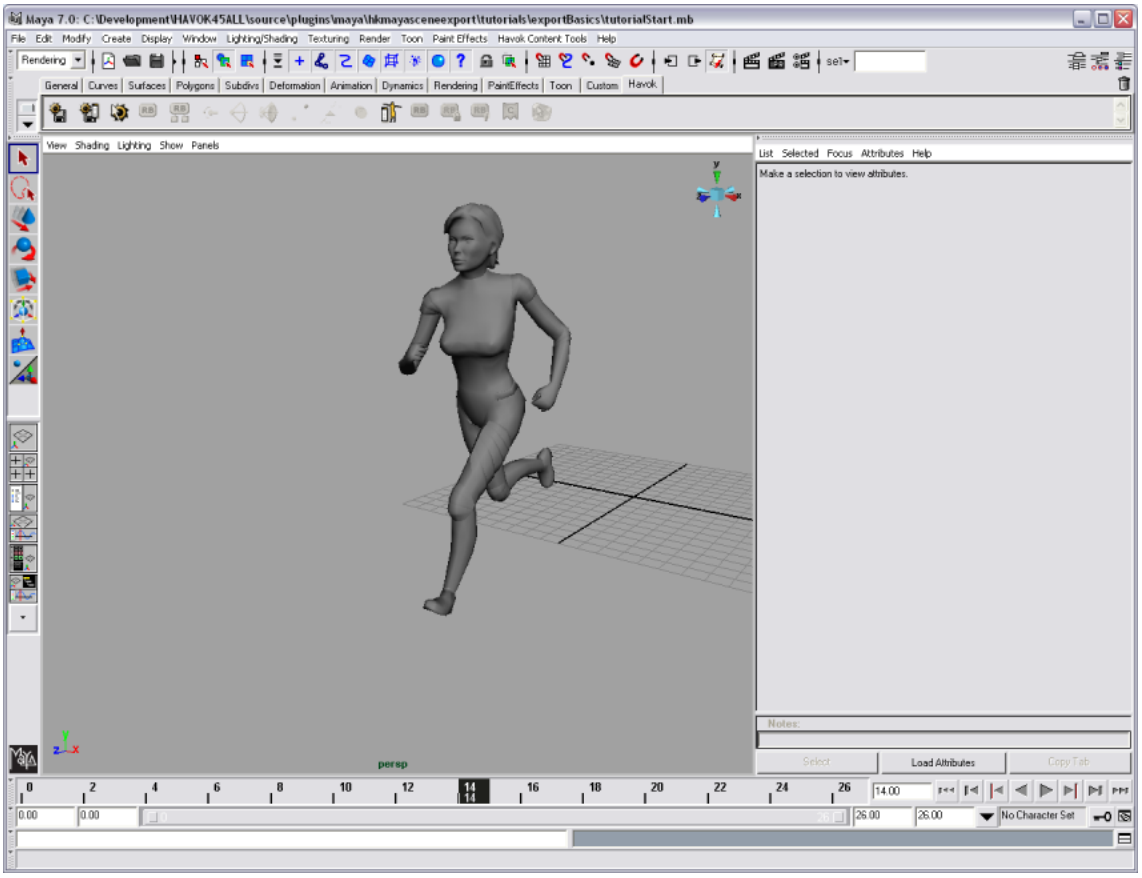
#### 4.5.5.1 Getting Started

Ensure that you have installed and loaded the Havok Content Tools for your version of Maya. You should see a **Havok Content Tools** menu in the Maya menu bar:




If the menu is not present then refer to the section on loading the plugins.

Open the file `"tutorials/exportBasics/tutorialStart.mb"` (installed with the Maya Havok Content Tools, usually in `"Program Files/Havok/HavokMayaModules"`):




The scene contains the animation of a girl running (a single cycle). If you examine the scene, you will notice that the animation is constructed using a set of bones which modify a mesh through the use of Maya skinning algorithms.

In this tutorial we will export this information (bones, animation, skin weights, etc.) from Maya into a format that the Havok Animation/Complete SDK can interpret.

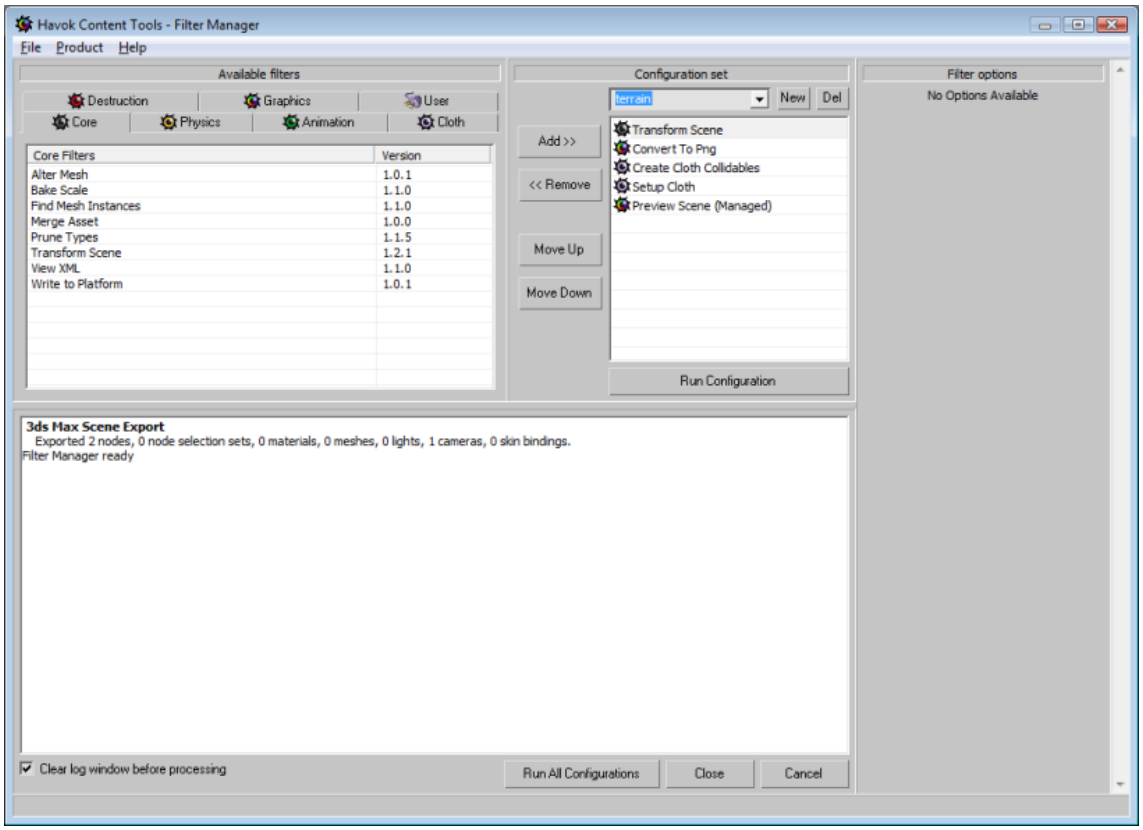
**Note:**

Even if you are not a user of Havok Animation or Havok Complete, you can still follow this tutorial as the Havok Content Tools packages contain filters for all products. This tutorial focuses on introducting the basic concepts behind asset export and processing so it is relevant for all Havok products.

#### 4.5.5.2 Exporting the Scene

To export the current scene, simply click on the **Export** button in the **Havok Content Tools** menu or shelf .

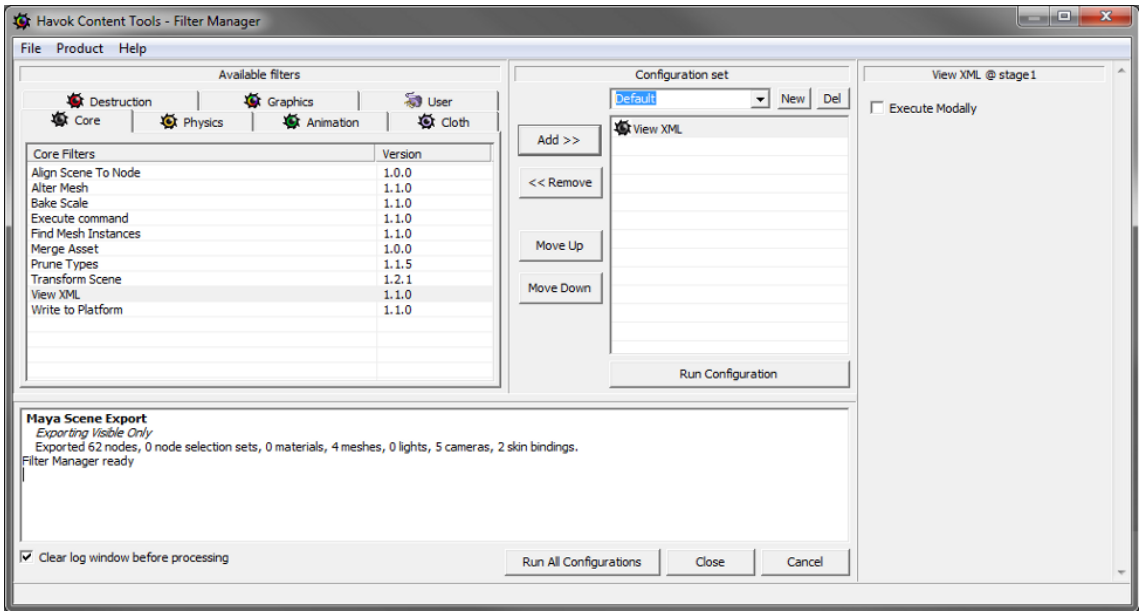
This invokes the Maya Scene Exporter - the scene is navigated and relevant information is pulled from the Maya nodes, attributes, meshes, etc., and converted into a modeler-independant format. After that, the content is passed to the filter manager - its UI should appear after a short time:



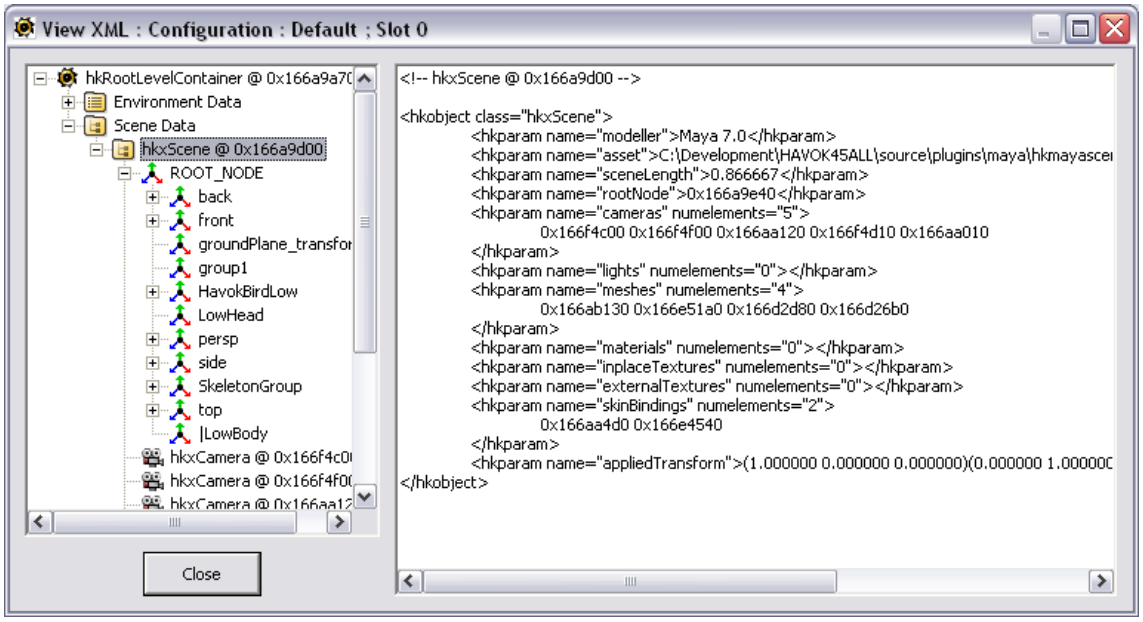

The **Configuration Set** pane (in the middle of the window) lists the filters that will be applied to the content we just exported. It is now empty, which means that nothing will happen to the content. You can add filters by selecting them from the **Available Filters** tabs and clicking on the **Add** >> button.

Let's start by adding a very useful filter for debugging purposes: the View XML filter.

- On the **Available Filters** window, click on the **Graphics** tab
- Double-click on the **ViewXml** filter (or select it and press **Add** > >)
- That filter should now appear at the top of the **Current Filters** list




Filters are executed in the order they appear in the **Current Filters** list; if a filter modifies the content, that modified content is what is used by the next filter (check the filter pipeline documentation for details). The View Xml filter is special in that it does not modify any content - it just opens a window that presents the current content in a human-readable XML format. Let's check what we exported from Maya by clicking on the **Run Configuration** button:




Notice that filter manager is processing an "`hkxScene`" object (scene data), which contains information about nodes, meshes, materials and skin bindings. This is what the Maya Exporter extracted from our current scene, and is the raw information that we are going to process further until we have an appropriate set of objects to be used by our runtime.

Close the XML window and return to the filter manager. Remove the View Xml filter by selecting it and

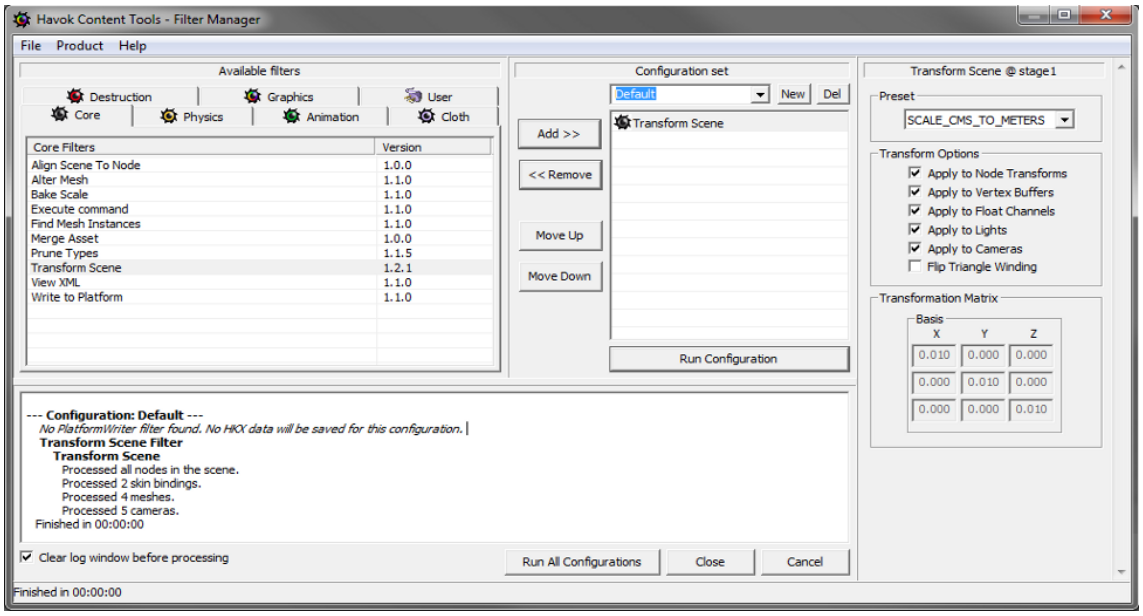
clicking on << **Remove** or hitting **DEL** .

**Tip:**

Feel free to add the View Xml filter at different locations during this and other tutorials in order to examine how processing is done by different filters. This filter is a very useful debugging tool.

#### 4.5.5.3 Processing the Scene

Let's now add some useful filters for processing the scene. The first filter that we'll add will be the Transform Scene filter (from the **Core** tab):

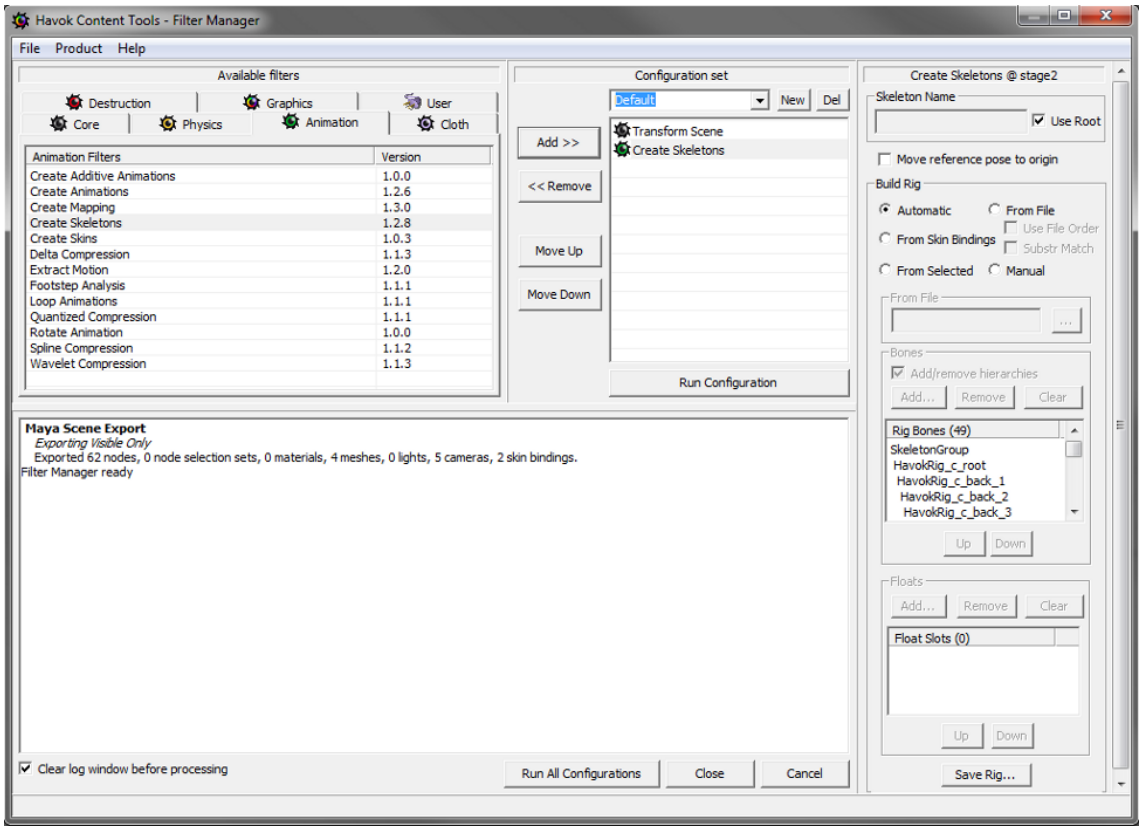



This filter is very useful and, in the majority of cases, it will be the first filter that you will need to add to any filter configuration. This filter applies an arbitrary transform to all the objects (nodes, meshes, etc.) in the scene. In this case, we want to apply it here because the internal units in Maya are centimeters, whereas the Havok runtime units are meters. Hence we use the **SCALE_CMS_TO_METERS** preset in the options of the filter (the right panel in the filter manager).

The next filter we will apply is the Create Skeleton filter (from the **Animation** category). This takes a look at the nodes in the scene and creates an "`hkaSkeleton`" object (used by the Havok Animation/Complete SDK).

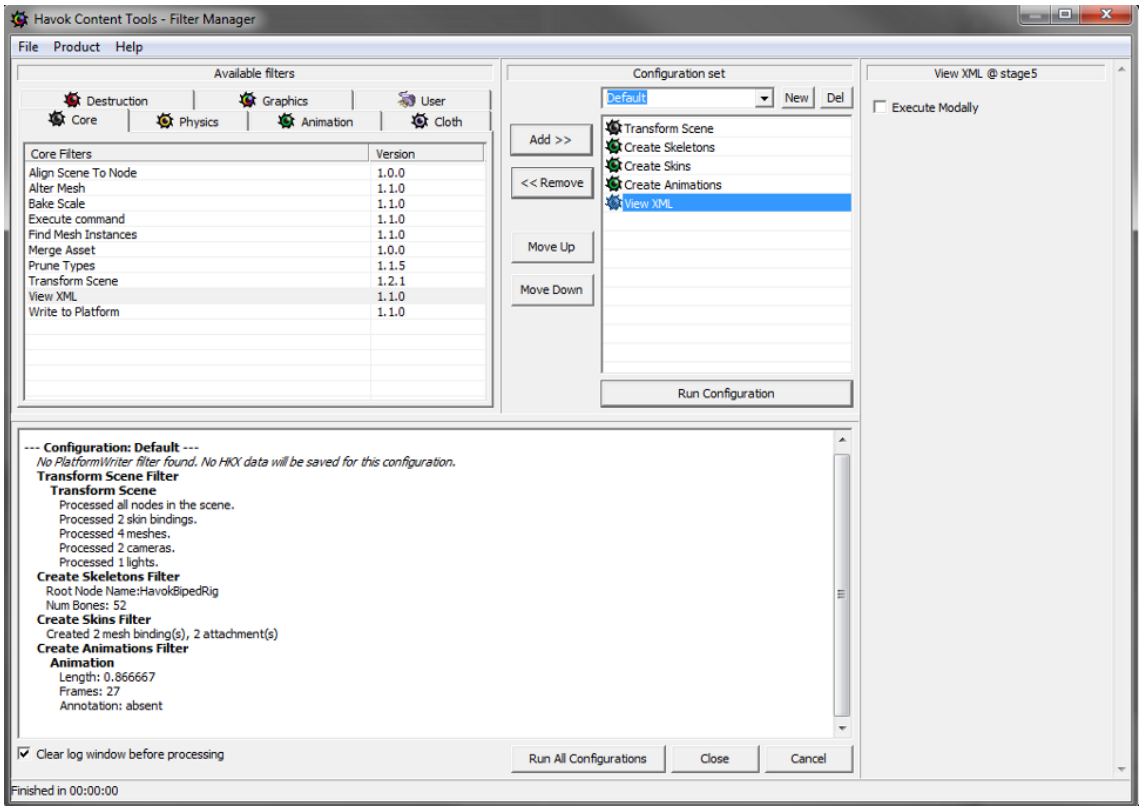
**Note:**

If your current Product Selection is set to Havok Physics, some of the filters will appear in red, and when run, warnings will be generated. You can either ignore those warnings, or change your product selection temporarily to Havok Complete.

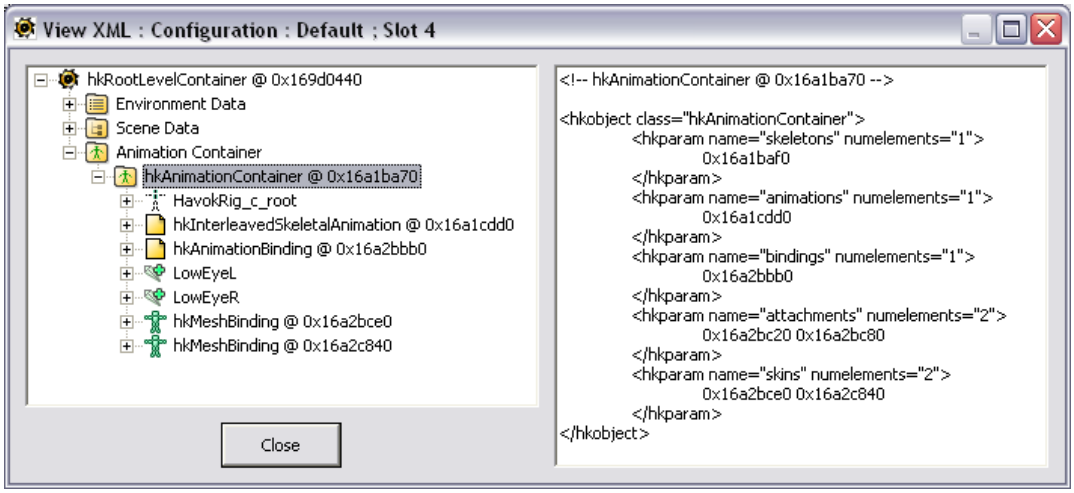



As you can see, this filter also has options associated with it (these appear in the right hand panel). In this case, switch from " **Automatic** " to " **From Skin Bindings** " - this will reduce the number of bones in the skeleton to just those associated with the skin. You can check the documentation for this and other filters if you want to learn more about how they work:

After this filter, add a Create Skin and a Create Animations filter (also in the **Animation** category) - using the default options for each. Finally, place the View Xml filter again at the bottom of the filter list:




When you run (press the **Run Configuration** button) the filter processing, the View XML filter will show you the effect of all the processing:



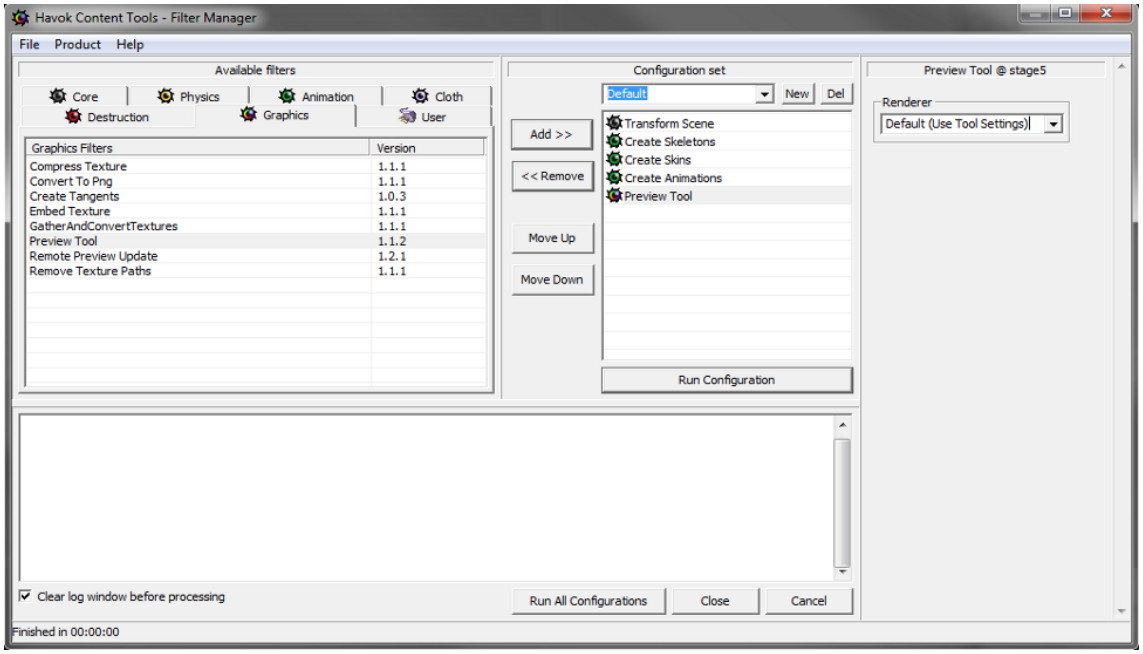

Notice how a new root-level object, an *Animation Container*, is now part of the content. This contains Havok Animation/Complete objects such as `hkaInterleavedSkeletalAnimaton`, `hkaSkeleton`, `hkaMeshBinding`, `hkaBoneAttachment`, etc., that we should be able to use at runtime.

#### 4.5.5.4 Previewing the Scene

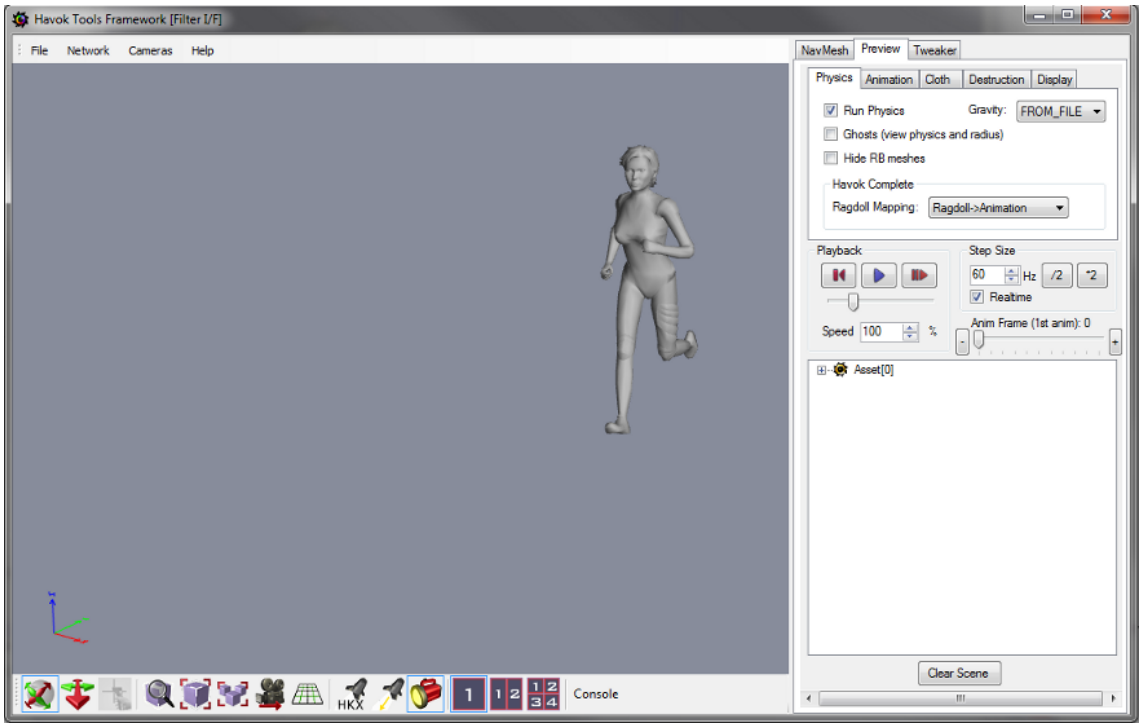
Although examining the contents by viewing the XML output give us an indication on how our processing went, in most cases visual inspection is the best way to ensure that the processed content matches our expectations.

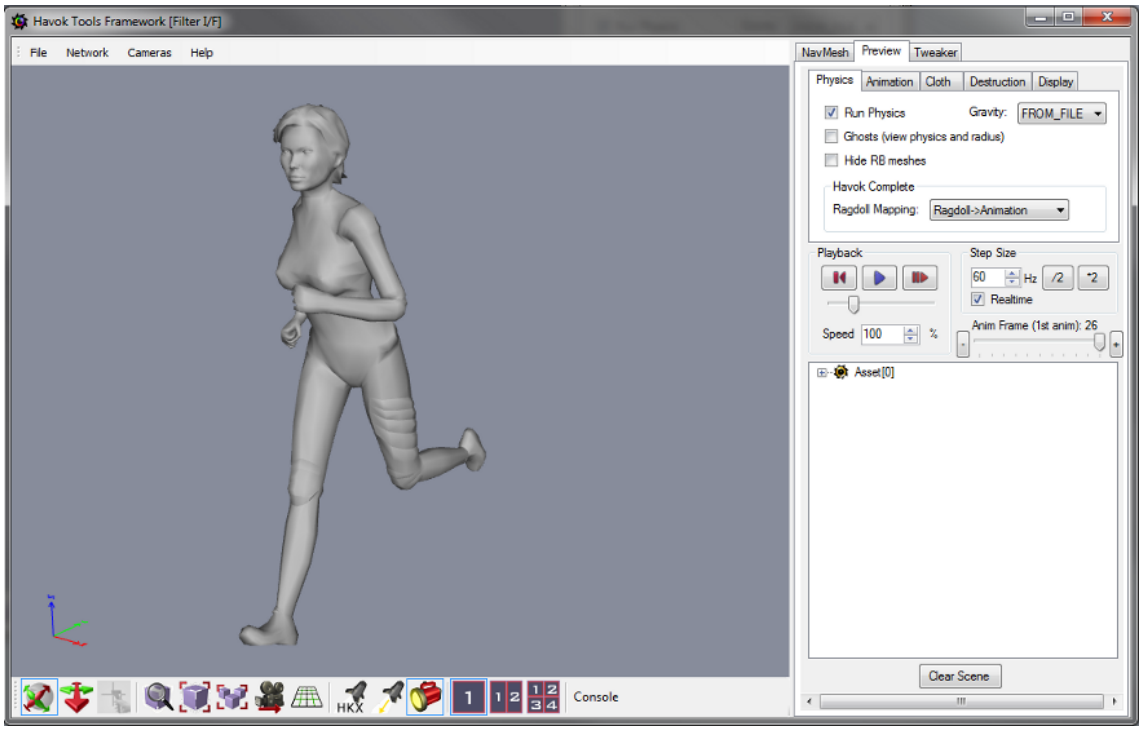
Havok provides the Preview Tool filter (in the **Graphics** category) just for this purpose - this filter is built using the Havok runtime, including some display libraries, and will play back any Havok content it finds.

Remove the View XML filter, add a Preview Tool filter at the end of the filter list, and click again on **Run Configuration** :




A modeless window will appear with our animation and skinning running in real-time:



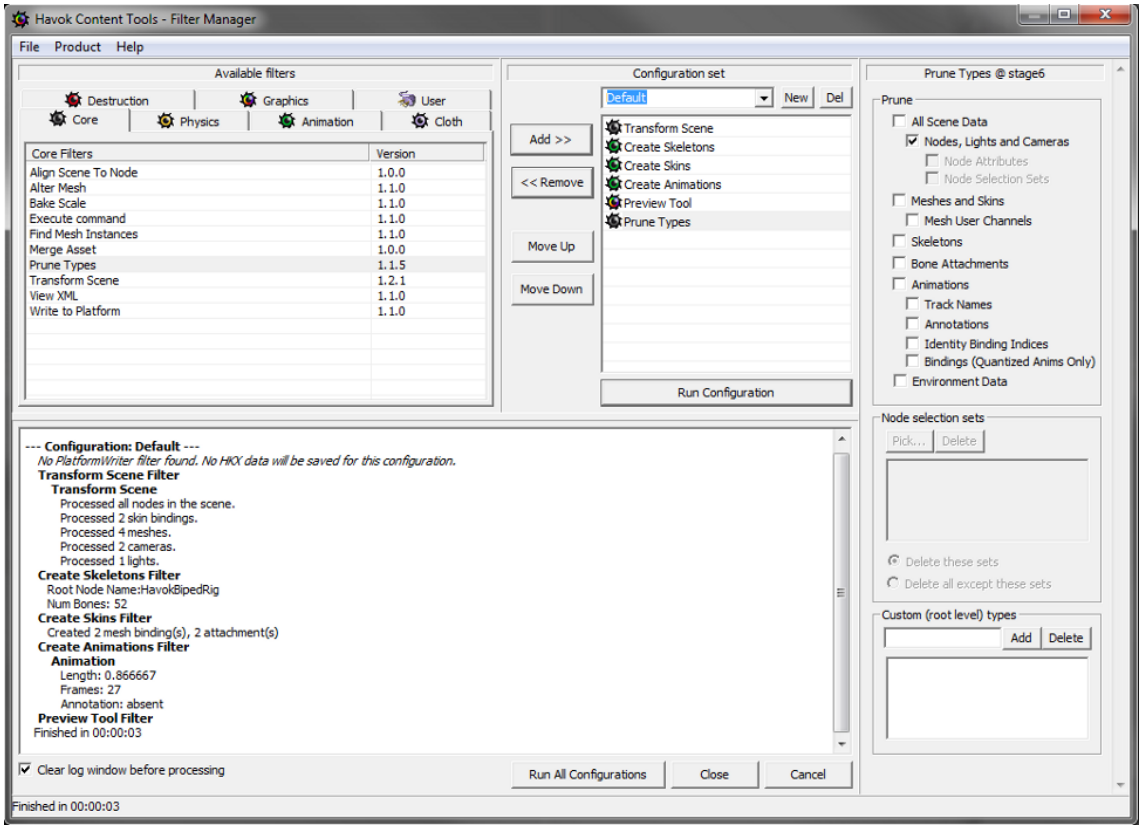




The preview window gives you a set of options to change the speed of playback, the current frame in the animation, etc. Check the Preview Tool filter documentation for more details.

#### 4.5.5.5 Pruning Some Data

Most of the filters either modify (e.g. the Transform Scene filter) or add (e.g. the Create Skeleton filter) information to the scene. Often it is useful to remove some of the data in an asset which is no longer relevant (particularly at the end of the processing). For example, once we have created our skeletons and animations, the original data regarding nodes and their keyframes is probably no longer needed (since it has been transformed into other classes).

We can add a Prune Types filter (in the **Core** category) after the Preview Tool filter. Leave the default options as they are: this will remove nodes (and their keyframes), lights and camera information:

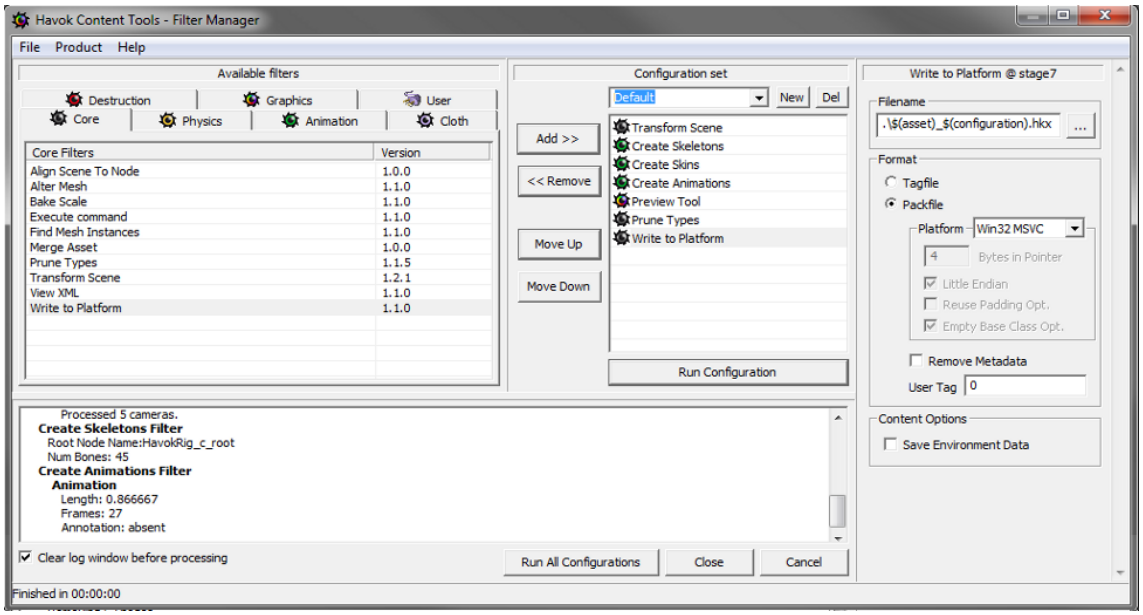



**Note:**
You could also add this filter before the Preview Tool filter, but since the Prune Types filter will remove camera information, the preview will have to use a default camera (rather than reflect the camera in the original scene).

#### 4.5.5.6 Writing the Processed Contents

So far, all of the processing done by the filters has happened in memory, and has not been written to any file.

The Write to Platform filter (in the **Core** category) will serialize the processed content to a file. It can write either XML format or binary format for many different platforms. Add the Platform Writer filter at the end of the filter configuration - by default it will save a binary .hkx file for the Win32 platform in the same location as the asset. Process the asset once again (click on **Run Configuration** ):




Observe in the log window that the processed asset is now saved alongside our Maya file, ready to be loaded into the Havok SDK.

This finalizes this first tutorial: you can find the final maya scene in the `"tutorialEnd.mb"` file.

We'd like to encourage you to experiment further with different filters and filter options. Check the results in the filter processing log, as well as the output of the View Xml filter, you should be able to get a good idea of what each filter does.

If you are a Havok Animation/Complete user, the following sections go a little further in processing this scene by doing some motion extraction. If you are a Havok Physics user, skip the following section and go to the Physics Basics Tutorial.

#### 4.5.5.7 Appendix (Animation/Complete Only): Extracting Motion

The animation we just exported has some *motion* on it: in between the first and last frame of the animation there has been some displacement on the character (it has moved forward).

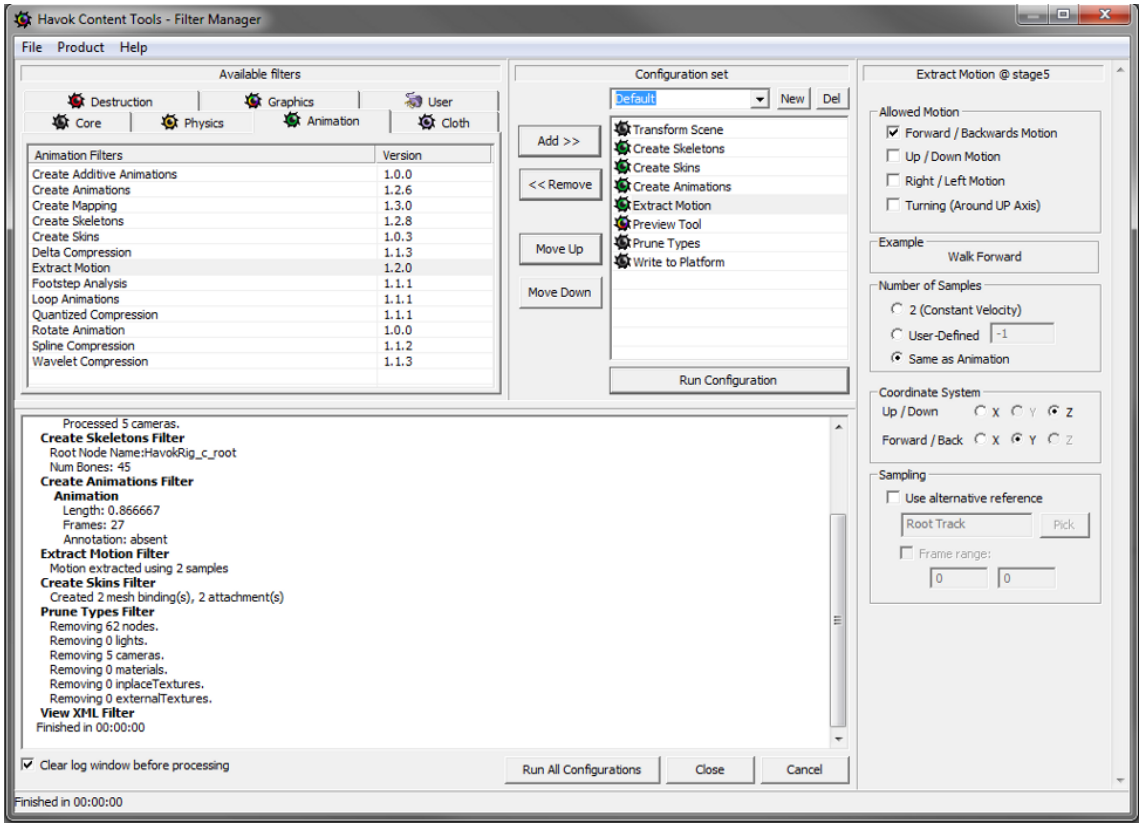
Motion extraction, explained in detail in the *Havok Animation* section of this manual, is the process on which we examine an animation and divide it into two: an motion/displacement component (the moving forward during running) and a local animation component (the movement of arms, legs and body during the run cycle).

So, after motion extraction our running animation should become a run-on-the-spot animation, complemented by some information on how the character should displace when that animation is applied (the extracted motion).

In this appendix to the tutorial we are going to do motion extraction to our run animation.

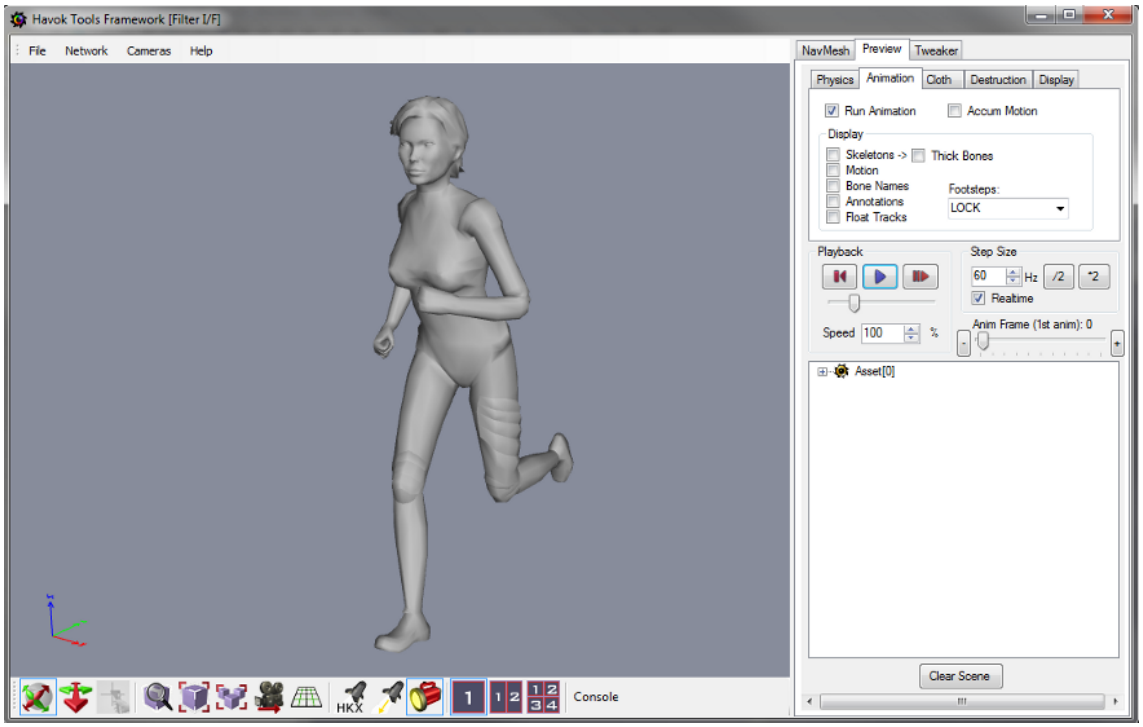
- Open the filter manager again (if necessary) by clicking on **Export**

- Add the Extract Motion filter (in the **Animation** category) to the filter list, just after the Create Animations filter (use the **Move Up** and **Move Down** buttons to position the filter in the right order):

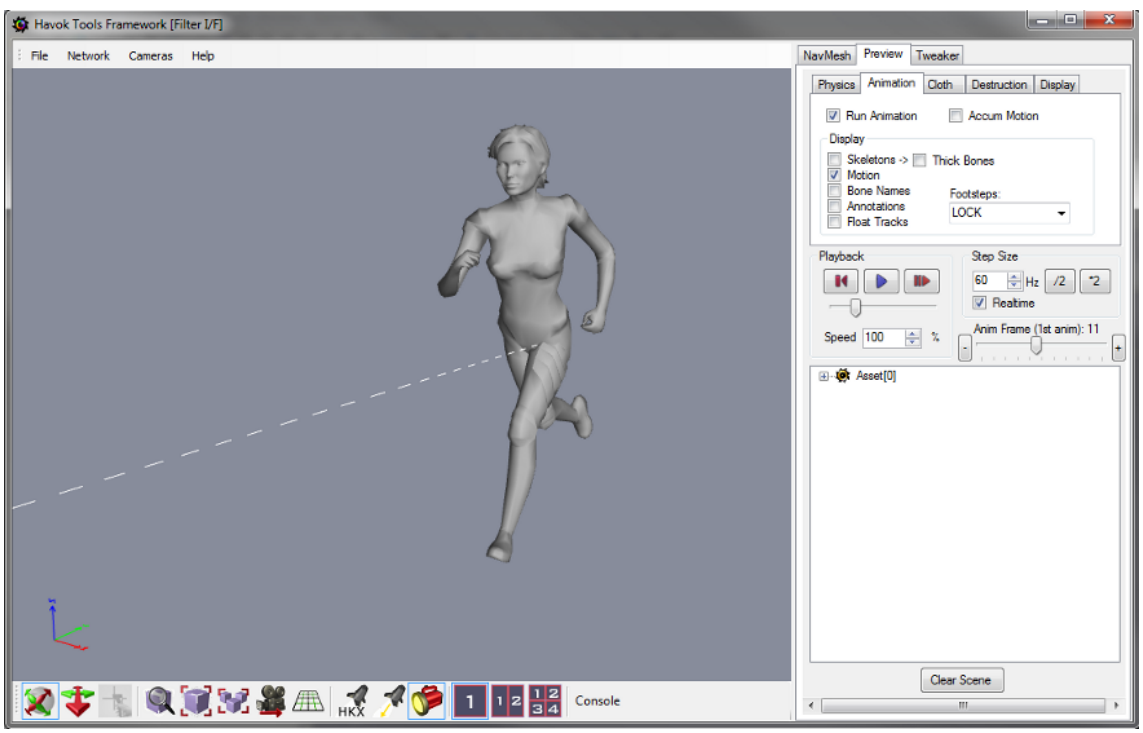



The filter options allow you to specify what kind of motion do you want to extract from the animation. In our case the motion is a forward motion so we'll leave the **Forward/Backwards** check box on. But, since in our animation the up direction is Y and forward direction is Z we need to specify that in the **Coordinate System** options (the defaults are Z, X).

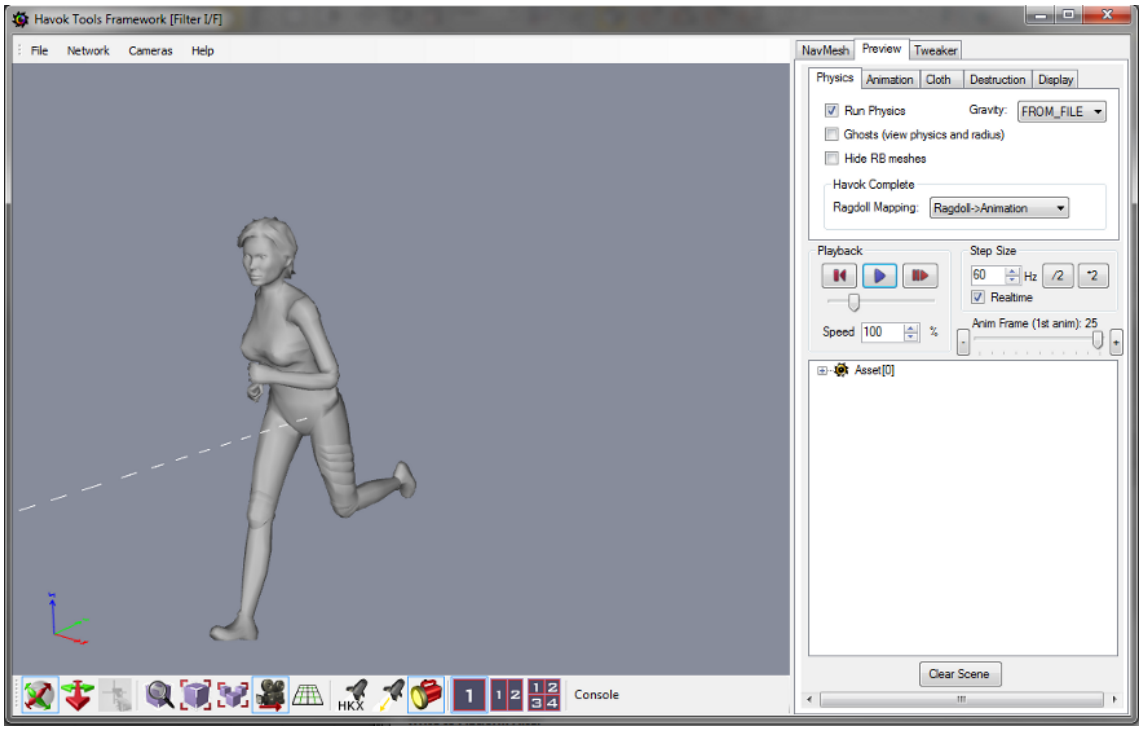
If we now run our filters, it will open the Preview dialog: you can see how now our animation is now a run-on-the-spot animation:




The Preview Tool filter knows about motion and it allows you to visualize it. In the preview dialog, click on **View** > **Extracted Motion** :




A dashed line is display showing the motion extracted from the animation. The preview can also apply and accumulate the extracted motion - on the **Animation** menu, click on **Accumulate Motion** :




Notice how the character now moves according to the motion that was extracted - notice also how that motion accumulates: the character moves forward continuously over animation loops (instead of warping back to the starting position at the beginning of each loop).

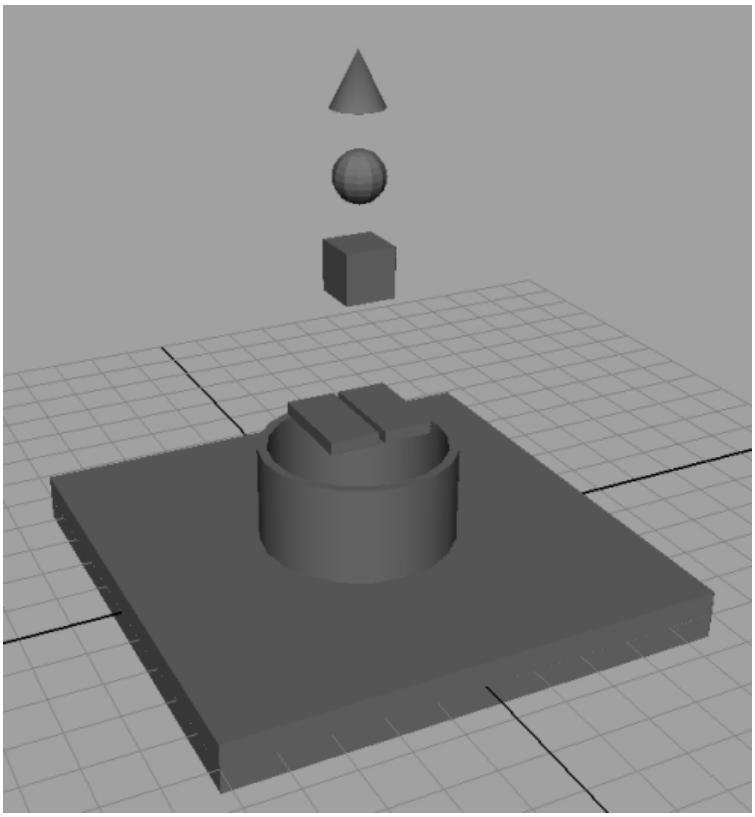
You can find the tutorial scene with motion extraction applied in the file `"tutorialEnd_MotionExtracted.mb"`.

### 4.5.6 Tutorial: Physics Basics

In this tutorial we are going to explore the basic operations for setting up rigid bodies and constraints in Maya, and processing them through the filter pipeline. This tutorial assumes that you are familliar with the basics of exporting and processing scenes, covered by the Export And Animation Tutorial. We also recommend that you take a look at the rigid body and constraint concepts sections of this documentation.

#### 4.5.6.1 Getting Started

Start by loading the scene `"tutorialStart.mb"`, located in the `"tutorials/physicsBasics"` folder of your Maya module (`"Program Files/Havok/HavokMayaModules/..."` by default):

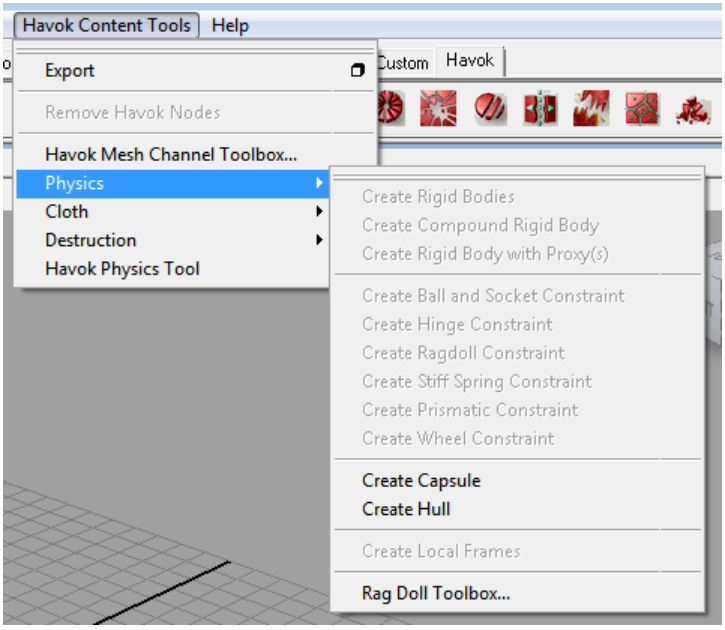


In this scene, we have a set of objects: a cone a sphere, a box, a trap door, a can and a ground. We want to attach rigid body properties to them so they behave as they would in real life.

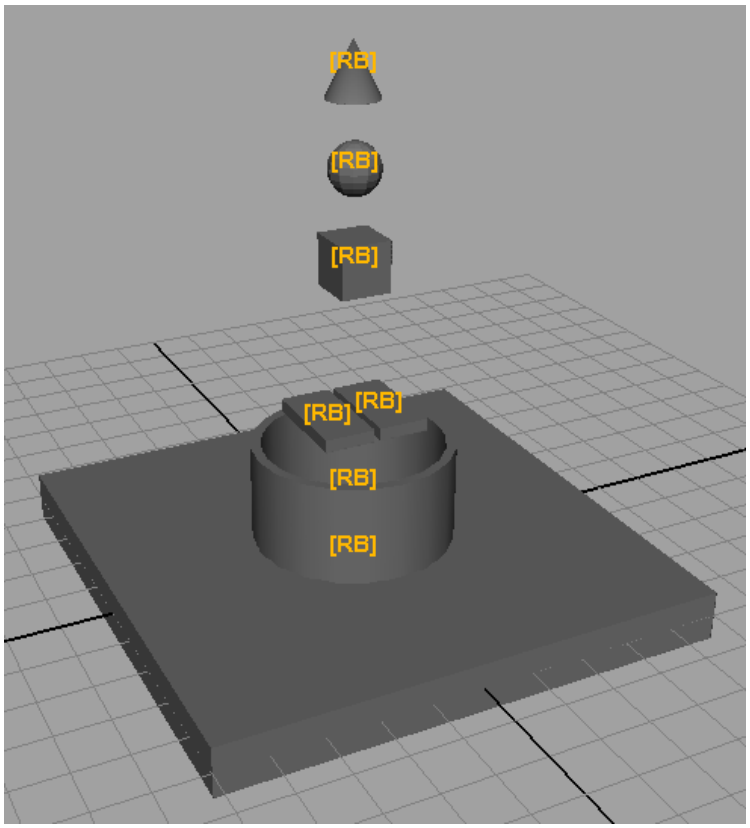
#### 4.5.6.2 Setting Up Rigid Bodies

To start with, we need to make these objects become rigid bodies, i.e., attach rigid body information to them so we can later on create rigid bodies and simulate them using the Havok SDK.

The Havok Content Tools provide a set of physics tools in Maya in order to do so easily. These tools are available through the **Havok Content Tools** menu and toolbar:

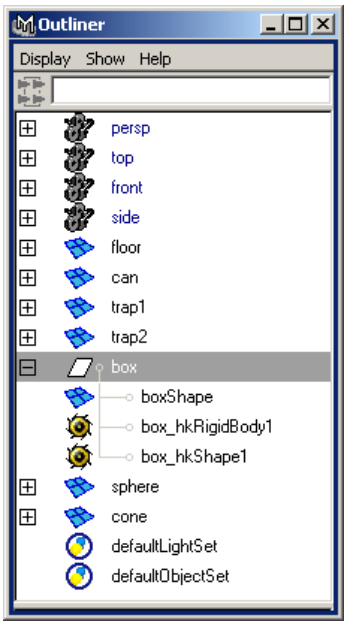


In our case, we want to create a rigid body out of every single object, so let's start by doing so. Select all the objects in the scene, and click on the **Create Rigid Body** button or menu option. Notice how all the objects in the scene are now are labeled with the text " [**RB**] ", indicating that these objects have now rigid body properties (the square brackets indicate that they are fixed bodies):




Note also that the Havok Physics Tool has been activated. This enables the mode in which Havok properties are represented in the viewport. The tool is customisable - you can access the tool settings at any time by double-clicking its icon in Maya's toolbox.

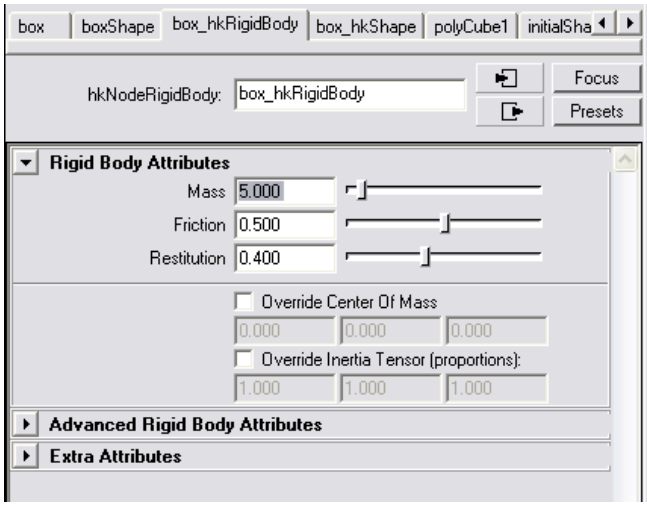
In Maya, rigid body properties are stored in nodes. For example, if you select the suspended box and then open Maya's outliner panel (Window | Outliner), you should see that two nodes have been connected to the box:




The rigid body node contains information about the dynamic behaviour of the object: mass, friction and restitution are specified here. The shape node contains information about the representation of the object for collision detection.

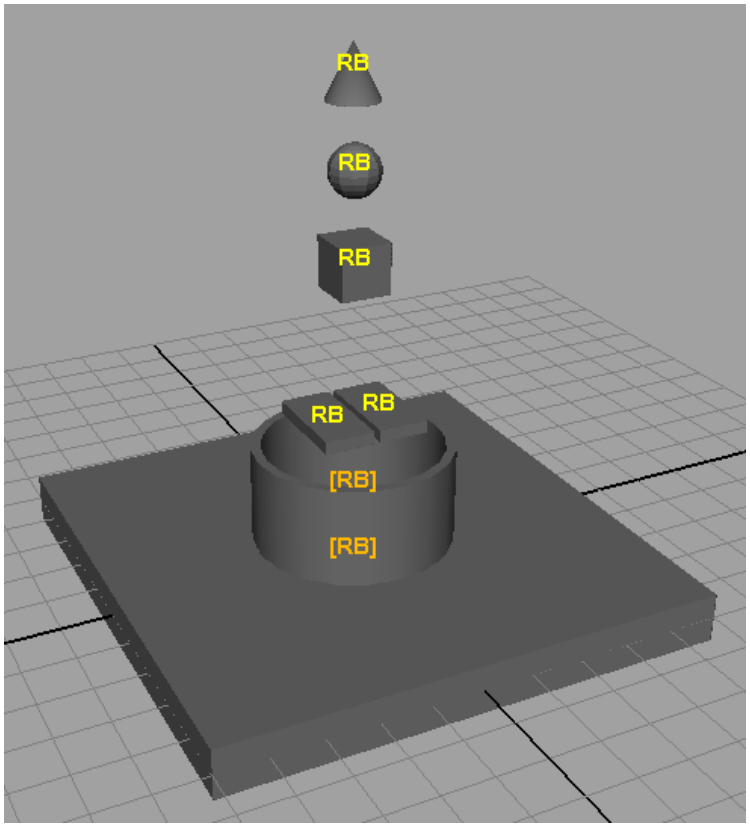
We will be exploring these nodes a little bit more later on. For now, select the rigid body node and open the attribute editor. Notice how the mass of the box is currently zero - that is the default mass for new rigid bodies. Since Havok will consider a mass of zero to be an indication of the rigid body being fixed in space (not moving), we want to make sure that we specify a mass for all the dynamic objects in our scene - that would be the cone, the sphere, the box and the trap doors.

Set the **Mass** attribute of each dynamic to a value different than zero (5kg for example). Do this by opening the attribute editor and either: selecting each rigid body node directly in turn; selecting each object and locating the rigid body node tab:




After doing this, we have a scene set up with 7 rigid bodies, 5 of them dynamic/movable and 2 of them

(the ground and the can) fixed. We can easily identify which are fixed and which are not:




**Note:**
The 'RB' labels, like all other Havok content, is only visible while the Havok Physics Tool is active. Remember to activate this tool whenever you wish to view or edit any Havok content in the scene.

#### 4.5.6.3 Previewing the Scene

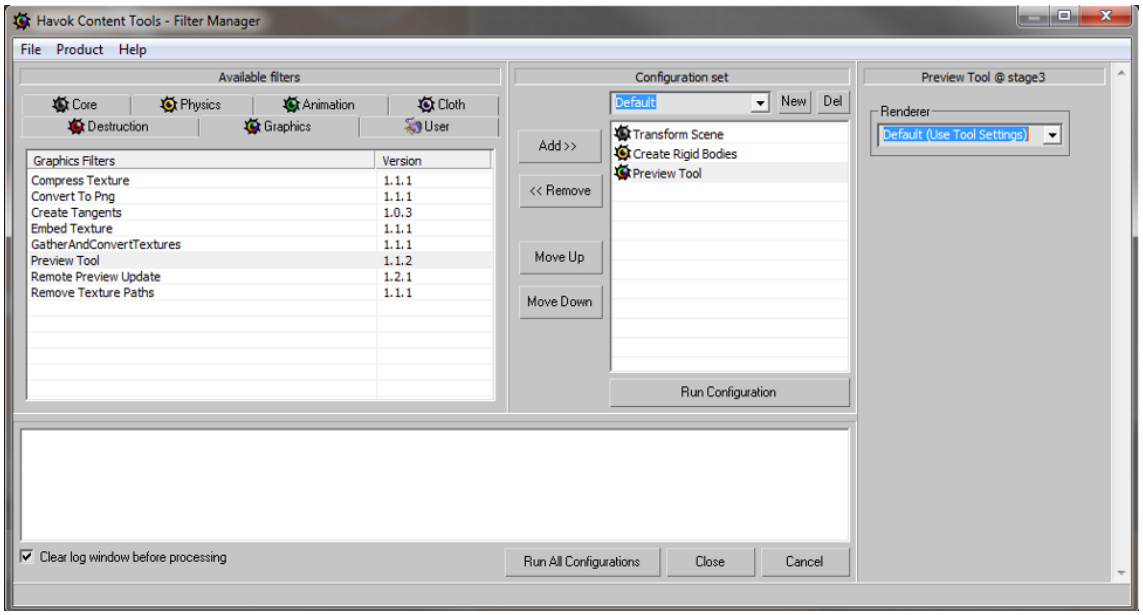
Now that we have some content set up, we'd like to preview it and check that everything behaves as expected. In the previous tutorial we saw how to export, process and preview assets using the Maya scene exporter and the filter pipeline. We are now going to do the same, but this time using some processing specific to physics assets.

Start the export and processing by clicking on the **Export** button or menu option. This should open the filter manager, with an empty filter setup.

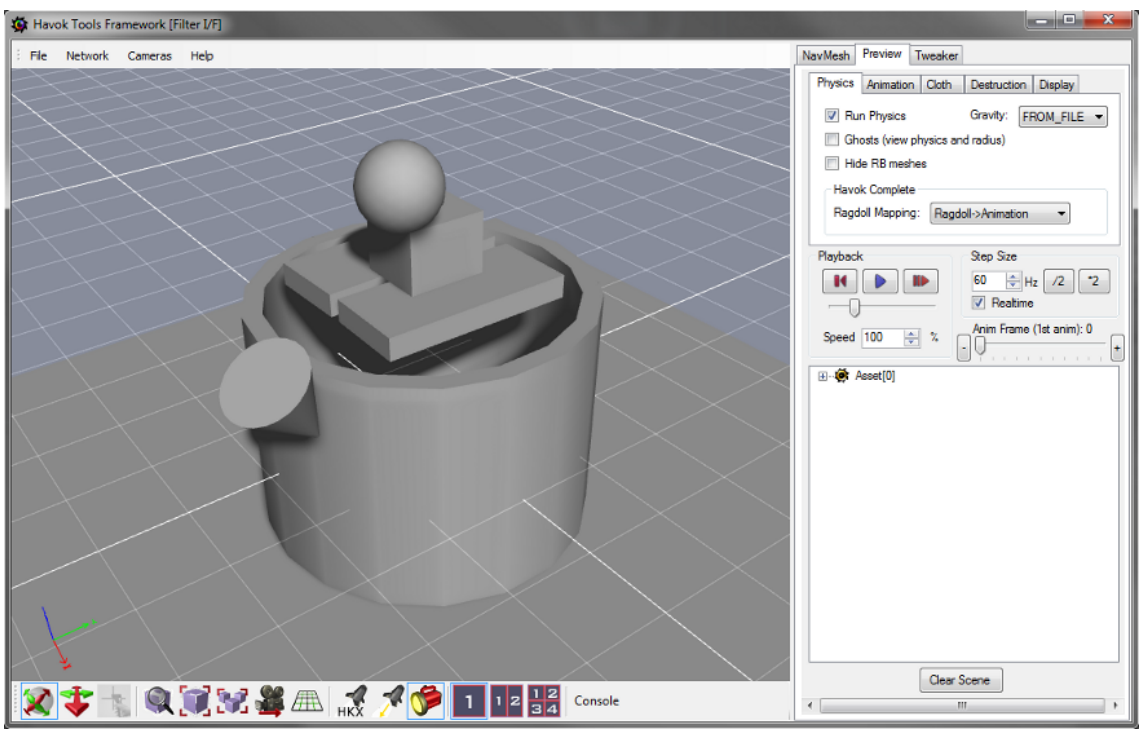
Insert, in this order the following filters:

1. Transform Scene ( **Core** category): Since Maya works internally with centimeters and we work in meters, set the **Preset** parameter to **SCALE_CMS_TO_METERS**
2. Create Bodies ( **Physics** category): This filter will detect the rigid body information we assigned to our objects and will create rigid bodies based on it. Use the default options.

3. Preview Tool ( **Graphics** category): We want to visually inspect the behaviour of our objects, so we will use this filter to render and simulate them.




If we now process our asset (press the **Run Configuration** button), a preview window will appear with our scene:

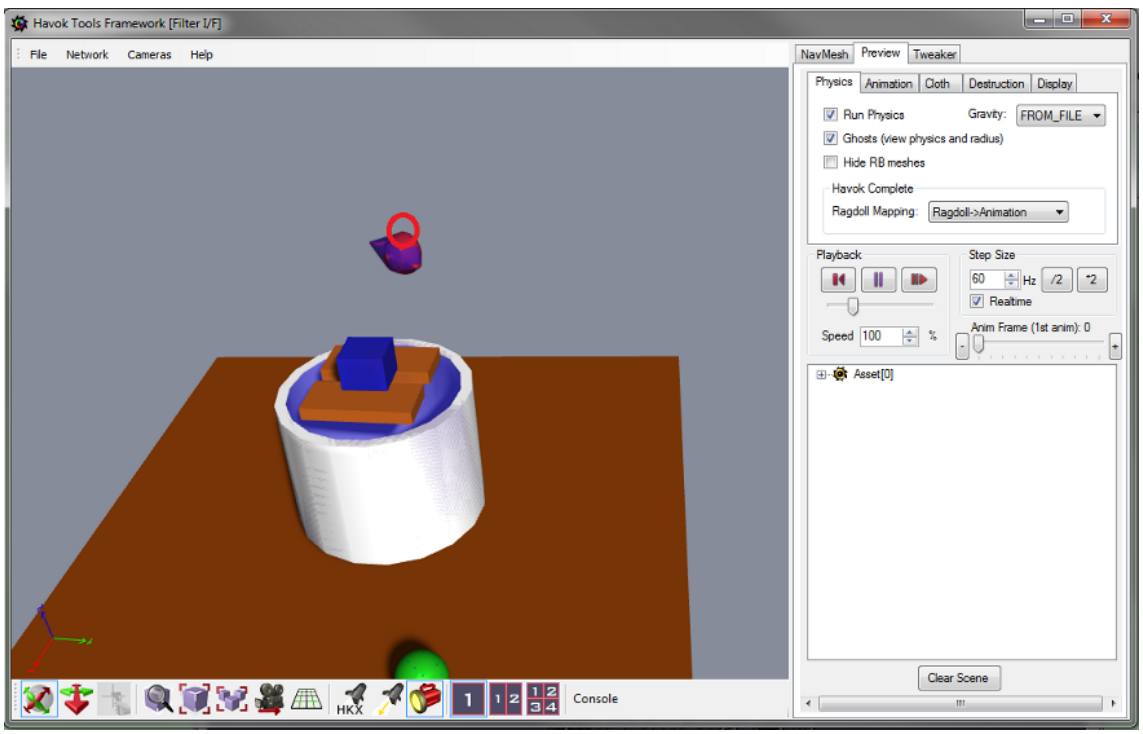



Press the **Play/Stop** button to start the simulation.

You will notice that all the objects fall as expected, but there is couple of things which are not quite right:

- None of the objects fall inside the can - it is like there is an invisible lid on it. If you switch on the menu option **View** > **Physics Ghosts** you should see a ghost representation of the rigid bodies displayed in the preview (in alpha blended blue). That shows that there is indeed a lid on the can. It looks like the can is being simulated as a closed volume!
- The trap doors do not behave as such, and are just falling as any other object. This is because we have not set up any constraint to specify that they should behave as hinged doors. We will take a look at this later on this tutorial.

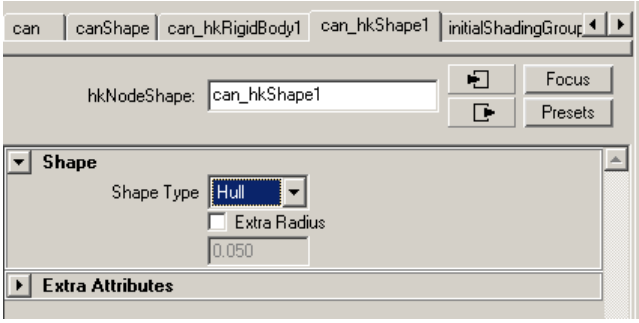
**Mouse Picking**
You can interact with the objects in the scene by using *mouse picking*: with the mouse cursor over one of the movable objects, press the **SPACE** bar and, without releasing it, move the cursor around - the rigid body will behave as if a spring was attached between itself and the mouse pointer. Release the **SPACE** bar to release the object.




When you are ready, close the preview and the filter manager and return to our Maya scene.

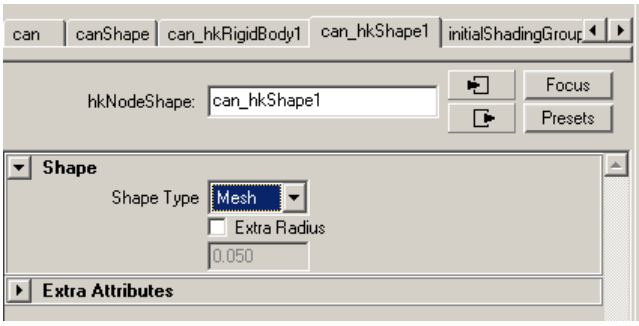
#### 4.5.6.4 Shape Properties

Let's take a look at the can. Since our problems are related to collision detection, we'll take a look at the shape information associated with it. Select the shape node attached to the can object - by selecting the can in the viewport and finding the appropriate attribute editor tab:



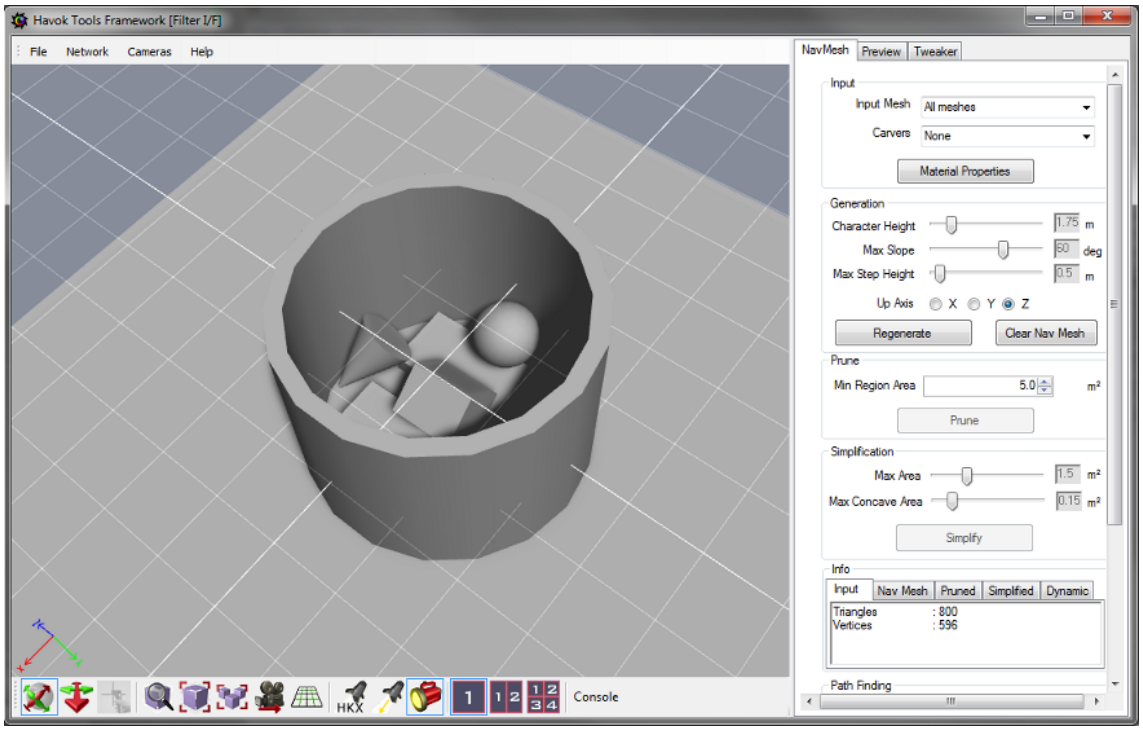

Notice how the Shape Type parameter is set to "Hull" (Convex Hull). When creating a rigid body from a mesh, the default value for this Shape Type parameter is taken by examining the object type in Maya. Thus, the default Shape Type for boxes is (Bounding) Box, the default Shape Type for spheres is (Bounding) Sphere, etc. For other arbitrary objects the default Shape Type is Convex Hull[4]. The Convex Hull of an object is the tightest convex volume that includes all the vertices of the object. The best way to visualize a convex hull is by imagining shrink-wrapping the object. In our case, using the convex hull of the can results on simulating a closed cylinder, rather than an open one, leading to the behaviour we observed in the preview.

In our case we want to use the exact mesh of the object, so we need to change the Shape Type parameter to **Mesh** :




If you now **Export** the scene again and process the asset (press **Run Configuration** ), you will see that the objects now properly fall inside the can during the simulation:

[4] The reason why the default Shape Type is Convex Hull and not Mesh is that convex objects are much faster to simulate than concave meshes and, in many cases, simulating a mesh as its convex hull gives very good results. Also, movable (not fixed) concave objects (objects with Shape Type set as Mesh) are not really suitable for interactive simulations (a warning will be generated if used).



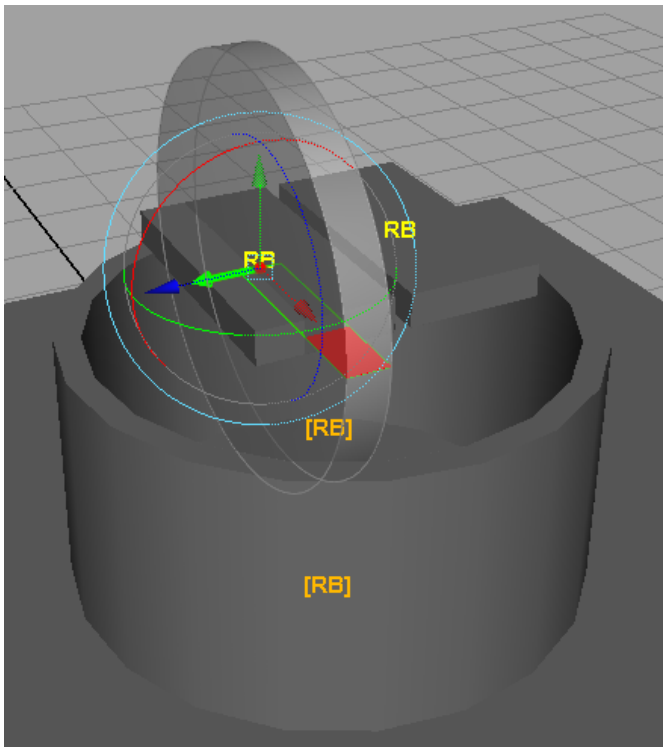

Close the preview and the filter manager and return to our Maya scene.

#### 4.5.6.5 Setting Up Constraints

The last thing we want to improve in our scene is the behaviour of the trap doors: currently they behave as the other objects - they just fall. We want them to behave as if they were constrained to their location by hinges.

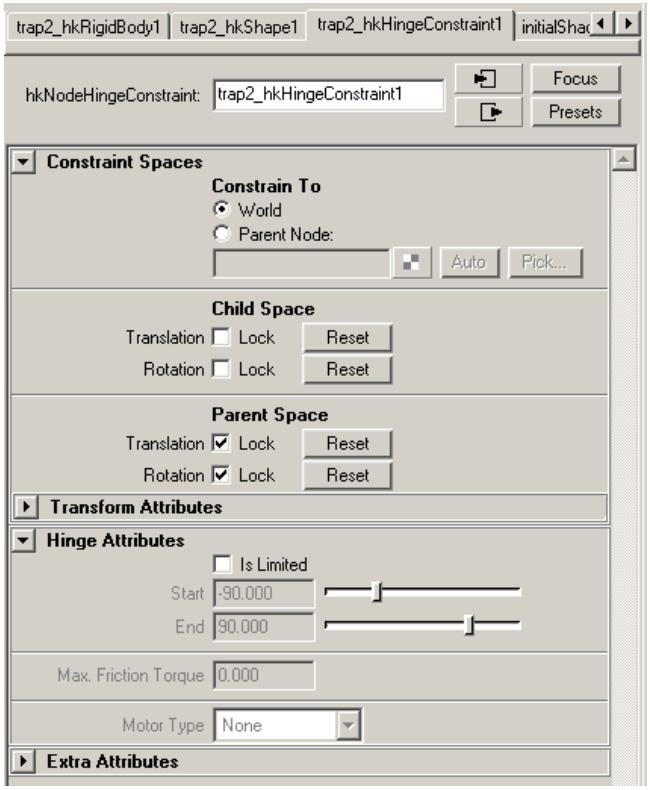
The Havok Content Tools for Maya give you the ability to set up and visualize constraints in Maya. In our case, we want to create a hinge constraint - a constraint that will limit the movement of the trap doors to a rotation around a single axis.

Select one of the trap doors and click on the **Create Hinge Constraint** button or menu option:




This adds a **Hinge** constraint node to the object, and activates the Havok Physics Tool so that we can visualize and manipulate it in the viewport. Note that the display size of the constraint is matched to the current size of Maya manipulators (which can be changed using the + / - keys).

Locate the attributes of the hinge constraint in the attribute editor for the trapdoor object:



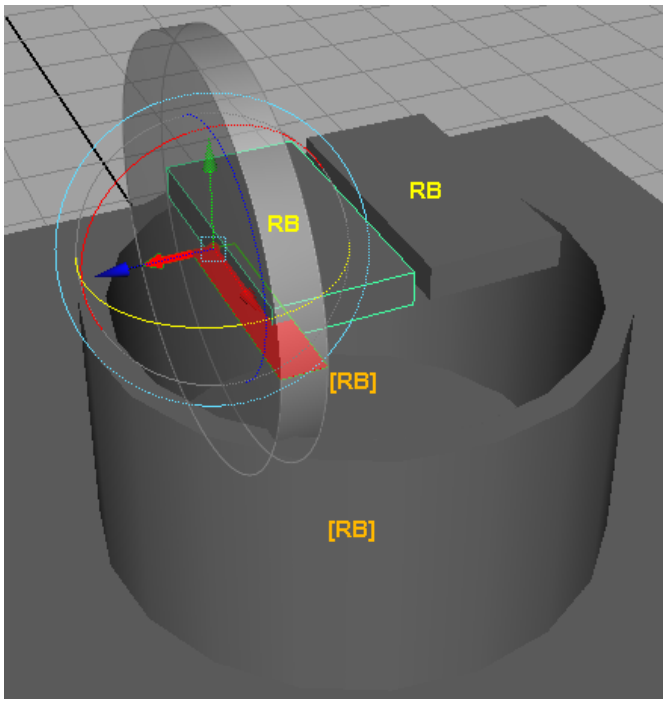

First of all, notice how the **Constrain To** parameter is set to **World** : this means that the constraint will only affect our rigid body, and its position, orientation and limits will be absolute (rather than relative to another (parent) rigid body). This is what we want.

The constraint is represented in the viewport as an axis and a volume of allowed rotation (with another axis specifying where zero-rotation is). By default these axes are aligned to the object's Z and X axis, and is placed at the pivot point. In our case, however, we'd like to place them on the sides and orientate them so the trap door would move up and down rather than sideways. We can change the position of the constraint in two ways: by modifying the object's pivot or by manipulating the constraint spaces.

**Positioning the constraint by modifying the rotate pivot**

With the trap door selected, select Maya's translate tool and press the ' **Insert** ' key - this will put the tool into pivot mode. Then, drag the pivot of the trap door to the side which we want to act as the fixed point of the door.

Press ' **Insert** ' again to put the tool back into normal mode, and active the Havok Physics Tool to visualize the constraint:



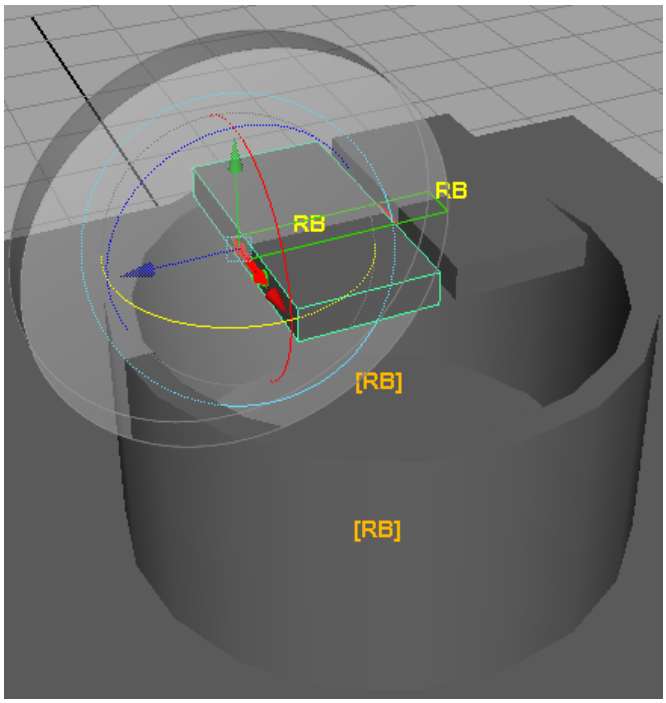

Observe that the constraint's position follows the (rotate) pivot point of the object. However, we cannot change the orientation of the hinge by moving the pivot point - we can only do this by manipulating the constraint spaces.

**Note:**

An option is provided by the Havok Content Tools to 'include rotate pivot in transforms'. If this option is disabled then moving the rotate pivot of an object will have no effect whatsoever on the constraint. See the Export Options for more.

### Positioning/orientating the constraint by manipulating its constraint spaces

Let's now set up the orientation of the hinge constraint for the trap door. Activate the Havok Physics Tool if is not already active, so that we can manipulate the constraint spaces. This constraint provides manipulators in the viewport to change its orientation. Drag the manipulators in the viewport to orient the hinge so that the main axis points along the length of the trap door and the zero-rotation axis (the one within the volume) points towards the other trap door:



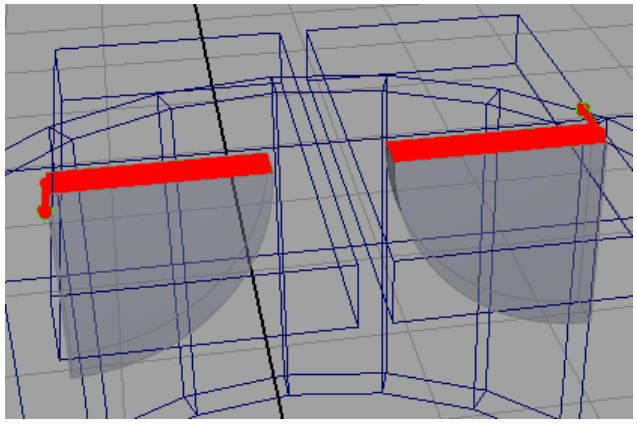

Note that you can also change the position of the constraint by dragging the tripod manipulator. However this is not necessary in this case since the constraint is aligned to the pivot point of an object, which we have moved already.

Repeat the process with the other trapdoor to create a hinge constraint with the appropriate position and rotation. In this case, try positioning the constraint by manipulating the constraint space only, instead of moving the pivot point.

**Limiting the hinges**

By default, hinge constraints allow full freedom of rotation around the main axis. It is often desirable to limit that amount of rotation to a specific range.

Select one of the trap doors, and locate its hinge constraint attributes in the attribute editor. Enable the " **Is Limited** " checkbox. Notice how the **Min** and **Max** parameters are now enabled, and how the limits are reflected in the hinge volume in the viewport (the limits are specified counterclockwise from the zero-rotation axis). Set the limits on each of the hinges to appear as follows:



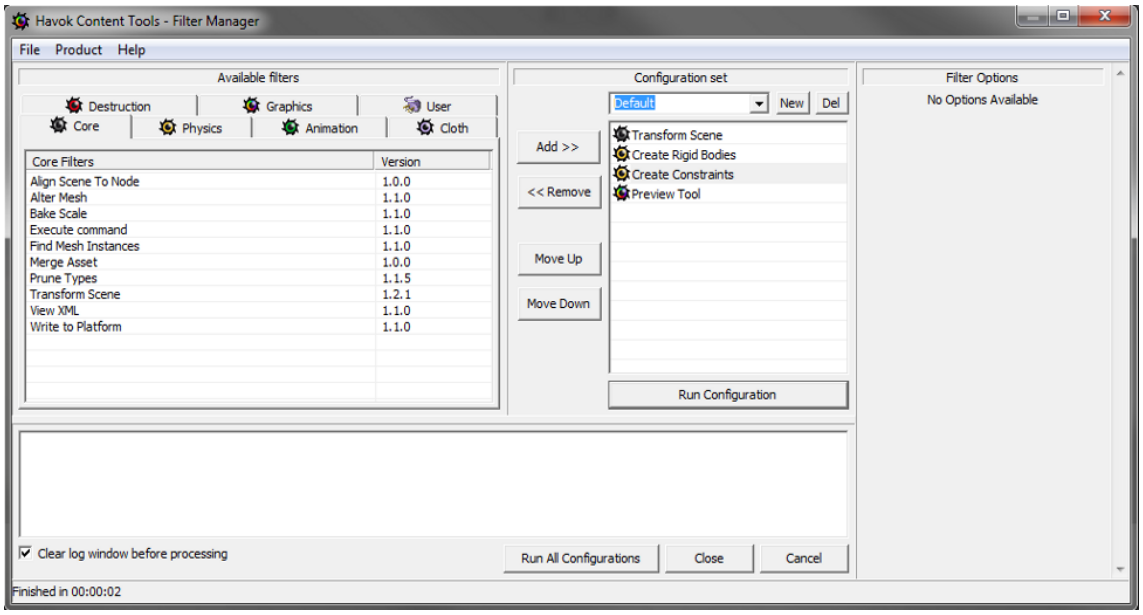
(Note that the Havok Physics Tool is set to **Show All Constraints** for this screenshot)

#### 4.5.6.6 Previewing the Scene

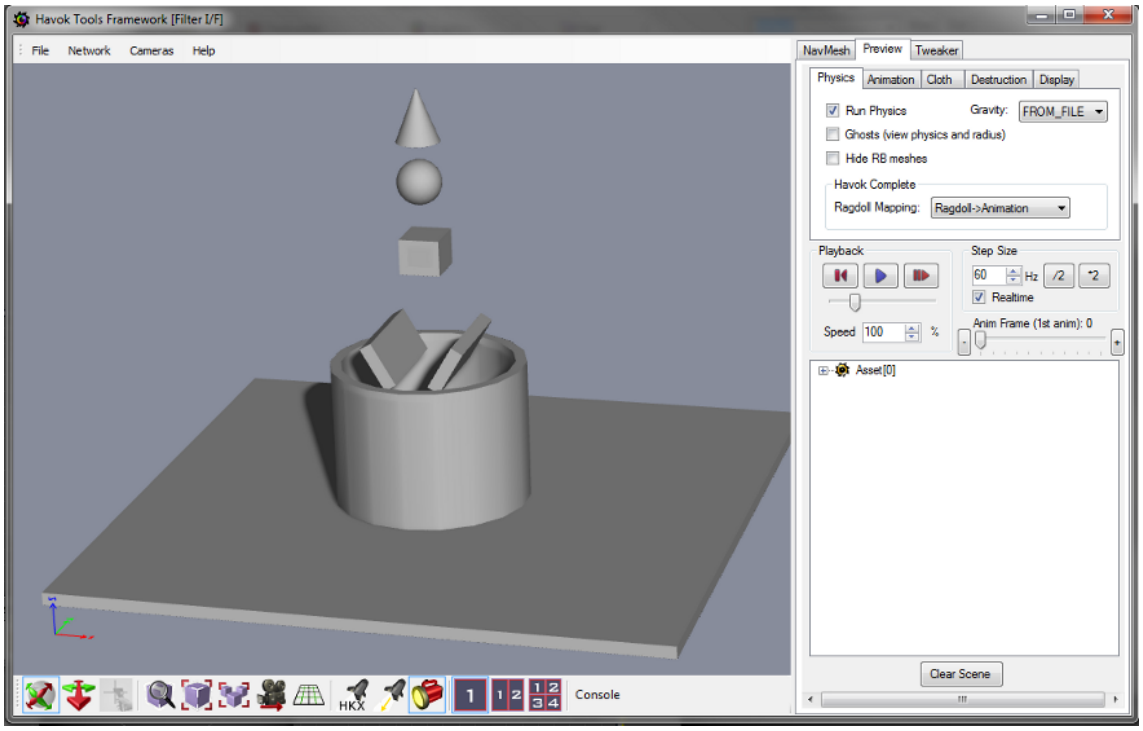
Now that we have set up our constraints, let's see what effect they have on the simulation.

Open the filter manager again by exporting the scene .

We now need to add a new filter, the Create Constraints filter (in the **Physics** category), in order to create constraint objects from the data we set up in Maya. Place it after the Create Bodies filter, and before the Preview Tool filter:




If you now execute the processing by pressing **Run Configuration** , you will see the constraints in action:



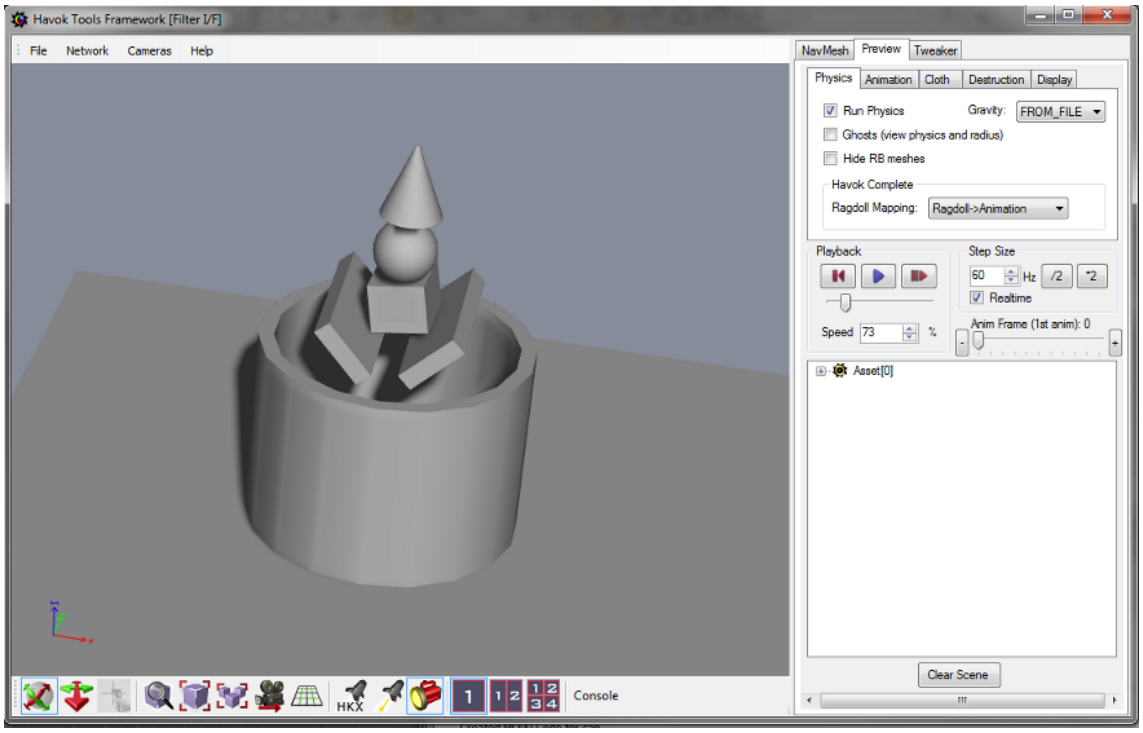

You can check that the limits we applied to the hinges are working by using mouse picking - you will notice that the doors won't move outside the limits we specified.

Close the preview window and the filter manager, and return to Maya.

#### 4.5.6.7 Adding friction to constraints

Our setup is almost finished. The last thing we'd like to fix is the fact that the trap doors open even before the objects hit them (due to gravity). We'd like to add some friction to the hinges so the doors only move when pushed by the objects falling on them. Select each of the trap doors and locate the hinge constraint tab in the attribute editor. Change the **Max Friction Torque** parameter to 40. This parameter specifies how much torque (angular force) is absorbed by the hinge's friction - only torques above this value will cause the hinge to move.

If you now export and preview the scene, you will notice how the trap doors remain closed until the objects fall on top of them.



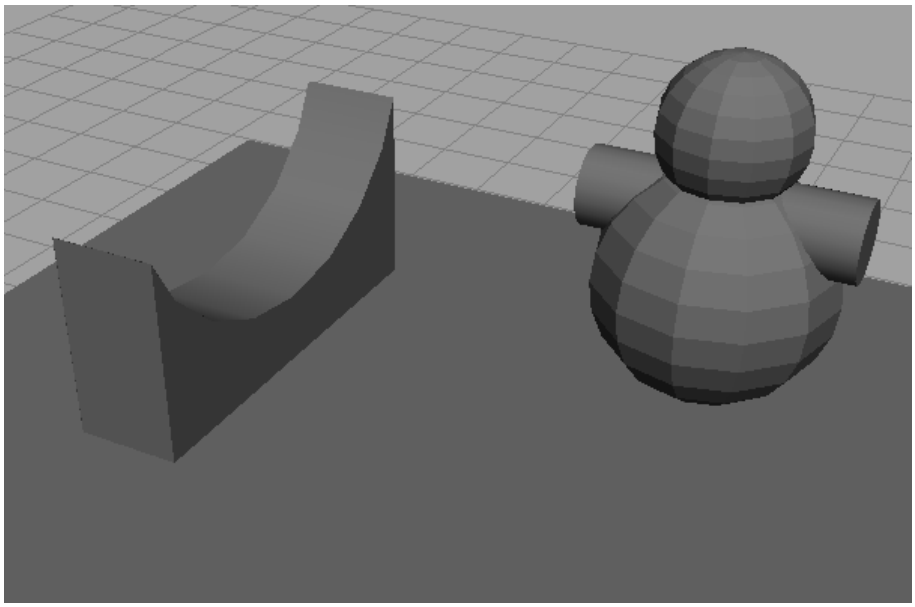

And this finishes our tutorial. You can find the fully completed tutorial in the scene file "`tutorialEnd.mb`". Try varying the properties of the rigid bodies and constraints, and observing the effects by running the filter setup.

### 4.5.7 Tutorial: More on Rigid Bodies

In this tutorial we will explore some more concepts regarding rigid body setup, such as creating compound rigid bodies and changing the center of mass. This tutorial assumes that you are acquainted with the basics of exporting and processing assets and working with rigid bodies. We recommend you follow the first two tutorials (Export and Animation Basics and Physics Basics) before proceeding with this one.

#### 4.5.7.1 Getting Started

Start by loading the scene "`tutorialStart.mb`", located in the "`tutorials/moreOnRigidBodies`" folder of your Maya module ("`Program Files/Havok/HavokMayaModules/...`" by default):

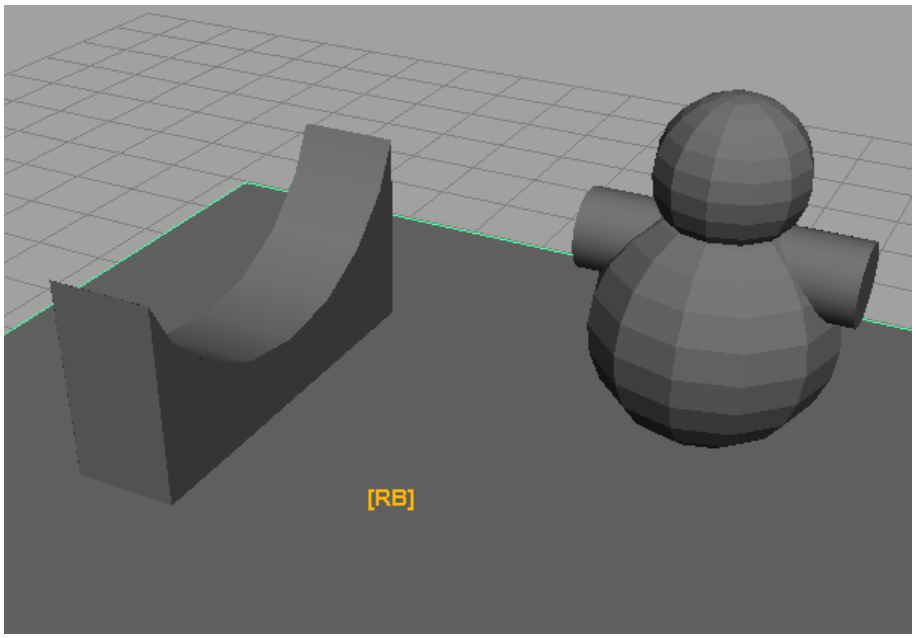


In the scene you will find the following:

- Two spheres and a cylinder, making up the shape of a little toy man
- A ground box
- A concave object (a little ramp)
- Two (hidden) convex objects that together make a shape similar to the ramp

#### 4.5.7.2 Creating a Fixed Rigid Body

Let's start by making a rigid body out of the ground box. Select it and press the **Create Rigid Body** button or menu option.



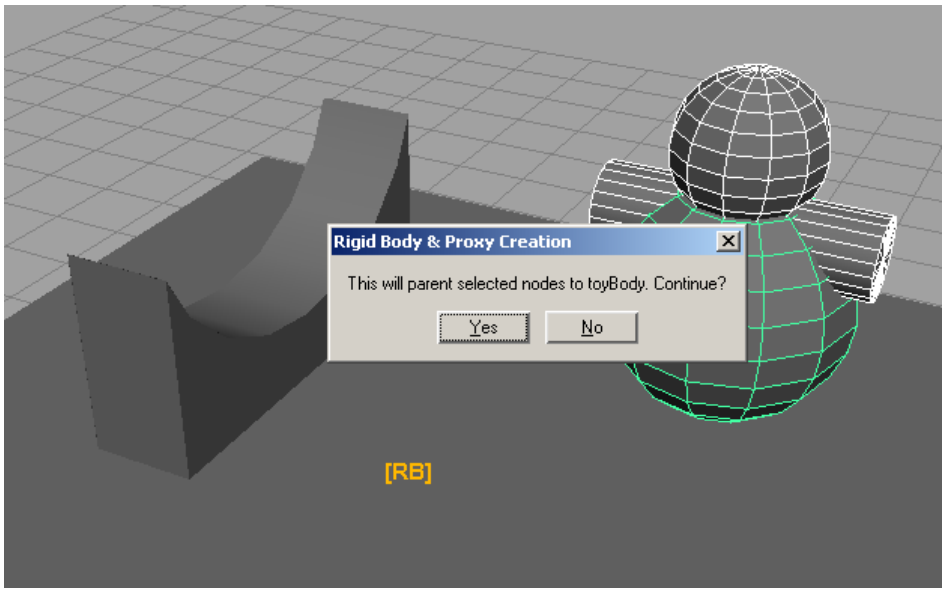

Since the default mass is zero, the ground box will be fixed in space. That's exactly what we want.

#### 4.5.7.3 Creating a Compound Rigid Body

For the toy man we want to create a single rigid body from the two spheres and the cylinder. When we create a rigid body like this out of multiple objects, we are creating a compound rigid body.

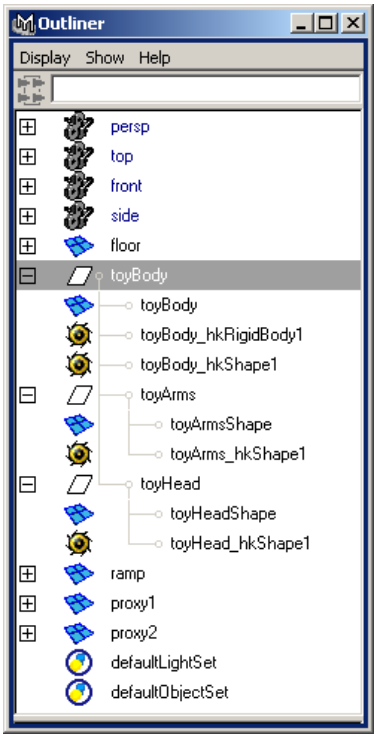
When creating a compound rigid body, one of the objects needs to take the role of the rigid body - this object will hold the rigid body information, and the rigid body will be named after it. The other objects need to be children of that rigid body object. If the objects are not parented already, the last object selected will be considered to be the rigid body object and the other objects will be reparented accordingly.

Select, in turn, the head, the arms and finally the body of the toy (holding SHIFT for multiple selection). Then click on the **Create Compound Rigid Body** button or menu option:

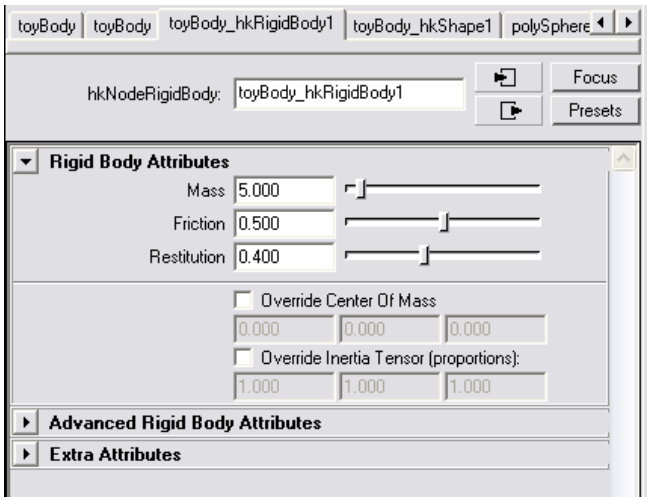



A confirmation dialog appears. This is because, as we mentioned, compound rigid bodies need to form a hiearchy. The Havok Content Tools have assumed that the last one selected (*ToyBody*) should be the parent (the one that will hold the rigid body information) - but is asking us for confirmation before proceeding to reparent the arms and the head. Click **Yes** to continue.

After this, *ToyArms* and *ToyHead* will be parented to *ToyBody*. Shape information (a Havok Shape Node) will be added to all three objects (as the three of them are used for collision detection). Rigid Body information (the Rigid Body Node) will only be added to *ToyBody*. You can verify this setup using the outliner:



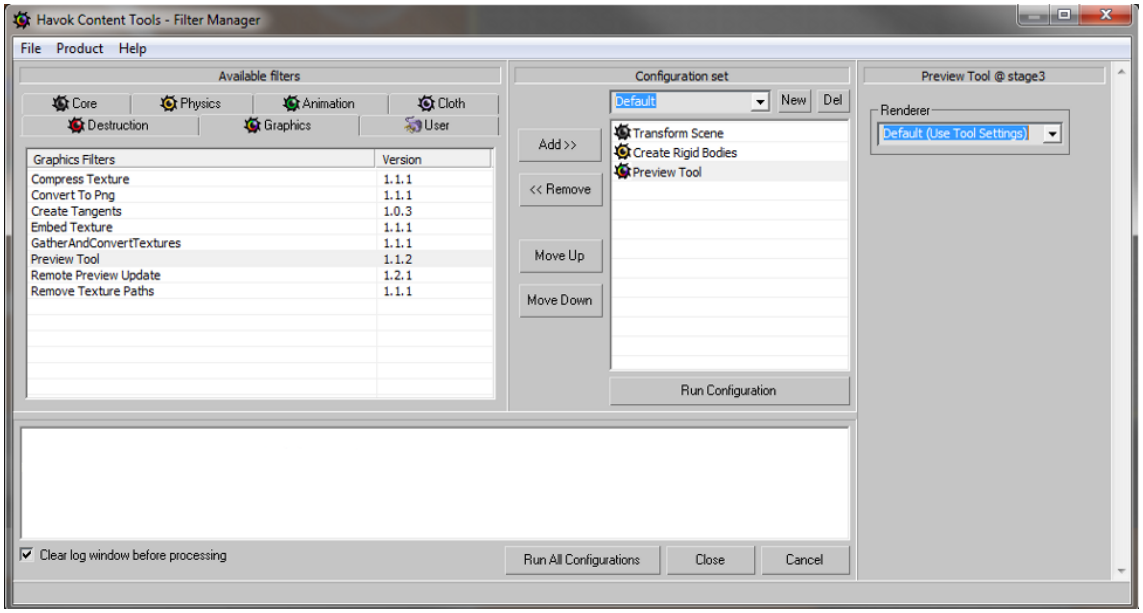
Notice that the rigid body has a default mass of zero. Since we want it to be dynamic, select the attribute editor tab for the rigid body node and set the mass to a different value, let's say 5 Kg:



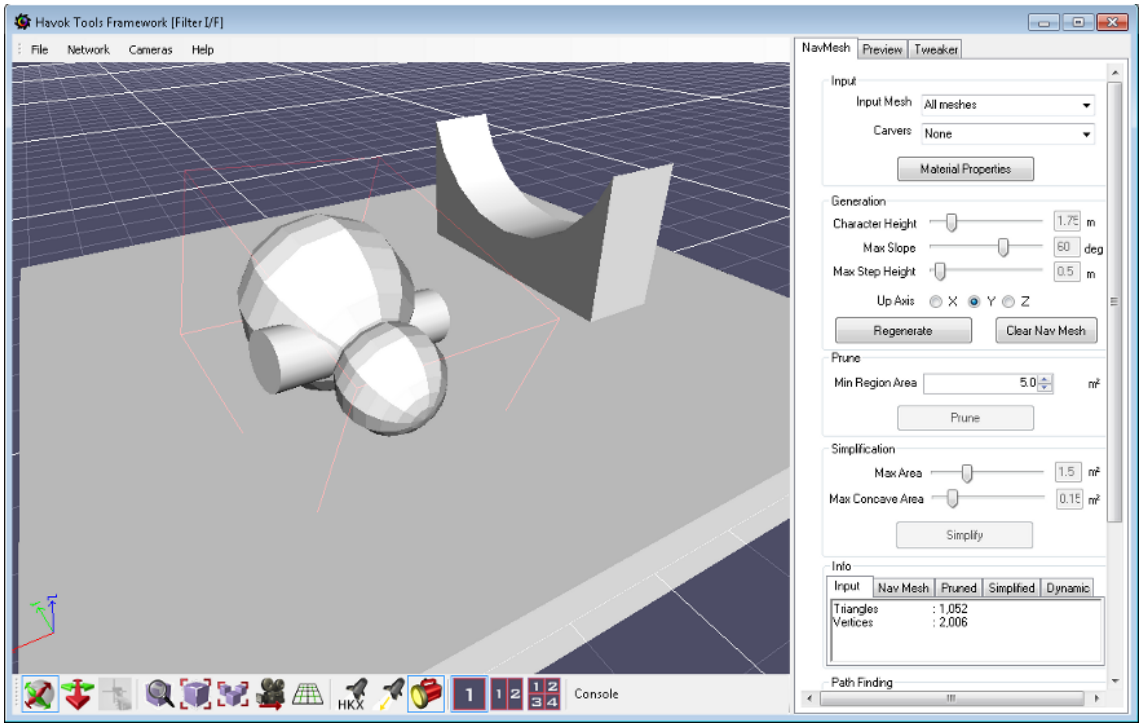

It is also useful to locate the attributes of the shape nodes: notice how the arms of the toy have the *Shape Type* parameter already set to " **Cylinder** " and the head to " **Sphere** ". The Havok Content Tools will automatically detect some Maya shapes and set the appropriate Havok Shape Type for them when creating rigid bodies - however it is still useful to always check the shape type value to ensure that it is as expected.

### Previewing the Scene

Let's now preview the behaviour of our compound rigid body. Click on **Export** and, in the filter manager, add the Transform Scene (using the **SCALE_CMS_TO_METERS** preset), the Create Bodies and the Preview Tool filters (as we did for the previous tutorial):



Click on **Run Configuration** - this will process the asset and open a preview window:



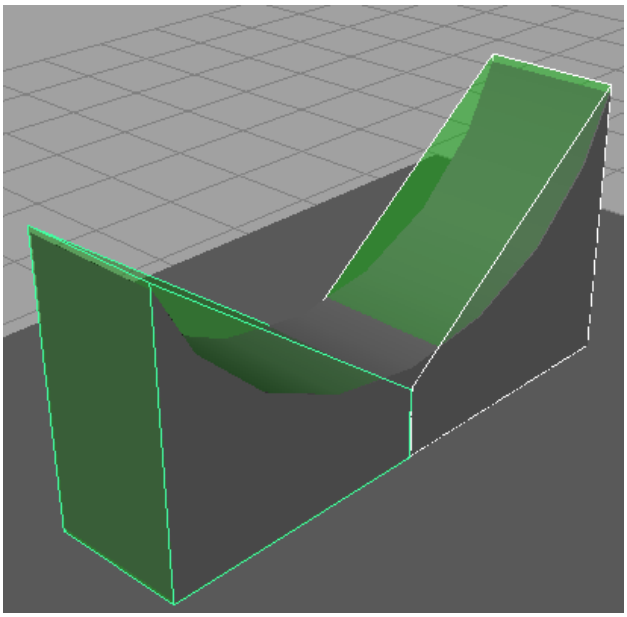

As you can see (use mouse picking to interact with the toy man) the three objects behave as a single rigid body.

Close the preview and the filter manager and return to Maya.

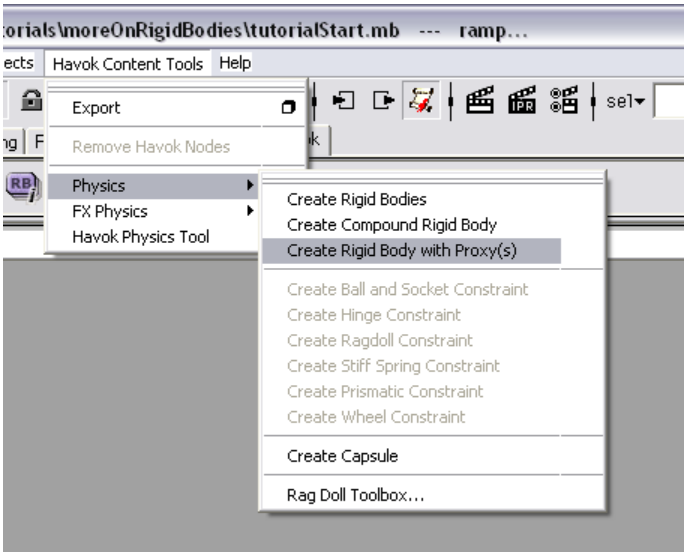
#### 4.5.7.4 Creating a Rigid Body with Proxy Shapes

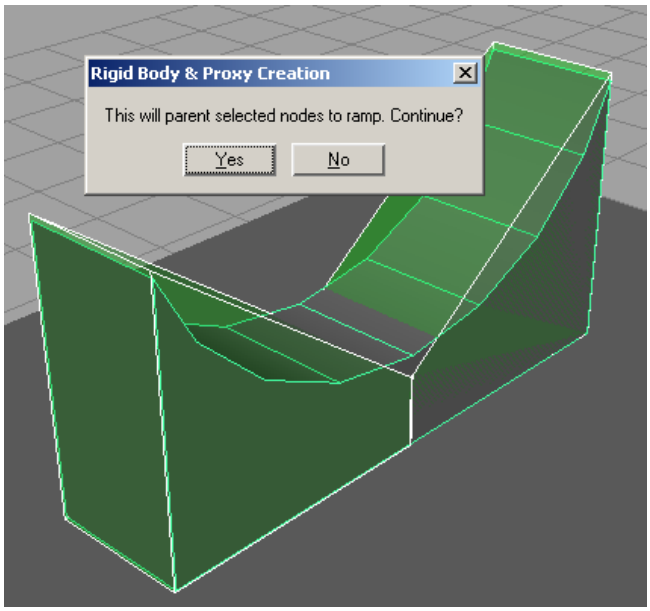
Sometimes we may want to simulate an object as a rigid body, while using another object or set of objects for the collision detection rather than the mesh of the object itself. The usual example of this case is when we may want to simulate a concave object as the combination of multiple convex pieces. Creating a rigid body with proxy shapes is very similar to creating a compound rigid body. The only difference is that, in this case, the object containing the rigid body information (the rigid body node) does not contain shape information (the shape node) since its mesh is not used for collision detection. Shape information is instead taken only from its descendants.

Let's see this in action in this scene. Notice how the white ramp is a concave object. Ideally, we'd like to approximate its rigid body shape it using several convex objects (since simulating convex objects is far more efficient than simulating concave objects). We will use two convex pieces provided to create a proxy shape representation of the ramp. Locate and unhide the two proxy objects (*proxy1*, *proxy2*), then move the objects so the two pieces and the concave ramp match their locations in the scene. We have coloured the proxy obejcts transparent green to make it easier to distinguish them:



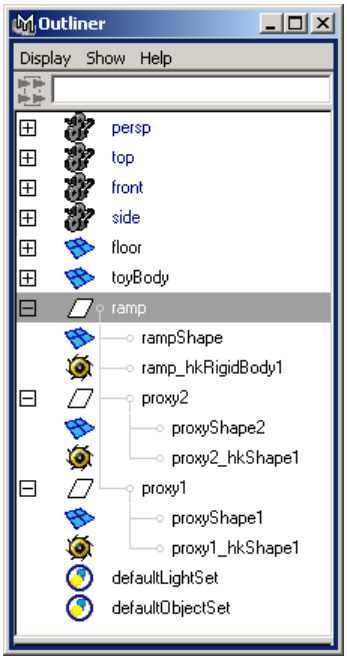
We now want to convert the ramp to a rigid body while using the two convex pieces as the shapes. So, select the two green convex shapes first and then the concave ramp, then click on the **Physics** > **Create Rigid Body with Proxy(s)** menu option.






In the same way as for compound bodies, the rigid body object must be the parent of the shape objects - the dialog above notifies that the proxies must be reparented. Click on **Yes** to confirm. (Alternatively, start by first parenting the two convex pieces to the ramp manually, and then with the three objects selected click on **Create Rigid Body with Proxy(s)** . )

Notice how the *ramp* object now has a rigid body node only; and how the two convex pieces (which are now children of it) have a shape node only:

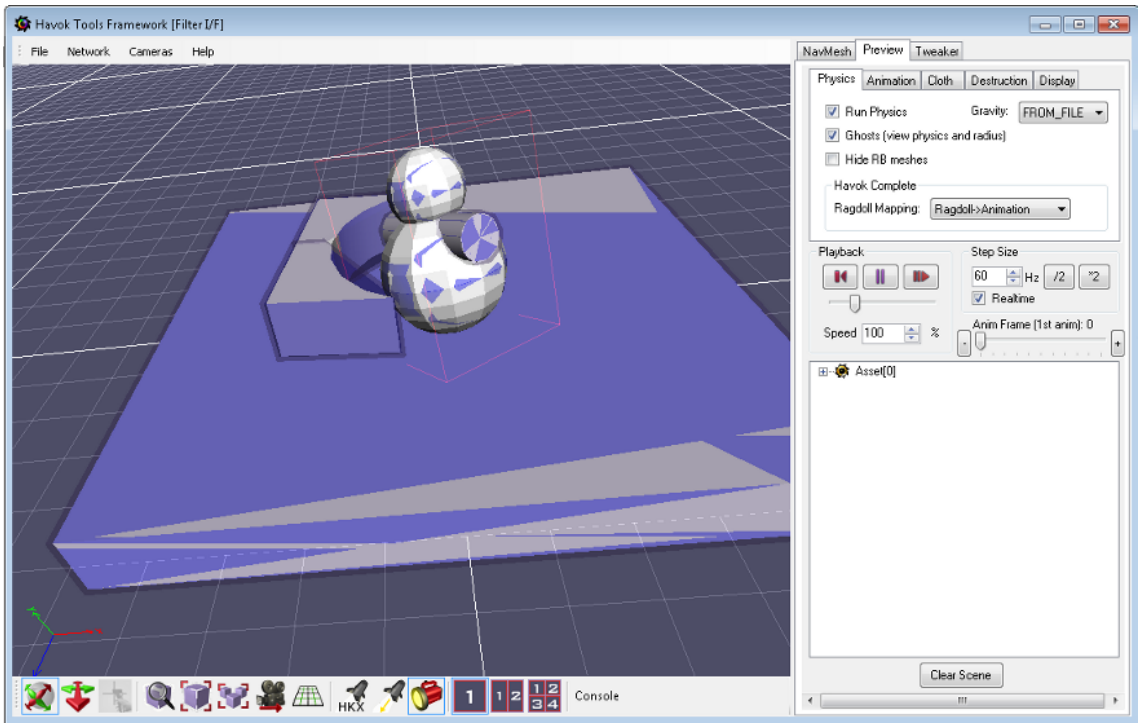



Since the two convex pieces are neither boxes, spheres, capsules or cylinders, their *Shape Type* attribute should be set to " **Convex Hull** ". Ensure that this is indeed the case. Also notice how, again, the

default mass of our rigid body is zero. Since we want it to be dynamic (movable), it should be set to something different (5Kg for example).

**Previewing the Scene**

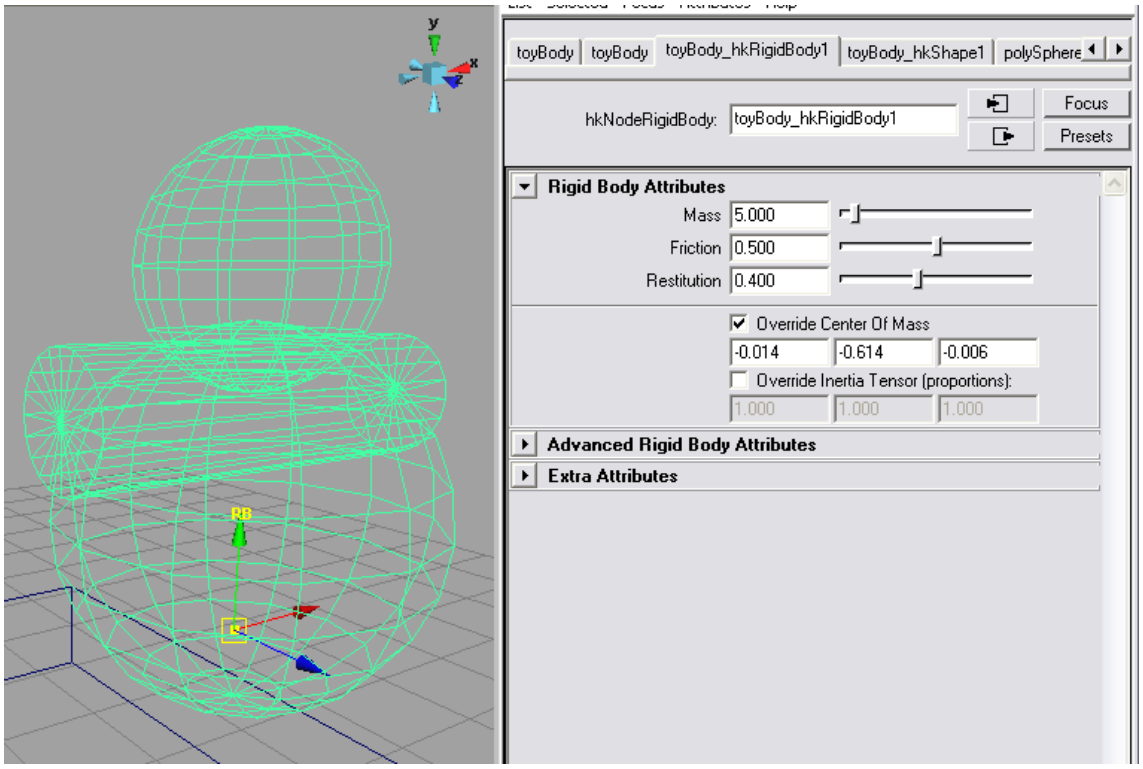
Let's see how the ramp behaves in our preview. Export the scene and press on **Run Configuration** to execute the preview. You should be able to see how the ramp is simulated using the convex pieces. To view exactly what is simulated, enable **Physics Ghosts** in the **View** menu of the preview. Notice how the two convex pieces, and not the concave mesh, is being used to simulate the ramp:




#### 4.5.7.5 Changing the Center of Mass

We'd like our little man to behave like a tumbler - one of those toys that keep themselves straight. Those toys work by having a very low center of mass (usually by having a lead weight inside them). The Havok Content Tools allow you to easily change the center of mass of rigid bodies.

In Maya, select the *ToyBody*'s rigid body node, and in the **Mass** section of the attribute editor, check the **Override Center Of Mass** check box. This enables editing of the local center of mass coordinates. If the Havok Physics Tool is enabled, the center of mass is represented as a point in the viewport which can be manipulated by dragging it:

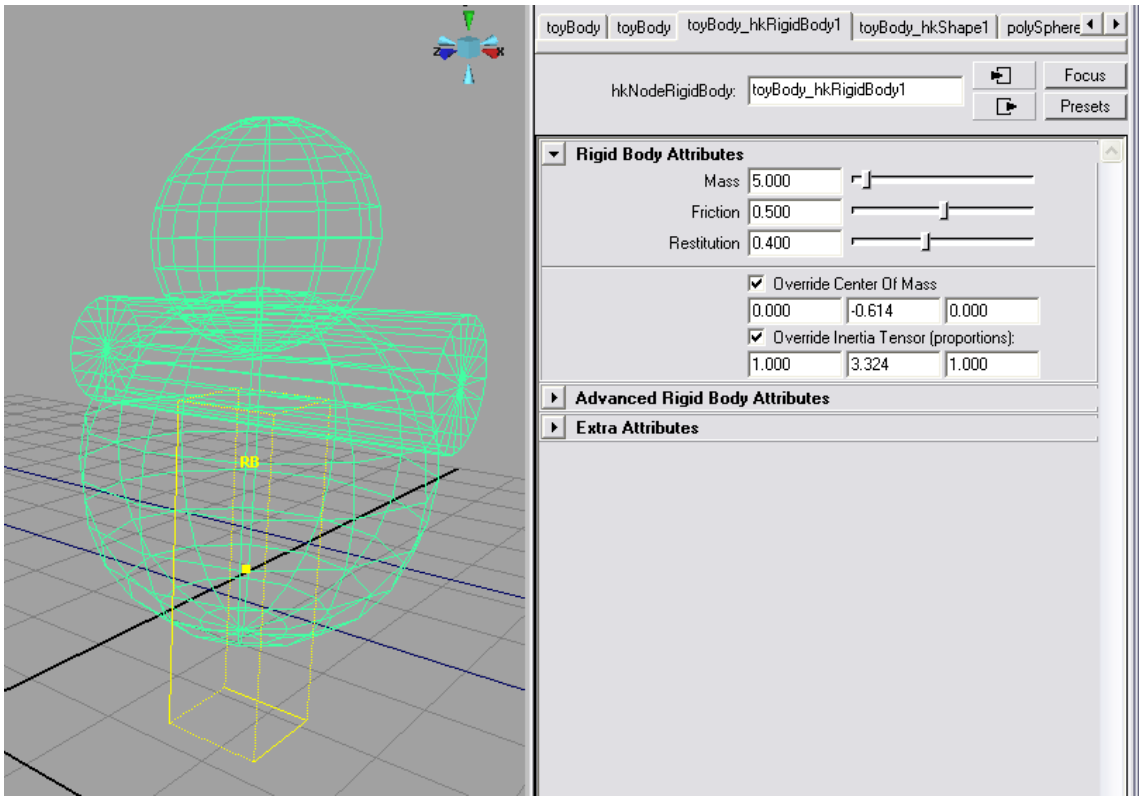



Postion the center of mass towards the base of the toy. If you execute the preview again (export and press **Run Configuration** ), you see how our little man now keeps himself upright as he is dragged around the scene.

#### 4.5.7.6 Changing the Inertia Tensor

We'd now like to customize the motion of our little man even more by changing the proportions of its inertia tensor.

Select its rigid body node again, and check the box **Override Inertia Tensor (proportions)** . This enables editing of the inertia tensor proportions of the rigid body. Again, the Havok Physics Tool provides a visual representation of the inertia tensor proportions - in this case as a scaleable unit cube which can be manipulated by dragging the scale handles:




If you now preview again, you will see that our little man will tend to wobble more around this vertical direction, while it won't lean as much as before. Experiment further with the center of mass and inertia tensor values, observing the results using the preview.

And this finishes our tutorial. You can find the fully completed tutorial in the scene file "`tutorialEnd.mb`".

### 4.5.8 Tutorial: Rag Doll Setup

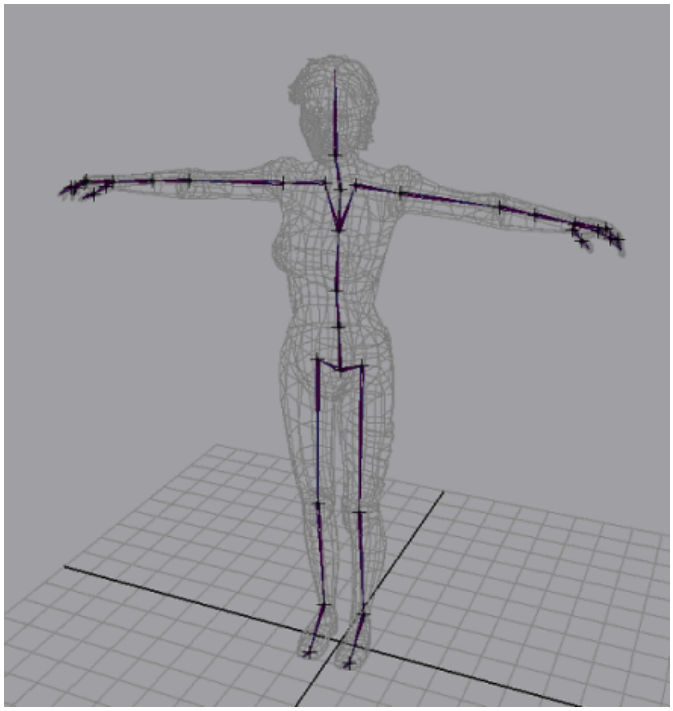
The Rag Doll Toolbox allows the user to create and edit a new constrained rigid body hierarchy, to match to an existing bone hierarchy. In this walkthrough we will:

- Create and associate rigid bodies with an existing skeleton using the Rag Doll Toolbox
- Create appropriate constraints between the rigid bodies using the Rag Doll Toolbox
- Export and test the rag doll, using appropriate filters in the Havok Filter Manager

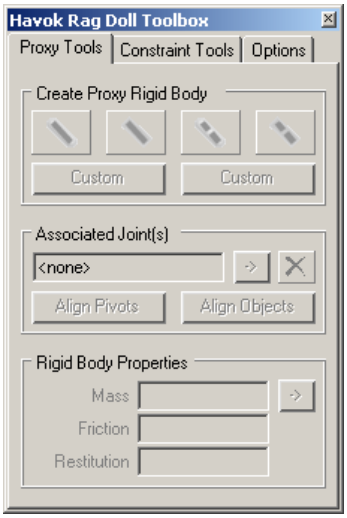
**Note:**
Maya tends to work in terms of 'joints' rather than 'bones' - in this tutorial the two terms may be used interchangably. Please read Issues with Maya Joints for more related information.

#### 4.5.8.1 Getting Started

The sample files used in this tutorial may be found in "`tutorials\ragDollToolbox`", under the same location as your installed Havok Maya modules (normally "`Program Files\Havok\HavokMayaModules\...`"). Open the **"havokGirl1.mb"** sample file – this contains a skeleton which has been skinned with a polygon mesh:



Open the Rag Doll Toolbox via the main **Havok Content Tools** menu or the toolbar icon :

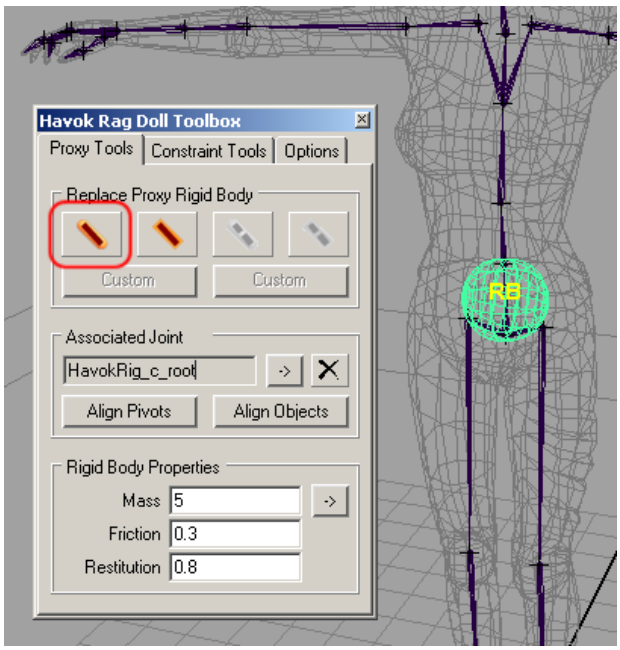


#### 4.5.8.2 Creating Rigid Bodies

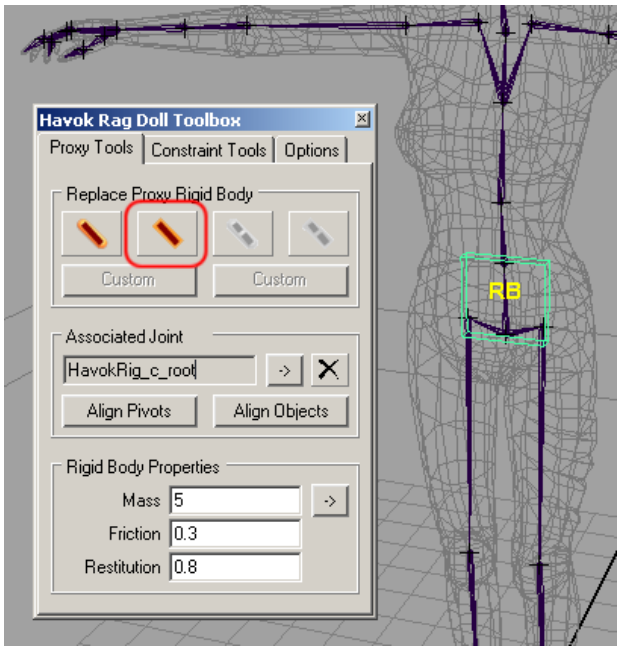
We will now create a set of rigid bodies to represent the physical rag doll for this character's bone structure.

**Simple Proxies**

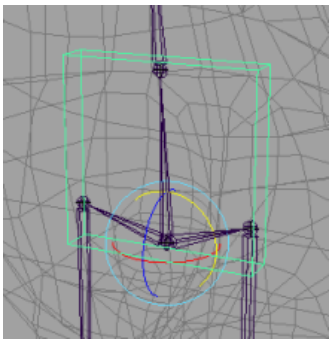
With the **Proxy Tools** tab of the Rag Doll Toolbox open, select the "*HavokRig_c_root*" pelvic joint. Since a single unassociated 'bone' is now selected, the first set of buttons in the "Create Proxy Rigid Body" section of the toolbox becomes enabled, allowing rigid body proxies to be created for the selected bone. Click the first button to create a new capsule proxy for the pelvic bone. A new capsule shape will be created to fit around the joint and its children, with Havok rigid body and shape nodes already attached, and initialized to a set of default values:




If you would prefer to use a box proxy instead, simply select the newly created capsule proxy and click the second of the two enabled buttons. This will replace the proxy with a box version, while retaining any rigid body properties you may have changed:




Observe that in each case above, the new proxy is created unparented (so as not to pollute the skeleton hierarchy). Observe also that the rotate pivot point of a new proxy automatically becomes aligned to that of the joint:



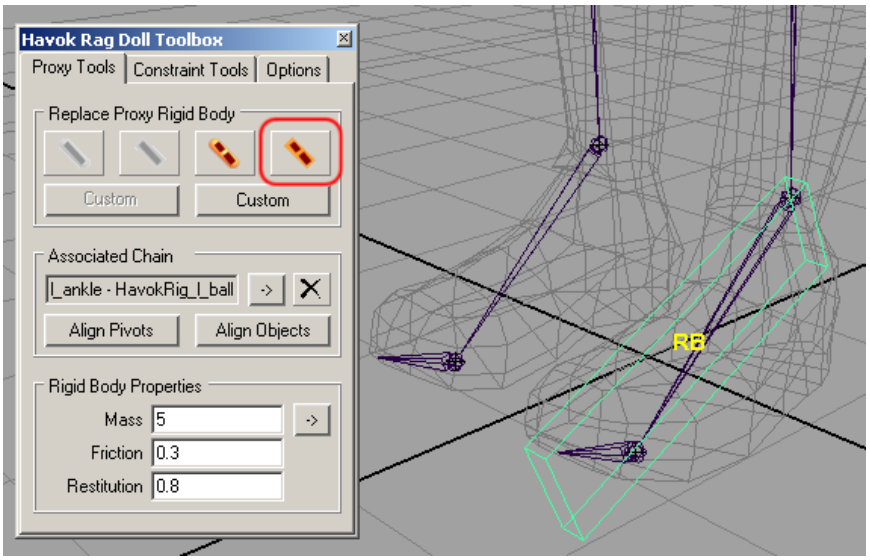
**Note:**

A bone can have at most one associated proxy. Therefore, if any selected bone already has an associated proxy then all buttons to create a new proxy will be disabled.

### Chain Proxies

Suppose that you would like to create a single proxy to represent a chain of bones, for example a foot in this example. Select the two bones representing a foot (holding SHIFT for multiple selection)[5]. When a chain of bones is selected and none of them already has an associated proxy, the second set of buttons in the "Create Proxy Rigid Body" section of the toolbox becomes enabled. Click one of the 'chain' buttons to create a capsule/box proxy for the selected bones:

[5] Unfortunately Maya highlights all descendants of selected nodes in the viewport. This can make it difficult to see which bones are directly selected while creating chain proxies. Selecting nodes using the outliner panel helps if this is causes difficulty.

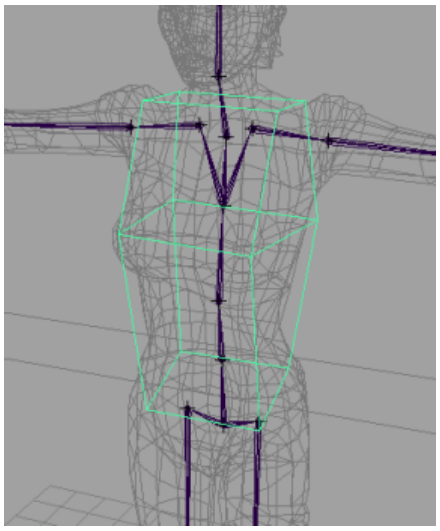



Observe that as before, the new proxy is created with the world as its parent. Also observe that in this case the rotate pivot point of the new proxy automatically becomes aligned to that of the most senior joint in the selection.

Observe also that the new proxy is not yet oriented or positioned in a useful way. This will be fixed later in this tutorial.

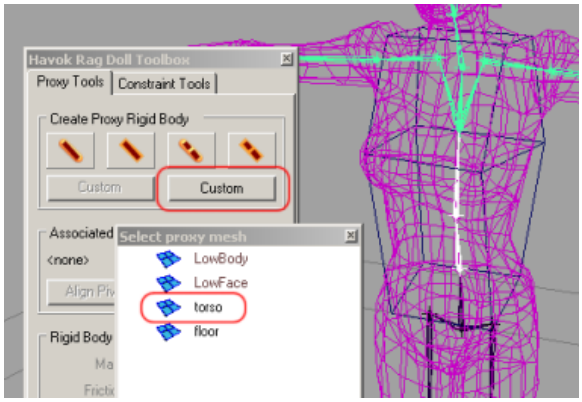
### Custom Proxies

Sometimes a box or a capsule is not enough to accurately represent the volume around a bone or a chain of bones. For example, take the torso region of this model. In this case, instead of creating a box/capsule using the toolbox, we will first create a custom mesh and then associate it with the spine bones. Create a new polygon mesh in world space to represent the character's torso, as shown. Use any of Maya's standard tools to create the mesh:



To associate this mesh as a chain proxy for the spine bones, select the following three bones

(holding SHIFT for multiple selection): "*HavokRig_c_back_1*", "*HavokRig_c_back_2*", and "*HavokRig_c_back_3*". Now click the " **custom** " button in the toolbox - this will open a selection window where you can pick the existing mesh to use as a proxy. Select your custom mesh and click "OK":




This mesh has now been associated as a proxy for the set of spine bones, and behaves like any other chain proxy - i.e: it inherits default Havok properties; it can be replaced by clicking the other chain proxy buttons; its pivot is aligned to the most senior of the associated joints, etc.

**Reshaping/resizing Proxies**

After a proxy has been created, it probably won't accurately represent the skin volume. To correct this, the proxy can be moved/rotated/etc. using the standard tools to achieve the desired fit, with one caveat: **non-uniform scaling should be avoided** . The reason for this is that the proxy meshes used in the modeler act as guides for creating the rigid body shapes (based on the 'Shape Type' attributes). If any non-uniform scale is introduced into the proxy meshes then the exported rigid bodies may not accurately match the proxies displayed in the modeler. This is true for capsules in particular.

Therefore the preferred method of editing a proxy is to use the attribute editor to change the construction attributes (width/height/etc.) and use the translate/rotate tools to position the proxy.

As observed, whenever a new proxy becomes associated with a bone, its rotate pivot point is automatically aligned to that of the bone. After moving or rotating a proxy, the **Align Pivots** button in the 'Associated Bone' section of the toolbox **should always be pressed** to ensure that the proxy pivot matches the joint position. This helps to create accurate rag dolls during the filter process.

Reshape and reposition any proxies created so far to achieve a good fit with the skin that they represent.

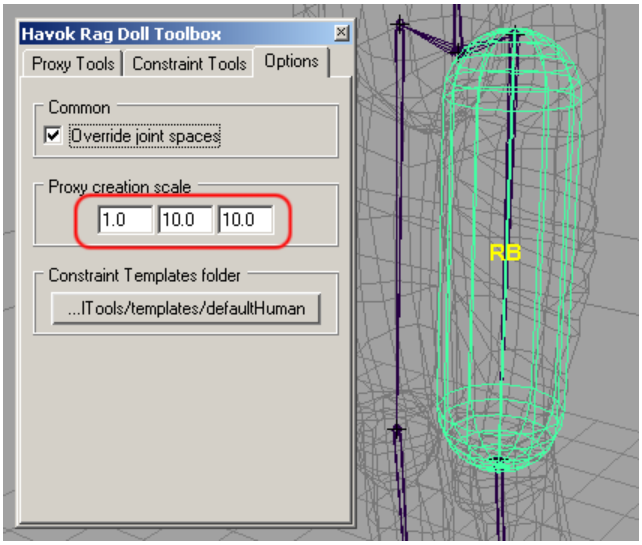
**Using the 'Proxy Creation Scale' option**

If your character's bones are much narrower than the volume of your character's skin, as in this case, then most new proxies which you create using the Rag Doll Toolbox will require reshaping to match the skin volume. The 'Proxy Creation Scale' option is provided to help this scenario.

This option sets a scaling factor which is used during the creation of new capsule/box proxies. This can be thought of as an additional local [X,Y,Z] scale applied to a bone before a new proxy is created to fit around it., where 'X' is **usually** the primary bone axis (see Maya Joint Issues).

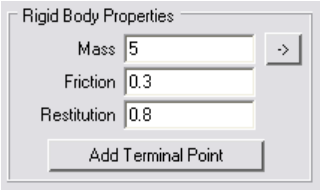
For example: in the *Options* tab set the values to [1,10,10], then in the *Proxy Tools* tab create an new capsule proxy for a thigh bone. This results in a proxy which is ten times wider than the bone but has

the same length:




### Rigid Body Properties

The "rigid body properties" section of the Proxy Tools tab allows quick access to rigid body properties of selected proxies. As you create proxies, think about these properties and try to set appropriate values for each proxy:

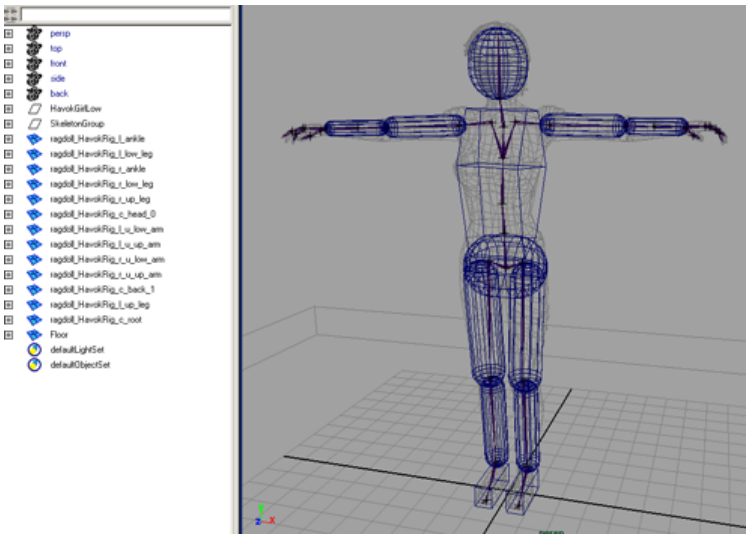



These properties can also be applied in bulk, for example to set the friction property for all of your proxies you can select them all together and input some value into the 'Friction' field above.

Apply suitable properties to the rigid bodies as you create them. For instance, the torso would be a lot heavier than a foot.

### Finishing with rigid bodies

Continue creating, reshaping and positioning proxies for this character's skeleton. Remember to align the pivots once you have positioned your proxies, and to set appropriate rigid body properties. You may wish to open the sample " **havokGirl2.mb** " file - this contains the original skinned skeleton with rigid bodies created and associated to most of the bones:



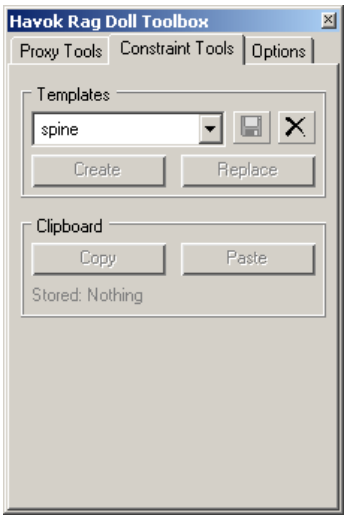

Examine the proxies before continuing. Points of interest:

- All proxies are parented to the world – the skeleton hierarchy is unchanged
- All proxies have their pivot points aligned to their associated bone(s)
- Some proxies are associated to bone chains
- The torso uses a custom (convex) mesh
- Masses differ for each proxy – e.g. the torso is heavier than the feet
- Some proxies overlap, others don't - there is no requirement either way
- Some bones are not associated (e.g. hands). We don't require that amount of rigid body detail for this tutorial. Feel free to add additional proxies if desired.

You may export and preview the scene before continuing if desired. An appropriate filter setup is included with the scene. On previewing, the rigid bodies fall to the floor and are not connected to each other or to the skin, as expected.

#### 4.5.8.3 Creating Constraints

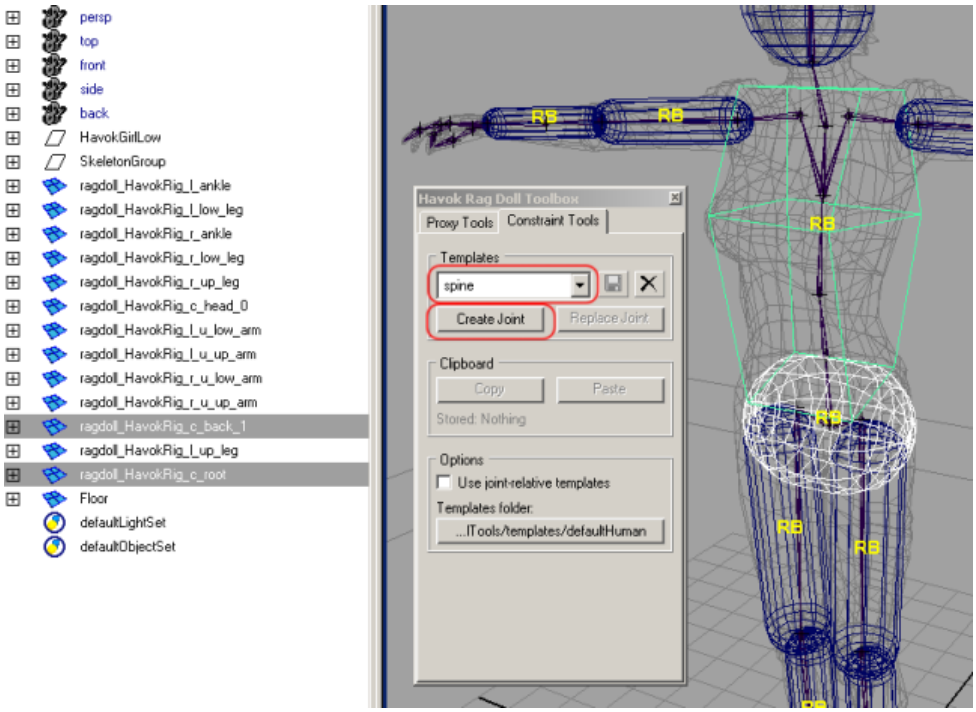
At this point, you should have a scene with a skinned skeleton and a set of rigid proxies associated with the bones. To finish setting up our rag doll, we need to organize the proxies into a constrained hierarchy. The **Constraint Tools** tab of the Rag Doll Toolbox contains the tools to achieve this.




### Applying Constraints

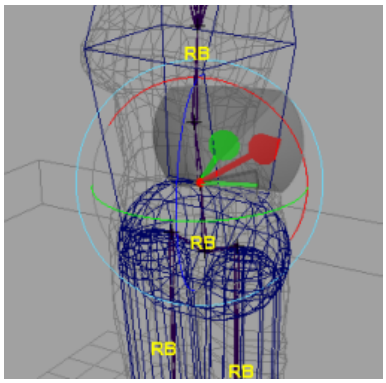
Templates are representations of Havok constraints, saved as text files. For instance a "knee" template might describe a limited hinge constraint with specific limits and other attributes. They provide a useful way of wrapping a constraint type and attributes together in a re-usable way. Templates are stored together in a specific directory (which is customisable via the 'Options' tab). The Havok Content Tools installer installs a set of default templates for human characters, in the default template directory. We will use these templates.

To apply a template, a pair of rag doll proxies must first be selected. Select the pelvis and torso proxies together, then select the "Spine" template from the drop-menu and click the **Create** button:



Observe that several things happen at this stage:

1. The two proxies are automatically re-parented into the correct hierarchical order
2. A constraint node is added to the child proxy (the torso)
3. The constraint's properties are set from those stored in the template




The new constraint may not be oriented correctly initially. This is due to a known issue with Maya - see Maya Joint Issues for more.

**Modifying Constraints**

Once a template is applied, you can always modify its properties by changing the constraint parameters using the attribute editor panel and/or the Havok Physics Tool. You can then save (and possibly replace) the constraint as a new template by clicking on the " **Save** " icon beside the template combo box.

**Replacing Constraints**

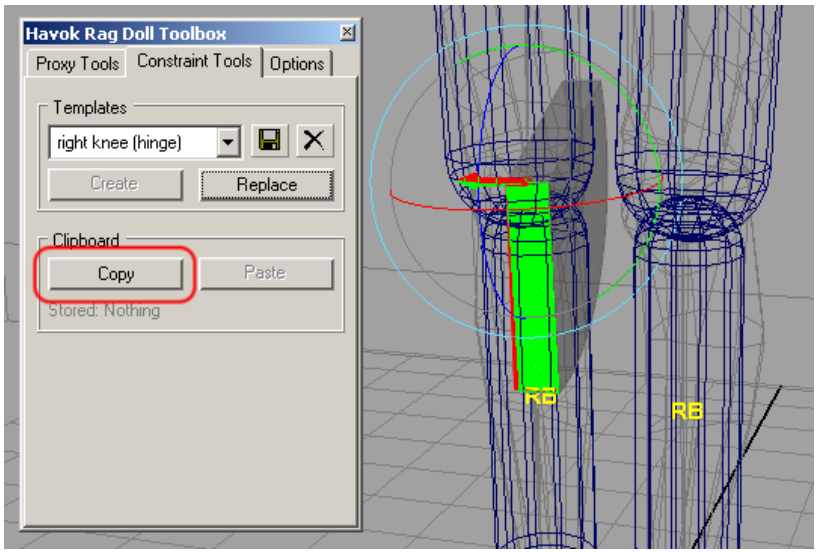
To replace a constraint in the scene which was added using templates, we need to select only the child proxy which contains the constraint node (the torso in this scenario), select a new template from the drop-menu, and click the **Replace** button. In this case no re-parenting is performed, since the proxies should already be in their correct hierarchal order from the first application of a template.

Try replacing the spine constraint with various other templates.

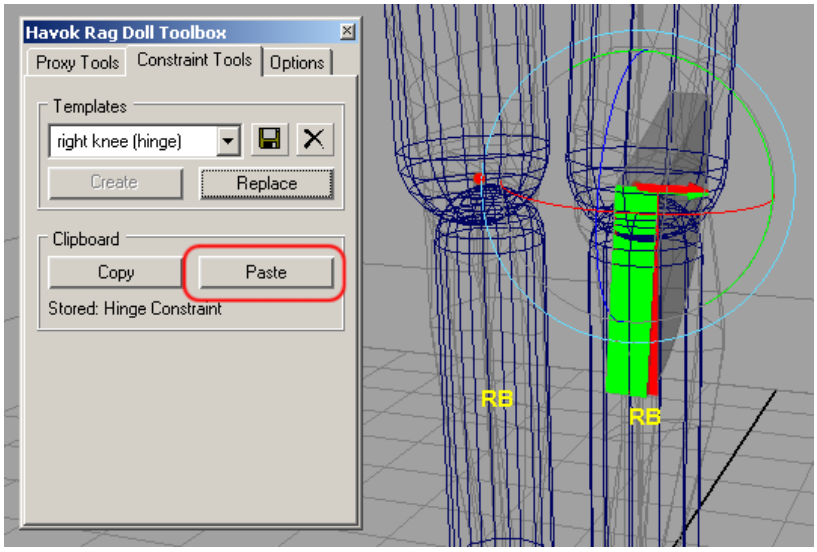
**Copy and Paste**

Symmetric characters usually have similar constraint properties on their left and right side. The copy and paste mechanism of the toolbox is useful in this case.

For example, create a "knee" constraint between one of the thighs and its calf. Fix the orientation if necessary. Now, tweak some attributes of the constraint (limiting angles, max friction torque, etc.). When ready, press the **Copy** button. This generates a templated version of this constraint in memory.

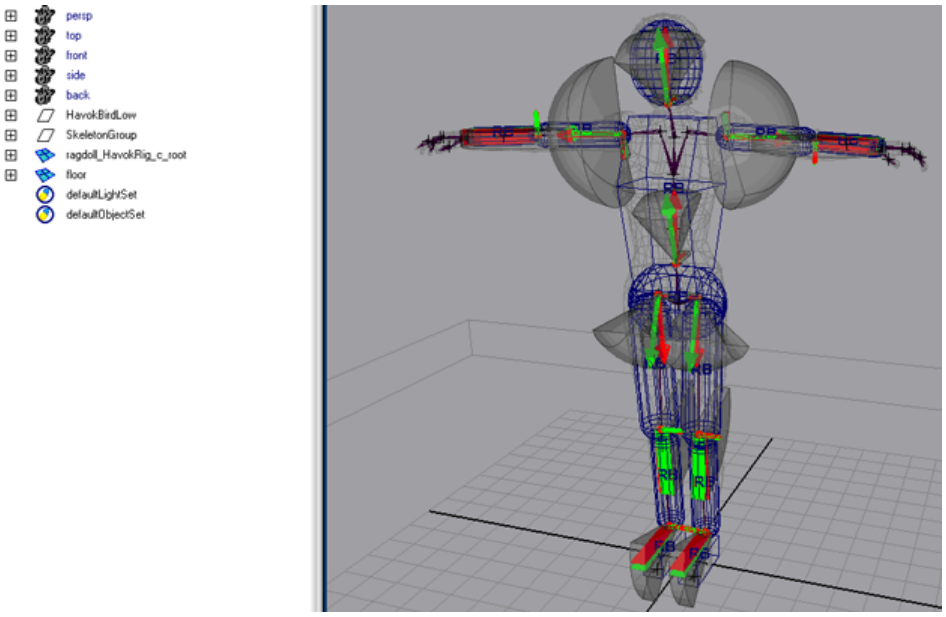



To apply this in-memory template to the other leg, simply select the other thigh and calf together and press the paste button. This acts in the same way as creating a new constraint using the 'Create' button, but uses the in-memory template instead. Unfortunately this may also require some manual fix-up of the orientation.




### Finishing with Constraints

Continue applying templates to each pair of proxies, until the rigid body hierarchy is complete. You may wish to open the sample " **havokGirl3.mb** " file - this contains the skin, skeleton, and the constrained proxy hierarchy.




This screenshot is taken with the Havok Physics Tool active and set to show all constraints.

Examine the scene before continuing. Points of interest:

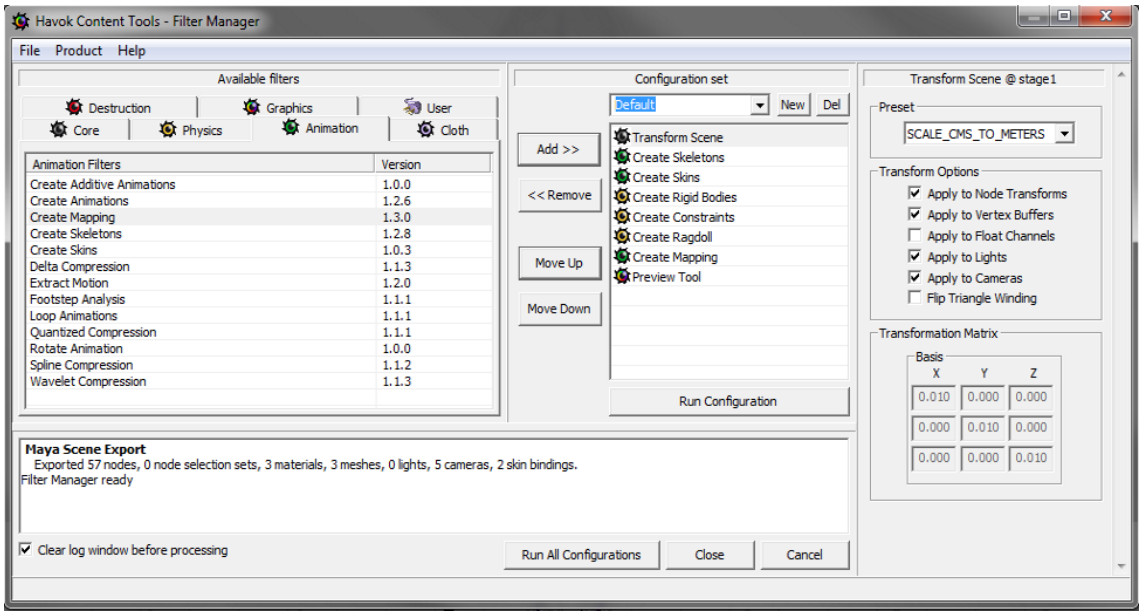
- The proxies now form a single hierarchy, matching that of their associated bones
- Each proxy has a single constraint (except for the root) , with its hierarchical parent set as the constraint parent

#### 4.5.8.4 Exporting the Rag Doll

At this stage your scene should contain at least:

- The original skeleton hierarchy
- The original skin
- The new constrained proxy hierarchy

Export the scene to the filter manager using the Havok exporter. The sample scene includes the appropriate set of filters to create the rag doll. These are:




- **Transform Scene**

  Maya exports everything in cm's and Havok works in meters, so for Maya we always need a SCALE_CMS_TO_METERS transformation.

- **Create Skeleton**

  Creates a skeleton object from the set of bones. Choose 'Skin bindings' in the 'Build Rig' section.

- **Create Skin**

  Creates a skin object for the skeleton.

- **Create Rigid Bodies**

  Creates the rigid body objects from the proxies we created (and the floor).

- **Create Constraints**

  Creates the constraint objects which constrain the rigid bodies together.

- **Create Rag Doll**

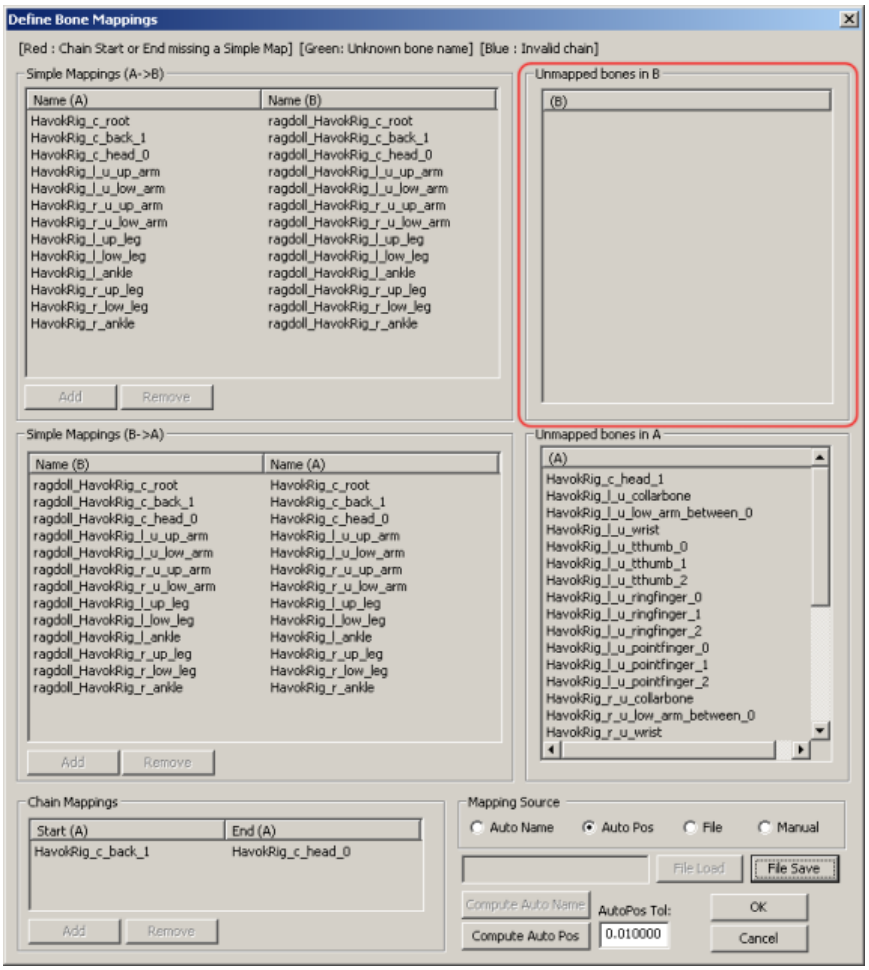
  Collects the rigid bodies and constraints into a rag doll object. Choose ' **Automatic** ' in the ' **Detect Rag Dolls** ' section.

- **Create Mapping**

  Maps the skinned skeleton to the rag doll object, so that the rag doll can drive the bones (which in turn drives the skin).

  For ' **Skeleton A** ', pick '`HavokRig_c_root`' (the skeleton) For ' **Skeleton B** ', pick '`ragdoll_HavokRig_c_root`' (the rag doll)

  Click on ' **Define Mappings** ' to check the mapping:




For a successful mapping, all of the bones in one skeleton should be mapped to (some of) the bones in the other skeleton. In this case we can see that The are no unmapped bones in the 'B' skeleton (the rag doll) so this is a successful mapping.

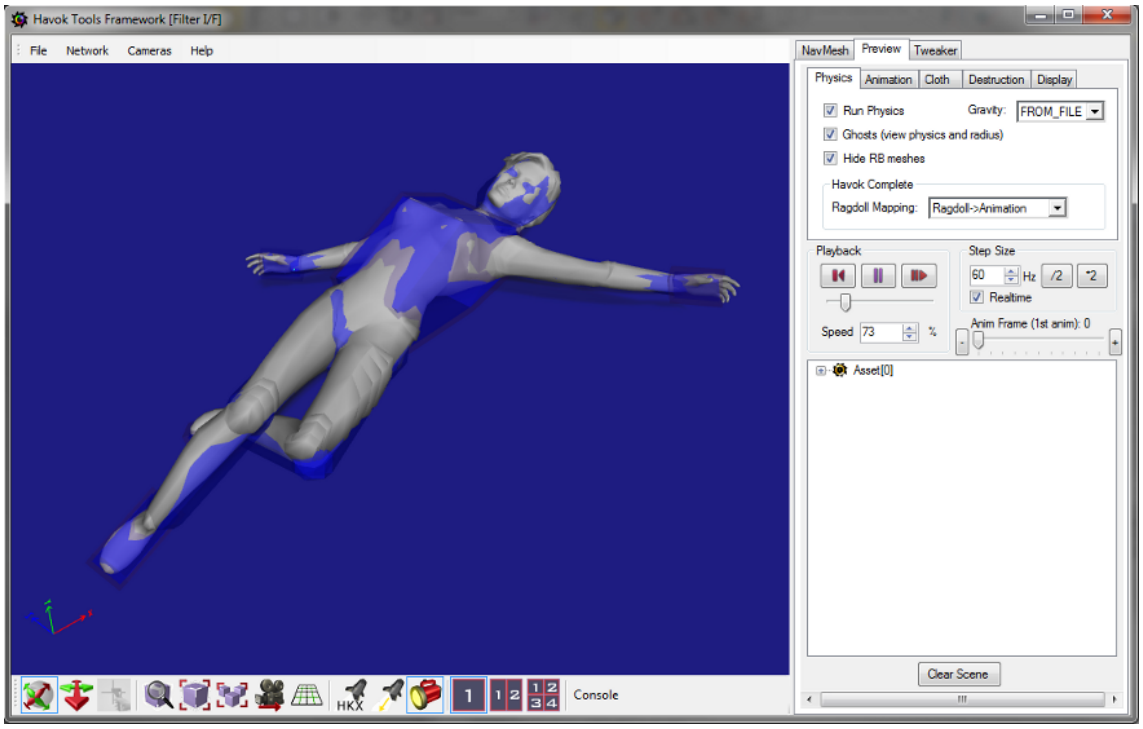
The default behaviour of the mapping filter is to match the skeletons by comparing the positions of the bones in each. This is why it was important to ensure that the pivot points of the rigid bodies were aligned to the pivot points of the joints.

Click ' **OK** '.

- **Preview Tool**

  So that we can interactively test the rag doll behaviour.

Run the filter manager. When the preview window appears, play the physics and interact with the rag doll using the space bar to pick and drag the rigid body shapes:




If the rag doll does not behave as expected, close the filter manager and review the rigid bodies and constraints in Maya. Tweak the attributes if necessary and continue to test using the filter manager until satisfied.

At this point you may want to serialize the completed rag doll to a file. See the export turorial for details on how to do this.

## 4.6 XSI Tools

### 4.6.1 Introduction

*Note that the XSI toolchain for Havok Content Tools is deprecated and as such will not be receiving product updates and bug fixes as often as the 3ds Max and Maya toolchains.*

In this chapter we are going to present the XSI specific components of the Havok Content Tools. These include:

- The XSI Scene Exporter
- The XSI Physics Tools
- The XSI Animation Tools

We also present several tutorials:

- Export and Animation Basics: How to set up, export and process an animation.
- Physics Basics: How to set up, export and process a physics scene.
- More On Rigid Bodies: A few more concepts about rigid body setup.
- Rag Doll Setup: How to set up, export and process a rag doll and its association with a high-res skeleton.

Check also:

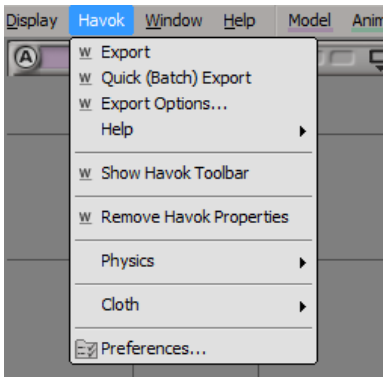
- Extending the XSI Tools: For information about exporting custom data and scripting access to XSI.
- Common Concepts: For common concepts useful for all Havok Tools (regardless of modeler).
- The Havok Filter Pipeline: For details on how the content created by any modeler can be processed.

## 4.6.2 XSI: Scene Exporter

Havok's XSI Scene Exporter is responsible for exploring XSI's scene graph, collecting scene information, and passing it to the filter pipeline for processing.

### 4.6.2.1 Accessing the Scene Exporter

The scene exporter is implemented as a custom XSI command. The best way to access the exporter functionality is to use the **Havok** menu in XSI:




You can also access functionality in the XSI Scene Exporter using the **Havok** toolbar. Use the **Havok** > **Show Havok Toolbar** or the **View** > **Toolbars** > **Havok** menu option in order to show it. You can also switching to the **Havok** custom layout (through the **View** > **Layouts** > **Havok** menu option) - the toolbar should appear docked in the bottom left corner of the screen. You can also include this toolbar as an element of your own custom layouts.

Let's focus on the options related to the Scene Exporter:

- **Export**

  This invokes the Scene Exporter, which will navigate the current scene graph extracting information about meshes, lights, nodes, attributes, etc., according to the current export options. After this is done, the Havok Filter Manager will be invoked in order to process this data.

- **Options...**

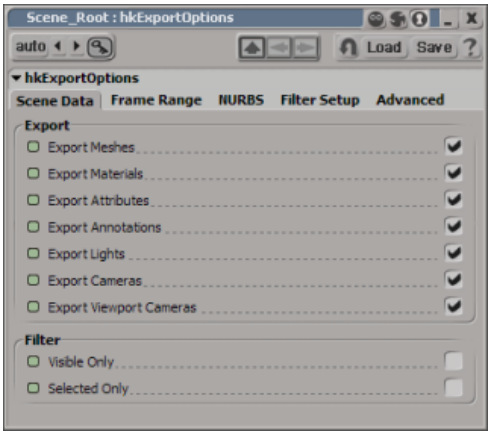
  This command will open a Property Editor within XSI where various export options can be modified. These options are described next.

#### 4.6.2.2 Export Options

The export options for the XSI Scene Exporter are stored as a *custom parameter set* in the scene. To view these options, click the **Havok** > **Options...** menu option or toolbar button.

The options are distributed among multiple tabs:

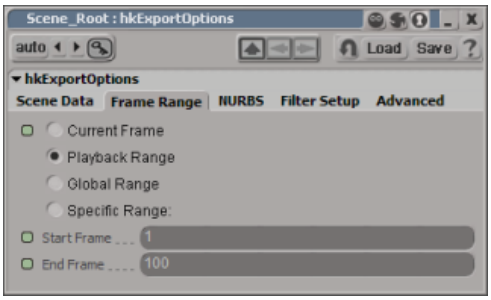
**Scene Data**



Options in this tab determine which types of objects are exported from the scene. The screenshot above shows the default settings.

- **Export Meshes** : Controls whether meshes should be exported.
- **Export Materials** : Controls whether materials and textures should be exported.
- **Export Attributes** : Controls whether custom Havok attributes should be exported. **Note that the Physics Tools depend on these attributes being exported.**
- **Export Annotations** : Controls whether annotations should be exported.
- **Export Lights** : Controls whether lights should be exported.
- **Export Cameras** : Controls whether cameras should be exported.
- **Export Viewport Cameras** : Controls whether any 'user' viewport cameras in use should be exported.
- **Visible Only** : If enabled, only objects which are currently visible in XSI will be exported.
- **Selected Only** : If enabled, only objects which are currently selected in XSI (along with all their ancestors) will be exported.

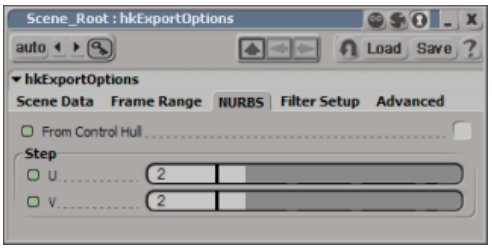
### Frame Range




Options in this tab specify how animated transforms are exported.

- **Current Frame** : Samples the scene only at the current frame. This option is useful if no animation data is required, as it speeds up the export process.
- **Playback Range** : Samples the scene using the current playback range.
- **Global Range** : Samples the scene using the global animation range.
- **Specific Range** : Allows you to specify a sampling range from **Start Frame** to **End Frame** .

### NURBS



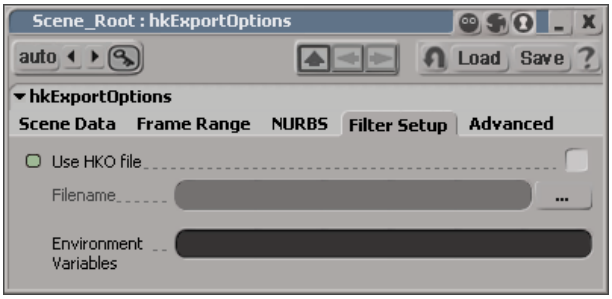

The Havok scene data format supports only polygonal meshes. Any NURB surfaces will be converted to a polygonal representation during export (the XSI scene won't be modified). These options control how this conversion is performed. Notice that these options reflect the parameters of the XSI **nurbsToMesh** operator - check the XSI documentation on this operator for more details.

- **From Control Hull** : Controls the placement of the polygon mesh's vertices.
  - When this option is **off** , vertices will be located at the knot positions and along the interpolated NURBS surface. This is the most common option.
  - When this option is **on** , vertices will be placed at the positions of the surface's control points. This can be useful if the mesh is going to be used to create a subdivision surface.
- **Step U, V** : The number of edges on the polygon mesh for each span between successive knots on the original surface

**Note:**

Skinned NURB surfaces are not supported by the XSI Scene Exporter - the NURBS surface will be converted to a polygonal representation but the skins binding (bone weights) won't be exported (since they are based on the NURBS surface and not the new polygonal mesh).

### Filter Setup

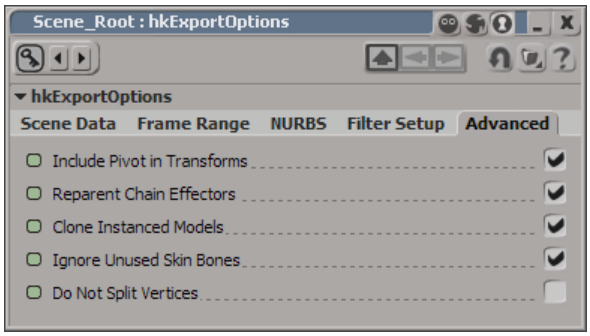



These options allow customization of the filter manager and individual filter behaviour.

- **Use HKO file / Filename** : Filter configurations (set up in the Filter Manager) are stored as part of the XSI scene, but sometimes it is useful to specify a custom set of options explicitly - for example, during batch processing, or to avoid having to build a filter set up from scratch in new assets. A set of filter configurations can be stored in an HKO file. Enabling this check box allows you to choose such an HKO file which contains the desired filter configurations to use for processing the asset. Check the Configurations section for more details.

- **Environment Variables** : Any variables specified here (through scripting for example) will be added to the exported scene under the "`hkxEnvironment`" root level variant. These variables may then be used by individual filters.

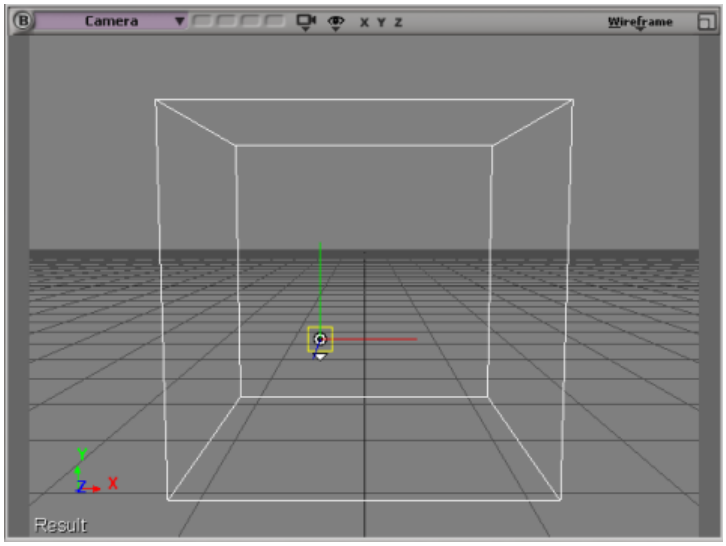
### Advanced



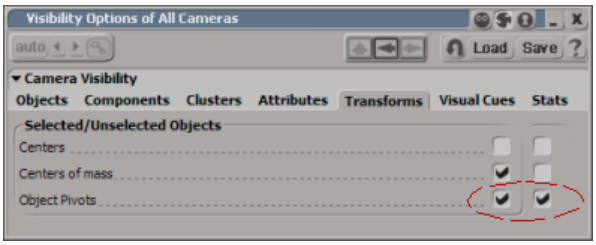

- **Include Pivot in Transform**

  It is often desirable to have control over the transform of an object without modifying the object itself. For example, the transform of a newly created box is usually specified at the center of the box. Changing the translation of this transform would move the box with it.

  XSI also defines the concept of *a pivot*, which is an extra transform associated with an object. Moving the pivot of an object doesn't move the object itself. By default the pivot is aligned with the transform of the object, but you can move it by holding **ALT** during a transform manipulation.




  You can also tell XSI to display a representation of the pivot in the viewports by changing the **Transform** settings in the **Visibility Options** menu:



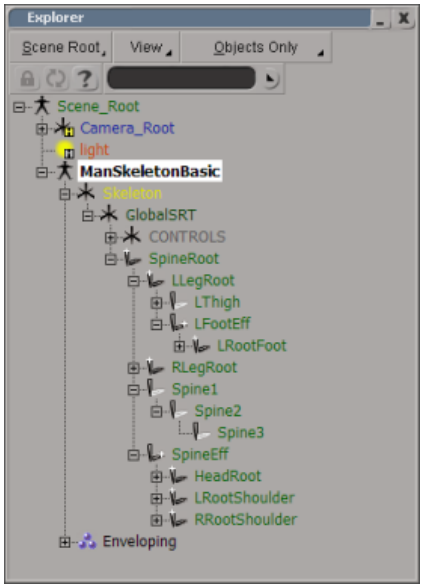

The pivot is also used as the center of manipulation for the translation, rotation and scale tools. Thus, for hierarchical structures, it usually represents the location of the joint associated with the object.

For all of these reasons, it can be very useful to export the transforms of objects in the scene as that of the pivot (which the user can fully control) instead of the object (which the user has limited control). The **Include Pivot in Transform** option enables this behavior. When this option is on, the exported transform of objects in the scene takes the pivot translation into consideration.

In particular, the Physics Tools extensively use the ability to manipulate the pivot in order to place constraints and align proxy rigid bodies to bones. Therefore in most cases you should leave this option enabled.

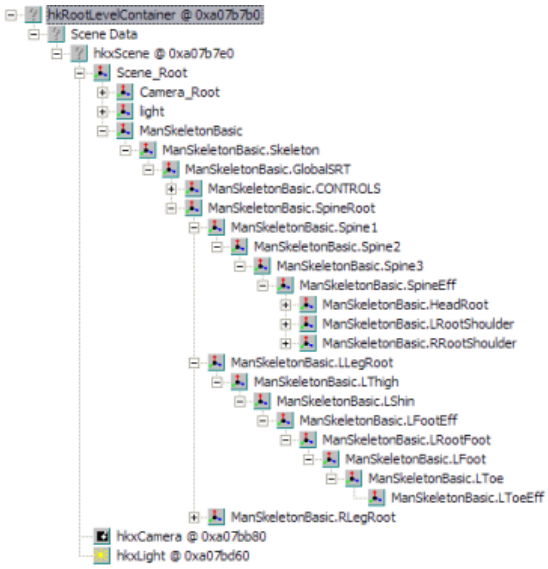
- **Reparent Chain Effectors**

  It is quite common in XSI to model skeletons as multiple chains of bones not necessarily hierarchically connected in the same way they operate. For example, the node structure of the default biped (man skeleton) model in XSI is as follows:

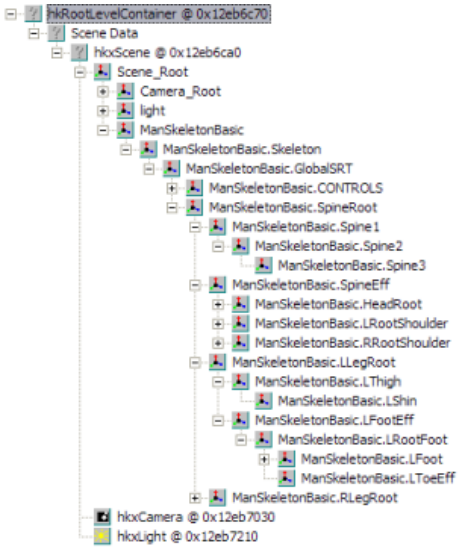



While you may expect the Head to be parented to the last link in the spine, notice how *HeadRoot* is parented to *SpineEff* instead of *Spine3*. Similar parenting can be found on the feet and arms.

In XSI the effector is treated as part of the chain, even if it's not a child of the last bone. In order to replicate this "implicit" parenting, the Havok XSI Scene Exporter can optionally detect chain effectors and parent them to the end bone of the associated chain. This is controlled by the **Reparent Chain Effectors** option. With that option **on** , the node hierarchy for the above skeleton is exported as a standard skeleton hierarchy:

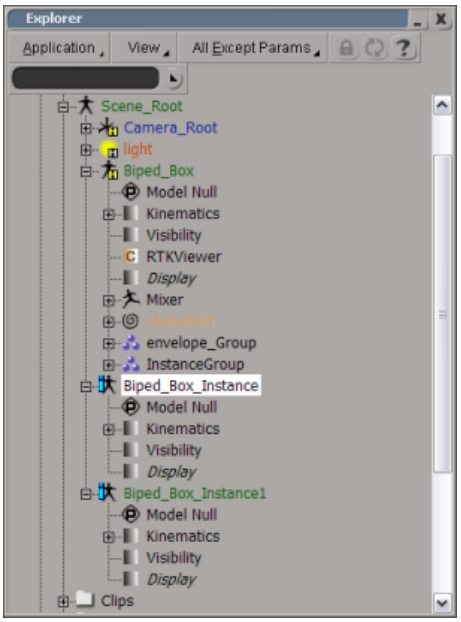



While switching the option **off** the node hierarchy exported will match that of XSI:

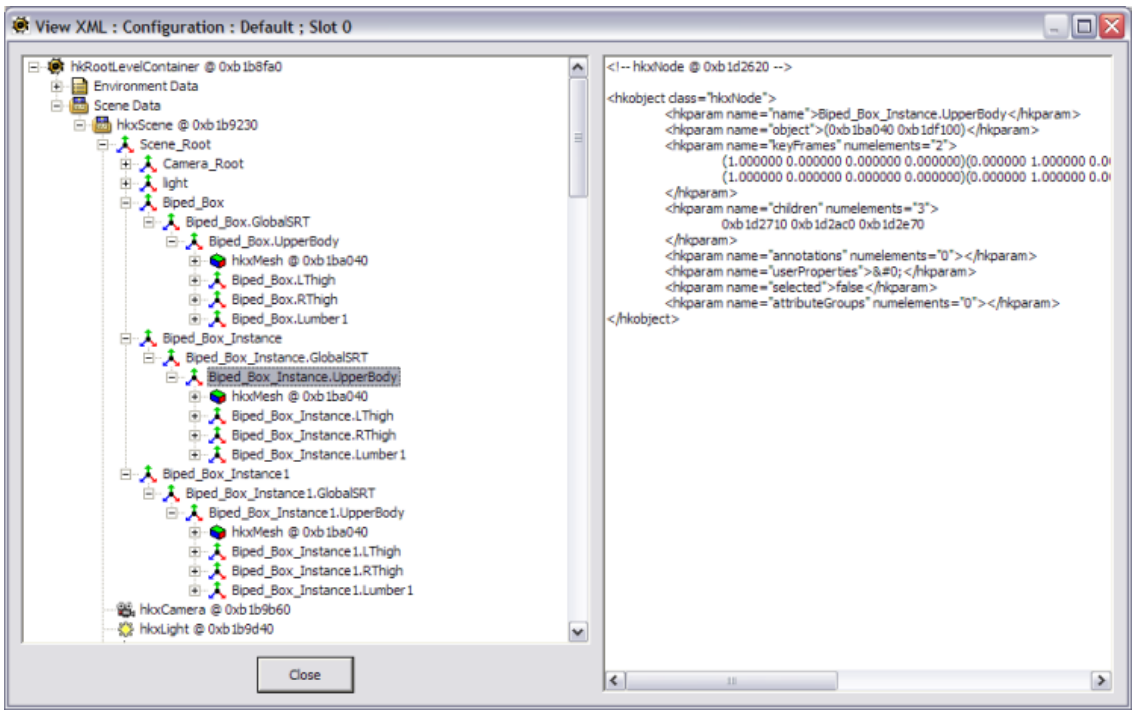



- **Clone Instanced Models**

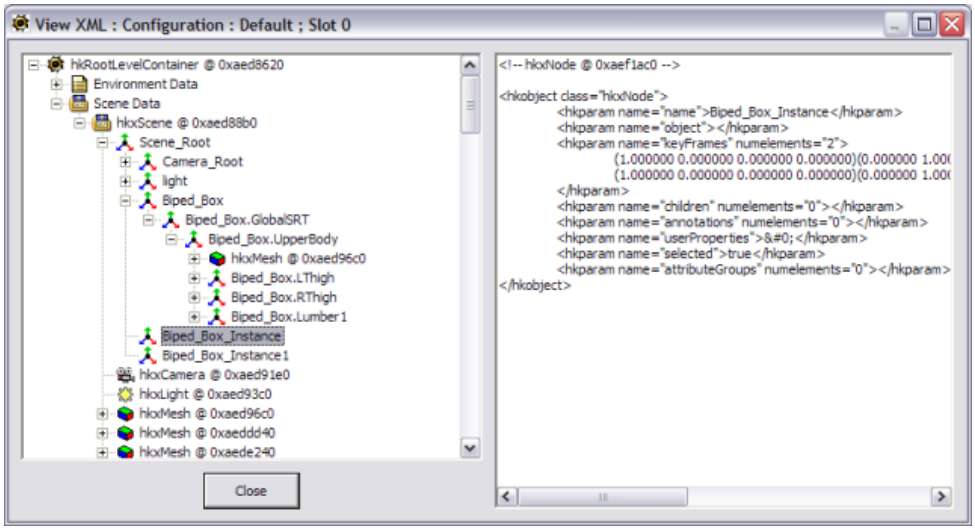
  XSI provides functionality for instancing models - it allows to replicate node hierarchies (full models) in the scene without any duplication of information - each instance has its own root (instanced model) which is just a transform - the children of the model are implicitly duplicated (they don't appear under the instanced model).




The Havok XSI Scene Exporter allows you to convert this implicit cloning into a more explicit one. If the **Clone Instance Models** option is selected, the node hierarchy under the original (master) model will be replicated under each of the instances in the exported scene.




If this option is off, the instanced models are exported as a single nodes with no extra children:




- **Ignore Unused Skin Bones**

  If this option is selected, the XSI Scene Exporter will perform an extra check for the bones reported by XSI as deformers of a given skin, and ignore them if they don't actually deform it (if all the associated weights are zero). This may reduce the size of the exported skin bindings and it may help create simpler rigs.

- **Do Not Split Vertices**

  By default, the XSI Scene Exporter will export as many mesh vertices as required in order to store texture and normals. So, for example, while for a cube only 8 geometric vertices would be required, in order to store per-face normal and texture coordinates, 24 vertices need to be exported.

  Selecting this option will override the default behaviour, and multiple vertices that have the same position will be exported as one, regardless of their normal or UV coordinates. This may reduce the size of the exported mesh - but due to the lack of accurate normal and texture coordinates, the mesh may not display as it originally did in the modeler.

## 4.6.3 XSI: Physics Tools

### 4.6.3.1 Introduction

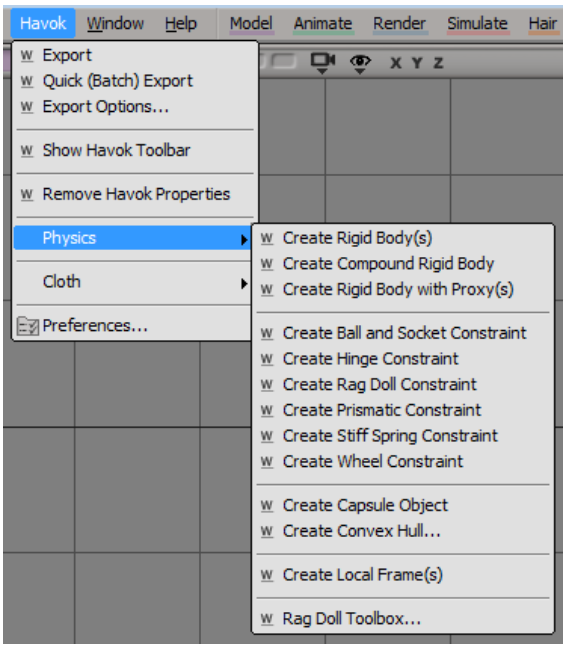
All modeling packages, including XSI, have the ability to specify scene and animation data that has a close relationship to the equivalent runtime objects in Havok (and provide many sophisticated tools to manipulate that data). However, for physical setup the tools available in the modeler are either limited or do not necessarily match the Havok runtime features. Moreover, as the construction of complex physical systems (such as rag dolls) can be a complicated task, providing tools to automate and facilitate these and another operations become important.

It is for these reasons that the Havok Content Tools include special XSI tools (parameter sets, operators, commands, views, etc.) for setting up physical information in an XSI scene. In the following sections we will explore these tools.

Advanced users should first read the Common Concepts chapter for in-depth details on some of the rigid body and constraint concepts.

**Accessing the Physics Tools**

The physics tools for XSI can be accessed through the **Havok** > **Physics** menu or through the **Havok** toolbar:



### Havok Parameter Sets

Custom parameter sets (also known as custom properties) are the main way in which additional (user) information can be attached to an object in XSI. Most of the Havok physics data in a scene is added through the use of such parameter sets, which are registered with XSI by the Havok Physics plugin.

For reference, the main parameter sets used by the Havok Physics Tools for XSI are:

- Rigid Body Parameter Set: Adds rigid body (dynamics) information to an object.
- Shape Parameter Set: Adds shape (collision detection) information to an object.
- Constraint Parameter Sets (Ball and Socket, Hinge, Rag doll): Add constraint information to an object.

- Local Frame Parameter Sets Add local frame information to an object.

We will be examining these parameter sets closer in the following sections.

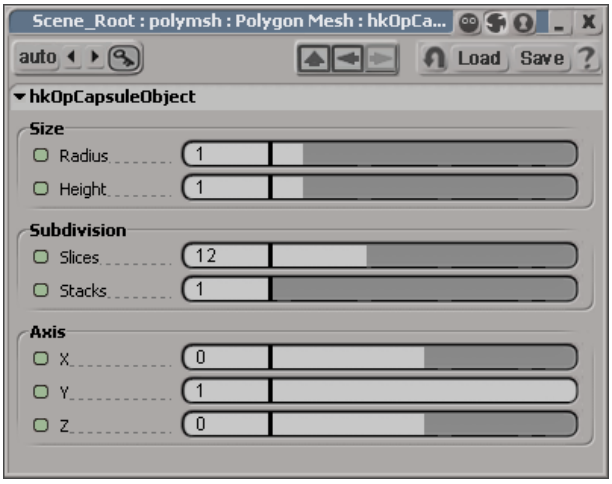
The Havok Content Tools also provide some other parameters sets, generally for storing options and some internal data. These parameter sets are recognizable by the common '`hk`' prefix.

While it is possible to manually add these parameter sets to your scene (through scripting etc.), it is highly recommended that you use the Havok menu and toolbar items to do so as doing otherwise may leave your scene in an inconsistent state.

You may remove any Havok Physics properties from selected objects at any time by choosing the **Remove Havok Properties** menu option.

### Havok Operators - Capsule

Unfortunately, XSI does not provide a capsule primitive. Capsules are a very useful primitive in Havok, so the Havok Content Tools provide an `hctOpCapsuleObject` operator. This allows the physics tools to create and modify capsule meshes easily. A polygon capsule can be created at any time by clicking on the **Create Capsule Object** menu option. The capsule operator provides several parameters to control the resulting polygon mesh:

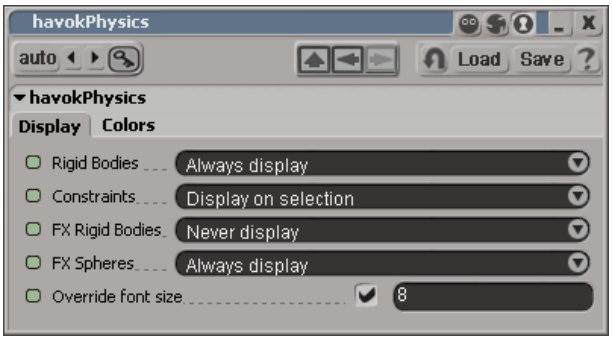



### Havok Views

The Havok Content Tools also include the *Rag Doll Toolbox*, implemented as a custom view in XSI. This is a set of controls which are layered on top of the core physics tools. The purpose of the toolbox is to simplify the creation of rag doll representations of skeletal structures - it takes care of common tasks such a fitting rigid bodies to bones, aligning pivots, setting up hierarchies, creating constraints, etc.

### Physics Preferences

Most Havok Physics parameter sets in a scene are represented in the viewports in some way. It is useful to have some control over this representation - the Havok Physics Preferences provide this control. You can open the preferences by clicking on the **Preferences...** menu option, or by locating the *havokPhysics* category in the application preferences:




These options mainly control the display of Rigid Body and Constraint parameter sets in the viewports. These display settings are as follows:

- **Always Display** : Always show these items in the viewports.
- **Display on Selection** : Show these items whenever the parameter set or its parent 3D object is selected.
- **Never Display** : Never show these items in the viewports.

There is also an option to specify the font size to use for Havok labels etc., in the viewports. If the **Override** option is disabled, the labels will be drawn at the same size as any other XSI text.

The second tab allows users to specify colors to use for various components which are drawn in the viewports, allowing you to easily adjust the display to suit your setup.

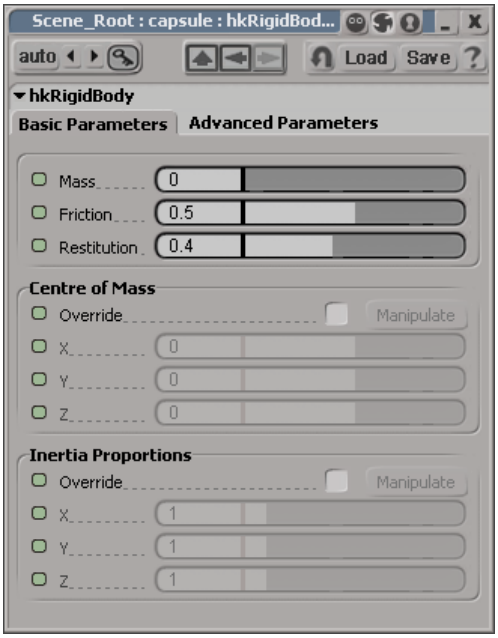
**Note:**
These preferences are normally saved with the application, not with the scene.
However, in some situations it may be necessary to save them with the scene - see the troubleshooting section for more.

#### 4.6.3.2 Rigid Bodies

In this section we will describe how to create and manipulate rigid bodies and shapes in XSI. For information about concepts and properties associated with rigid bodies and shapes, please check the Rigid Body Concepts section.

**Rigid Body Parameter Set**

A Havok rigid body parameter set defines the root of a rigid body and contains information on the dynamics of the body. Any object which contains a rigid body parameter set will be converted to a runtime rigid body by the Create Rigid Bodies filter (provided there is at least one associated shape parameter set).



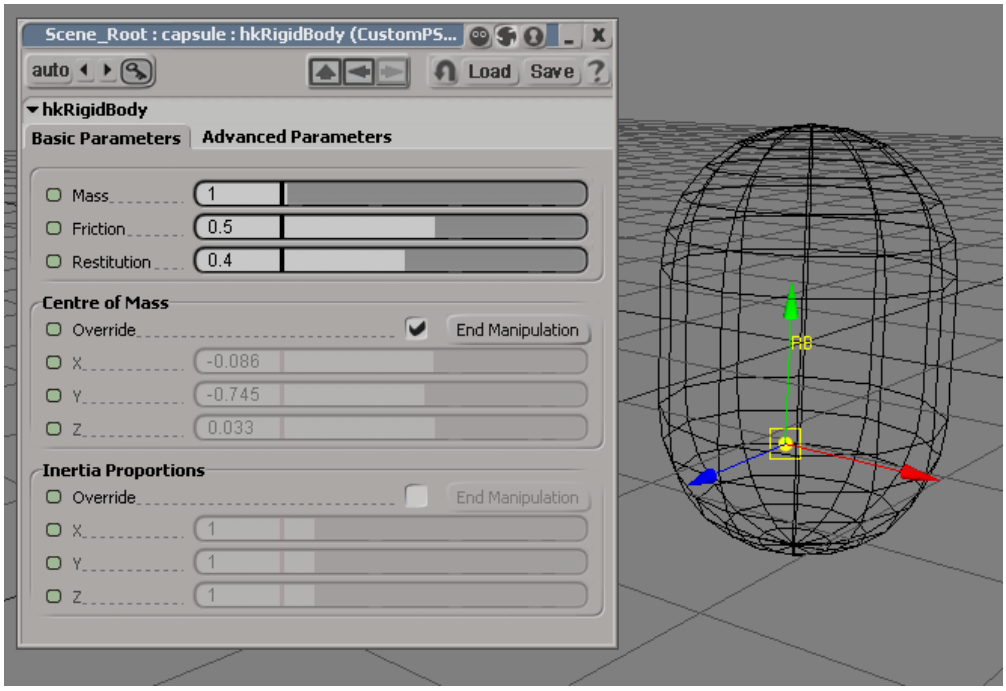

Check the common Rigid Body Properties section for specific details on these parameters.

Rigid bodies are represented by ' **RB** ' labels in the viewport. If a rigid body has zero mass, its label changes to ' **[RB]'** to indicate that the body is fixed.

**Overriding the Center of Mass and/or the Inertia Tensor**

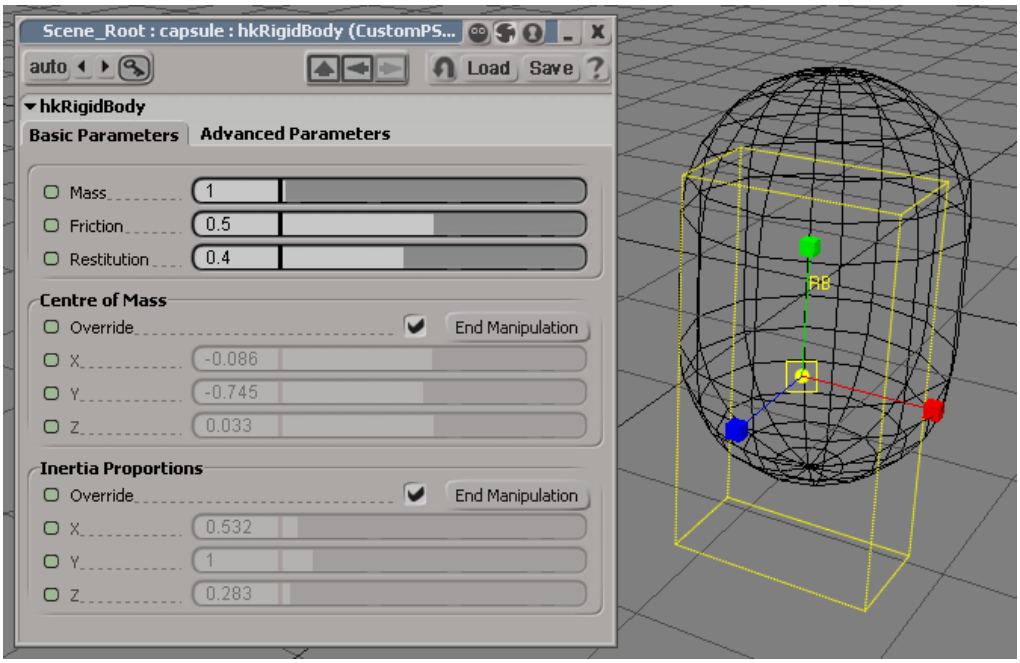
By default, the center of mass and inertia tensor of a rigid body will be automatically calculated during export, based on the rigid body geometry and mass distribution. You can, however, override these values and specify them within XSI, either by editing the parameter values directly, or by manipulating them in the viewports.

Selecting the **Override Center Of Mass** check box will display a point in the viewport, representing the center of mass of the object. To manipulate this point, click the **manipulate** button in the parameter set's property page. This sets up a temporary mechanism whereby we can drag a gizmo in the viewport to drive the parameter values:



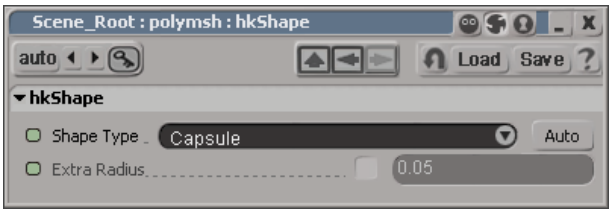

To exit this 'manipulation' mode, you may either press the **end manipulation** button, or change the selection in the viewport. This method of manipulating parameters is used by several of the Havok Physics tools.

Similarly, selecting the **Override Inertia Tensor** check box will display a box in the viewport which represents the inertia distribution of the object. Again, to manipulate this tensor, you may click its **manipulate** button, which in this case provides a scaling gizmo:




### Shape Parameter Set

A Havok shape parameter set contains information about how an object's mesh will contribute to the collision detection setup of a rigid body. One or more shape parameter sets must be associated with a rigid body parameter set in order for the Create Rigid Bodies filter to produce a runtime rigid body.




Check the common Shape Properties section for specific details on these parameters.

Whenever a new shape parameter set is created by the Havok Content Tools, its *Shape Type* parameter is automatically set based on the mesh which it applies to. However it is still useful to check each shape type parameter to ensure that it suits your requirements.

Shape parameter sets currently have no representation in the viewports.

### Creating a Simple Rigid Body

Most of the rigid bodies you will ever want to create will be simple rigid bodies - where both the rigid body and shape information is associated with the same object.

You can create a simple rigid body by selecting an object in XSI and clicking on the **Create Rigid Body** menu option or toolbar button.

This will add both a rigid body parameter set and a shape parameter set to the selected object.

#### Creating a Compound Rigid Body

A compound rigid body is that which has more than one shape (i.e. more than one collision detection primitive). In a compound rigid body one object has both rigid body and shape information while one or more of its children provide additional shape information.

You can create a compound rigid body by selecting two or more objects in XSI and clicking on the **Create Compound Rigid Body** menu option or toolbar button.

If the selected objects are parented to each other, the topmost object in the hierarchy will automatically hold the rigid body and shape parameter sets while each of the other objects will hold a shape parameter set. If the objects are not already parented, they will be parented automatically (after user confirmation), by considering the last object selected as the parent of the new hierarchy.

#### Creating a Rigid Body with Proxy(s)

A rigid body with proxy(s) is one where the object which contains the rigid body parameter set doesn't contain any shape parameter set - the shape parameter sets are provided by one or more of the children (i.e. the proxies). In that sense, a rigid body with proxy(s) is similar to a compound rigid body, the only difference being that the object which has the rigid body information has no shape information.

You can create a rigid body with proxy(s) by selecting two or more objects in XSI and clicking on the **Create Rigid Body with Proxy(s)** menu option.

If the selected objects are parented to each other, the topmost object in the hierarchy will automatically hold the rigid body parameter set while the rest of the objects will become the proxies and just hold shape parameter sets. If the objects are not already parented, they will be parented automatically (after user confirmation), by considering the last object selected as the parent of the new hierarchy.

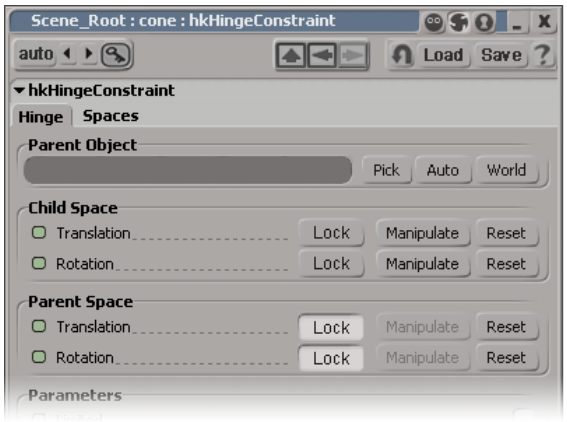
### 4.6.3.3 Constraints

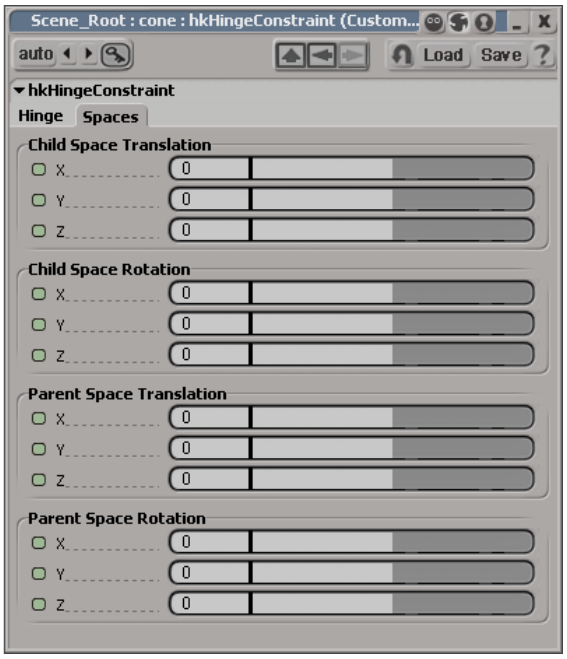
In this section we will present how rigid body constraints are created and manipulated in XSI.

#### Constraint Parameter Sets

A Havok constraint parameter set contains information which will be used to create a runtime constraint on export. Since a constraint must act on a rigid body, a constraint parameter set is useful only if the parent object also has a rigid body parameter set. If a pair of rigid bodies are to be constrained, a constraint parameter set is attached to only one of the bodies (the *child*), and it maintains a reference to the other body (the *parent*). Again, please consult the common Constraint Concepts section of this manual for more details.

There are currently three types of Havok constraint parameter set - the specifics of which are described in the common Constraint Types section of this manual. Each of these types provides the same basic constraint parameters, which specify the child and parent spaces:




Each constraint type generally provides additional parameters which are relevant to that type only. Each type also represents itself in the viewport in a way which reflects the type and its parameter settings.

### 'Parent Object' controls

These controls allow you to choose which other rigid body object (if any) is the constraint attached to.

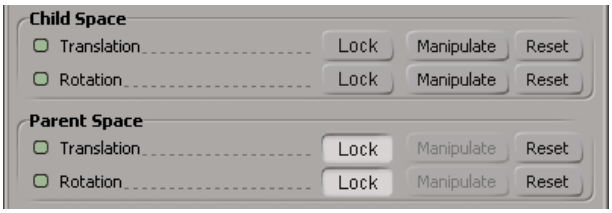
- **Pick** : Starts a picking session which allows you to specify any other object in the scene as the parent object. The picked object must contain a rigid body parameter set.
- **Auto** : Examines this object's hierarchical parent - if it contains a rigid body parameter set, then it will be set as the parent object. This is usually the desired setup.
- **World** : Clears the parent object. The constraint will attach the rigid body to a fixed location in space.

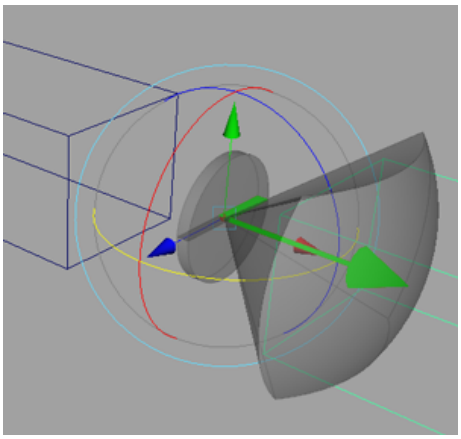
### 'Constraint Space' controls

Constraints can be considered as having two distinct spaces: the *child* (in the local space) and the *parent* (in the parent object's space). See the common Constraint Spaces section for more details.

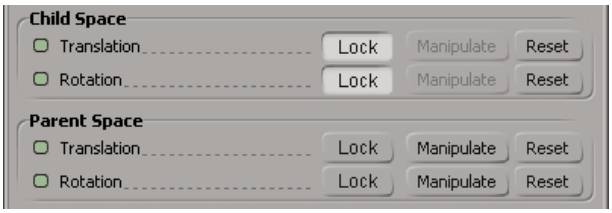
The individual [X,Y,Z] components of each space are editable via the controls in the 'spaces' tab. However it is often easier and more intuitive to manipulate the spaces in the viewport using the various **manipulate** buttons. Once the selection is changed or another button is pressed in the property panel, the manipulation mode will end.

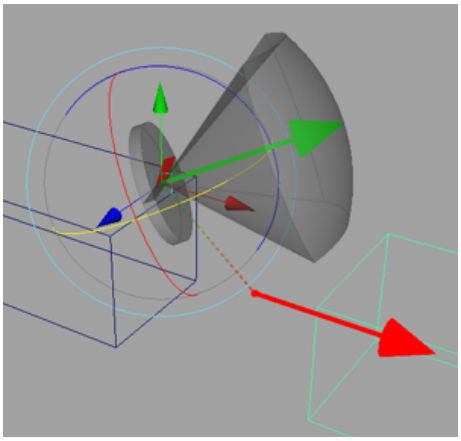
The state of the **Lock** buttons determine which spaces may be manipulated, and in the case of the parent space also determine how the spaces are interpreted. If the parent space is locked in both translation and rotation (the default setting) then the constraint space behaves as a single entity - both the child and parent spaces move together:



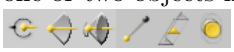
If the parent space is unlocked in either translation or rotation then the constraint parent space can be manipulated independently of the child space:



In all cases, the **Reset** buttons return a constraint space parameter to its initial setting, effectively moving the space to the local origin of the relevant object.

### Creating a Constraint

You can create a constraint by selecting either one or two objects in XSI and using the **Create <xxx> Constraint** menu options or toolbar buttons. 

Each selected object must already have a rigid body parameter set attached. Based on the number of selected objects, the constraint is created as follows:

- If one object is selected, the constraint parameter set will be attached to that object. If its parent object is also a rigid body then that object will be set as the constraint parent object, otherwise no parent object will be specified.
- If two objects are selected, and one is a descendant of the other, the constraint parameter set will be attached to the descendant and the ancestor will be set as the constraint parent. If the two objects are not explicitly parented, the constraint parameter set will be attached to the first object and the second will be set as the constraint parent.

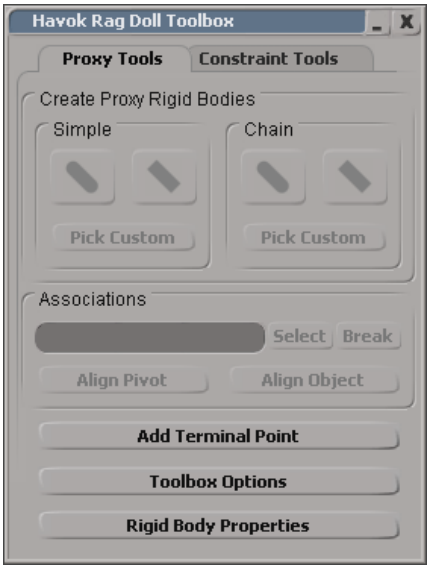
A single rigid body object may have any number of constraints attached to it.

#### 4.6.3.4 The Rag Doll Toolbox

**Introduction**

One of the most common applications of rigid body simulation is the creation of articulated systems of rigid bodies representing skeletal characters. This usually involves the creation and alignment of rigid body proxies for the character's bones, as well as the setup and alignment of constraints which represent the character's joints. The *Havok Rag Doll Toolbox* provides a set of tools to facilitate these operations. It is built as a custom view in XSI that works on top of the lower-level physics tools.

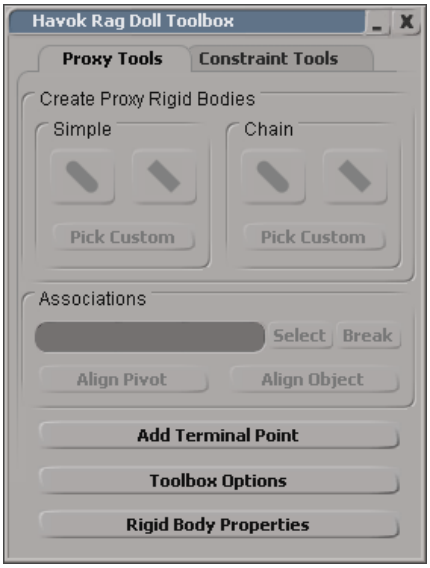
You can open the Rag Doll Toolbox using either the **Rag Doll Toolbox...** menu option or toolbar button.



The toolbox appears as a floating window with two tabs: the *Proxy Tools* are used to create and associate rigid bodies with bones; the *Constraint Tools* are used to organize and constrain these rigid bodies together:

The toolbox may also be opened in any viewport by selecting it from the ' **custom display host** ' submenu of the viewport's menu.

### Proxy Tools




The first task when creating a rag doll representation of a character is to construct a set of rigid bodies to represent a selection of the character's bones. These rigid bodies are usually simple collision primitives such as capsules and boxes, which are very efficient to simulate by the physics runtime (capsules in particular).

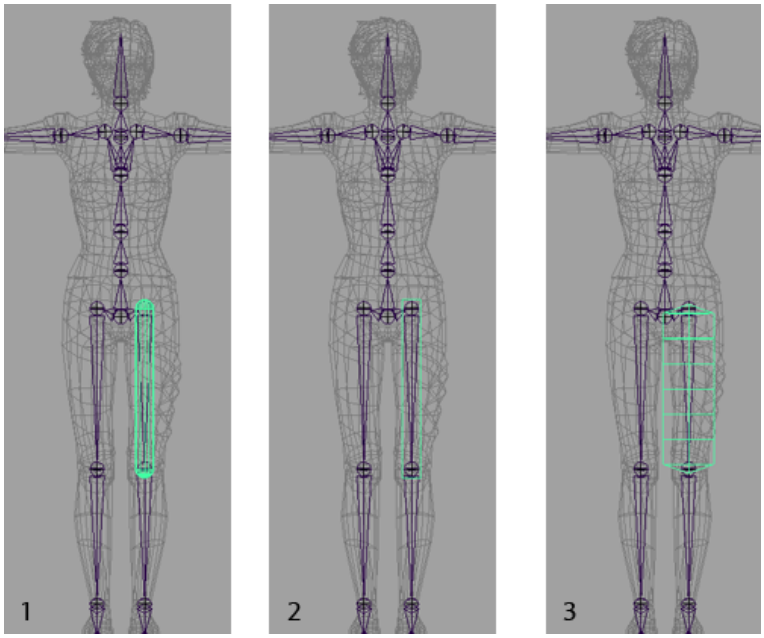
The Proxy Tools provide the following functionality:

- Given a bone, create a proxy representation (a box, a capsule, or a custom mesh) and align and associate the proxy with the original bone.
- Given a chain of bones, create a single proxy representation of that chain and align and associate that proxy with the chain of bones.
- Given a proxy rigid body (associated with a bone or chain of bones), replace it with another proxy.
- When creating proxies, it is very important that their pivots are aligned to those of the bones they represent, since, at runtime, movements on the bones will be converted to movement on the rigid bodies and vice versa. The proxy tools track this association and provide the ability to realign the proxy pivot/object at any time.
- Quickly access and modify the parameters of one or multiple rigid bodies.

### Create/Replace Proxy Rigid Bodies

These buttons allow you to create proxies and associate them with bones, and are enabled based on the current selection in XSI. Note that the Toolbox Options contains several settings which apply to proxy creation.

The first group of buttons create *simple proxies* - those which are associated with a single bone.




1. **Create Capsule Proxy(s)**

   With one or more bones selected, this will create a capsule rigid body for each selected bone. It will also associate and align each capsule with each bone.

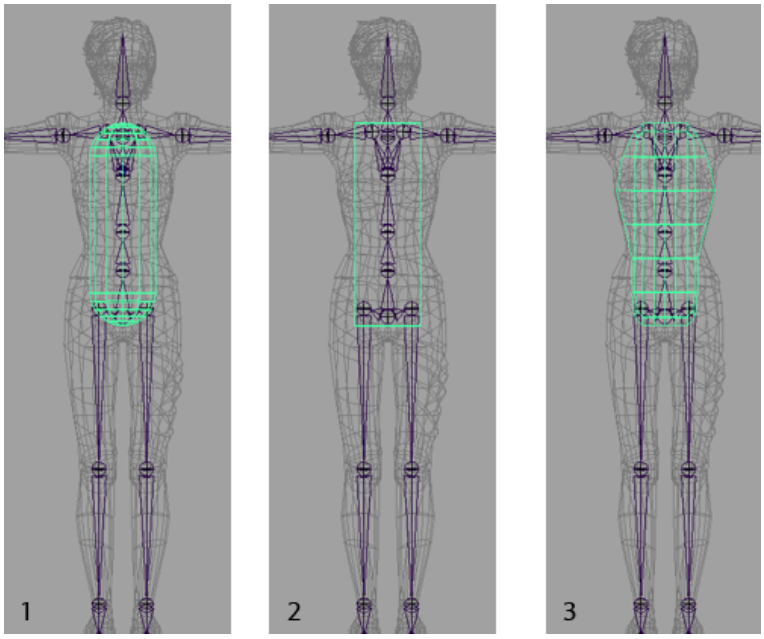
2. **Create Box Proxy(s)**

   With one or more bones selected, this will create a box rigid body for each selected bone. It will also associate and align each box with each bone.

3. **Pick Custom**

   With a single bone selected, this will start a picking session where you can select any mesh object in the scene to become the proxy rigid body for the selected bone. It will then associate and align that rigid body with the selected bone.

The second group of buttons create what we call *chain proxies* - those which are associated to a set of bones in hierarchical order. Chain proxies are useful in order to create single rigid bodies to represent multiple bones, such as a rigid body bounding multiple spine links.




1. **Create Capsule Proxy(s)**

   With a chain of bones selected, this will create a capsule rigid body bounding all of them. It will also associate and align the capsule with the chain of bones.

2. **Create Box Proxy(s)**

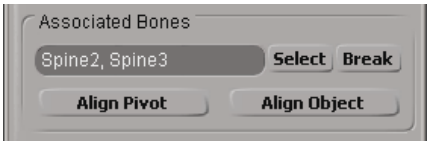
   With a chain of bones selected, this will create a box rigid body bounding all of them. It will also associate and align the box with the chain of bones.

3. **Pick Custom**

   With a chain of bones selected, this will start a picking session where you can select any mesh object in the scene to become the proxy rigid body for the chain of bones. It will then associate and align that rigid body with the chain.

Whenever a rigid body proxy (either single or chain proxy) is selected, these *Create Proxy* buttons become *Replace Proxy* buttons - the proxy they create will replace the selected proxy, and the association with the original bone will be updated accordingly.

**Associations**



These controls are used to display and edit the associations between selected rigid body proxies and bones. They are enabled whenever one or more rigid body proxies (rigid bodies created by one if the " **Create Proxy** " options above) are selected, or one or more associated bones are selected.

If a set of associated proxies are selected, the edit box displays the associated bones. If a set of associated bones are selected, the edit box displays the associated proxies. The **Select** button will change the current selection to that displayed in the edit box.

The **Break** button will remove the associations that are stored with the selected proxies/bones. The proxies will remain as rigid bodies in the scene, but their association with the original bones will be lost.

The **Align Pivots** button will ensure that the pivots of the each selected proxy match that the pivots of the associated bones (or the first bone of the associated chain), without transforming the proxy mesh. Use this button whenever you have modified a proxy's transform and want to ensure that the pivot is aligned correctly:

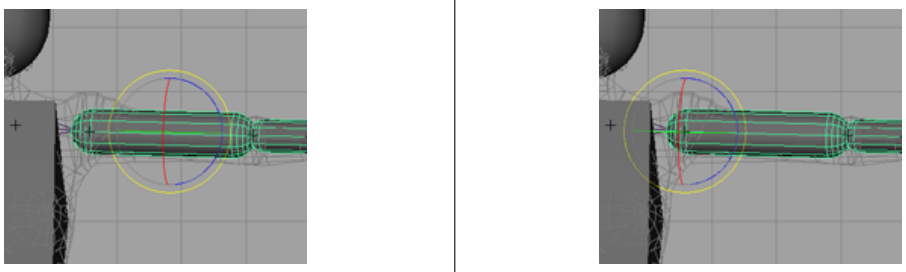


Table 4.5: Aligning Pivots

The **Align Objects** button will ensure that the pivot of each selected proxy matches the pivot of the associated bone by moving and/or rotating the proxy. Use this button whenever you move or rotate a bone and want to update the associated proxy accordingly:

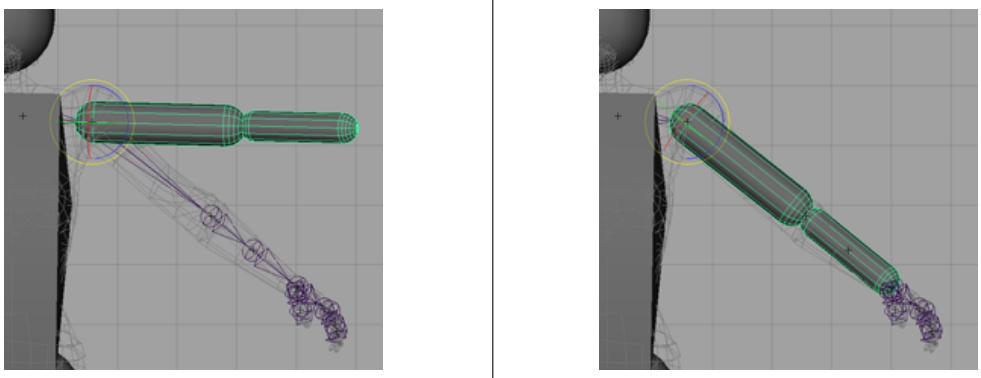


Table 4.6: Aligning Objects

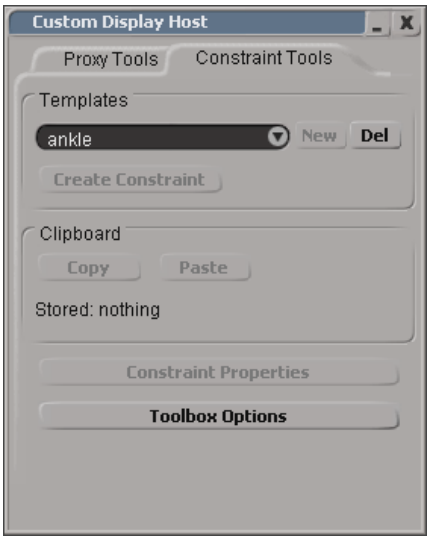
**Rigid Body Properties**

The **Rigid Body Properties** button allows quick access to the rigid body parameters of a set of selected rigid body objects. This is particularly useful when you want to modify multiple rigid bodies at the same time. Simply select a set of proxies and click this button to open a multi-edit property panel.

### Add Terminal Point

The **Add Terminal Point** button is used to add extra leaf bones to skeletons to be used for raycasting, specifically to prevent ragdoll penetration with landscapes using the hkaDetectRagdollPenetration utility. Selecting a ragdoll proxy and clicking the add terminal point button creates a null transform as the child of the proxy. The null is positioned on the local bounding box of the proxy at the center of the face furthest from the proxy pivot.

### Constraint Tools

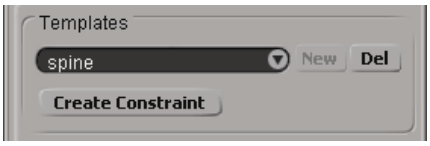



Once the rigid body proxies have all been created, the next step in setting up a rag doll is to organize them into a constrained hierarchy to match that of the original skeleton.

The Constraint Tools provide the following functionality:

- Given a pair of proxies, reparent them in a way which reflects the original skeleton, and apply a constraint between them.
- Save and reuse constraint setups between characters, as *template* files.
- Easily replace a constraint with another templated version.
- Quickly copy and paste constraints within the same scene.
- Quickly access and modify the parameters of a constraint.

### Templates



These buttons allow you apply any of a set of saved constraint setups (templates), and allows you to create new templates or delete existing ones. The dropdown list show all the available templates. This list is built by looking at all templates stored in the current *templates folder*, the location of which can be changed from the Toolbox Options.

The **New** button is enabled whenever a constraint parameter set (or its parent 3D object) is selected. It allows you to save the parameters of that constraint as a new template file (templates are saved as `*.txt` files). The **Del** button allows you to permanently remove the selected template (deleting it from the templates folder).

The **Create/Replace Constraint** button is that which does the main work. This button is enabled only when two proxies are selected. When a constraint is created/replaced, the following steps take place:

1. Ensure that hierarchy of the selected proxies is correct, by looking at the hierarchy of the associated bones. If not correct, it reparents the proxies accordingly.
2. Create/replace a constraint parameter set on the 'child' object - the one which is lower down in the hierarchy. The type of constraint which should be created is stored in the template file.
3. Set the constraint's parent object as the one higher up in the hierarchy.
4. Set the constraint's parameter values based on those stored in the template.

**Note:**

When working with templates, some assumptions need to be made regarding the local axis of the bones. In particular, the main axis (an axis pointing towards the length of the bone, in the "outside" direction) and the bend axis (an axis around which a positive rotation (using the right hand rule) causes a bending of the joint) need to be defined. These axes are currently assumed to be +X and –Z, since this is the convention used by XSI's default skeletons. The Havok Content Tools default templates were created using this convention. If a proxy's transform does not follow this convention, then any applied templates will need to be rotated to the correct orientation.

### Clipboard

While working on a symmetric character's constraints, it is often desirable to replicate constraints on each side of the character. The clipboard provides a quick way to create a temporary template which can then be applied throughout the rest of the character.

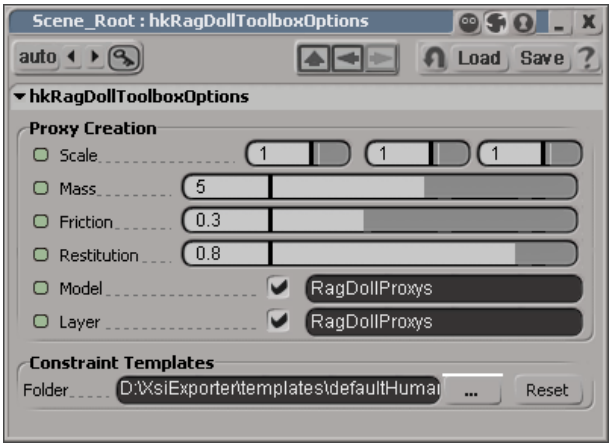
The **Copy** button is enabled only when a rigid body with a single constraint is selected. It will create a template of that constraint in memory, and display the stored constraint type for reference.

The **Paste** button operates in the same way the as the **Create/ Replace Constraint** button, with the difference being that it applies the template stored in memory rather than the template selected in the drop down list.

### Constraint Properties

The **Constraint Properties** button allows quick access to a constraint's parameters. Simply select a constrained rag doll proxy to enable this button. When the button is clicked, a property panel will open for the relevant constraint parameter set.

**Toolbox Options**




The toolbox options are stored as a custom parameter set in the scene, and are accessible via the **Toolbox Options** button on each tab of the toolbox.

Whenever new proxies are created, they are automatically sized to fit the bones they are associated with. However, bones are often much narrower than the skin volume and so the resulting proxies are generally not wide enough. The **Scale** options the ability to set a scaling factor to be used during the creation of new capsule/box rigid body proxies. Once a bounding box has been created in the local space of a bone (or a chain of bones) this additional scale is applied to the bounding box before a new proxy is created to fit the bounding box.

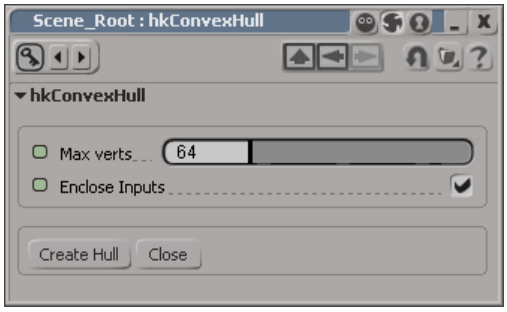
The **Mass** , **Friction** , and **Restitution** values are those which are applied to any newly created proxy, and are self-explanatory.

The **Model** and **Layer** options determine how new proxies are placed in the scene. By default any new proxies are grouped under an XSI 'model', and placed in a unique layer. This makes it easy to hide/lock/etc., the proxies as a whole without affecting the rest of your scene.

#### 4.6.3.5 The Convex Hull Tool

In this section we describe a simple tool for generation of convex hulls from selected meshes. This tool generates an approximate convex hull from the currently selected objects (either meshes or nurbs) with a user-specified maximum number of vertices **Max verts** . The hull is approximated if the requested number of vertices is less than the actual number of vertices present in the complete convex hull. This is useful in a number of contexts, such as generation of assets for platforms which require <64 vertices per convex body. If the specified maximum exceeds the number of vertices in the true hull, the tool just generates the true hull. Note that if multiple meshes/nurbs are selected, the hull tool generates a single hull from all of their vertices combined.

When **Enclose input** is checked (default), the convex hull when approximated using **Max verts** , is forced to enclosed the input(s)




The convex hull tool utility consists of fields for the two parameters **Max verts** and **Enclose input** described above, and a **Generate Hull** button to generate the hull using these parameters. Generated hulls are placed in the scene root with the name "`foo_Hull01`" where "`foo`" is the name of the first object in the selection. Subsequently generated hulls accumulate in the scene root but are given unique names.

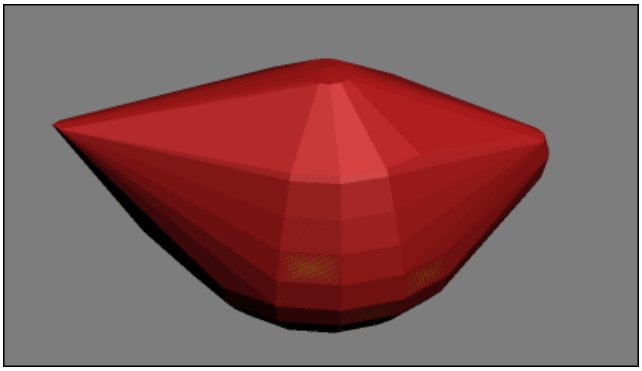


Figure 4.37: Full convex hull of a teapot

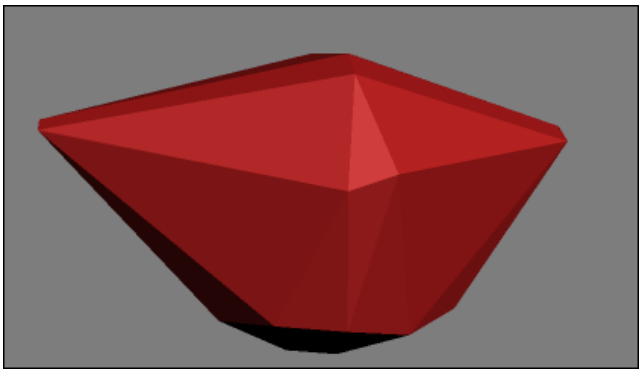


Figure 4.38: Approximated convex hull of a teapot with **Enclose input** checked

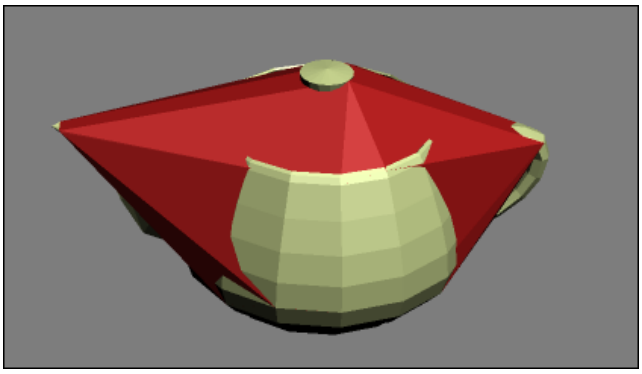
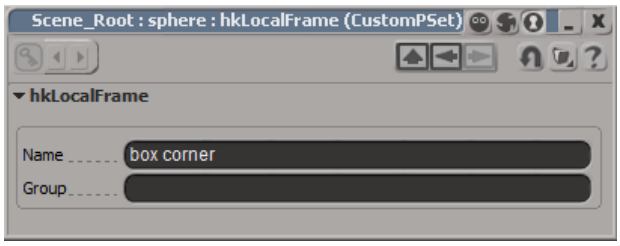


Figure 4.39: Approximated convex hull of a teapot with **Enclose input** unchecked

#### 4.6.3.6 Local Frames

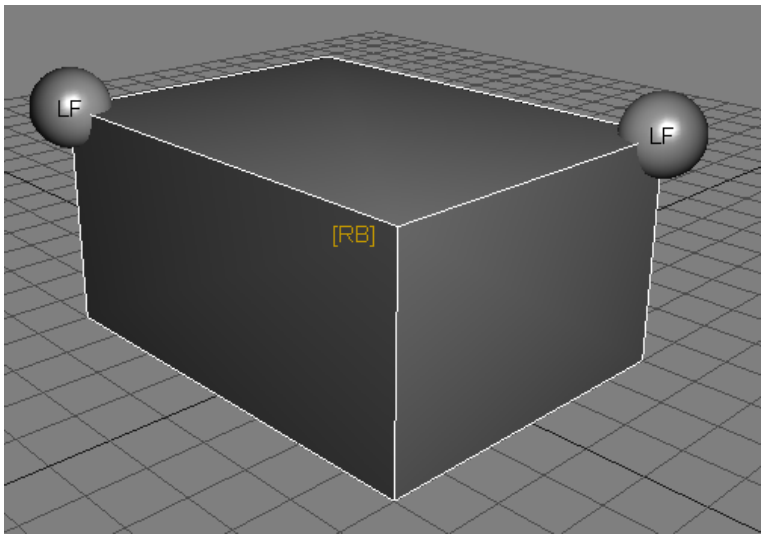
This section describes how to create **local frames** in XSI. To learn more about the concept of a local frame please see the Local Frame Concepts section.

You can create a local frame by selecting an object in XSI and clicking on the **Create Local Frame** menu option or toolbar button . This will add a **local frame parameter set** to the selected object as shown here:




The parameter set has a **Name** parameter in which you can name the local frame and a **Group** attribute in which you can name the group to which the local frame belongs.

An object that has a local frame parameter set will be displayed with the label **"LF"** in the viewports. The following image shows a rigid body box with two local frames parented to it displayed in XSI:




Any object with a local frame parameter set will be converted to a runtime local frame in the Create Rigid Bodies filter (if parented to a rigid body) or the Create Skeletons filter (if parented to a skeletal bone).

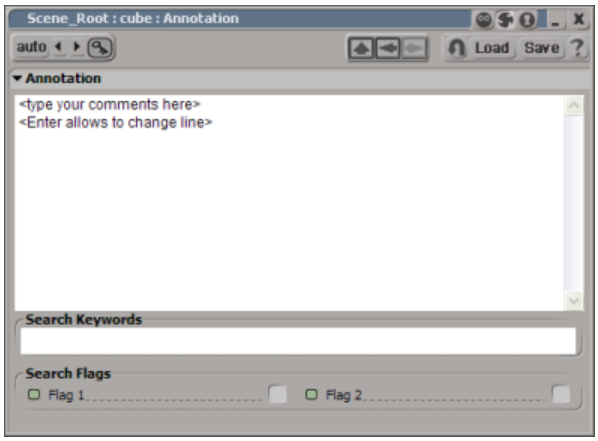
### 4.6.4 XSI: Animation Tools

Since XSI already provides a wealth of tools for creating content for animation playback, the Havok Content Tools use these features already provided by XSI without the need of providing custom extensions (as it does for the physics). Thus concepts like skeletons, bones, skins and animations in XSI transfer to the same concepts/objects in the Havok SDK. The following section outlines one of the less obvious connections between XSI and the Havok Animation SDK.

#### 4.6.4.1 Annotations in XSI

Please check the common Annotations section for details on annotations in the Havok Animation SDK.

XSI supports the concept of annotations as simple "extra information" associated with any object. An XSI annotation consists of:



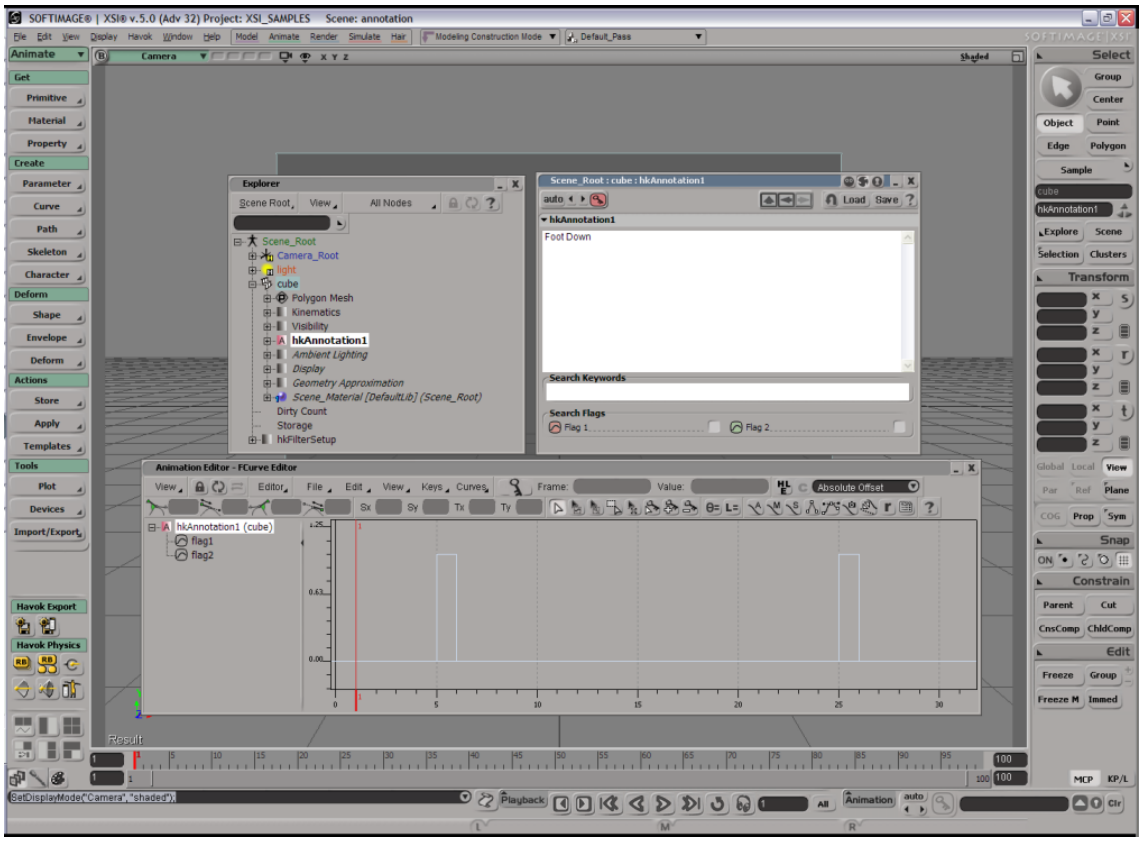

- An annotation text (non animatable).
- Search keywords (non animatable).
- Two boolean flags (animatable).

Notice that XSI annotation objects are associated to objects, but not to a particular time in the animation, while a Havok Annotation is associated both to a node and a time. In order to bridge the gap between the two types, the following logic is applied when creating Havok Annotations from XSI annotations:

- The XSI annotation has to be associated with a node (3D object) in the scene
  - This is because Havok annotations can only be associated to nodes
- The XSI annotation's name has to start with "`hkAnnotation`" (for example, "`hkAnnotation1`")
  - This is so the XSI Scene Exporter can decide which annotations are meant to be transformed into Havok Annotations
- The animation of the **flag1** boolean parameter is used to determine at which time(s) are the annotations added. Each time this flag transitions from **false** to **true** , a Havok annotation will be added.
- The text of the Havok annotation is taken from the text of the XSI annotation. If this text is empty, the name of the XSI annotation is used instead.

**Example**

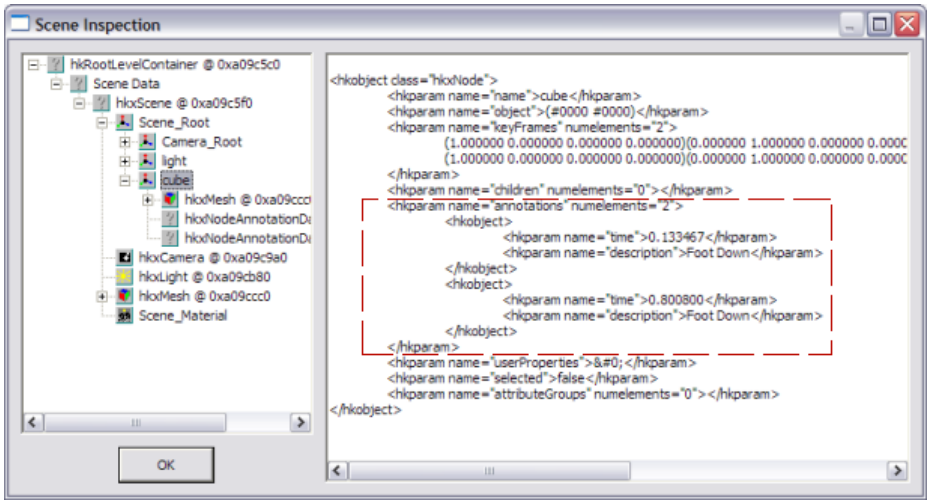
The screenshot below show an XSI annotation created so it exports as a Havok annotation.




Notice the following:

- The XSI annotation is associated to the *cube* node
- The XSI annotation is named "`hkAnnotation1`"
- The **flag1** parameter is animated so it transitions to **true** at frame 5 and 25
- The animation text is set to "*Foot Down*"

Since this XSI annotation matches the criteria we described above, it will be exported as a Havok annotation. The following screenshot shows a section of the scene data exported by the XSI exporter:

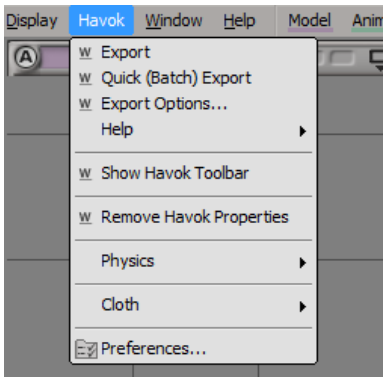



### 4.6.5 Tutorial: Export and Animation Basics

In this tutorial we are going to explore the basic operations involved in exporting and processing an asset. We are going to use the example of exporting skeletons, animations, etc., but the principles regarding how assets are exported, the filter pipeline and the use of the filter manager and individual filters for processing applies to all Havok products.

#### 4.6.5.1 Getting Started

Ensure that you have installed the Havok Content Tools for XSI. You should see a **Havok** menu in the XSI menu bar:

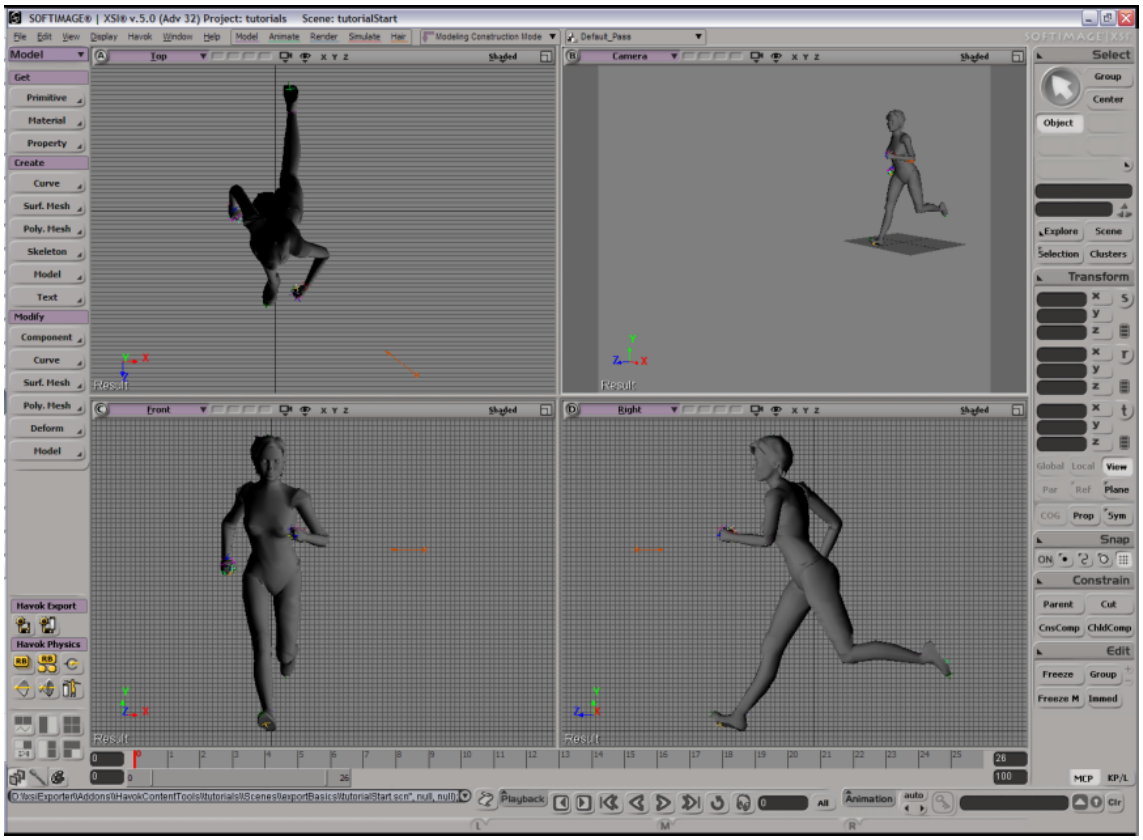



The **Havok** toolbar will be useful for this and the next tutorials, so it is useful to have it in the UI. You can switch to the Havok layout ( **View** > **Layout** > **Havok** menu option) or open the toolbar as a floating window ( **View** > **Toolbars** > **Havok** menu option)

Open the scene "`exportBasics/TutorialStart.scn`", in the *tutorials* project (which should be installed in `[INSTALLDIR]/Addons/HavokContentTools/tutorials`).

**Tip:**

If you haven't added this project to your project list we recommend you do so in order to quickly access the tutorial files.




The scene contains the animation of a girl running (a single cycle). If you examine the scene, you will notice that the animation is constructed by using a set of animated bones which modify a skinned mesh.

We want to export this information (bones, animation, skin weights, etc.) from XSI into objects that the Havok Animation/Complete SDK can interpret.

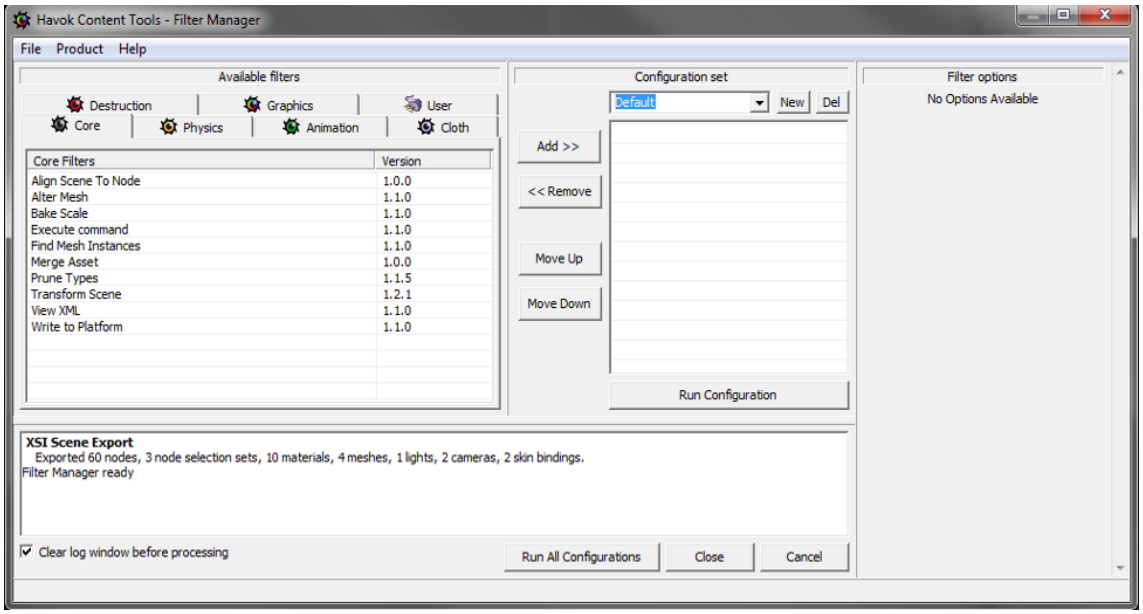
**Note:**

Even if you are not a user of Havok Animation or Havok Complete, you can still follow this tutorial as the Havok Content Tools packages contain filters for all products. This tutorial focuses on introducing the basic concepts behind asset export and processing so it is relevant for all Havok products.

#### 4.6.5.2 Exporting the Scene

Exporting the current scene is as simple as to click on the **Export** button in the **Havok** menu or toolbar.

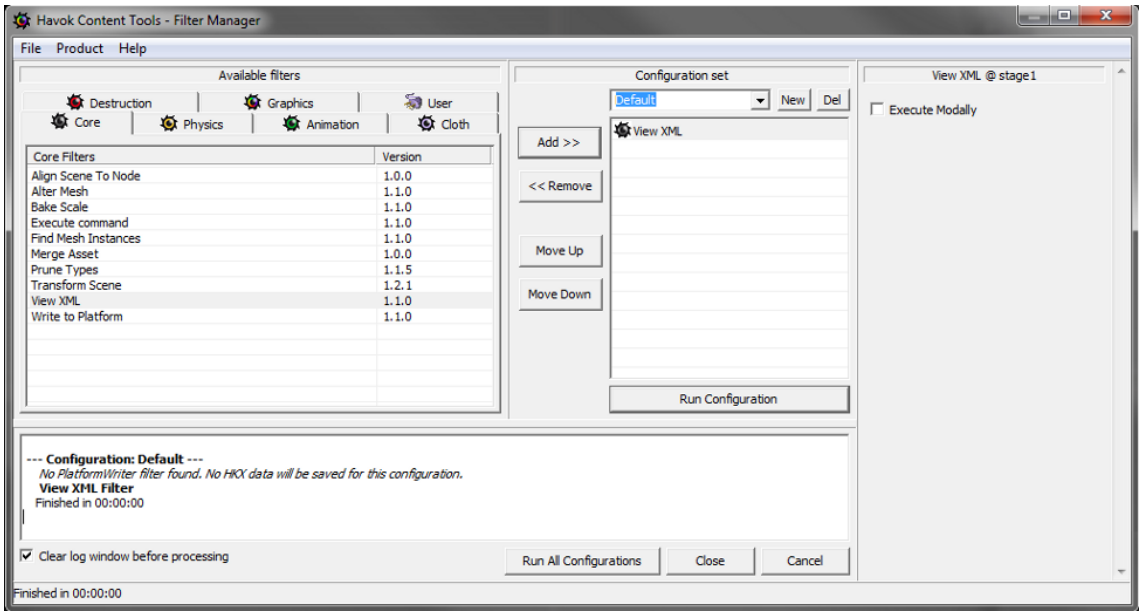
By doing this the XSI Scene Exporter is invoked - the scene is navigated and relevant information is pulled from the XSI nodes, properties, meshes, etc., and converted into a modeler-independent format. After that, that content is passed to the filter manager; its UI should appear after a short time:




The **Filter Order** window (in the middle of the window) lists the filters that will be applied to the content we just exported. It is now empty, which means that nothing will happen to the content. You can add filters by selecting them from the **Available Filters** tabs and clicking on the **Add**>> button.

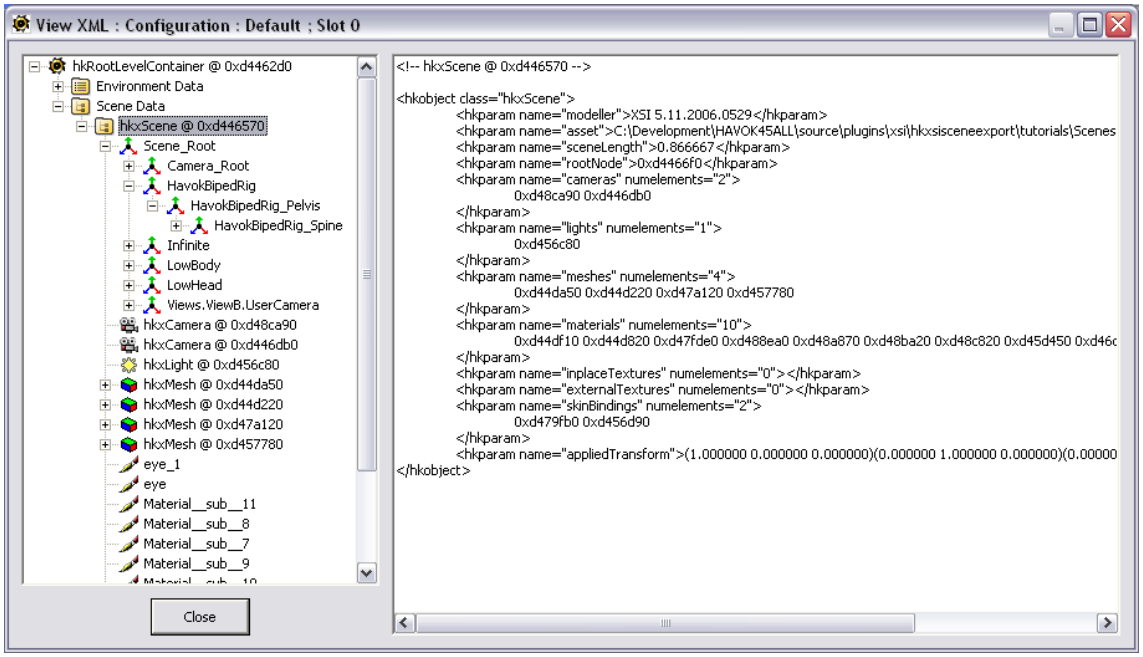
Let's start by adding a very useful filter for debugging purposes: the View XML filter.

- On the **Available Filters** window, click on the **Core** tab
- Double-click on the **ViewXml** filter (or select it and press **Add** > >)
- That filter should now appear at the top of the **Current Filters** list




Filters are executed in the order they appear in the **Current Filters** list; if a filter modifies the content, that modified content is what is used by the next filter (check the filter pipeline documentation for details).

The View XML filter doesn't modify any content, though. It just opens a modal window that presents the current content in a human-readable XML format. Let's check what we exported from XSI by clicking on the **Run Configuration** button.




Explore the contents of the exported data. Under the root level object, the main object that was exported is an `hkxScene` object, which contains information about nodes, meshes, materials and skin bindings. This

is what the XSI Scene Exporter extracted from our current scene, and is the raw information that we are going to process further until we can create a useful set of objects ready to be used by our runtime.

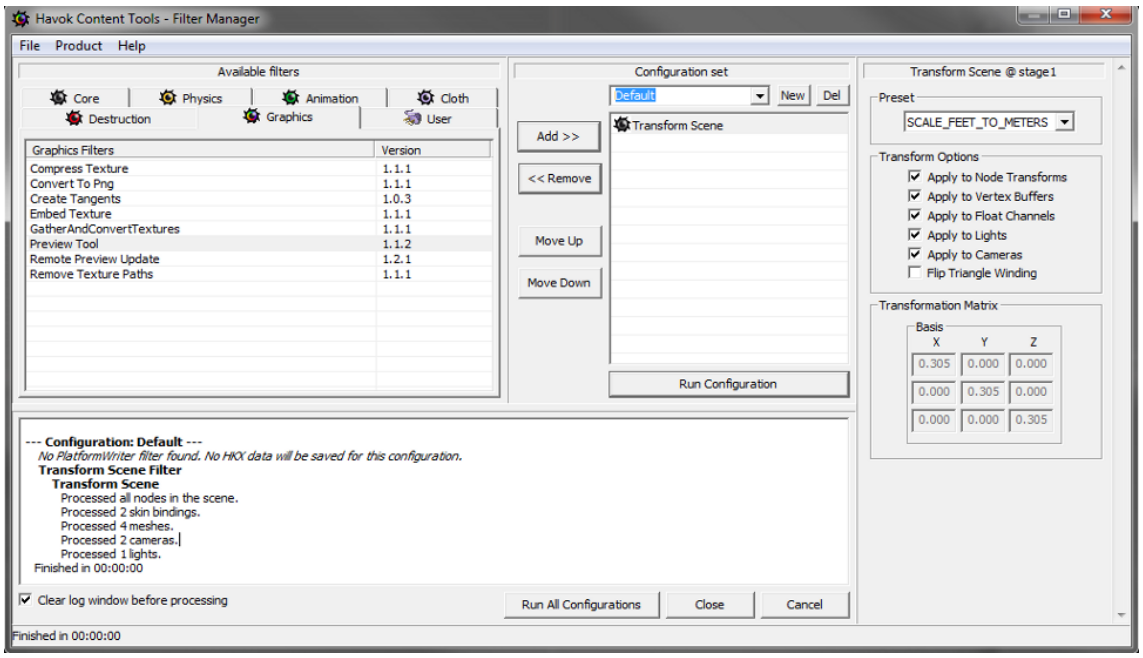
Close the XML window and get back to the filter manager. Remove the ViewXml filter by selecting it and clicking on << **Remove** or hitting **DEL** .

**Tip:**

Feel free to add the View XML filter at different locations during this and other tutorials in order to examine how processing is done by different filters. This filter is a very useful debugging tool.

#### 4.6.5.3 Processing the Scene

Let's now add some useful filters for processing the scene. The first filter that we'll add will be the Transform Scene filter (also from the **Core** tab):



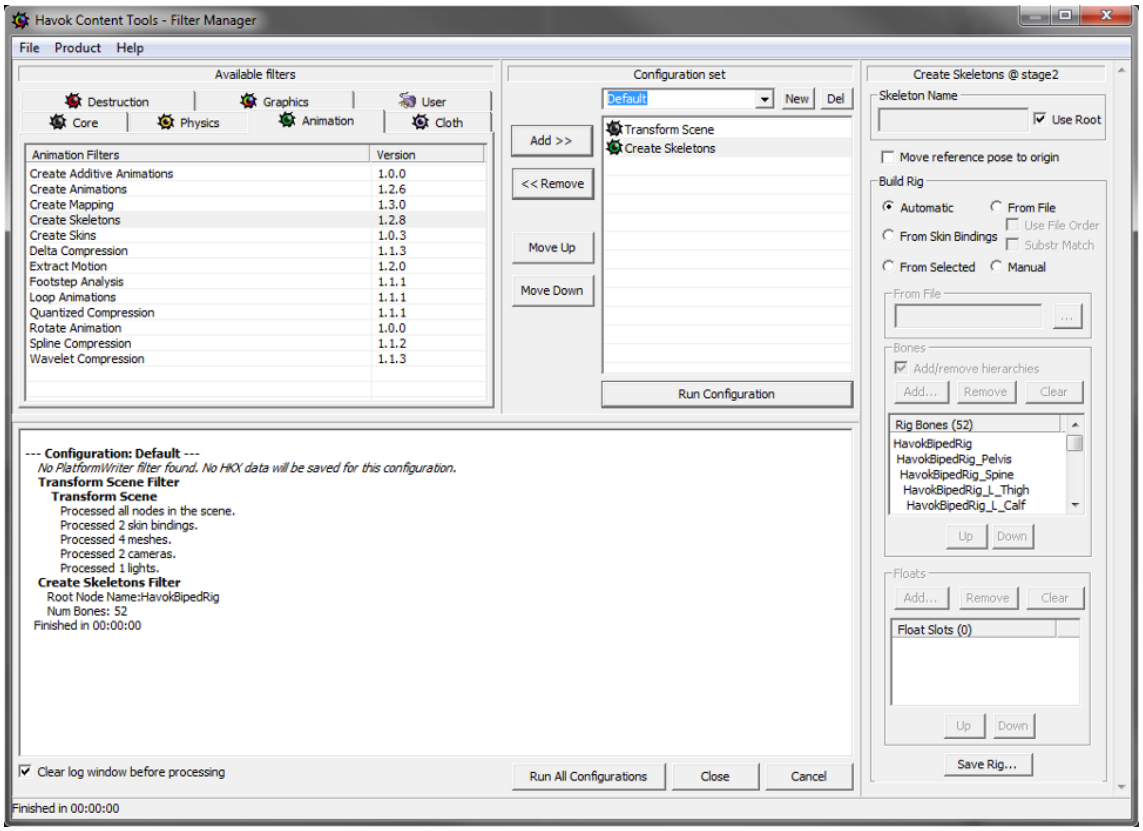

This filter is very useful and, in many cases, it will be the first filter that you'll add to any filter configuration. This filter applies an arbitrary transform to all the objects (nodes, meshes, etc.) in the scene. Our scene has been modelled in inches. Assuming that our runtime is modelled in meters, we use the **SCALE_INCHES_TO_METERS** preset in the options of the filter (the right panel in the filter manager) to scale the scene.

The next filter we will apply is the Create Skeleton filter (from the **Animation** category). This filter takes a look at the nodes in the scene and creates an `hkaSkeleton` object (used by the Havok Animation/Complete SDK).

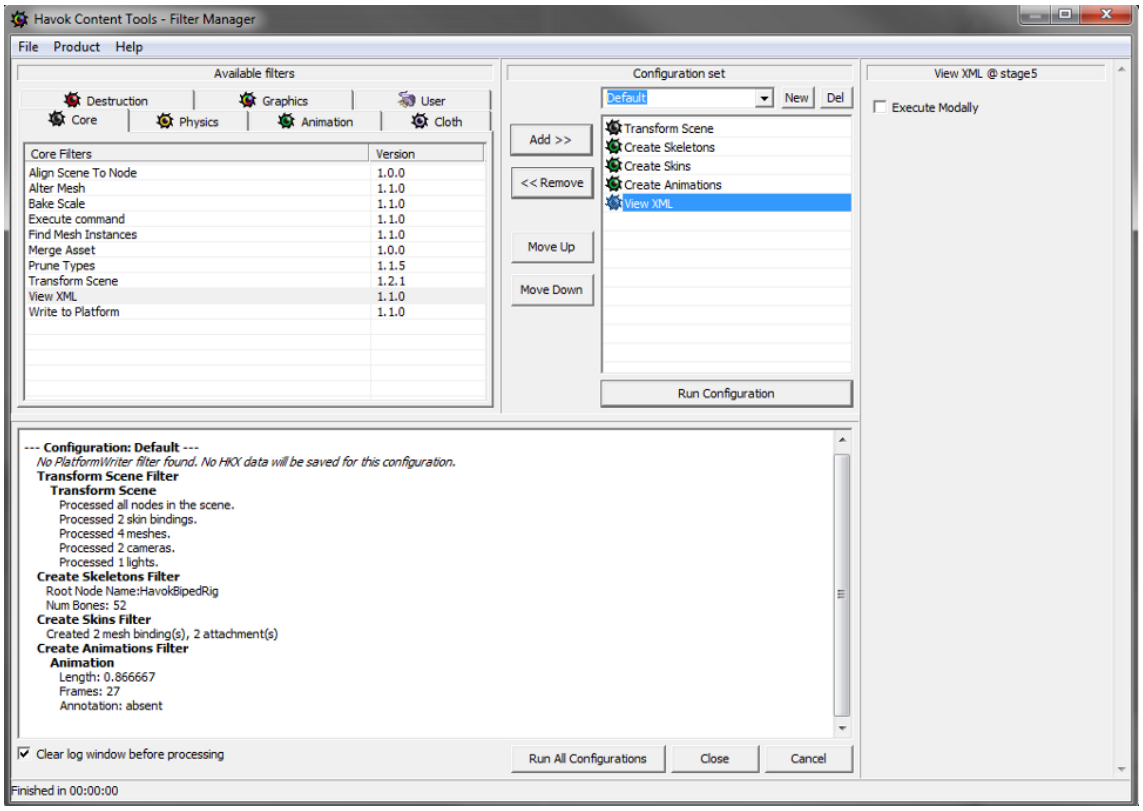
**Note:**

If your current Product Selection is set to Havok Physics, some of the filters will appear in red, and when run, warnings will be generated. You can either ignore those warnings, or change your product selection temporarily to Havok Complete.

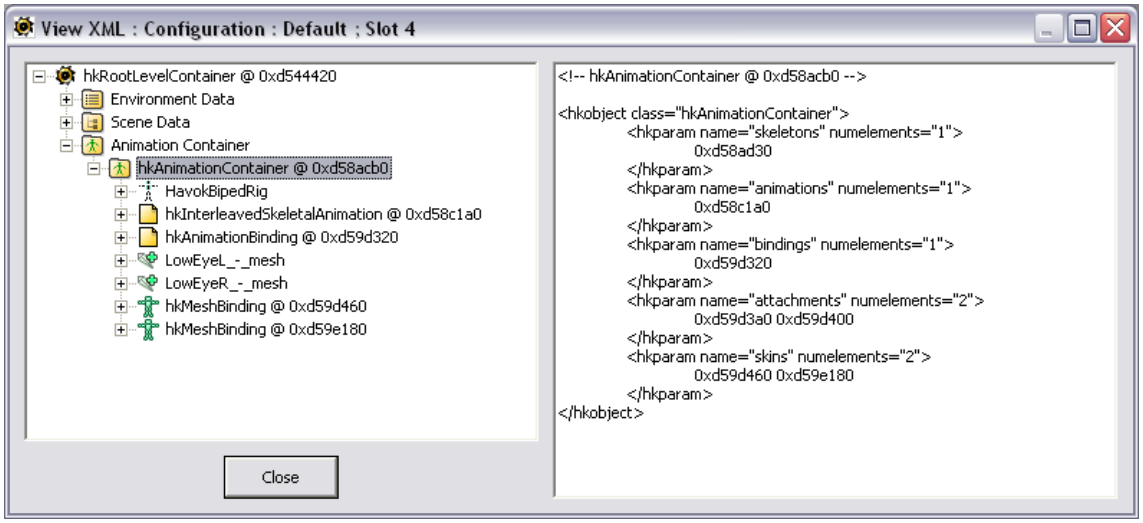
As you can see, this filter also has options associated with it (they appear in the right panel).




We will leave the default options as they suit us fine. You can check the documentation for this and other filters if you want to learn more about how they work. After this filter, add the Create Skin and Create Animations filters (also in the **Animation** category) - leave the default options as well. Finally, place the View XML filter again at the bottom of the filter list.




If you run (press the **Run Configuration** button) the filter processing, the View XML filter will show you the effect of all the processing:



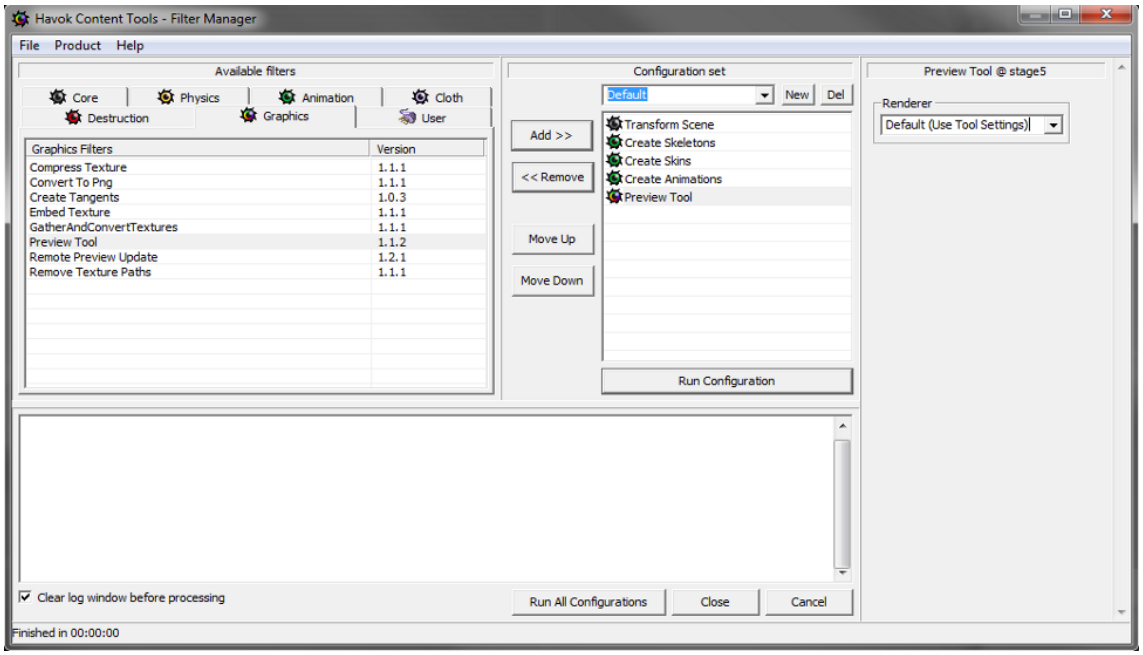

Notice how a new root-level object, an animation container, is now part of the content, and how it contains Havok Animation/Complete objects like `hkaInterleavedSkeletalAnimaton`, `hkaSkeleton` (HavokBipedRig), `hkaMeshBinding`, `hkaBoneAttachment`, etc., that we should be able to use at runtime.

#### 4.6.5.4 Previewing the Scene

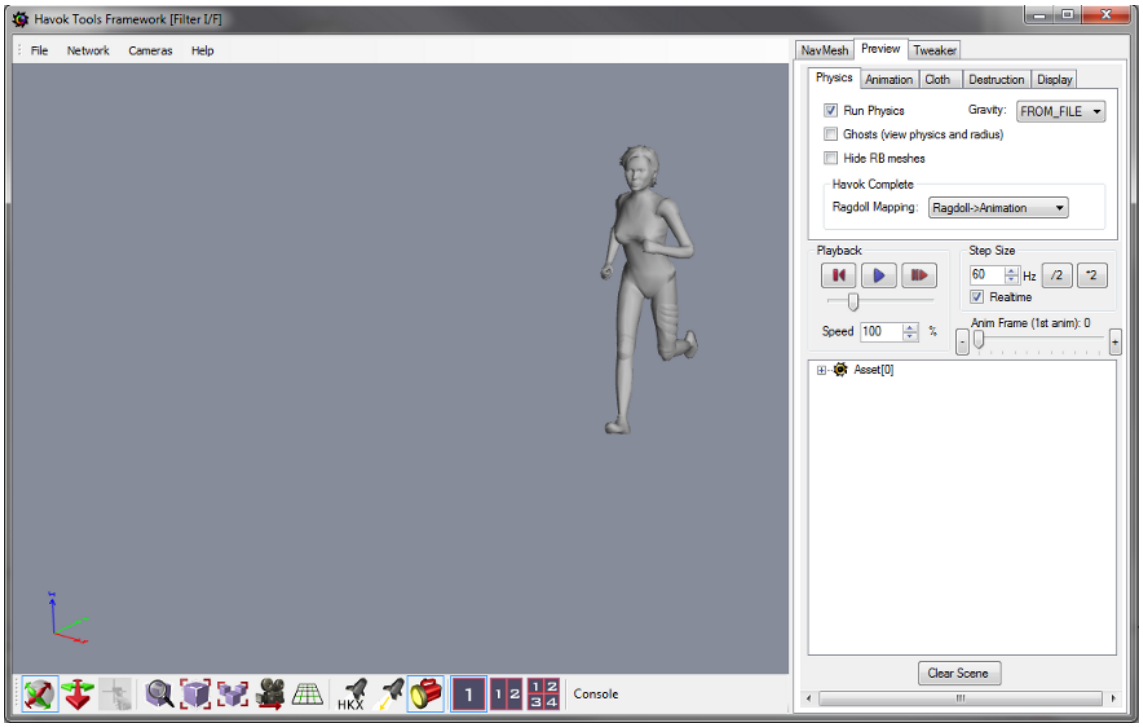
Although examining the contents by viewing the XML output give us an indication on how our processing went, in most cases visual inspection is the only way to ensure that the processed content matches our expectations.

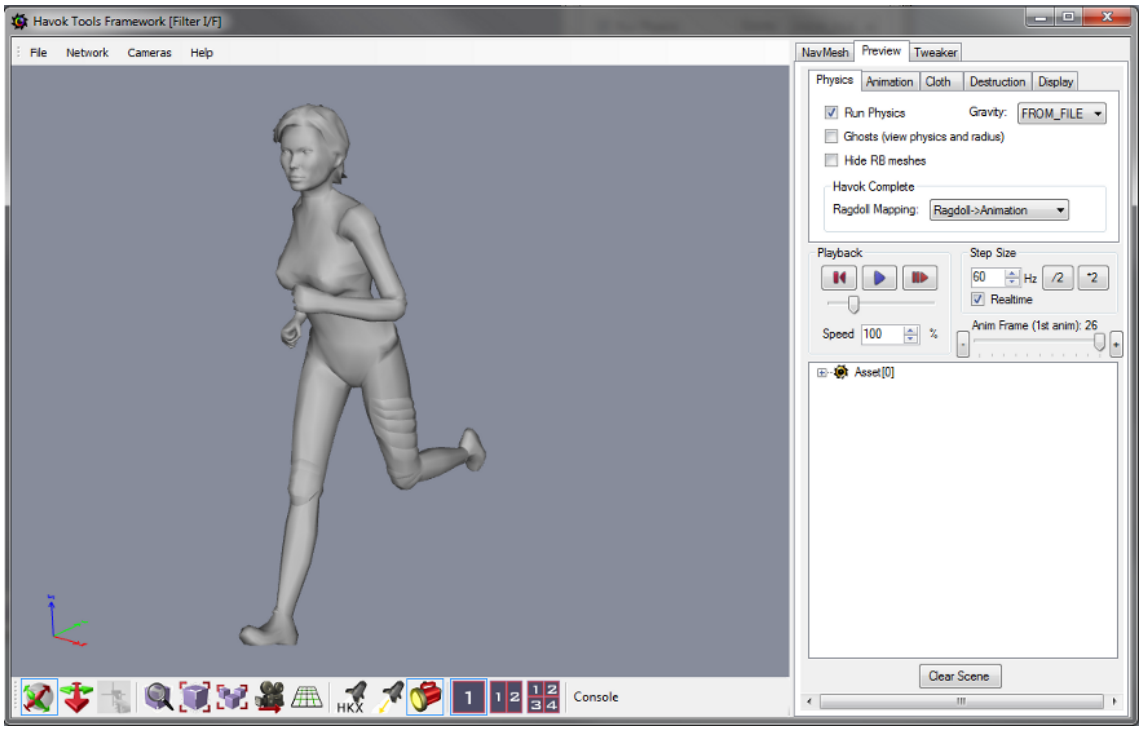
Havok provides the Preview Tool filter (in the **Graphics** category) just for that - this filter is built with the Havok runtime, including some display libraries, and will play back any Havok content it finds.

Replace the View XML filter at the end of the filter list with a Preview Tool filter, and click again on **Run Configuration** :




A modeless window will appear with our animation and skinning running in real-time:



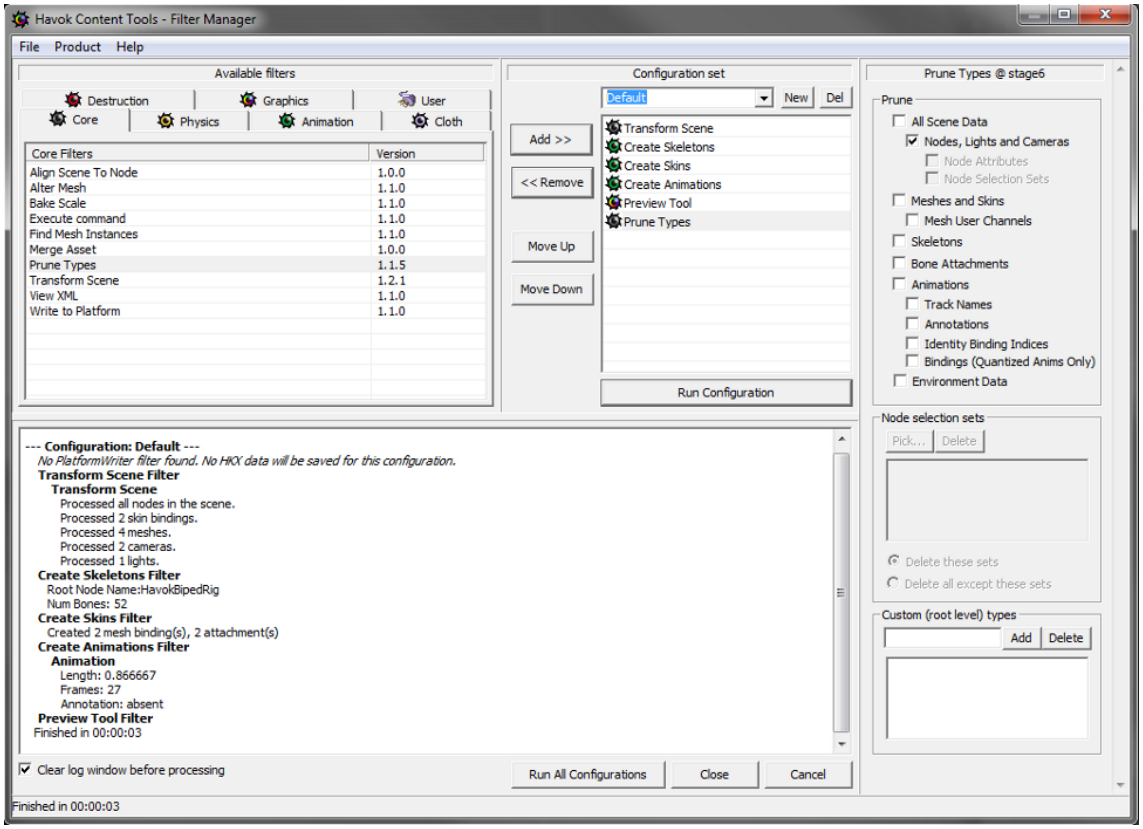




The Preview window gives you multiple options to change the speed of playback, the current frame in the animation, etc. Check the Preview Tool filter documentation for more details.

#### 4.6.5.5 Pruning Some Data

Most of the filters either modify (e.g. the Transform Scene filter) or add (e.g. the Create Skeleton filter) information to the asset. Sometimes it is useful to remove some of the data in the assets that is no longer relevant (particularly at the end of the processing). For example, once we've created our skeletons and animations, the original data regarding nodes and their keyframes is probably no longer needed (since it has been transformed into other classes).

We can add a Prune Types filter (in the **Core** category) after the Preview Tool filter. Leave the default options as they are - this will remove nodes (and their keyframes), lights and camera information:

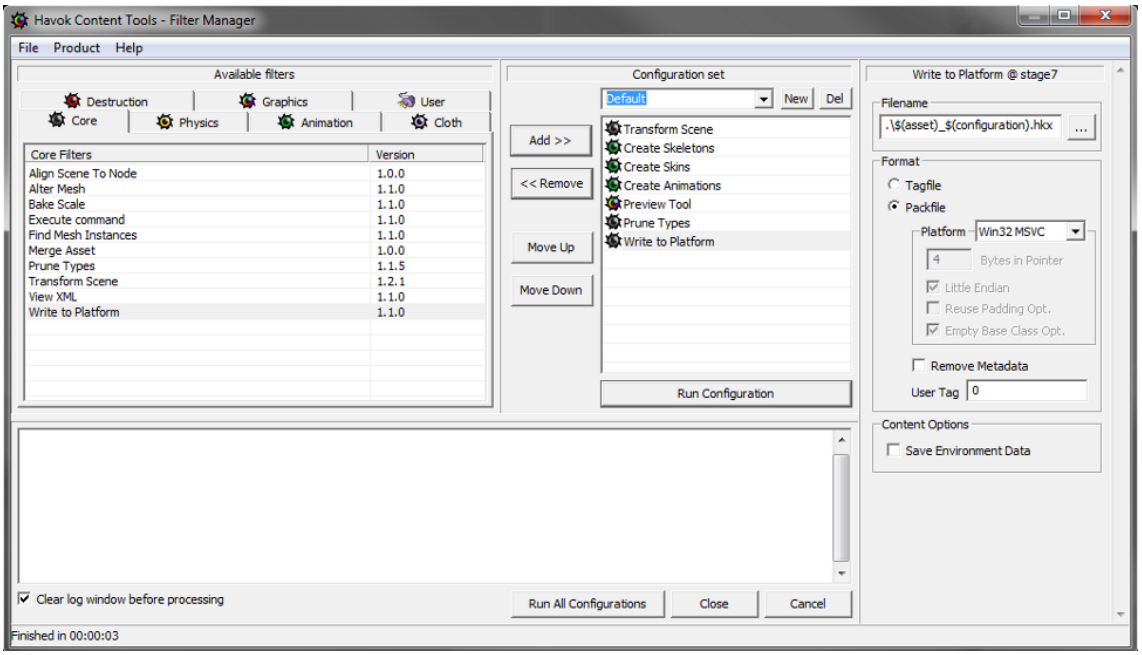



**Note:**
You could also add this filter before the Preview Tool filter - but since the Prune Types filter will remove camera information, the preview will have to use a default camera (rather than reflect the camera in the original scene).

#### 4.6.5.6 Writing the Processed Contents

So far, all the processing done by the filters has happened in memory, and hasn't been written to any file.

The Write to Platform filter (in the **Core** category) will serialize the processed content to a file. It can write either XML format or binary format for many different platforms. Add the Platform Writer filter at the end of the filter configuration. By default it will save a binary .hkx file for the Win32 platform in the same location as the asset. If you now process the asset again (click on **Run Configuration** ):




Observe the log at the bottom of the Filter Manager window. This reports that the file has been written alongside our XSI scene. This file is ready to be loaded into the Havok SDK.

This finalizes this first tutorial: you can find the final XSI scene in the "`exportBasics/tutorialEnd.scn`" file.

We'd like to encourage you to experiment with different filters and filter options. By checking the results in the filter processing log, as well as the output of the View XML filter, you should be able to get a good idea of what each filter does.

If you are a Havok Animation/Complete user, the following sections go a little further in processing this scene by doing some motion extraction. If you are a Havok Physics user, skip the following section and go to the Physics Basics Tutorial.

#### 4.6.5.7 Appendix (Animation/Complete Only): Extracting Motion

The animation we just exported has some *motion* on it - in between the first and last frame of the animation there has been some displacement on the character (it has moved forward).

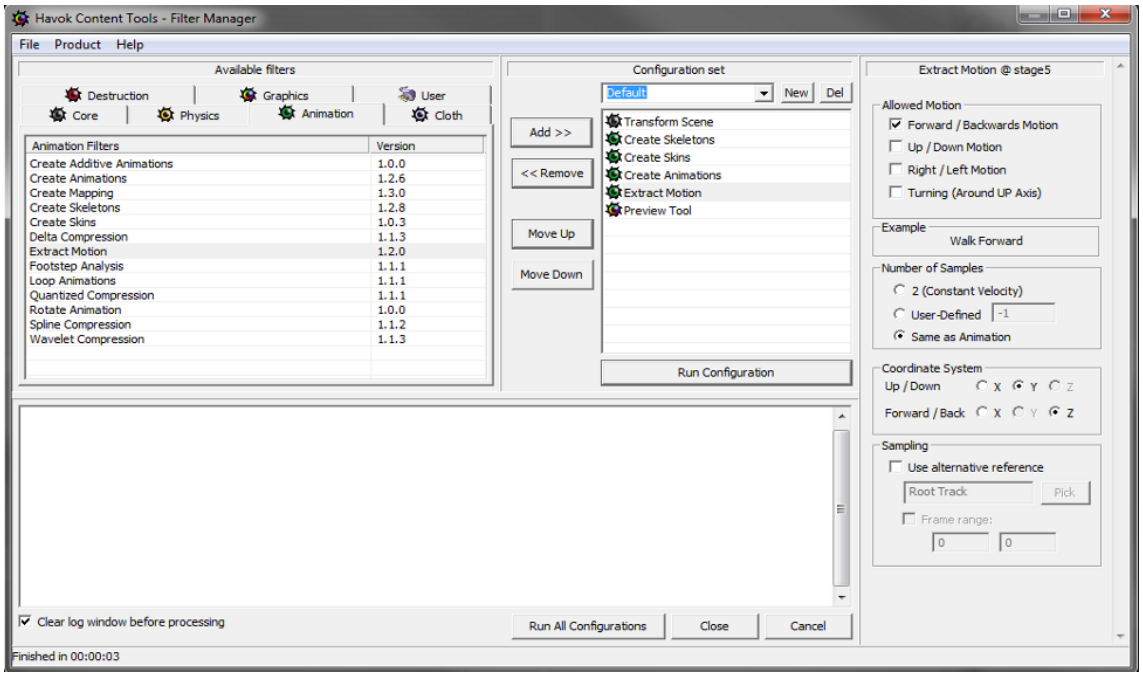
Motion extraction, explained in detail in the Animation chapter in this manual, is the process on which we examine an animation and divide it into two - a motion/displacement component (the moving forward during running) and a local animation component (the movement of arms, legs and body during the run cycle).

After motion extraction, thus, our running animation should become a run-on-the-spot animation, complemented by some information on how the character should displace when that animation is applied (the extracted motion).

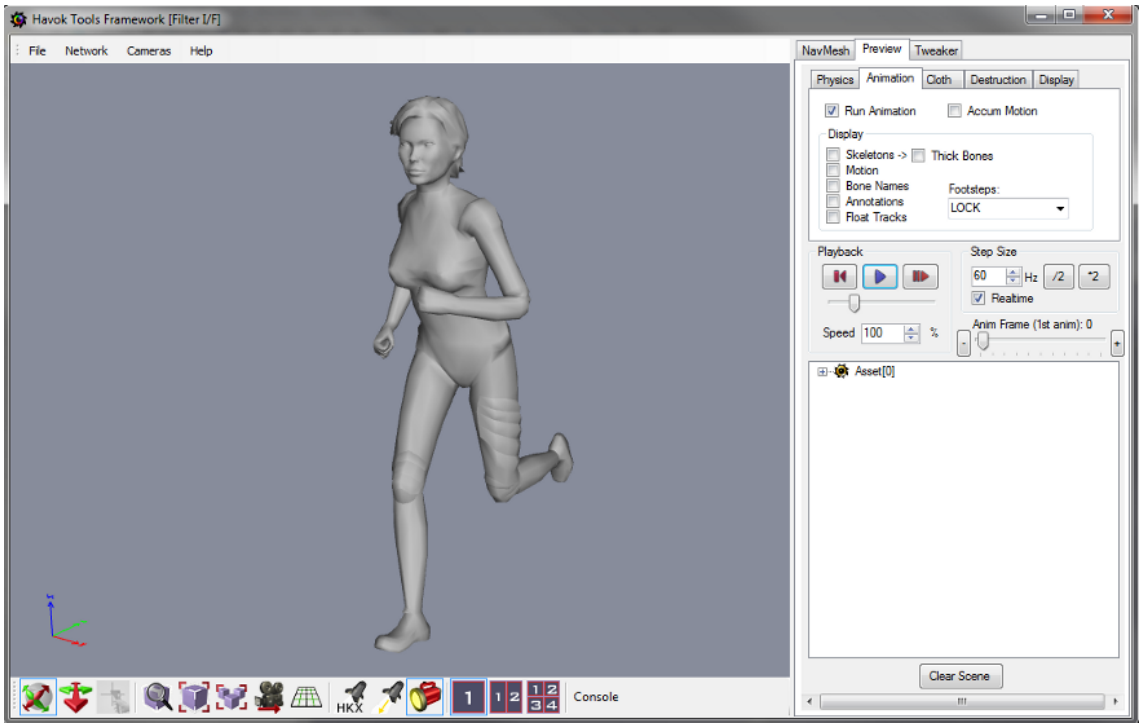
In this appendix to the tutorial we are going to do motion extraction to our run animation.

- Open the filter manager again (if you have closed it) by clicking on **Export**
- Add the Extract Motion filter (in the **Animation** category) to the filter list, just after the Create Animations filter (use the **Move Up** and **Move Down** buttons to place the filter in the right place).

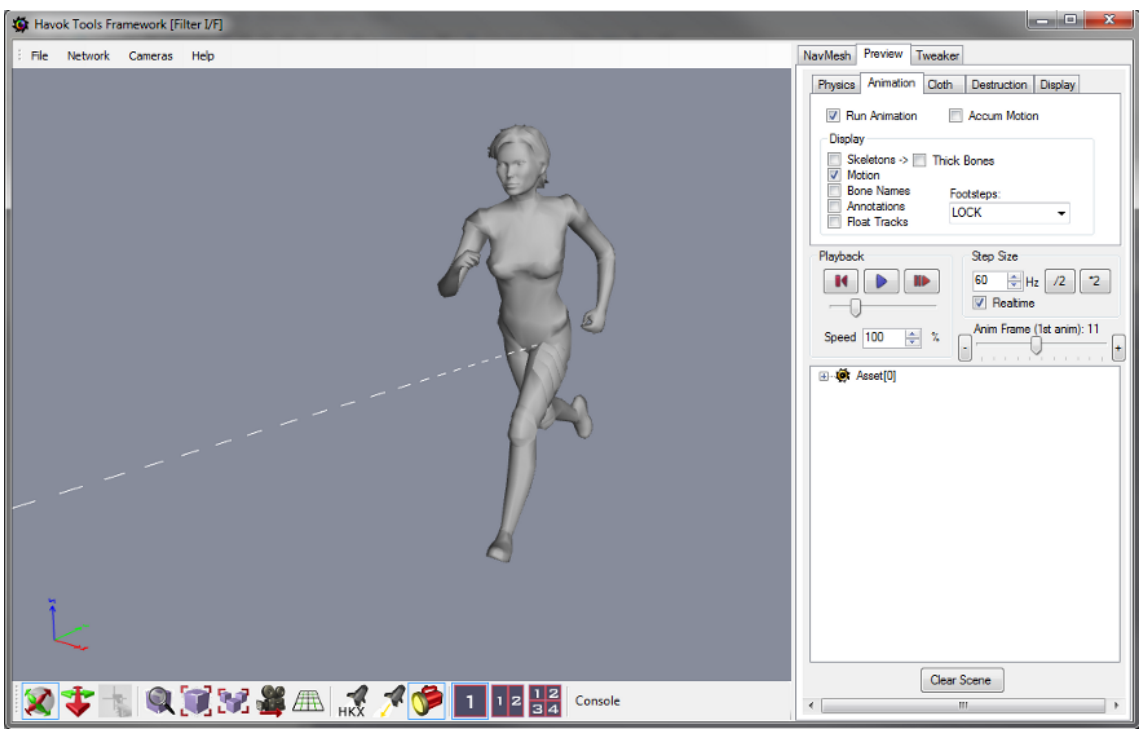
The filter options allow you to specify what kind of motion you want to extract from the animation. In our case the motion is a forward motion so we'll leave the **Forward/Backwards** check box on. But, since in our animation the up direction is Y (and not Z), change the **Up/Down** radio button in the **Coordinate System** box to **Y** . Similarly, since our character is moving forward alongside the Z axis, change the **Forward/Back** radio button to **Z** (instead of X)



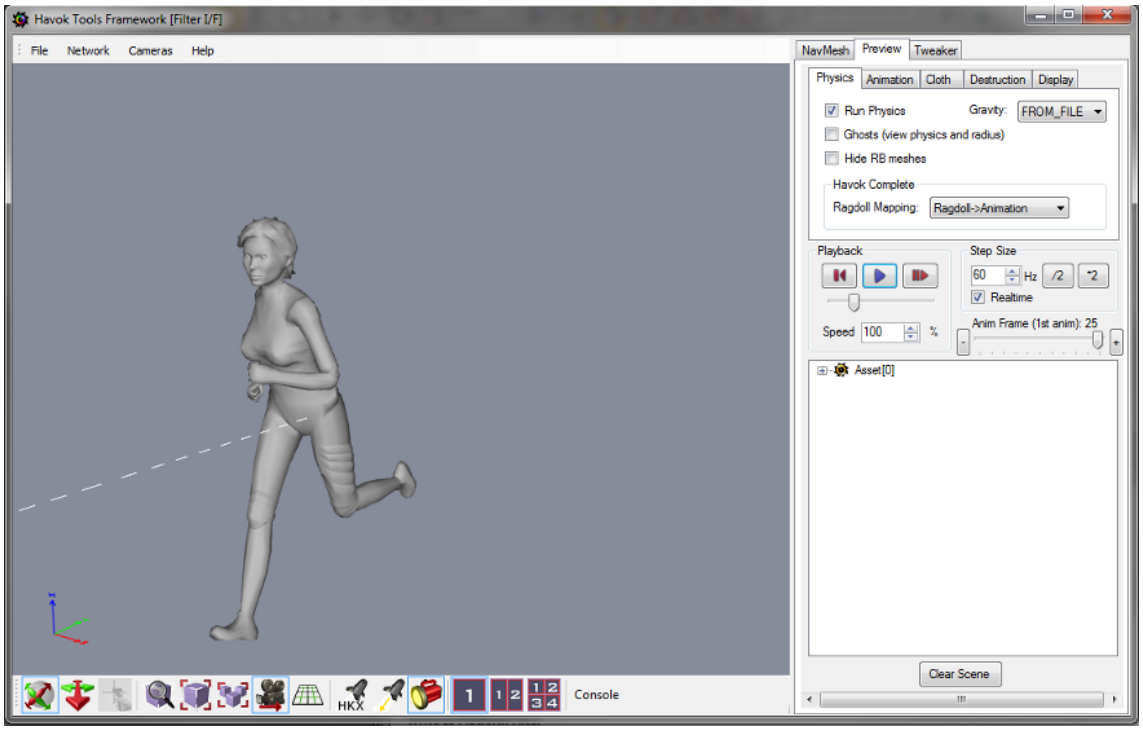

If we now run our filters, it will open the Preview dialog - you can see how now our animation is now a run-on-the-spot animation:




The Preview Tool filter knows about motion and it allows you to visualize it. In the preview dialog, click on **View** > **Extracted Motion** :




A dashed line is displayed showing the motion extracted from the animation. The preview can also apply and accumulate the extracted motion. In the **Animation** menu, click on **Accumulate Motion** :




Notice how the character now moves according to the motion that was extracted - notice also how that motion accumulates: the character moves forward continuosly over animation loops (instead of warping back to the starting position at the beginning of each loop).

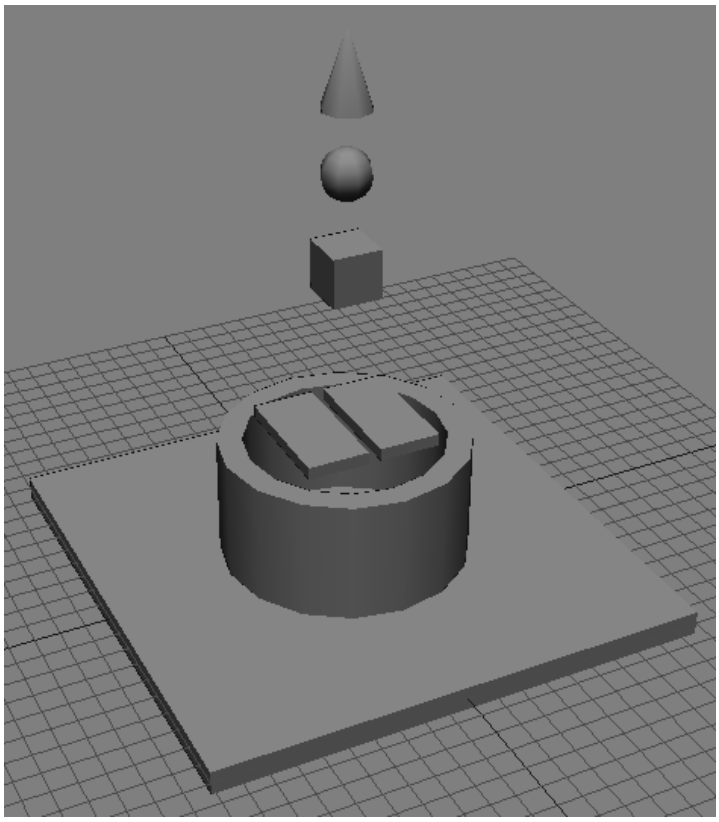
You can find the tutorial scene with motion extraction applied in the "`exportBasics/tutorialEnd_MotionExtracted.scn`" file.

### 4.6.6 Tutorial: Physics Basics

In this tutorial we are going to explore the basic operations for setting up rigid bodies and constraints in XSI, and processing them through the filter pipeline. This tutorial assumes that you are familiar with the basics of exporting and processing scenes, covered by the Export And Animation Tutorial. We also recommend that you take a look at the rigid body and constraint concepts sections of this documentation.

#### 4.6.6.1 Getting Started

Start by loading the scene "`physicsBasics/TutorialStart.scn`", in the *tutorials* project (which should be installed in `[INSTALLDIR]/Addons/HavokContentTools/tutorials`):

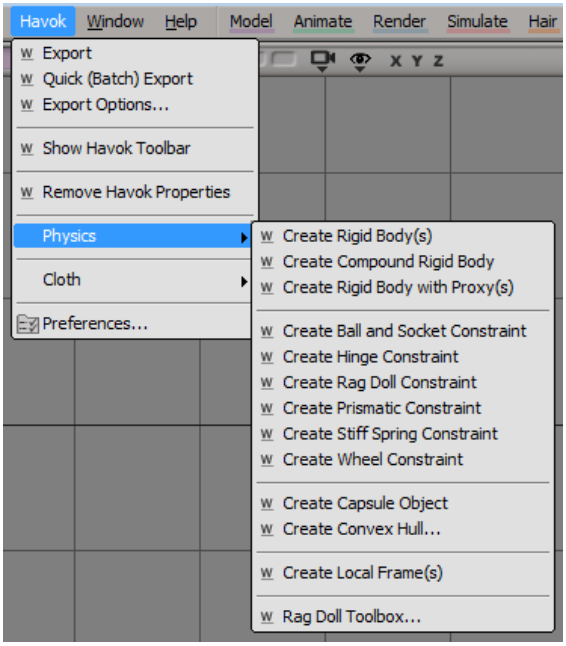


In this scene, we have a set of objects: a cone, a sphere, a box, a trap door, a can and a ground. We want to attach rigid body properties to them so they behave as they would in reality.

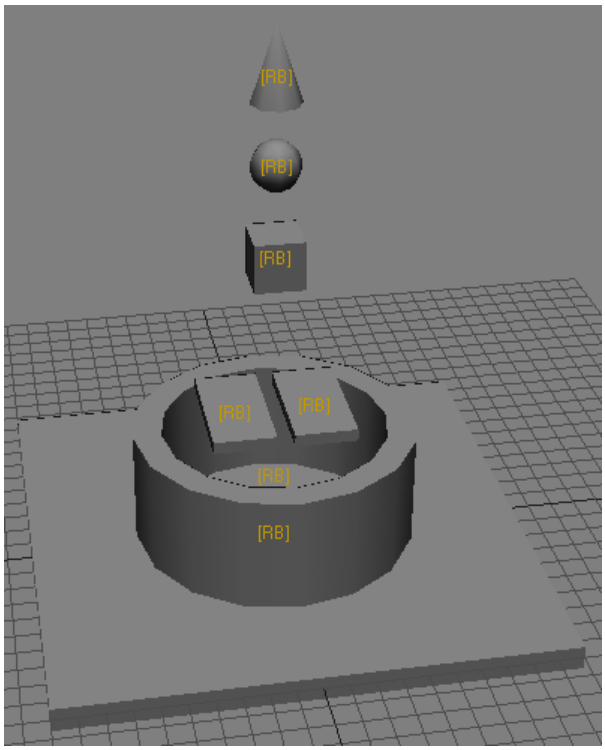
#### 4.6.6.2 Setting Up Rigid Bodies

To start with, we need to make the objects in this scene become rigid bodies, i.e., attach rigid body information to them so we can later on create rigid bodies and simulate them using the Havok SDK.

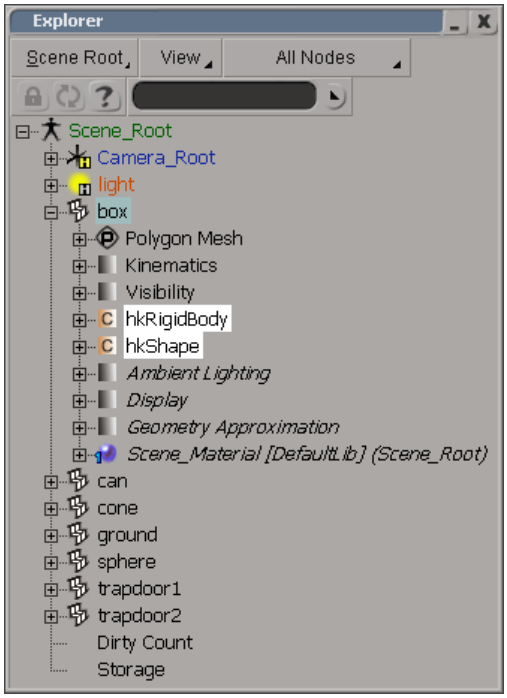
The Havok Content Tools provide a set of XSI physics tools in order to do so easily. These tools are available through the **Havok** menu and toolbar:



In this scene, we want to create a rigid body out of every single object, so let's start by doing so. Select all the objects in the scene, and click on the **Create Rigid Body** button or menu option. Notice how all the objects in the scene are now are labeled with the text " [**RB**] ", indicating that these objects have now rigid body properties. Note that the square brackets around the " **RB** " indicate that these are all currently fixed rigid bodies.



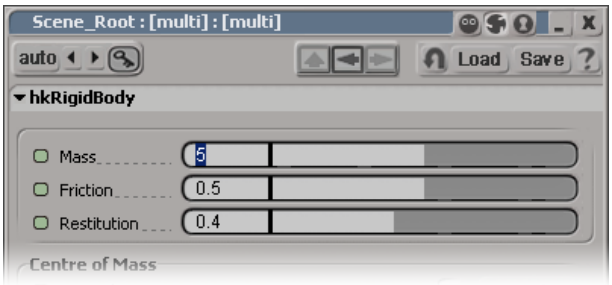

In XSI, rigid body properties are stored in custom parameter sets. For example, if you select the suspended box in XSI's explorer view, you should see that two parameter sets have been attached to the box:




The Rigid Body Parameter Set (`hkpRigidBody`) contains information about the dynamic behaviour of the object: mass, friction and restitution are specified here. The Shape Parameter Set (`hkpShape`) contains information about the representation of the object for collision detection.

We will be exploring these parameter sets a little bit more later on. For now, open the rigid body parameter set's property panel (by double-clicking on the `hkpRigidBody` parameter set). Notice how the mass of the box is currently zero - this is the default mass for all new rigid bodies. Since Havok will consider a mass of zero to be an indication of the rigid body being fixed in space (not moving), we want to make sure that we specify a mass for all the dynamic objects in our scene: the cone, the sphere, the box and the trap doors.

Set the **Mass** attribute of each dynamic object to a value different than zero (5Âăkg for example):




After doing this, we have a scene set up with 7 rigid bodies, 5 of them dynamic/movable and 2 of them (the ground and the can) fixed.

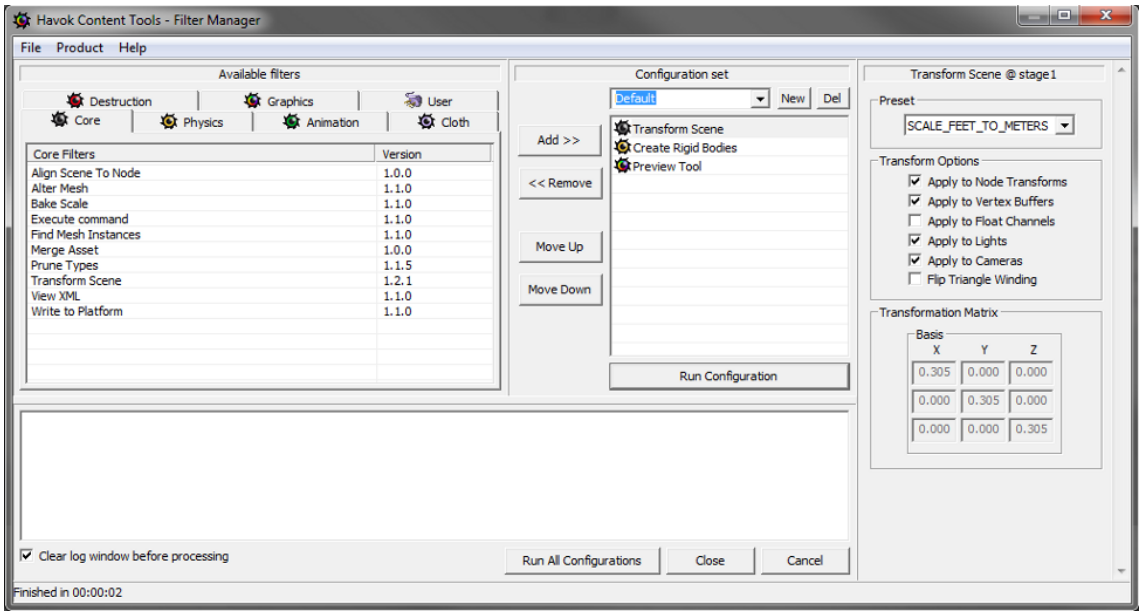
#### 4.6.6.3 Previewing the Scene

Now that we have some content set up, we'd like to preview it and check that everything behaves as expected. In the previous tutorial we saw how to export, process and preview assets using the XSI scene exporter and the filter pipeline. We are now going to do the same, but this time using some processing specific to physics assets.

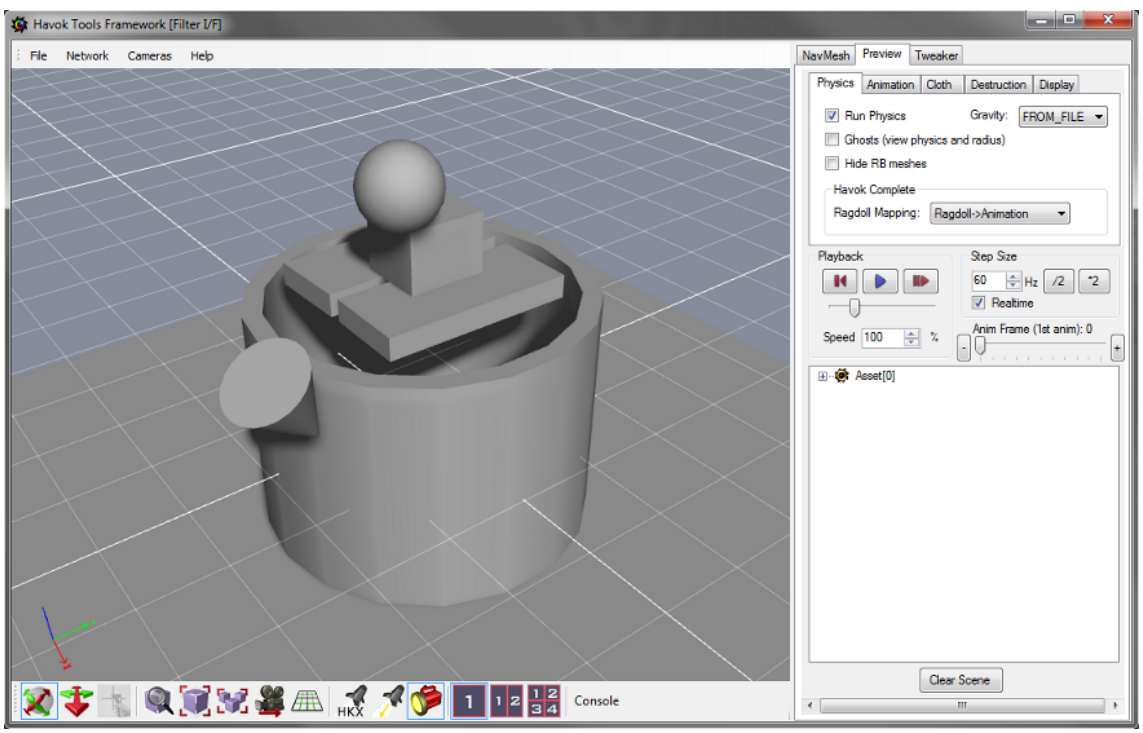
Start the filter manager by clicking on the **Export** button or menu option. You should begin with an empty filter setup - remove any filters which may be present.

Insert, in this order, the following filters:

1. Transform Scene ( **Core** category): Since the scene units are pretty big, and Havok works in meters, set the **Preset** parameter to **SCALE_FEET_TO_METERS** (for example).
2. Create Rigid Bodies ( **Physics** category): This filter will detect the rigid body information we assigned to our objects and will create rigid bodies based on it.
3. Preview Tool ( **Graphics** category): We want to visually inspect the behaviour of our objects, so we will use this filter to render and simulate them.




If we now process our asset (press the **Run Configuration** button), a preview window appears with our scene. Press the **Play/Stop** button to start the simulation:



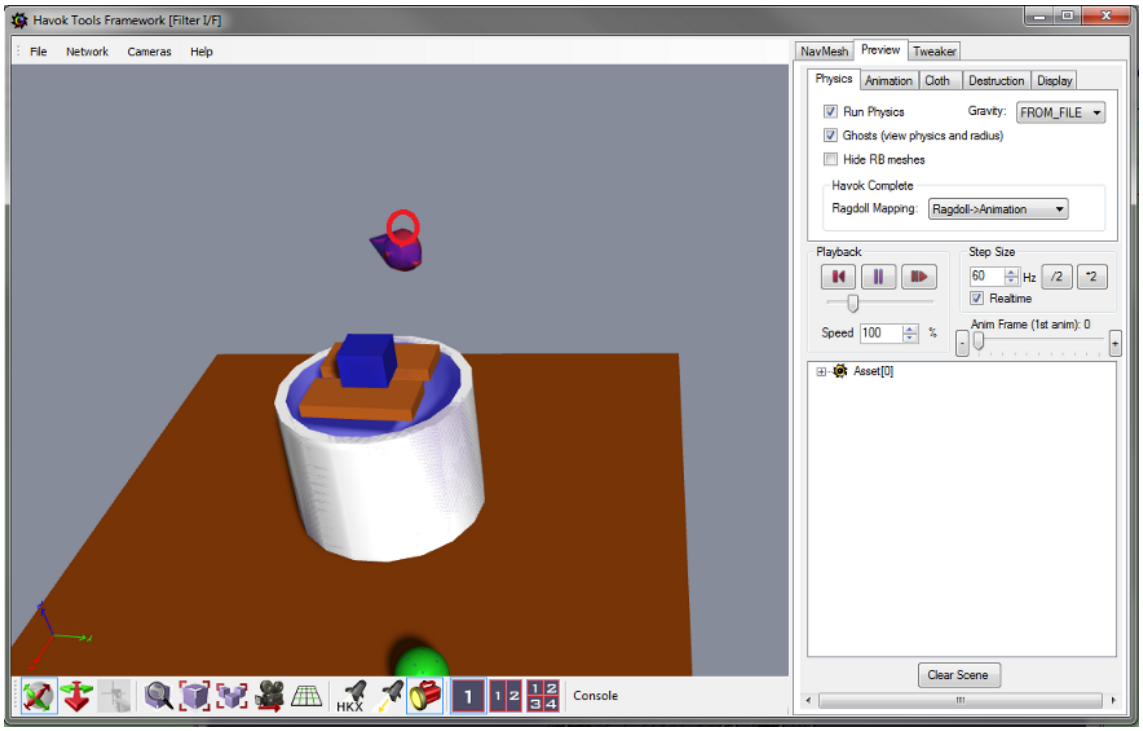

You will notice that all the objects fall as expected, but there are several things that are not quite right:

- The trap doors do not behave as such, and are just falling as any other object. This is because we have not set up any constraint to specify that they should behave as hinged doors. We will take a look at this later on this tutorial.

- None of the objects fall inside the can - it is as though there is an invisible lid on it.

If you switch on the menu option **View** > **Physics Ghosts** you should see a ghost representation of the rigid bodies displayed in the preview (in alpha blended blue). That shows that there is indeed a lid on the can. It looks like the can is being simulated as a closed volume!

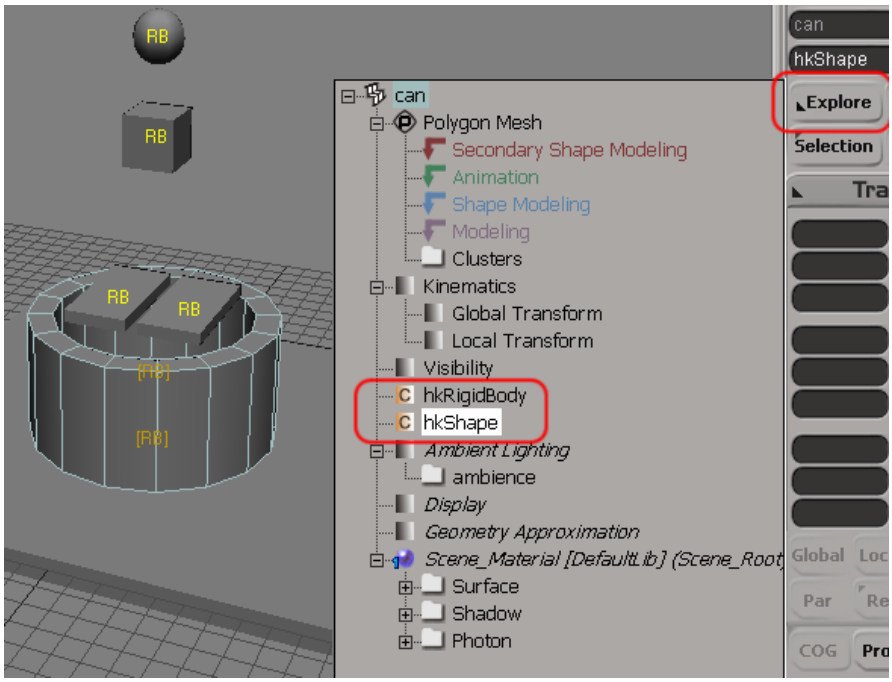
**Mouse Picking**
You can interact with the objects in the scene by using *mouse picking*: with the mouse cursor over one of the movable objects, press the **SPACE** bar and, without releasing it, move the cursor around - the rigid body will behave as if a spring was attached between itself and the mouse pointer. Release the **SPACE** bar to release the object.

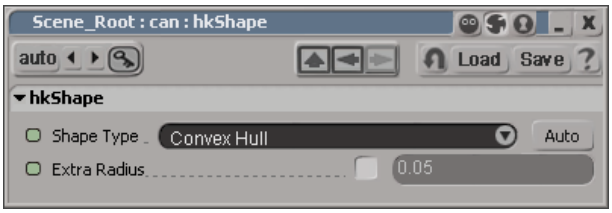



Close the preview and the filter manager and return to our XSI scene.

#### 4.6.6.4 Shape Parameters

Let's take a look at the can. Since our problems are related to collision detection, we'll take a look at the shape information associated with it. Open the shape parameter set's property panel - do this by selecting the can in the viewport, opening the 'selection' popup from the tools panel, and double-clicking on the `hkpShape` custom parameter set:



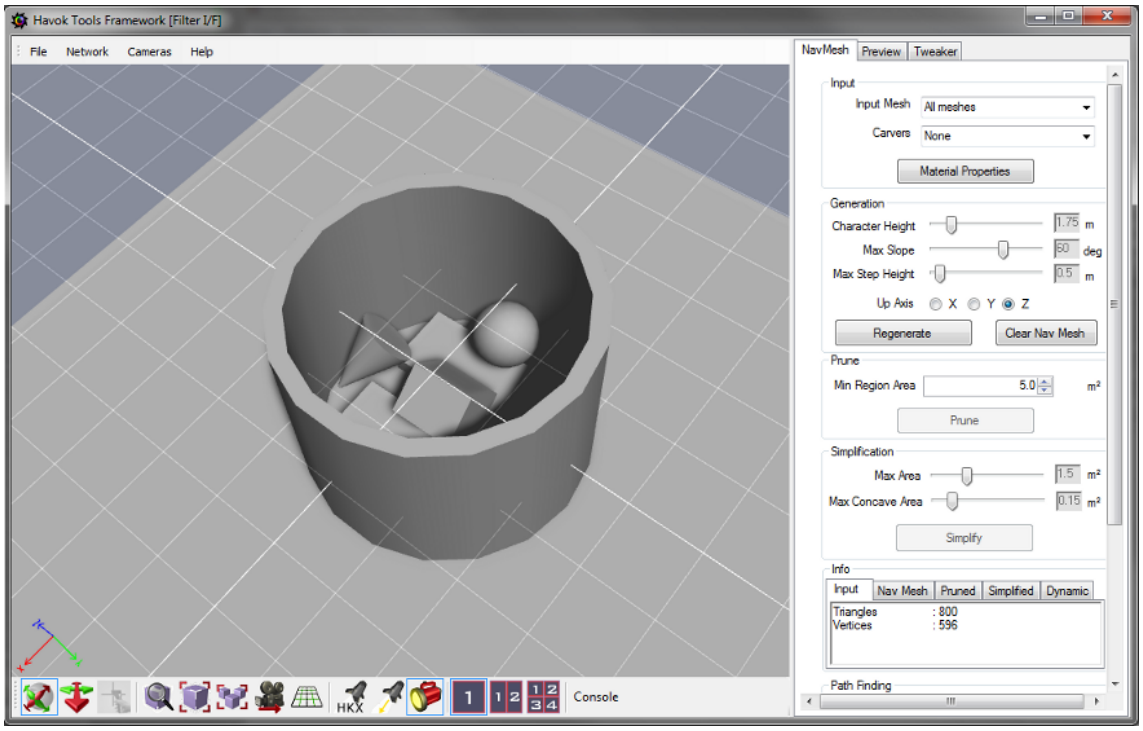

Notice how the Shape Type parameter is set to "Convex Hull".

When creating a rigid body from a mesh, a value for this Shape Type parameter is guessed by examining the object's geometry in XSI. Thus, the default Shape Type for boxes is (Bounding) Box, the default Shape Type for spheres is (Bounding) Sphere, etc. For other arbitrary objects the default Shape Type is Convex Hull. The Convex Hull of an object is the tightest convex volume that includes all the vertices of the object. The best way to visualize a convex hull is by imagining shrink-wrapping the object. In our case, using the convex hull of the can results on simulating a closed cylinder, rather than an open one, leading to the behaviour we observed in the preview.

In our case we want to use the exact mesh of the object instead of the convex hull, so we need to change the Shape Type parameter to **Mesh** .

**Note:**
The reason why the default Shape Type is Convex Hull and not Mesh is that convex objects are much faster to simulate than concave meshes and, in many cases, simulating a mesh as its convex hull gives very good results. In addition, movable 'mesh' objects are not really suitable for interactive simulations (a warning will be generated if used).

After changing the shape type to Mesh, if you now **Export** and process the scene again you will see that the objects now properly fall inside the can during the simulation:



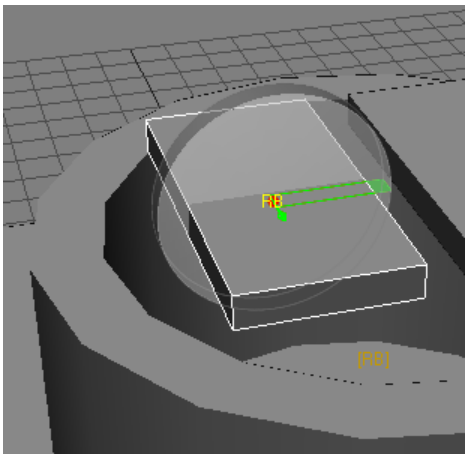

Close the preview and the filter manager and return to our XSI scene.

#### 4.6.6.5 Setting Up Constraints

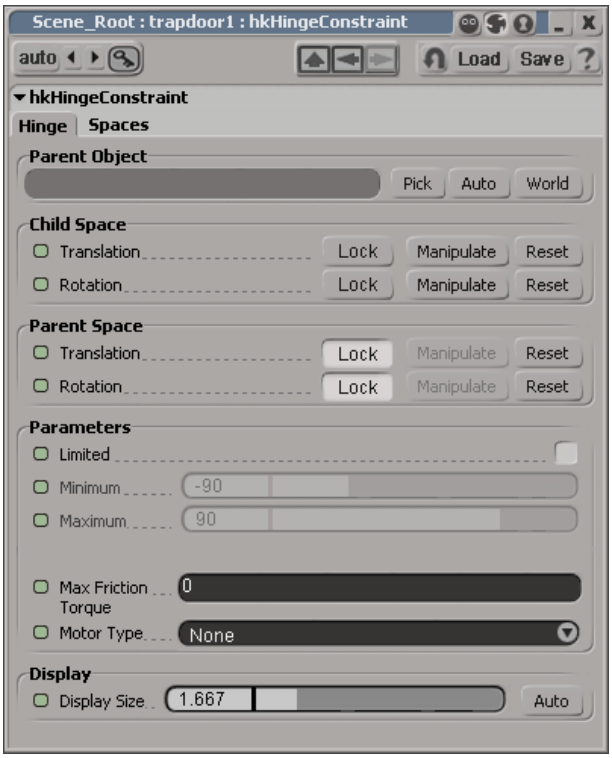
Another thing we want to improve in our scene is the behaviour of the trap doors: currently they behave just like the other objects (they just fall). We want them to behave as if they were constrained to their location by hinges.

The Havok Content Tools give you the ability to set up and visualize constraints in XSI. In our case, we want to create a hinge constraint - a constraint that will limit the movement of the trap doors to a rotation around a single axis.

Select one of the trap doors and click on the **Create Hinge Constraint** button or menu option. This adds a **Hinge** constraint parameter set to the object, which is represented in the viewport as shown:




Examine the parameters of the hinge constraint (i.e. the `hkHingeConstraint` parameter set) in a property editor:

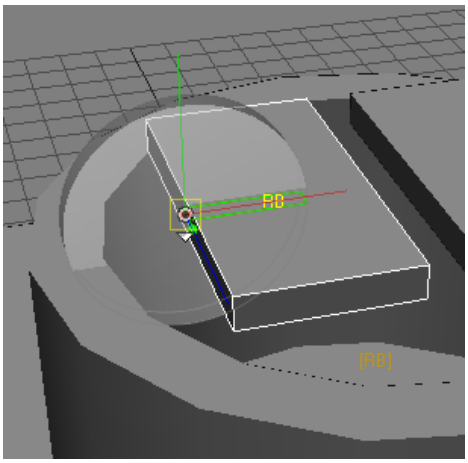



First of all, notice how there is no parent object specified: this means that this constraint will only affect this rigid body: its position, orientation and limits will be absolute, rather than relative to another (parent) rigid body. This is what we want.

The constraint is represented in the viewport as an axis and a volume of allowed rotation (with another axis specifying where zero-rotation is). By default these axes are aligned to the object's Z and X axis, and the constraint is placed at the object's pivot point. In our case, however, we'd like to place the constraint on the side of the trapdoor and orientate it so that the trap door will rotate up and down rather than sideways. We can change the position of the constraint in two ways: by modifying the object pivot or by manipulating the constraint spaces.

### Positioning the constraint by modifying the object pivot

With the trap door selected, select XSI's translate tool and hold the " **Alt** " key - this will put the tool into pivot mode. Then, drag the pivot of the trap door to the side which we want to act as the fixed point of the door. Observe how the constraint representation updates to follow the pivot point:

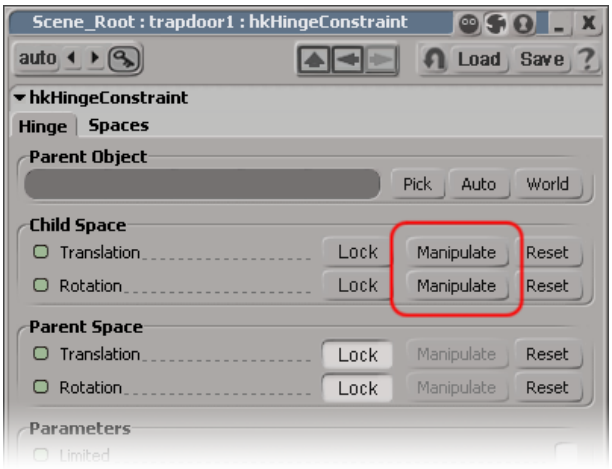



**Note:**
An option is provided by the Havok Content Tools to 'include pivot in transforms'. If this option is disabled then moving the rotate pivot of an object will have no effect whatsoever on the constraint. See the XSI Export Options for more.

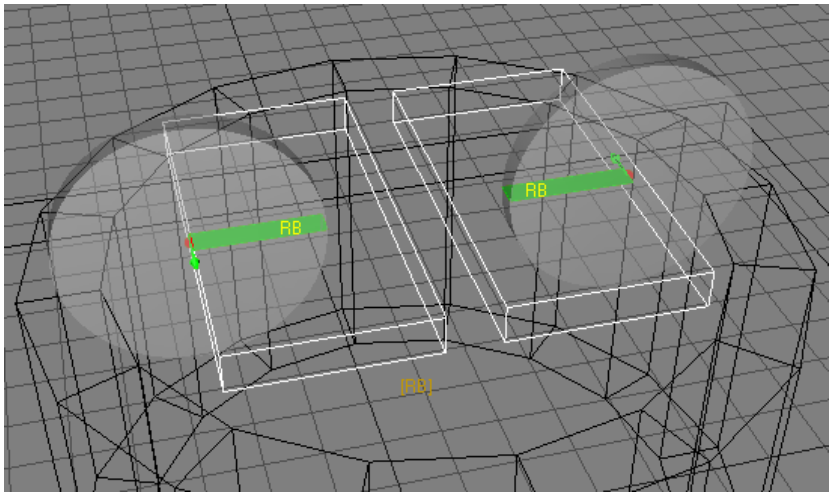
### Positioning/orientating the constraint by manipulating its constraint spaces

As observed, the constraint position normally follows the pivot point of the object. However, we cannot change the orientation of the hinge using the pivot point. We can only do this by manipulating the constraint space(s).

Let's now experiment with the orientation of the hinge constraint for the trap door. Open the property panel for the hinge constraint parameter set. In the 'child space' section under the 'hinge' tab, click one of the 'manipulate' buttons. Now drag the manipulator in the viewport to position/orient the hinge as desired. We want the main axis to point along the length of the trap door, and the zero-rotation axis (the one within the volume) to point towards the other trap door (as in the screenshot above).




Repeat the above process with the other trapdoor to create a hinge constraint with the appropriate position and rotation. In this case try positioning the constraint by manipulating the constraint space only, instead of moving the pivot point. Our setup should now look like this:

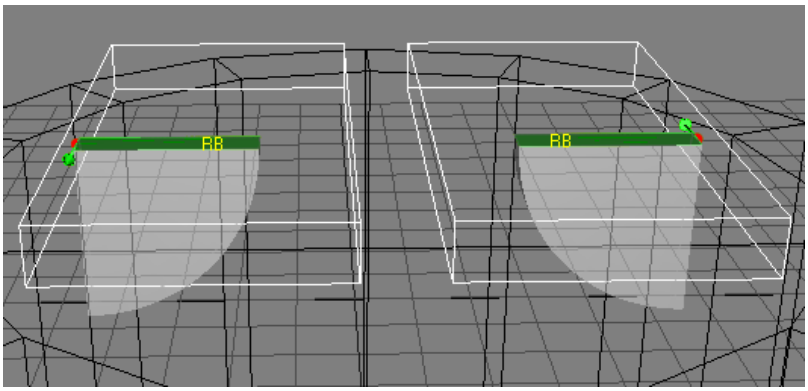



**Note:**
If you prefer, you can edit/tweak the spaces numerically, by switching to the 'spaces' tab in any constraint's property panel. This is useful if you want to specify an exact rotation, for example 90 or 180 degrees.

### Limiting the hinges

By default, hinge constraints allow full freedom of rotation around the main axis. It is often desirable to limit that amount of rotation to a specific range.

Select one of the trap doors, and examine its hinge constraint parameters in a property editor. Enable the **Limited** checkbox. Notice how the **Min** and **Max** parameters are now enabled, and how the limits are reflected in the hinge volume in the viewport (the limits are specified counterclockwise from the zero-rotation axis). Tweak the limits of each hinge so that they appear as follows:

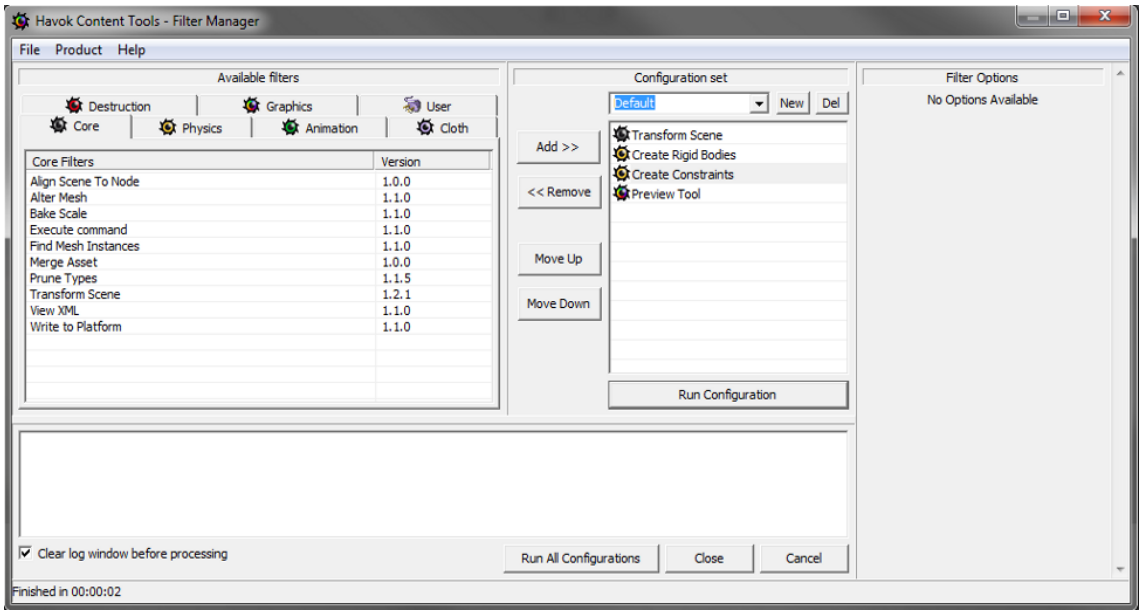



#### 4.6.6.6 Previewing the Scene

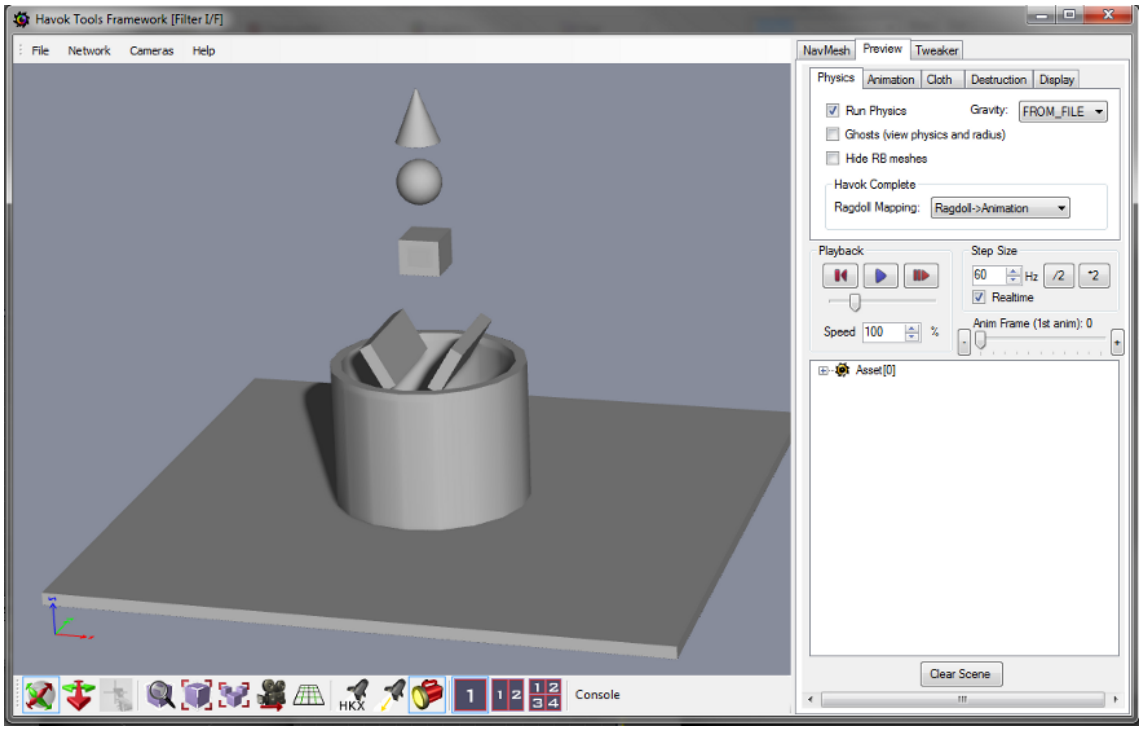
Now that we have set up our constraints, let's see what effect they have on the simulation.

Open the filter manager again by exporting the scene .

We now need to add a new filter, the Create Constraints filter (in the **Physics** category), in order to create constraint objects from the data we set up in XSI. This must be placed after the Create Rigid Bodies filter, and before the Preview Tool filter.




If you now execute the processing by pressing **Run Configuration** , you will see the constraints in action:




You can check that the limits we applied to the hinges are working by using mouse picking - you will notice that the doors won't move beyond the limits we specified.

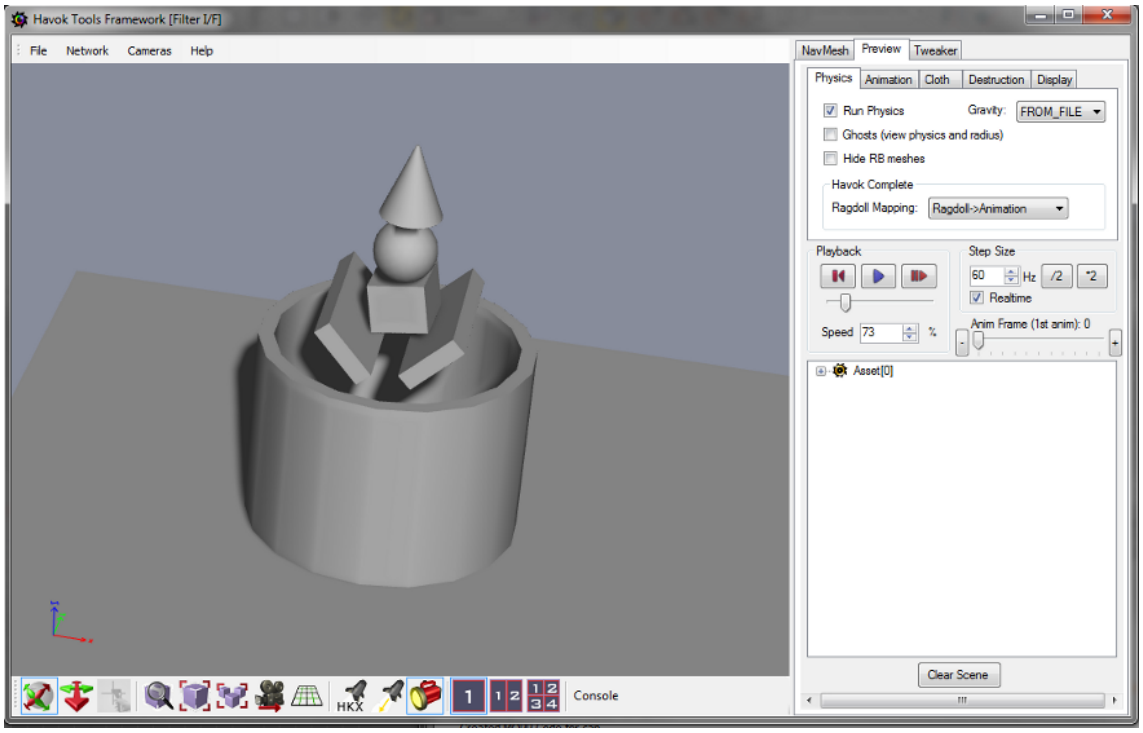
Close the preview window and the filter manager, and return to XSI.

#### 4.6.6.7 Adding friction to constraints

Our setup is almost finished. The last thing we'd like to fix is the fact that the trap doors open even before the objects hit them - this is due to the gravity in the previewer. We'd like to add some friction to the hinges so the doors only move when pushed by the objects falling on them.

Select each of the trap doors and open the hinge constraint parameter set in a property editor. Change the **Max Friction Torque** parameter to 40. This parameter specifies how much torque (angular force) is absorbed by the hinge's friction - only torques above this value will cause the hinge to move.

If you now export and preview the scene again you will notice how the trap doors remain closed until the objects fall on top of them.




And this finishes our tutorial. You can find the fully completed tutorial in the scene file "`physicsBasics/tutorialEnd.scn`".

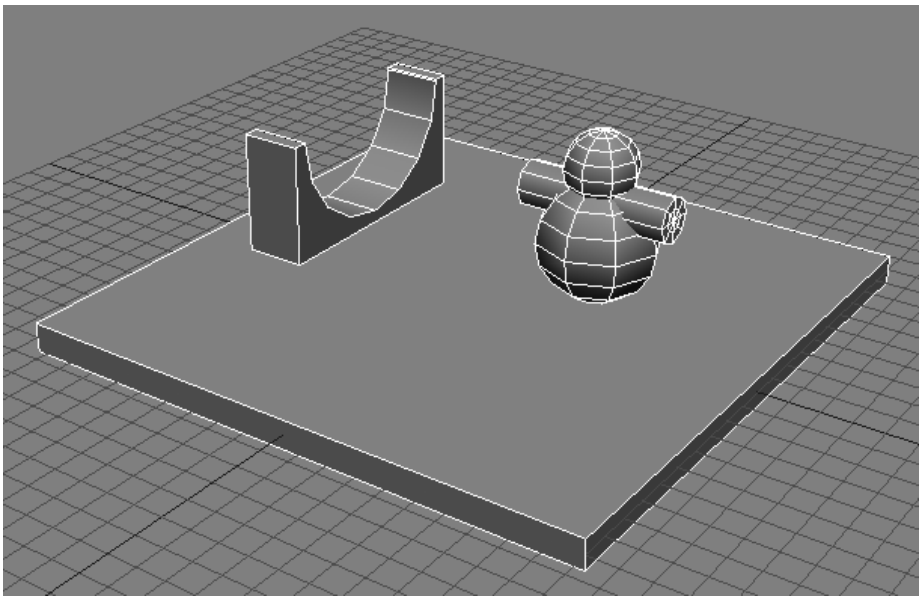
Now would be a good time to try tweaking the parameter values of each of the Rigid Body and Shape parameter sets in the scene, to familiarize yourself with the effects of each. Run the export and preview process any time you want to see the results.

### 4.6.7 Tutorial: More on Rigid Bodies

In this tutorial we will explore some more concepts regarding rigid body setup, such as creating compound rigid bodies and changing the center of mass. The tutorial assumes that you are acquainted with the basics of exporting and processing assets and working with rigid bodies. We recommend you follow the first two tutorials (Export and Animation Basics and Physics Basics) before proceeding with this one.

#### 4.6.7.1 Getting Started

Start by loading the scene "`moreOnRigidBodies/TutorialStart.scn`", in the *tutorials* project (which should be installed in `[INSTALLDIR]/Addons/HavokContentTools/tutorials`):



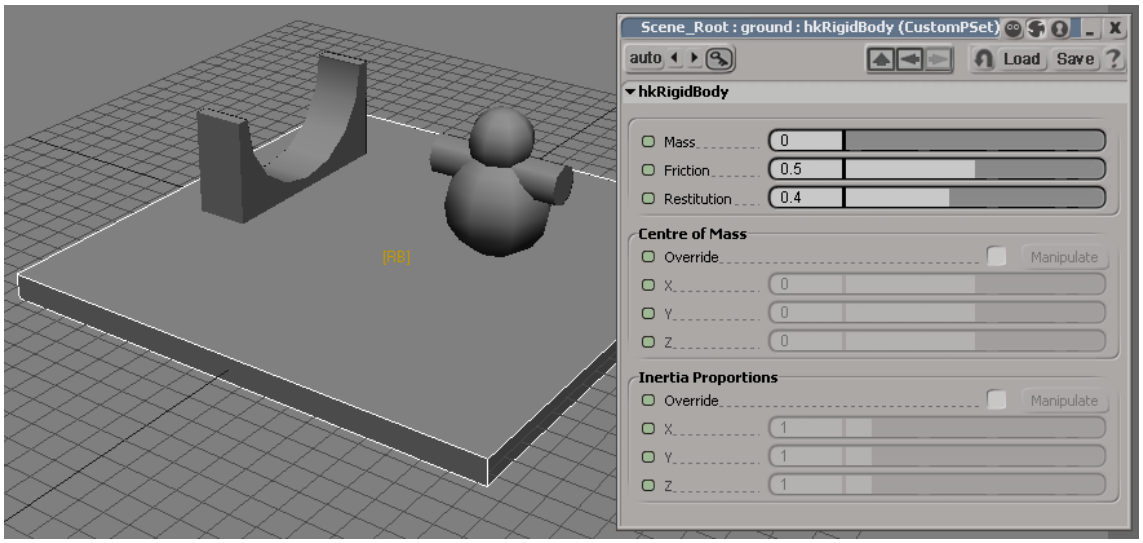
In the scene you should see the following:

- Two spheres and a cylinder, making up the shape of a little toy man
- A concave object (a little ramp)
- A ground box

We are going to create a rigid body object for each of the toy man and the ramp, and we will use the ground box as a fixed body so that we can preview the behaviour.

#### 4.6.7.2 Creating a Fixed Rigid Body

Let's start by making a rigid body out of the ground box. Select it and press the **Create Rigid Body** button or menu option.



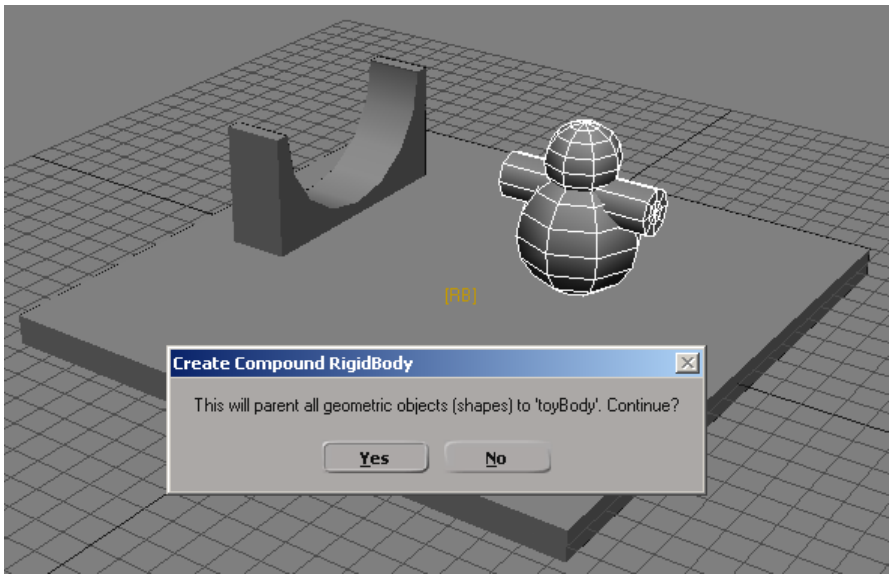

The default mass is zero, so the ground box will be fixed in space (notice the square brackets around the label, indicating a fixed body). That's exactly what we want.

#### 4.6.7.3 Creating a Compound Rigid Body

For the toy man we want to create a single rigid body from the two spheres and the cylinder. When we create a rigid body like this out of multiple objects, we are creating a compound rigid body.

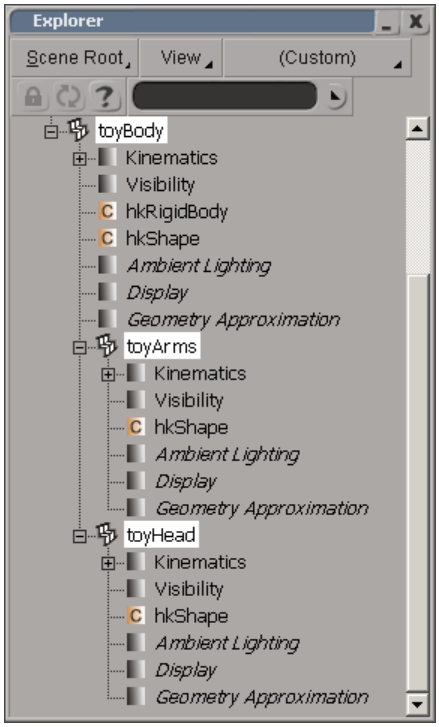
When creating a compound rigid body, one of the objects needs to take the role of rigid body - this object will hold the rigid body information, and the rigid body will assume this object's name. The other objects need to be children of this rigid body object. If the objects are not in the correct hierarchy already, the last object selected will be considered to be the rigid body object and the other objects will be reparented accordingly.

Select, in turn, the head, the arms and finally the body of the toy (holding SHIFT for multiple selection). Then click on the **Create Compound Rigid Body** button or menu option:

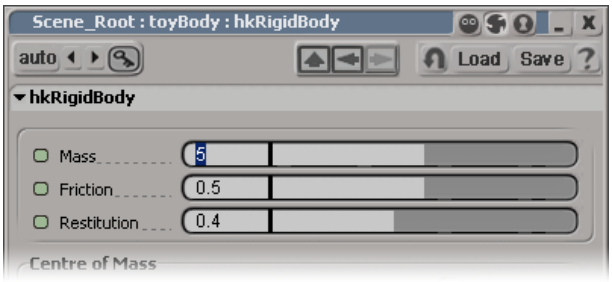



A confirmation dialog appears. This is because, as we mentioned, compound rigid bodies need to form a hierarchy. The Havok Content Tools have assumed that the last object selected (*toyBody*) should become the parent (the one that will hold the rigid body information) - but is asking us for confirmation before proceeding to reparent the arms and the head. Click **Yes** to continue.

Now, *toyArms* and *toyHead* become reparented to *toyBody*. Rigid Body information (a Havok Rigid Body Parameter Set) is be added to *toyBody*. Shape information (a Havok Shape Parameter Set) is added to all three objects (as the three of them are used for collision detection). Verify this setup using the scene explorer:



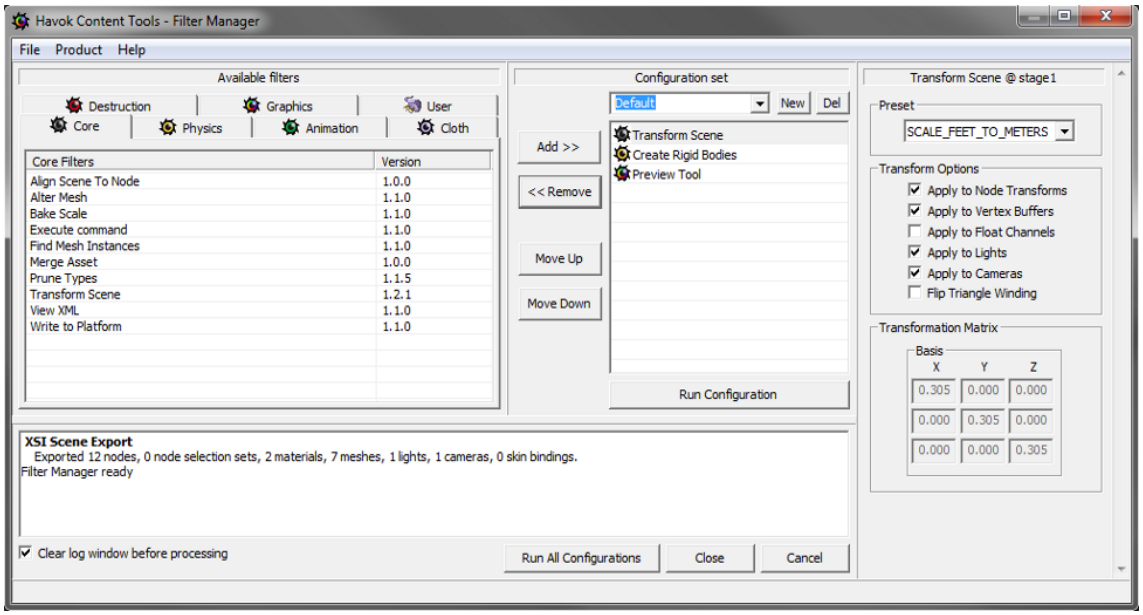
All rigid bodies have a default mass of zero. We want this rigid body to be dynamic, so open the property editor for the rigid body parameter set and set the mass to a different value, let's say 5 Kg:



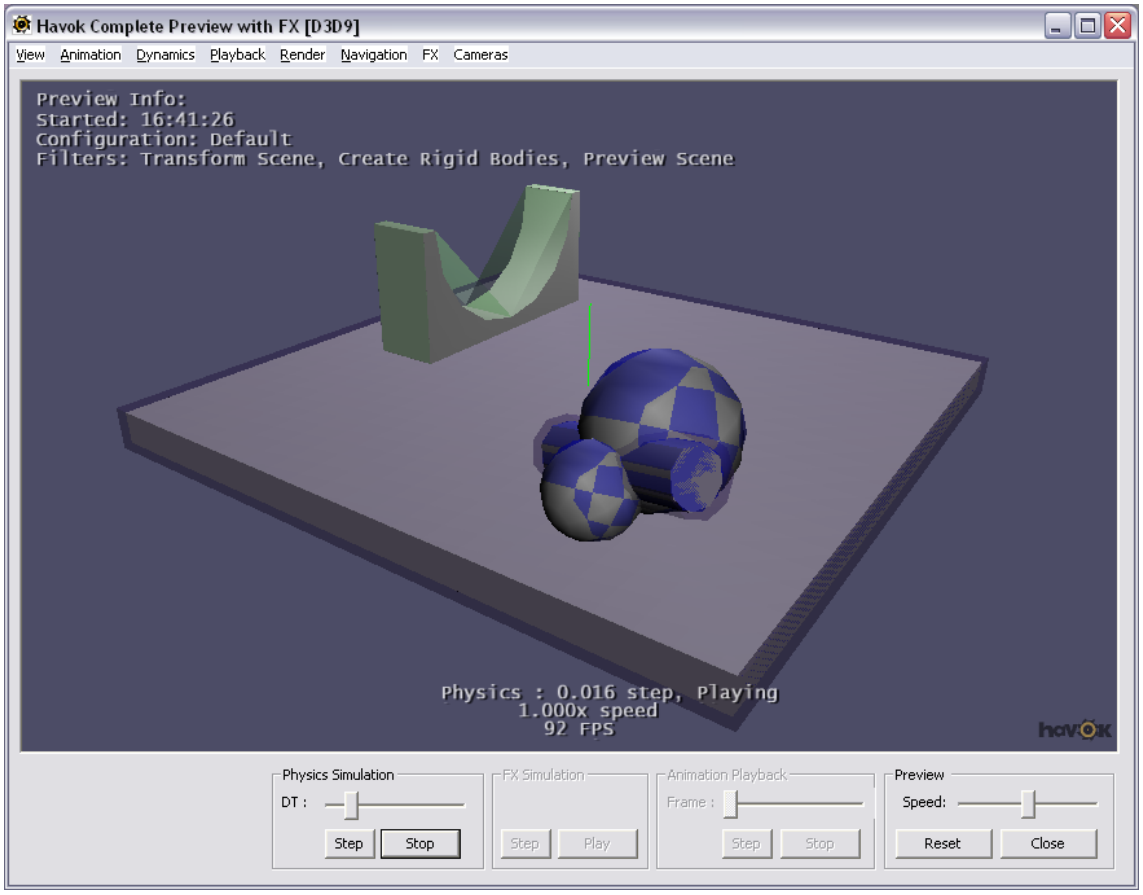

It is also useful to open the property editor for each Havok shape parameter set, and verify that the 'shape type' parameter is correct. The arms of the toy should be set to " **Cylinder** " and the head and body to " **Sphere** ". The Havok Content Tools will automatically detect some XSI shapes and set the appropriate shape type for them when creating rigid bodies, but one should always verify that the parameter has the desired setting.

### Previewing the Scene

Let's now preview the behaviour of our compound rigid body. Click on **Export** and, in the filter manager, add a Transform Scene filter (using the **SCALE_FEET_TO_METERS** preset), a Create Rigid Bodies filter, and a Preview Tool filter (as we did for the previous tutorial):




Click on 'Run Configuration' - this will process the asset and open the preview window. Press the 'Play/Stop' button to start the physics:




As you can see (use mouse picking to interact with the toy man) the three objects behave as a single rigid body.

To toggle whether the meshes (in gray) and/or the physics shapes (in blue) are displayed, use the **Display Meshes** and **Physics Ghosts** items in the previewer's **View** menu.

Close the preview and the filter manager and return to XSI.

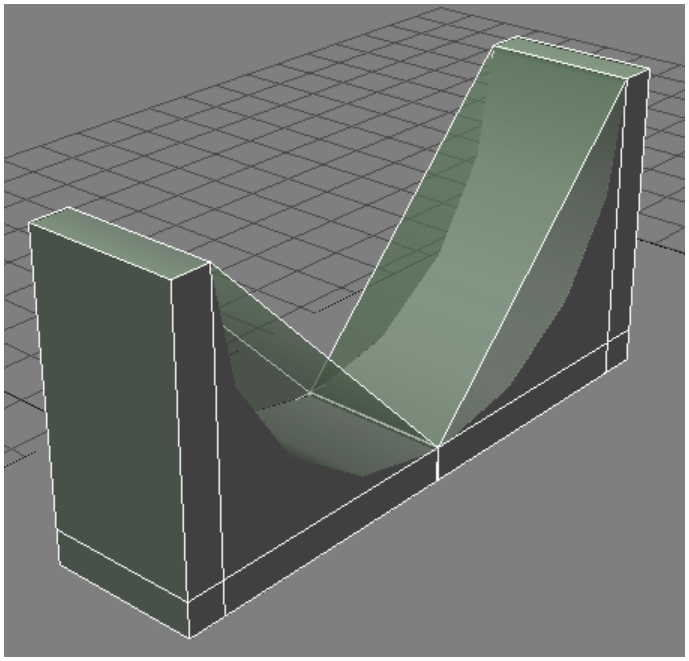
#### 4.6.7.4 Creating a Rigid Body with Proxy Shapes

Sometimes we want to simulate an object as a rigid body, but use another object or set of objects for the collision detection rather than the mesh of the object itself. The usual example of this case is when we may want to simulate a concave object as the combination of multiple convex pieces.

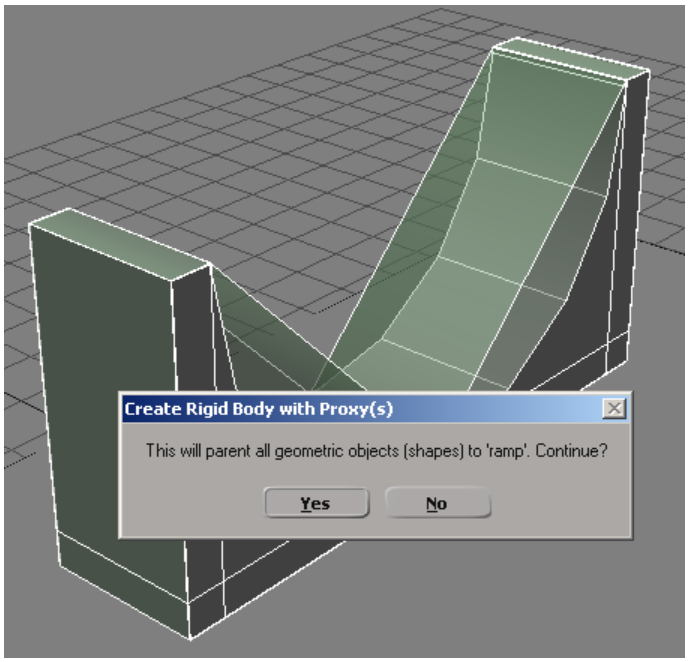
Creating a rigid body with proxy shapes is very similar to creating a compound rigid body. The only difference being that in this case, the object containing the rigid body information (a Rigid Body Parameter Set) does not contain any shape information (a Shape Parameter Set) since its mesh is not used for collision detection. Shape information is instead taken only from its descendants.

Let's see this in action in this scene. Notice how the ramp is a concave object. We'd like to approximate its rigid body shape using two separate convex objects, since simulating convex objects is far more efficient than simulating concave objects.

Locate the two hidden objects in the scene (*rampProxy1*, *rampProxy2*) and unhide them. We will use these two convex pieces to create a proxy shape representation of the ramp. Let's start by making sure that the two proxy objects are positioned so that they match the location of the ramp in the scene (we have assigned a transparent green material to the proxy objects to be able to distinguish them easier):

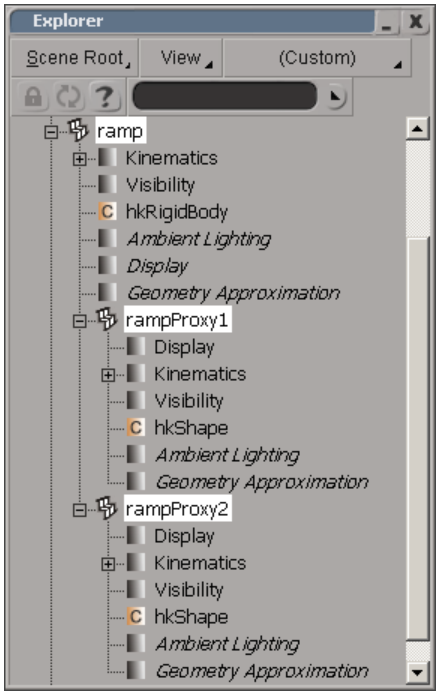


We now want to create a rigid body from the ramp while using the two convex pieces as the collision detection shapes. In the same way as compound objects are created, proxied objects require one object to be the parent of all the others. So, select the two proxy objects first and then the ramp (holding SHIFT for multiple selection, and using the scene explorer if necessary), then click on the **Havok** > **Physics** > **Create Rigid Body with Proxy(s)** menu option:




Click on **Yes** to confirm. Alternatively, you could start by first parenting the two convex pieces to the ramp manually, and then with the three objects selected click the **Havok** > **Physics** > **Create Rigid Body with Proxy(s)** menu option.

Notice how the *ramp* object now has a Havok Rigid Body Parameter Set only, and how the two convex pieces (which are now children of it) each have a Havok Shape Parameter Set only:

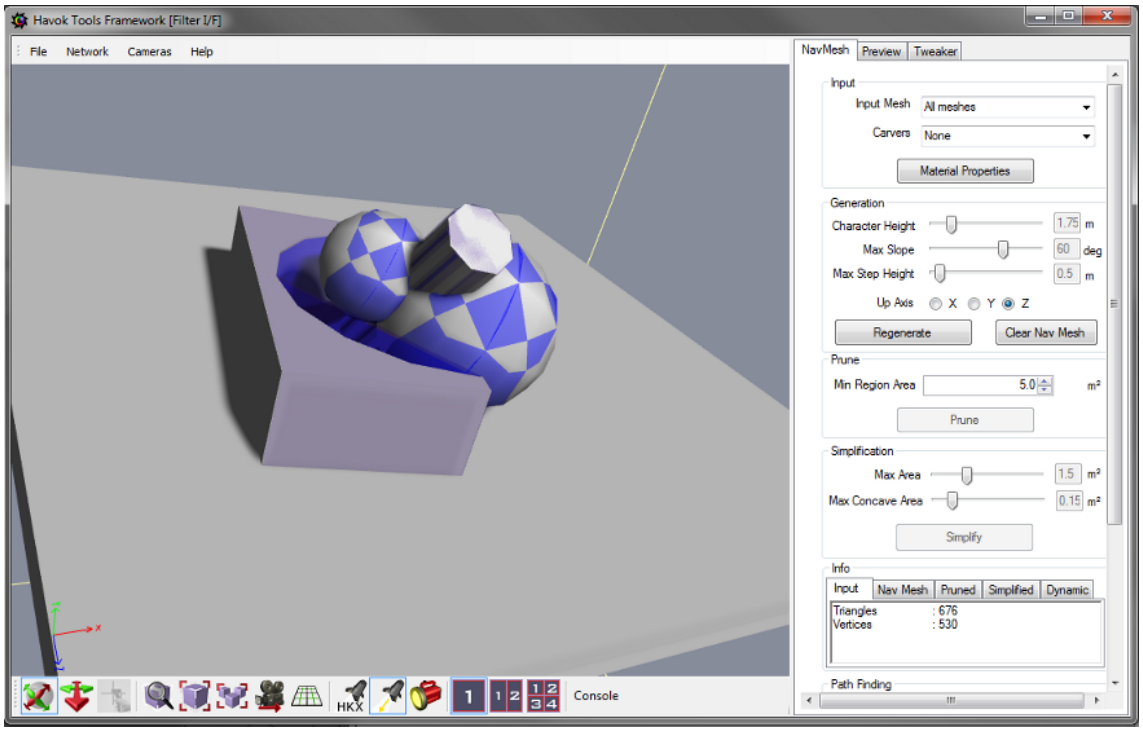



Since the two convex pieces are neither boxes, spheres, capsules or cylinders, their 'Shape Type' parameter

should be set to " **Convex Hull** ". Open each of the Havok Shape Parameter Sets and ensure that the shape type is set accordingly. Also notice how, again, the default mass of our rigid body is zero. Since we want it to be dynamic (movable), it should be set to something different (5 kg for example).

**Previewing the Scene**

Let's see how the ramp behaves in our preview. Export the scene and press on **Run Configuration** to execute the preview. Using mouse picking, you can observe how the ramp is simulated using the two convex pieces:




Again, to toggle whether the meshes (in gray) and/or the physics shapes (in blue) are displayed, use the **Display Meshes** and **Physics Ghosts** items in the previewer's **View** menu.

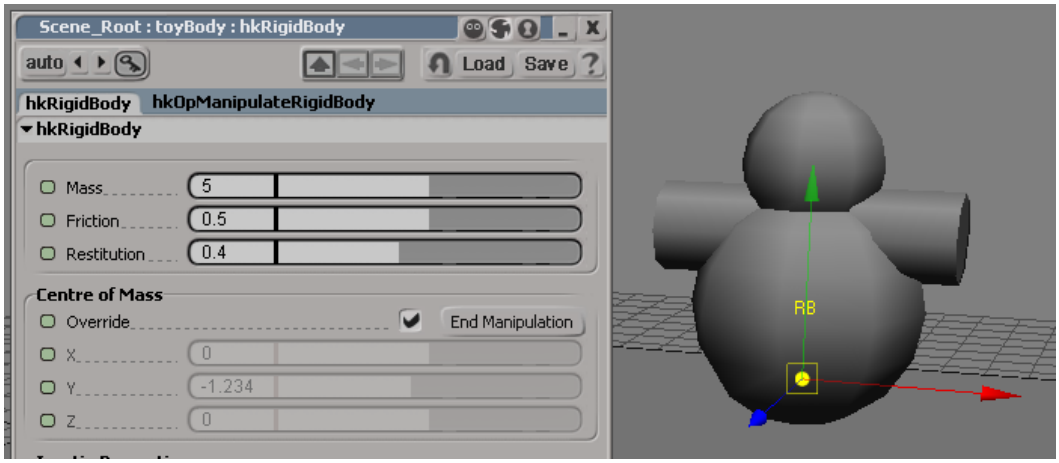
Close the preview and the filter manager and return to XSI.

#### 4.6.7.5 Changing the Center of Mass

We'd like our toy to behave like a tumbler - one of those toys that keep themselves straight. Those toys work by having a very low center of mass (usually by having a lead weight inside them). The Havok Content Tools allow you to easily change the center of mass of any type of rigid body.

In XSI, open the `toyBody`'s rigid body parameter set in a property editor, and check the **Override Center Of Mass** check box. This enables editing of the local center of mass coordinates. The center of mass is now represented as a yellow point in the viewport. Click the ' **Manipulate** ' button in the property editor, then drag the gizmo in the viewport to reposition the center of mass. You may enter the center of mass values manually instead if you prefer.

Position the center of mass towards the base of the toy:




**Note:**

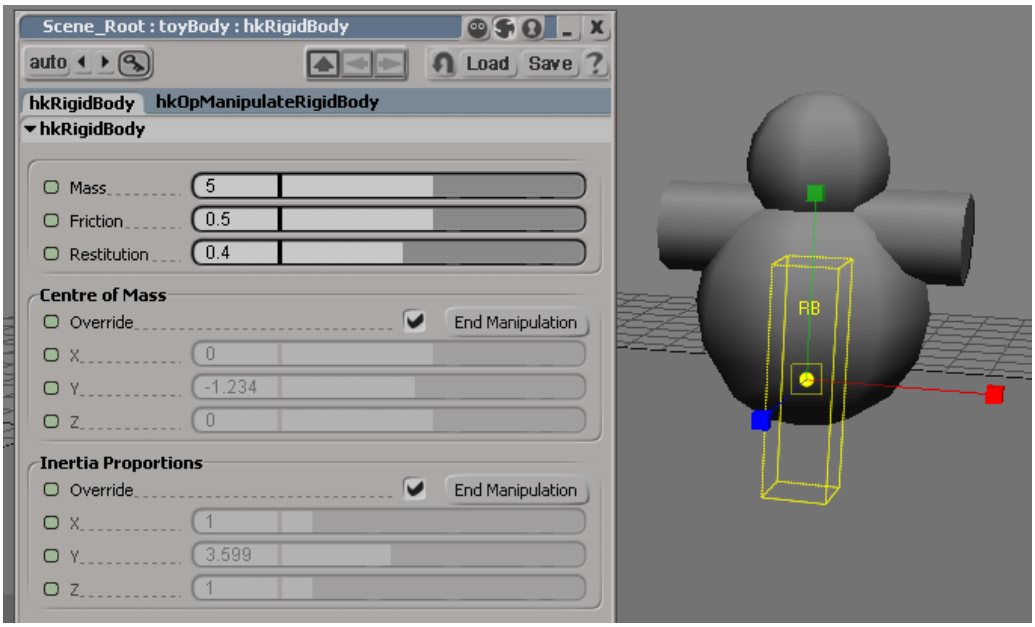
While manipulating the center of mass, the ‘ **Manipulate** ’ buttons change to ‘ **End Manipulation** ’. If you wish to tweak the center of mass values manually after a manipulation, you may need to press the ‘ **End Manipulation** ’ button first.

If you export and execute the preview again, you will see how the toy now tends to keep itself upright as it is dragged around the scene.

#### 4.6.7.6 Changing the Inertia Tensor

We’d now like to customize the motion of our toy even more by changing the proportions of its inertia tensor.

Select its rigid body parameter set again, and check the box **Override Inertia Tensor** . This enables editing of the inertia tensor proportions of the rigid body. The inertia tensor is now represented as a yellow box in the viewport. Click the ‘ **Manipulate** ’ button in the property editor, then drag the gizmo in the viewport to scale the inertia tensor (or tweak the values manually):




**Note:**

While manipulating the inertia tensor, the ' **Manipulate** ' buttons change to ' **End Manipulation** '. If you wish to tweak the inertia tensor values manually after a manipulation, you may need to press the ' **End Manipulate** ' button first.

If you now preview again, you will see that our little man will tend to wobble more around this vertical direction, while it won't lean as much as before. Experiment further with the center of mass and inertia tensor values, observing the results using the previewer.

And this finishes our tutorial. You can find the fully completed tutorial in the "`moreOnRigidBodies/tutorialEnd.scn`" file.

Try experimenting with the parameters further, or creating your own compound/proxied rigid bodies.

### 4.6.8 Tutorial: Rag Doll Toolbox

The "Rag Doll Toolbox" is a set of tools which facilitates the creation of proxies and the reuse of constraint setups. In this tutorial we will use the Rag Doll Toolbox to set up a (low res) hierarchy of constrained proxy rigid bodies to represent a (high res) hierarchy of bones.

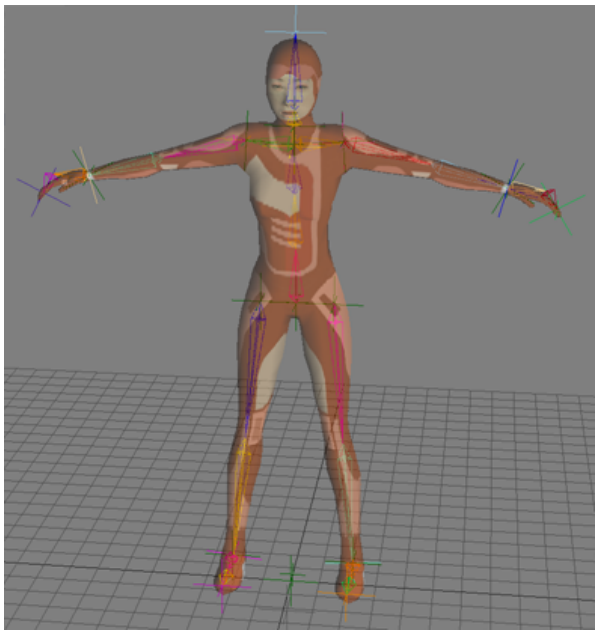
This tutorial consists of three distinct parts, in which we will:

1. Create and associate rigid bodies with an existing skeleton using the Rag Doll Toolbox.
2. Setup the hierarchy and create appropriate constraints between the rigid bodies using the Rag Doll Toolbox.
3. Setup and preview the mapping between the rigid body skeleton (ragdoll) and the original high-res skeleton.

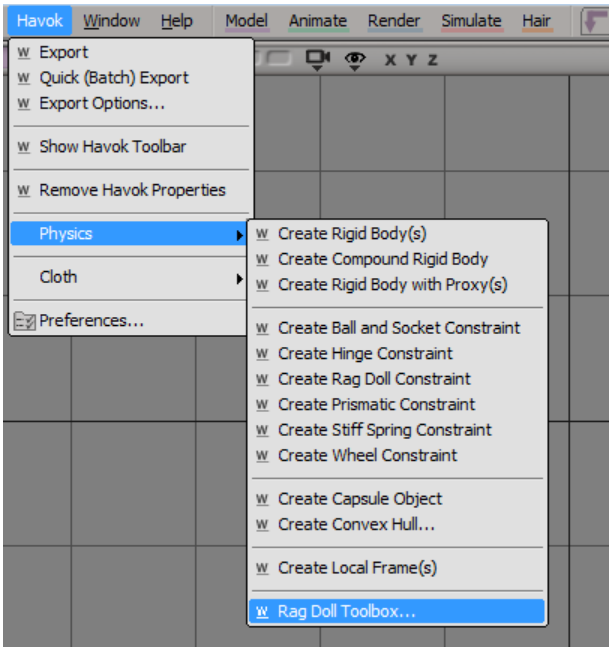
#### 4.6.8.1 Getting Started

For this tutorial, you can use any (skinned) skeleton. We will use the "`red_jaiqua.scn`" sample file which ships with XSI.

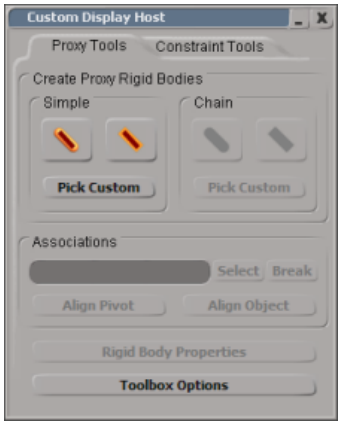
Start by loading the scene "`ragDollToolbox/TutorialStart.scn`", in the *tutorials* project (which should be installed in `[INSTALLDIR]/Addons/HavokContentTools/tutorials`):



Observe that this scene contains a skinned skeleton. Next, open and examine the Rag Doll Toolbox, via the main **Havok** menu or the associated button in the Havok toolbar:

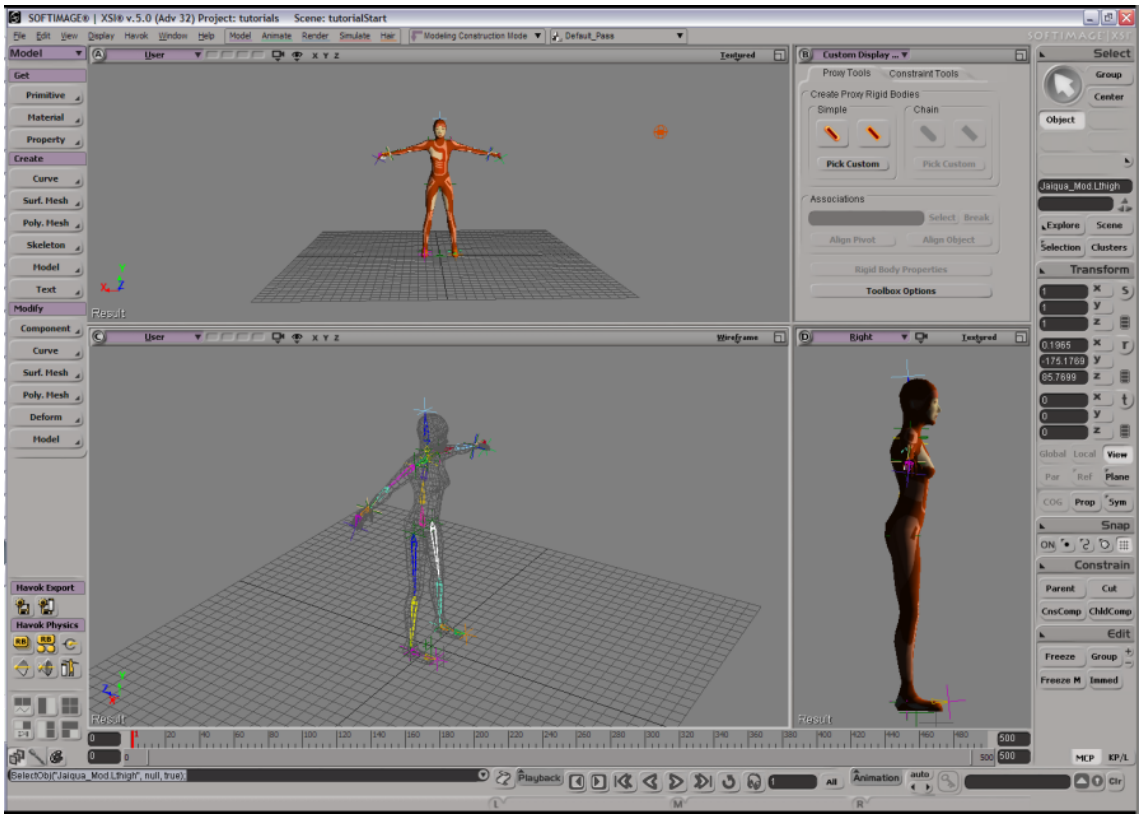


The rag doll toolbox appears as a Custom Display Host consisting of two tabs:




- **Proxy Tools** : Utilities to create proxy (rigid body) representations of animation bones, and manage associations between them.
- **Constraint Tools** : Utilities to setup hierarchies and create constraints between proxies, and reuse constraint data between characters.

XSI can also open Custom Display Hosts in any of the viewport windows by locating them in the viewports menu. This can be very useful if you are extensively using the tool:




#### 4.6.8.2 Creating Rigid Body Proxies

In this section we will focus on creating a rigid body representation of this character (a ragdoll) by creating and associating proxies to the character's bones. The **Proxy Tools** tab of the Rag Doll Toolbox provides the tools to achieve this.

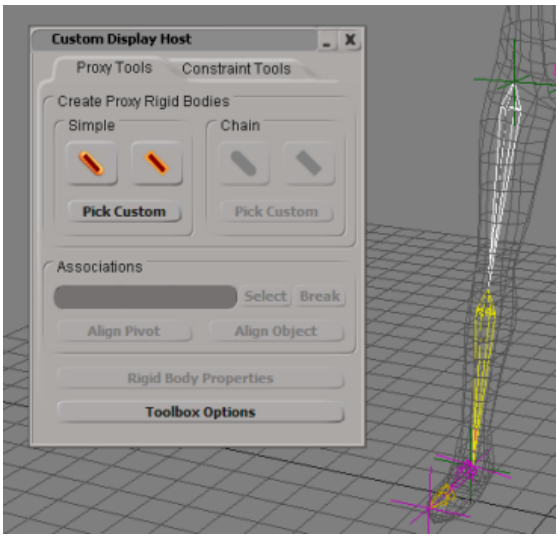
**Note:**

It is useful to hide/lock the skin at this point (if not already), since we will be working only with the bones.

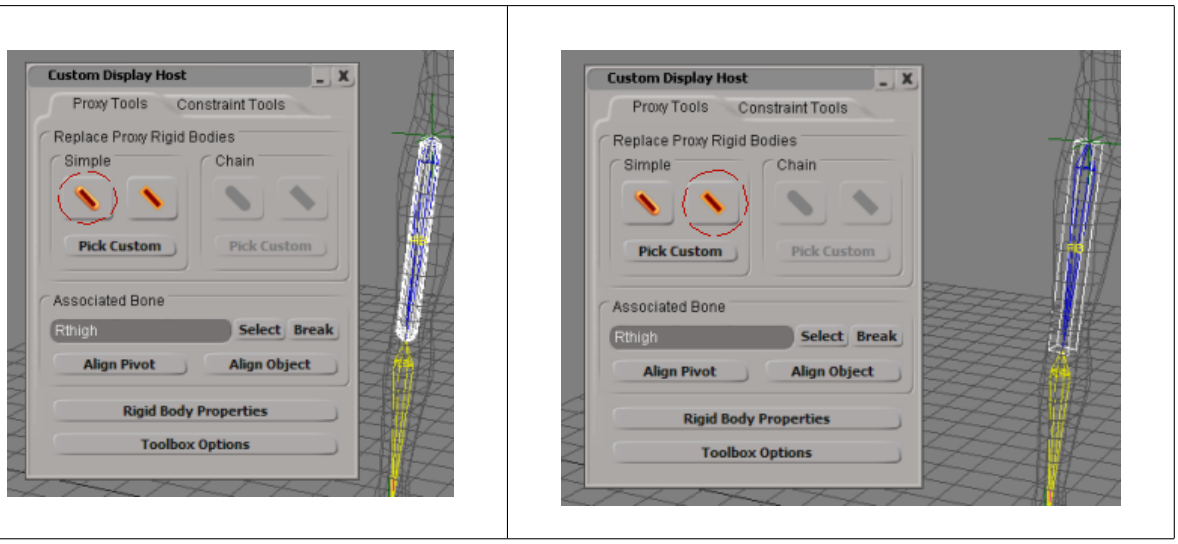
### Simple Proxies

Simple proxies are those which have a one-to-one association with a bone.

Start, for example, by selecting one of the thigh bones of the character. Notice how the set of buttons in the **Simple** section of the **Create Proxy Rigid Bodies** group now become enabled:

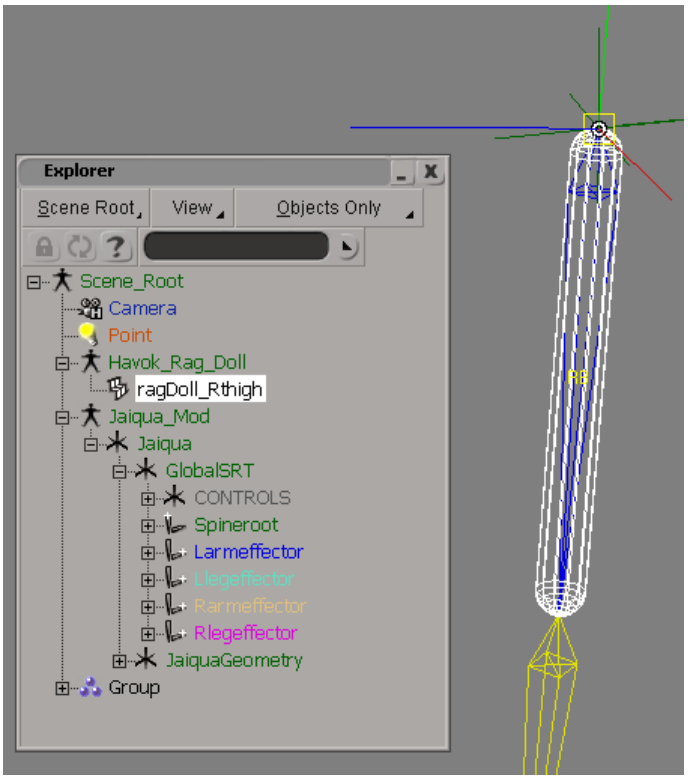



Click one of the buttons to create a new capsule/box proxy for the thigh bone. A new capsule/box shape will be created to fit around the bone, with Havok rigid body and shape parameter sets attached and initialized to a set of default values:




Once a proxy has been created, it is simple to replace it with a different type of proxy. Simply select the newly created proxy (if not already selected) and click of the same two buttons. This will replace the selected proxy, while retaining any rigid body parameters which may have been changed (or any constraints which may have been added). You can replace any type of proxy at any time.

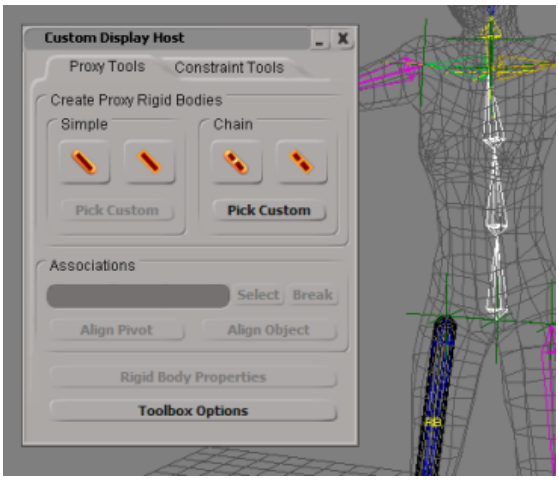
Observe that whenever you create/replace a proxy, the new proxy is placed in both a new model and a new layer (both called "`RagDollProxys`" by default), so as to keep it separated from the original skeleton hierarchy (this behaviour can be modified in the Toolbox Options). Also observe also that the pivot point of any new proxy automatically becomes aligned to that of the associated bone. This is an important step, as it ensures that the mapping we will create during export processing is accurate:



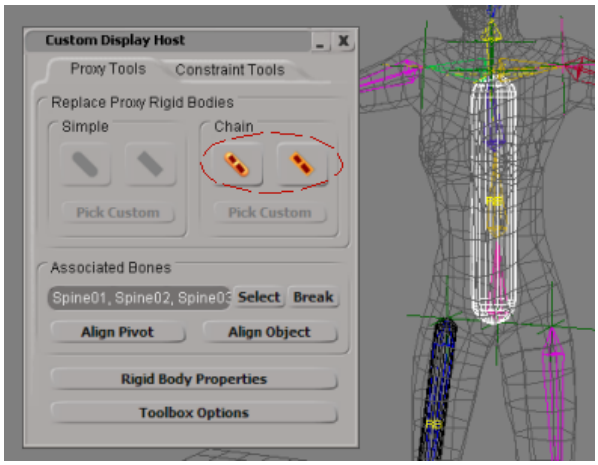

### Chain Proxies

Chain proxies are those which represent a chain of two or more connected bones.

Take, for example, the spine in this example. Select the three bones representing the spine. When a valid chain of bones is selected, the buttons in the **Chain** section of the **Create Proxy Rigid Bodies** group becomes enabled, allowing chain proxies to be created:




Click one of the buttons to create a capsule/box proxy for the selected bones. Once a chain proxy has been created it can be replaced in the same way as simple proxies - by selecting the proxy and clicking the capsule/box button again:




The only different between proxies created from a chain and proxies created from a single bone is that chain proxies keep a reference to both the start and end of the chain (while single proxies keep a reference to a single bone). Pivots of chain proxies are aligned to that of the first bone in the chain. For any other purposes, chain proxies operate the same way as simple proxies.

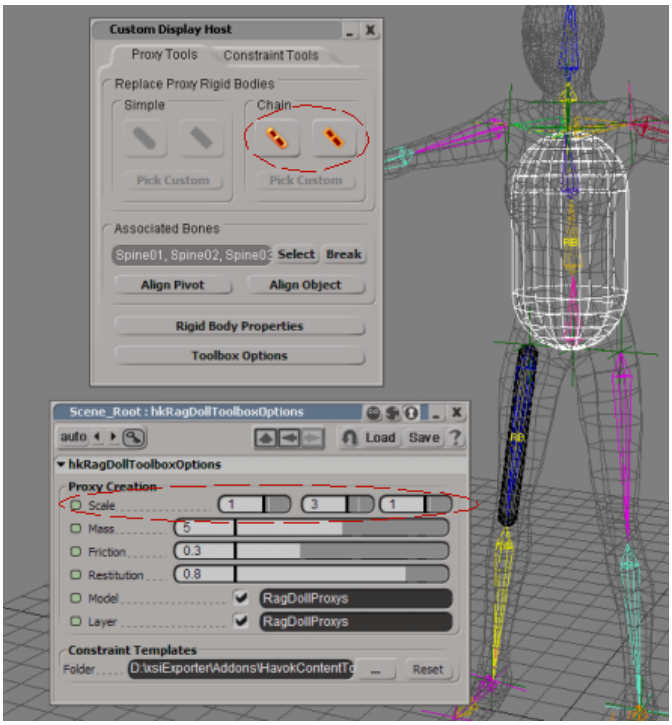
### Reshaping/resizing Proxies

After a proxy has been created, it may not accurately represent the skin volume. In this case the proxies we have created so far are too narrow. We have two methods to correct this:

- Changing default proxy creation scale

  This is usually the easiest and safest way to resize the proxies. When creating a proxy, the Rag Doll Toolbox will use a size that bounds the selected bone(s), scaled by a user-defined scale value. You can change this scaling vale by opening the **Toolbox Options** :

  The Proxy Creation Options group contains a set of X,Y,Z scale parameters. This represents the scaling factor applied to the bone's bounding box before a proxy is created for it. For example, to re-create the spine proxy from a larger bounding box: select the proxy, set the scale values to 1, 3, 3 , and press the capsule/box button again:




**Note:**

The primary axis of XSI bones is X by default. For example, by leaving the X value at 1 above, the proxy length stays the same as that of the bone.

**Tip:**

It is usually desirable to have some overlap between proxy representations of adjacent bones. This avoids objects going through the gaps at joints during runtime.

- Using the standard SRT tools

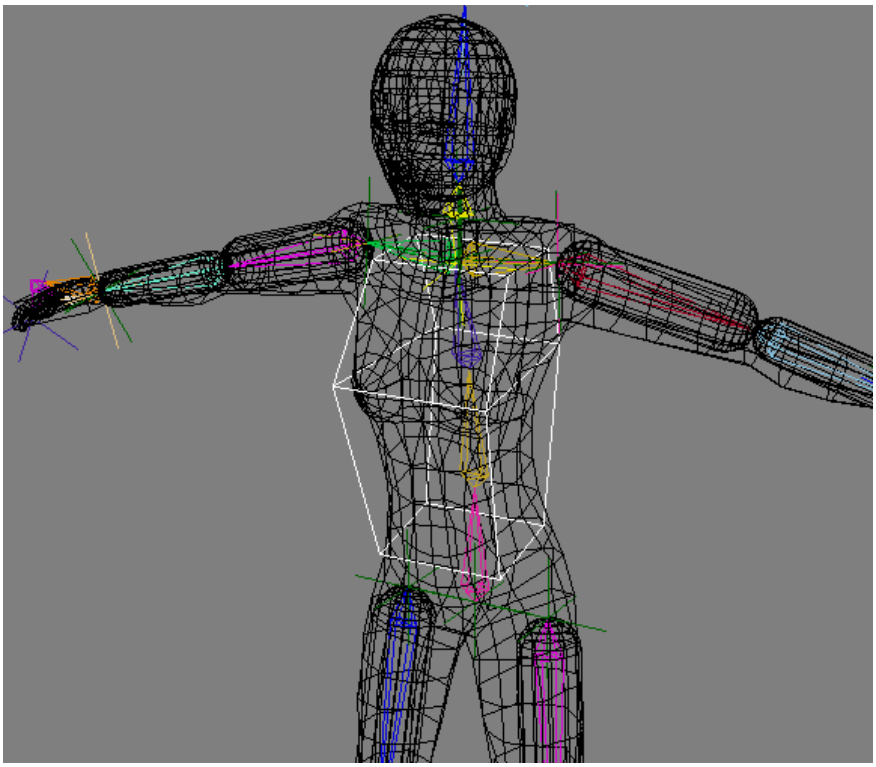
  Any proxies can be transformed as per usual using XSI's Scale, Rotate and Translate tools. However it is important that non-uniform scale should be avoided so that the rigid bodies are created properly during export. It is also important to press the **Align Pivots** button (in the 'Association' section of the toolbox) after performing any transformation to ensure that the proxy pivot matches that of the bone. This helps the rag doll mapping filter to create accurate mappings.

Experiment further with creating single and chain proxies for different parts of the character. Make sure that you are familiar with the general process before continuing.

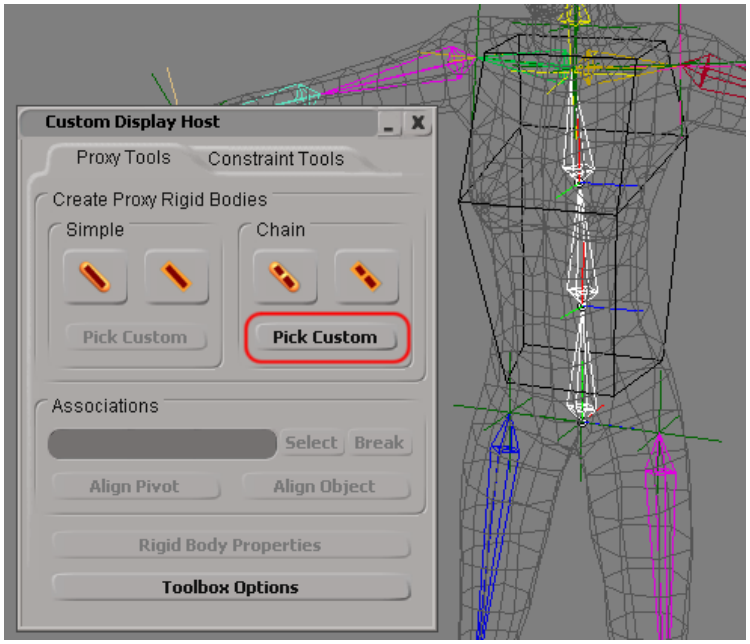
### Custom Proxies

Sometimes a box or a capsule is not enough to accurately represent the volume around a bone or a chain of bones. For example, take the torso region of this model. We have already created a box/capsule proxy associated to the spine bones. Lets replace this with a custom proxy. Start by deleting the existing spine proxy.

Create a new convex polygon mesh in world space to represent the character's torso, as shown, using any of the standard XSI tools:



To associate this mesh as a chain proxy for the spine bones, select the three spine bones, then click the **Pick Custom** button in the toolbox:




This will start a picking session - now pick your custom mesh.

The custom mesh now becomes associated as a proxy for the set of spine bones. From this point it

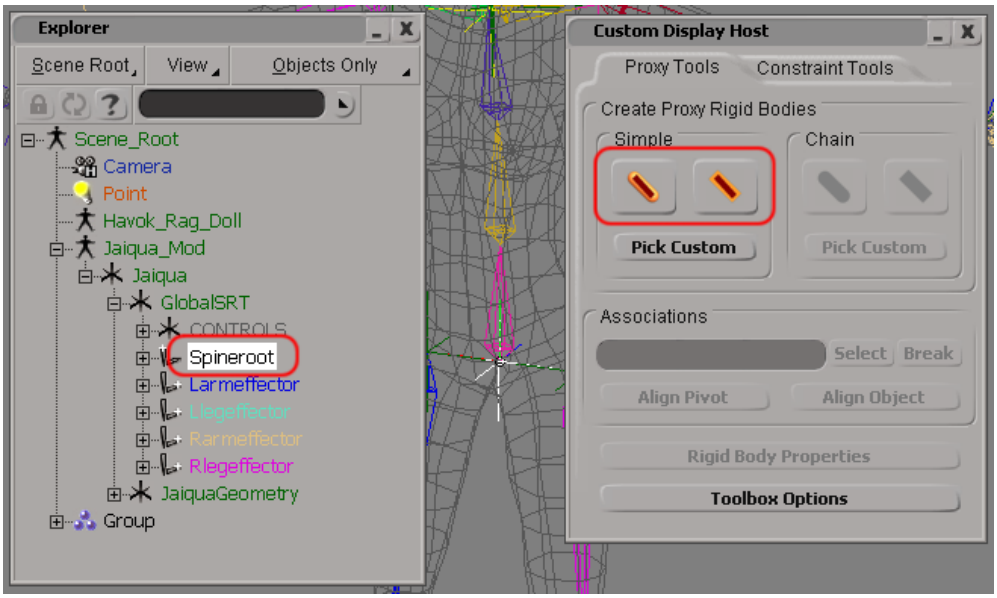
behaves like any other chain proxy - i.e: it inherits default Havok parameter sets; it can be replaced by clicking the other chain proxy buttons; its pivot is aligned to the first of the associated bones, etc.

One final step is to ensure that the "shape type" parameter of the new proxy is set to " **Convex Hull** ". This is required to accurately represent a mesh such as this. Open the `hkpShape` parameter set of the proxy and set the parameter accordingly.

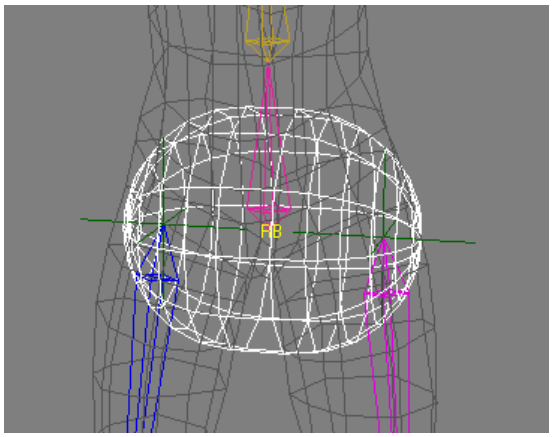
### Null-based Proxies

The scene we are working on does not have any specific 'pelvis' bone. However it is desirable for the rag doll representation to have a pelvis, both for realism and for mapping purposes. Using the following method, we can create a suitable proxy even though there is no suitable bone.

First, select the root of the spine chain, called *Spineroot* - use the explorer if necessary. Then click the create capsule/box button:




The proxy is created by taking the bounding box of the selected null object combined with any null children that it may have - in this case it considers the spine root plus the two leg roots. Tweak the scale values and recreate the proxy until it has the required dimensions:

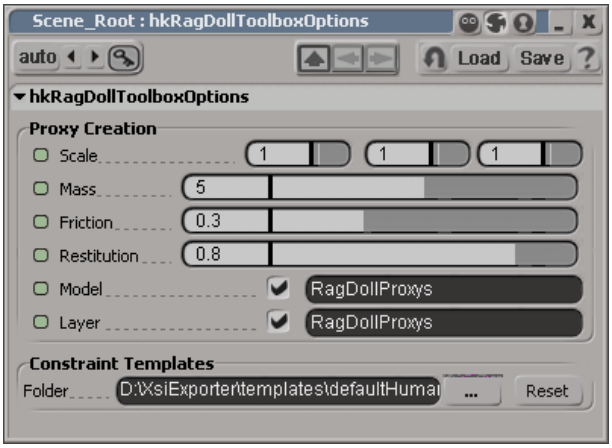



**Note:**

If the toolbox cannot determine a suitable bounding box for an object (e.g. a single null object with no children), it will always assume a bounding box of unit dimensions.

### Setting Rigid Body Properties

When a new proxy is created, it assumes a set of default parameter values. These defaults can be configured from the Rag Doll Toolbox Options, and are saved as part of the scene:

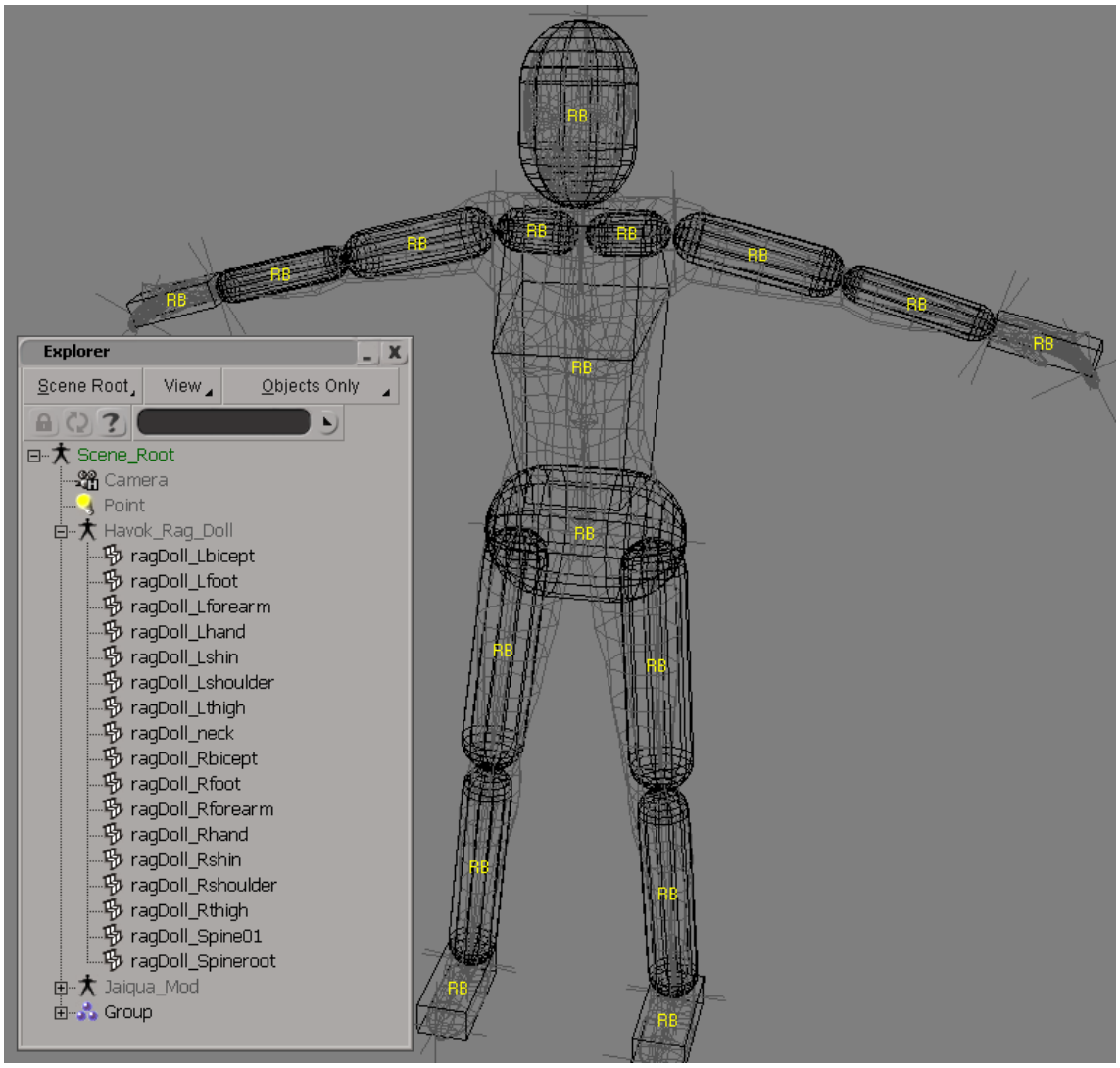



After a proxy has been created, it is simple to change the properties on an individual or group basis. Select one or more proxies, and click the **Inspect Rigid Body Properties** button in the Proxy Tools section. This opens a PPG from which the properties can be edited as per usual.

Now set appropriate rigid body properties for each proxy that you have created.

### Finishing with rigid bodies

Using the techniques above, you should now be able to create enough proxies to represent the character. The pivots of all your proxies should be aligned to the associated bone(s) - use the **Align Pivots** button on each proxy to be certain, and appropriate rigid body properties should be set for each proxy. The "`ragDollToolbox/tutorialProgress1.scn`" file contains a fully proxied character if you wish to use it:




The proxies in this scene were created as follows:

Simple proxies for:

- Thighs
- Calves
- Clavicles
- Upper arms
- Forearms

- Pelvis (aka spine root)

Chain proxies (with additional transformation and re-alignment) for:

- Hands
- Feet
- Head/Neck

Custom proxy for:

- Spine

As you can see from the screenshot above, all of the proxies are grouped together in a model, but they do not (yet) have any parenting.

#### 4.6.8.3 Creating Constraints

After creating the proxies, we are left with a flat collection of rigid bodies - they are not parented to each other, or connected in any way. To finish setting up the rag doll, we need to organize the rigid bodies into a constrained hierarchy. The **Constraint Tools** tab in the Rag Doll Toolbox gives us the tools to achieve this.

**Note:**
It is useful to hide/lock the bones at this point (if not already), since we will be working only with the proxies.

**Templates**

Templates are representations of Havok constraints, saved as text files. Each represents a particular type of constraint, plus information about its spaces, limits, etc., so that it can be reused between different characters. For example, an "elbow" template might describe a hinge constraint with certain limits and spatial orientation.

Templates are stored together in an specific directory. Although a default set of templates (`defaultHuman`) is provided, in most cases you will be creating your own templates to match the structure of your characters.

You can change the current location of the templates folder at any time from within the Rag Doll Toolbox Options.

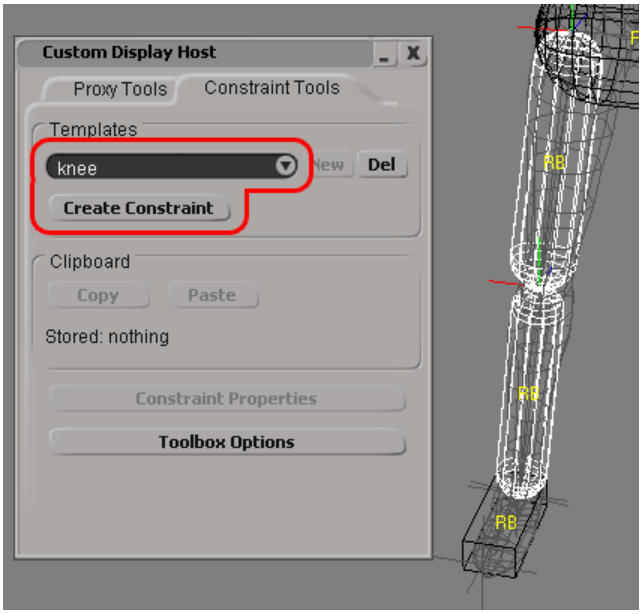
**Note:**
When working with templates, some assumptions need to be made regarding the local axis of the bones. In particular, the main axis (an axis pointing towards the length of the bone, in the "outside" direction) and the bend axis (an axis around which a positive rotation (using the right hand rule) causes a bending of the joint) need to be defined. These axes are currently assumed to be +X and –Z, since this is the convention used by XSI's default skeletons.
If a proxy's transform does not follow this convention, then any applied templates will need to be rotated into the correct orientation.

**Applying Templates**

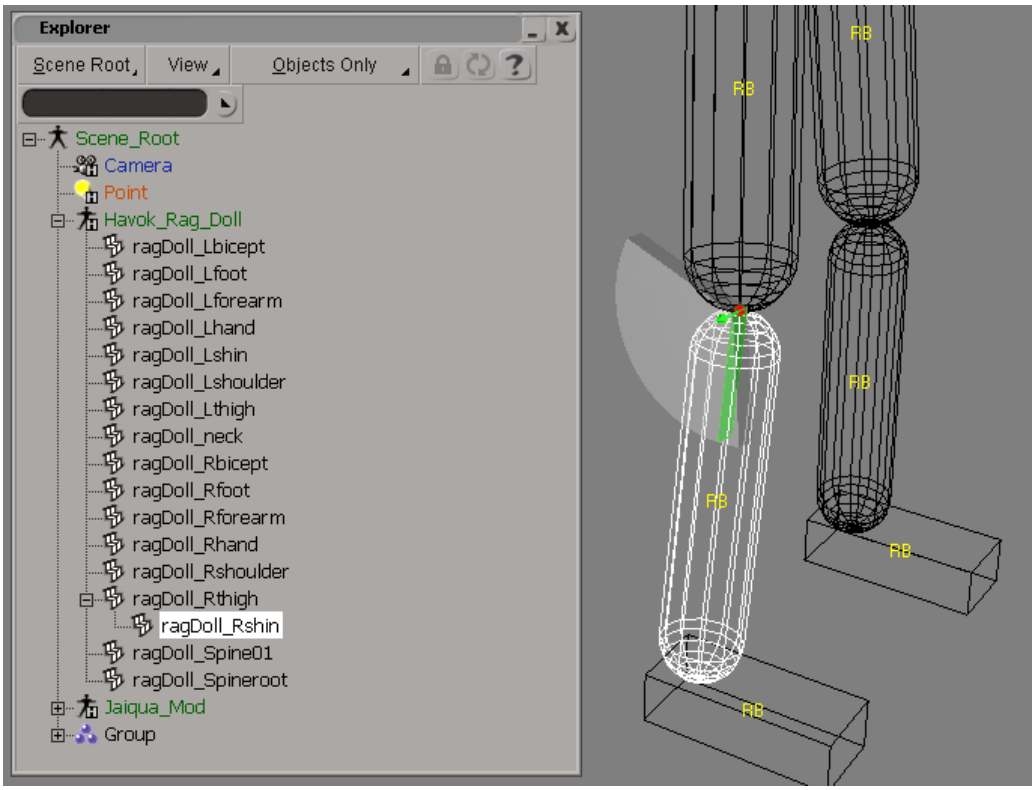
When we apply a template to a pair of proxies, several things happen in order:

1. The two proxies become parented to each other, matching the parenting of their associated bones.
2. A constraint is added to the child proxy, based on type stored in the template.
3. The parameters of the constraint are set from those stored in the template.

For example, select a pair of thigh and calf proxies. In the templates combo box, select the "`knee`" template and then click on **Create Constraint** :




Observe the result. The calf proxy becomes a child of the thigh proxy, and a hinge constraint is created on the calf:




### Replacing Constraints

After a template has been applied, it is simple to replace the existing constraint with a new template. Select either the pair of proxies or just the child proxy, then press the " **Replace Constraint** " button. Any existing constraint is deleted, and the same procedure occurs as when a new template is applied.

### Modifying Constraints

After a template is applied, you can always modify the parameters of the constraint as per usual. Use the **Inspect Constraint Properties** button for quick access to the constraint parameters. You can save a constraint as a new template (or replace an existing one) at any time by clicking on the " **New** " button beside the template combo box.

### Copy and Paste

Symmetric characters usually have similar constraint properties on their left and right side. The copy and paste mechanism of the toolbox is useful in this case.

With a constrained proxy selected, press the "copy" button to store its constraint as a template in memory. The constraint can then be applied to:

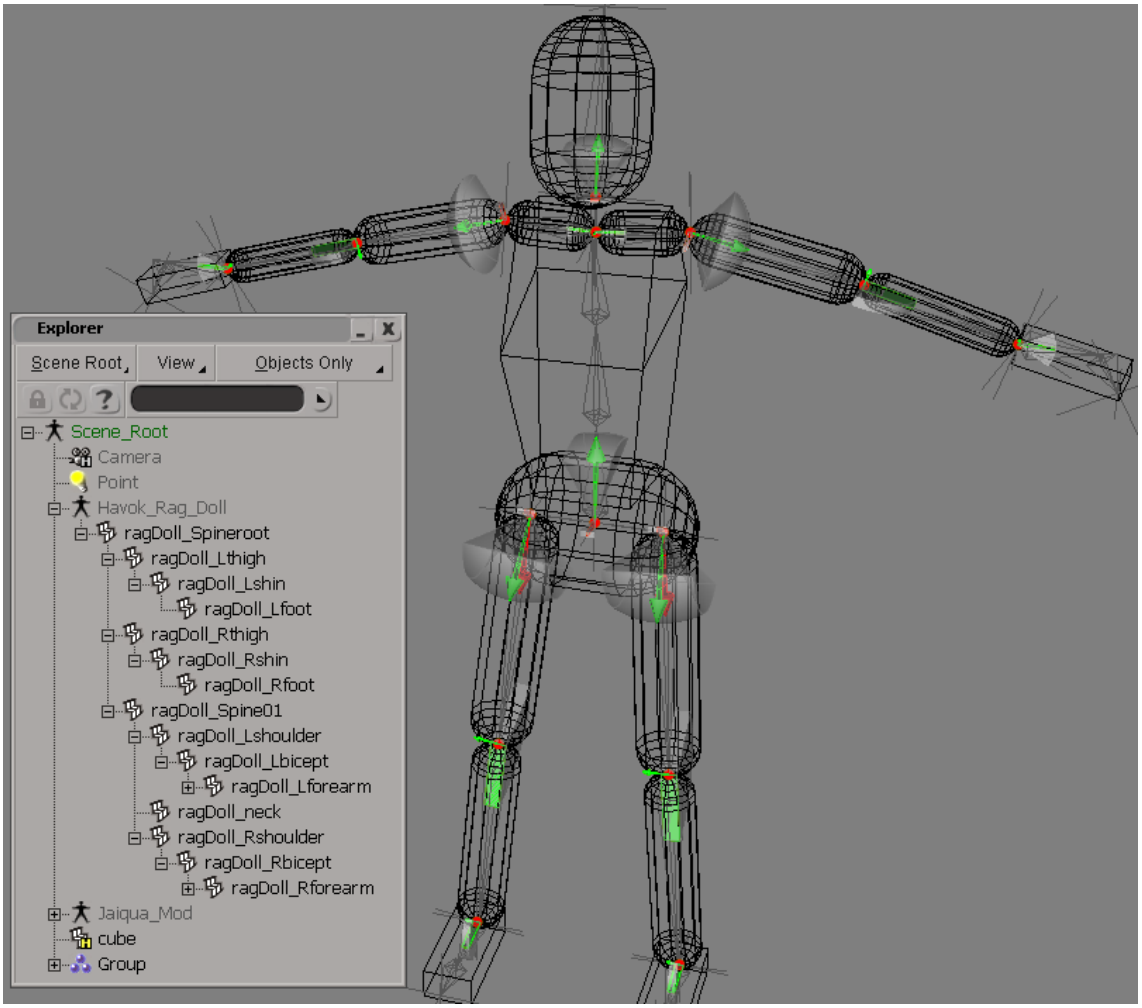
- A pair of unconstrained proxies, by selecting the pair and pasting. This behaves in the same way as creating a new joint.
- An already constrained proxy, by selecting the proxy and pasting. This behaves in the same way as replacing an existing joint.

For example, let's say we have modified the left knee friction torque, and we now want to apply the same constraint with the same values to the other knee:

- Select the calf, and change the **Max Friction Torque** parameter value of the constraint
- Click on **Copy** in the Constraint Tools Clipboard
- Select the opposite thigh and calf together in the viewport
- Click on **Paste** in the Constraint Tools Clipboard
- Observe that a new constraint has been created/replaced, with the same parameter values as the copied constraint

### Finishing with Constraints

Continue applying templates to each pair of proxies, using the default templates provided (feel free to modify any parameters). You may wish to open the sample "`ragDollToolbox/tutorialProgress2.scn`" file - this contains the skeleton, the skin, and the constrained proxy hierarchy. It also contains a simple floor so that the rag doll has something to land on when we preview the scene:



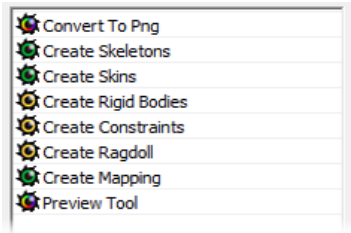

Observe that the proxies now form a hierarchy, matching that of the bones with which they are associated.

#### 4.6.8.4 Processing the Scene

Now that we have the rag doll set up to match the skeleton in the scene, we want to process so that:

- Runtime versions of the original skeleton and skin are generated
- The rigid bodies and constraints for the ragdoll are generated
- A "ragdoll instance" and skeleton for the ragdoll are generated
- A pose mapper between ragdoll skeleton and original skeleton is generated (so the ragdoll can eventually drive the skin)
- We can preview it

Export the scene to the filter manager using the Havok exporter. The following filter setup is required to process the ragdoll:




- **Convert To PNG**

  This embeds the textures into the scene data, so that we can view them in the preview.

- **Create Skeleton**

  This will create a skeleton instance from objects in the scene data. Choose the ' **Automatic** ' setting - this will automatically create a skeleton from the entire 'Jaiqua_Mod' model.

- **Create Skin**

  This will create a skin object and associate it with the skeleton we just created.

- **Create Rigid Bodies**

  This will detect the rigid body and shape properties in the scene and create the actual rigid body objects for the simulation.

- **Create Constraints**

  This will create actual physical constraints based on the setup we defined in the scene.

- **Create Rag Doll**

  This will create a Ragdoll instance based on the rigid bodies and constraint we just created. Choose the ' **Automatically** ' setting, which works fine in this case as there is only one rag doll in the scene.

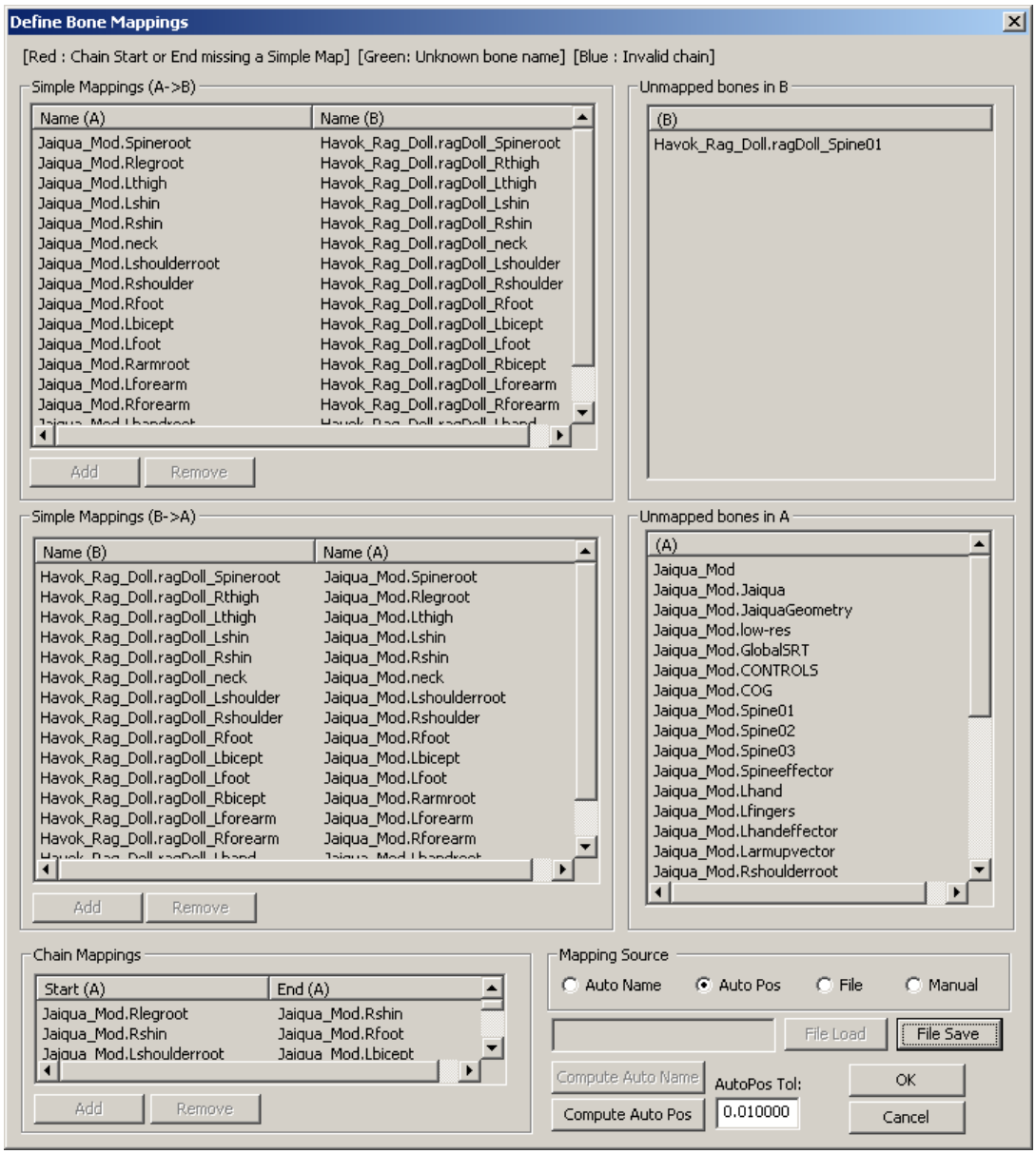
- **Create Mapping**

  This filters sets up skeleton mapper information that allows the transformation of poses between two different skeletons (in our case, the original boned skeleton and the Rag Doll skeleton).

  For ‘ **Skeleton A** ’, pick ‘Jaiqua_Mod’ (the skeleton root).

  For ‘ **Skeleton B** ’, pick ‘Havok_Rag_Doll.ragDoll_Spineroot’ (the rag doll root).

  Click on ‘ **Define Mappings** ’ to examine the mapping:




  The default behaviour of the mapping filter is to match the skeletons by comparing the positions of the bones in each. This is why it was important to ensure that the pivot points of the rigid bodies were aligned to the pivot points of the joints.

  For a successful mapping, all of the bones in one skeleton should be mapped to (some of) the bones in the other skeleton. We can see that this is not currently the case. We will fix this later.

  Click ‘ **OK** ’.

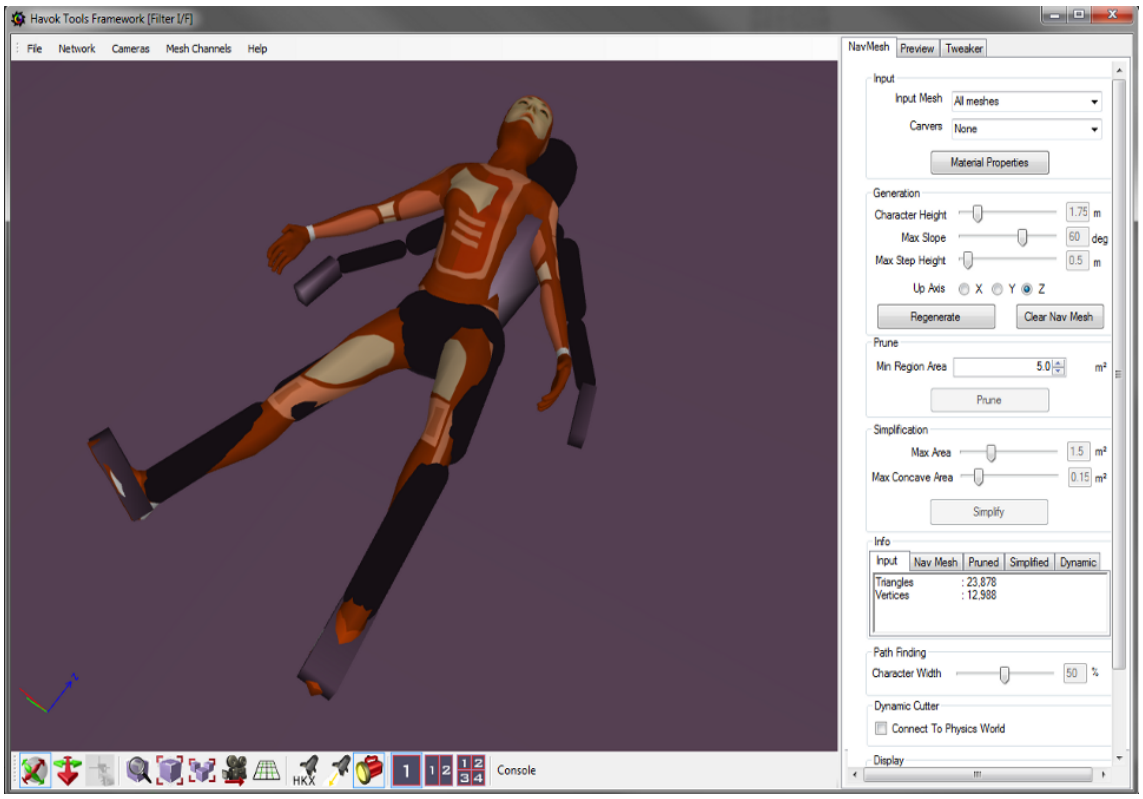
- **Preview Tool**

  So that we can interactively test the rag doll behaviour.

### Fixing the Mapping

Run the filter setup above. When the preview window appears, ensure that **Display Meshes** and **Physics Ghosts** are enabled in the 'View' menu.

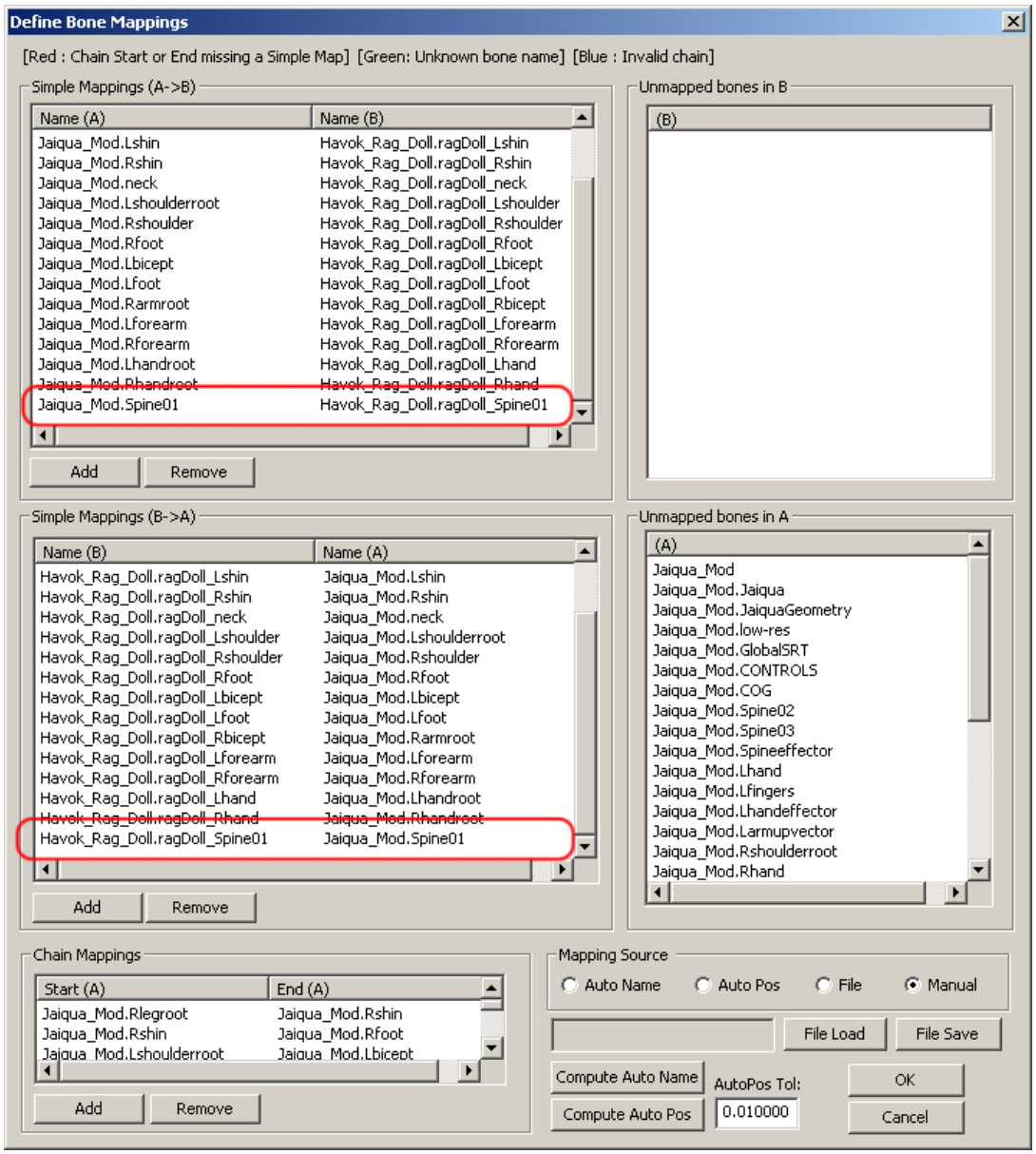
Play the physics and interact with the rag doll using the space bar to pick and drag the rigid body shapes. Observe that the character's upper body does not follow the rag doll correctly:



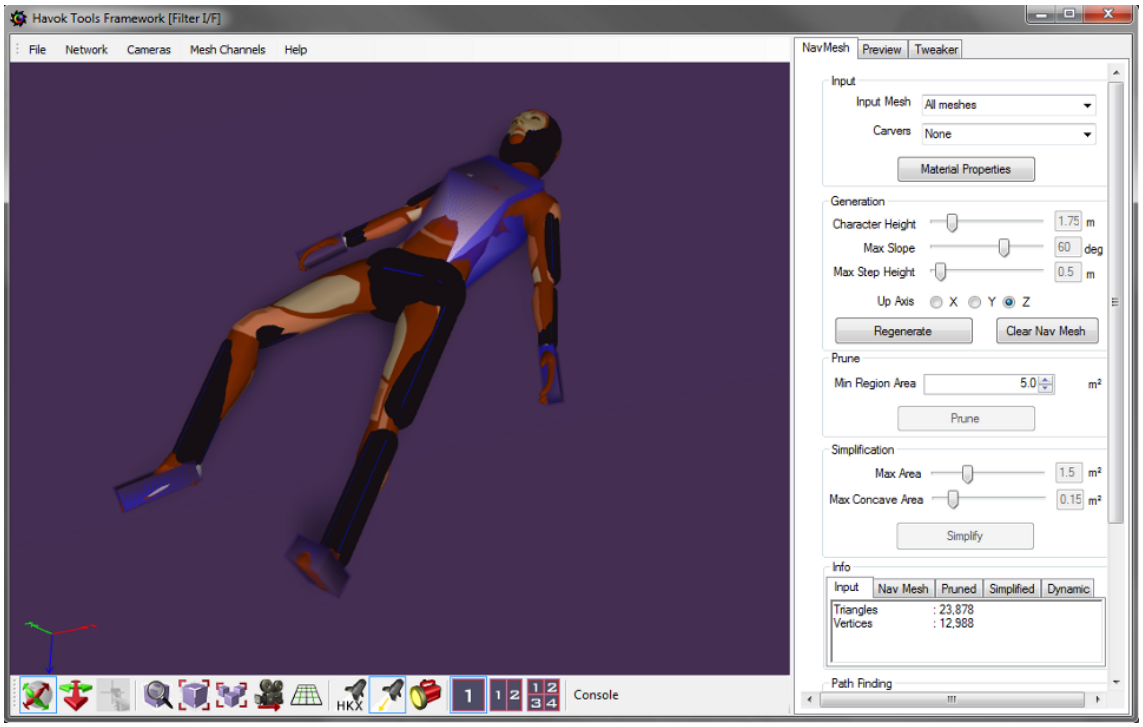

To fix this, close the preview and return to the 'Define Mappings' dialog in the 'Create Mapping' filter.

Observe that *Spine01* is unmapped in both directions. This happened because *Spine01* (the first bone in the spine) and *Spineroot* (the root of the spine) both have the same position - this confuses the mapper when set to 'Auto Pos', and leaves *Spine01* unmapped. Therefore we need to manually add this mapping:

- Switch the 'Mapping Source' to ' **Manual** ' - this allows us to edit the mapping.
- In the upper mapping section (A->B) press the ' **Add** ' button, and add a mapping from *Jaiqua_Mod.Spine01* to *Havok_Rag_Doll.ragDoll_Spine01*.
- In the lower mapping section (B->A) press the ' **Add** ' button, and add a mapping from *Havok_Rag_Doll.ragDoll_Spine01* to *Jaiqua_Mod.Spine01*.




Click ‘ **OK** ’, and run the filter setup again. Now observe that the rag doll maps correctly to the character:




**Note:**

Another solution in this case is to use the **Auto Name** mapping option instead of Auto Position. This compares the last element of the bone names in each skeleton (delimited by underscores/dots/space) and builds the mapping based on any matches, which works well in this scene. Try it to see for yourself.

### Tweaking the Scene

The rag doll setup is now complete. At this stage it is useful to return to XSI and begin to tweak parameter values. Try altering the Rigid Body masses as well as the Constraint limits and frictional values. You can run the filter setup at any time to test the result.

The "`ragDollToolbox/tutorialEnd.scn`" file contains the final tweaked rag doll. Note that a scene transform filter has also been added to the export pipeline to scale the character to a realistic size of ~1.5m.

## 4.7 Common Concepts

### 4.7.1 Introduction

This chapter contains information regarding basic concepts which are used by the Havok Content Tools, independent of the actual modeler being used to create the content.

### 4.7.2 Common: Controlling File Size

The final size of an asset file (written using the Platform Writer Filter) depends on a number of factors, and can be reduced by ensuring you only write the minimal information needed. By default the filters do not automatically optimize for file size, so it is easy to overlook these factors. Also in some cases it is a matter of preference how to divide the asset information amongst multiple files, so the user must make a decision as to how to best optimize the distribution.

#### 4.7.2.1 Methods for reducing file size.

Here is a complete list of the methods which can reduce file size:

- *Avoiding XML format.*

  The Platform Writer filter allows you to output the file in human readable XML, or in platform specific binary format. The XML should only used as an intermediate format or to aid debugging - it will clearly be much larger in size that the binary format which is made to be loaded straight into memory.

  **Warning:**
  Common mistake: Using XML format temporarily and forgetting to change back to binary.

- *Removing Metadata.*

  The Platform Writer filter allows you to remove the *metadata* (class information) from the file. Metadata in the form of `hkClass` information is required for operating on serialized assets (see Serialization documentation in the User Guide). However this information is usually not needed in the same file unless you are loading assets into a tool which has no metadata compiled in (or out-of-date metatdata). Also, metadata is shared by all objects of the same type so is clearly redundant for multiple files containing the same objects.

  **Warning:**
  Common mistake: Leaving this box unchecked for 'final' in-game assets.

  **Note:**
  The 'Environment Data' may be needed by the Platform Writer in order to construct the filename using environment variables, but it is not written out by default (unless you explicitly check the 'Environment Data' box) so it is safe to assume it will not contribute to overall file size.

- *Pruning Unneeded Objects.*

  The Prune Types filter allows you to remove objects of various types from your scenedata before they are passed to the Platform Writer for example. Although it may be necessary to create intermediate objects before other objects can be created later in the filter pipeline, it is possible to use this filter to prune them out again after they are no longer useful. An example of this would be creating a `hkaSkeleton` object using the CreateSkeleton filter in order that you can then create an animation with the CreateAnimation filter. If you then wish to save only the animation, you can then prune the skeleton using the Prune Types filter.

  To confirm that the file you are about to write has only the information you require in it you may wish to temporarily add a View XML filter before the Platform Writer - this will allow you to parse the hierarchy visually and confirm that only the desired objects are still present in the scene data. Of course you can also use the Standalone Filter Manager to reload any Win32/XML file you have previously saved and use the View XML filter to confirm its contents. Further information on pruning tips specific to Havok Animation and Havok Cloth can be found in Animation > Controlling File Size and Cloth > Controlling File Size.

#### 4.7.2.2 Expected Overhead Of Serialization

The final file size may be slightly larger than you might expect for the raw data contained in the objects. For example a compressed animation may say (in the Report window) that it is of size 8040 bytes but the final file size may be slightly larger, say 8800 bytes, even if all of the above methods for reducing the file size have been employed. There are three possible reasons for this:

- The header common to all serialized files has approximately a 400 bytes size overhead. This includes the version ID, the file layout and basic class info for vtable patchup etc.
- The Root Level Container class which is the class which holds all the animation objects is always serialized from the Content Tools which adds a further ∼176 bytes.
- It may be that there is a 'hidden' class, not immediately apparent in the file but required by the other objects for runtime - for example a `hkaAnimation` class is always associated with (owned by) an `hkaAnimationBinding` class. It is currently not possible to serialize one without the other, so a file contain an 'animation' actually contains two objects, the compressed animation and the binding.

### 4.7.3 Physics : Rigid Body Concepts

Rigid bodies are the basic building blocks of Havok simulations - for an object to be simulated, it must be a rigid body.

A simulated rigid body contains both dynamics (rigid body) and collision (shape) information. The Havok Content Tools separate these two elements into distinct objects for use in each supported modeler. If a Create Bodies filter (or a Create Rigid Bodies filter for Physics 2012) is applied during the export process, any rigid body information found in the scene is used to define the dynamics of a new runtime rigid body object, and any related shape information is used to define the collision properties of that rigid body.

Each modeler supported by the Havok Content Tools allows rigid body and/or shape information to be attached to any object in the scene. The means in which this information is attached depends upon the modeler: 3ds Max attaches the information through the use of modifiers, Maya attaches it through the use of nodes, while XSI attaches it though the use of custom parameter sets.

It is important to note that both rigid body and shape information is required for an object in the modeler for the Create Bodies filter to create a runtime rigid body object on export. If rigid body information exists without related shape information then no rigid body can be created. Similarly, if shape information exists but is unrelated to any rigid body information, then it is ignored. The shape information may be placed in the rigid body object's children as well or instead of in the rigid body object itself, which allows for the creation of compound/proxied rigid bodies.

This section describes each of the rigid body and the shape properties as used in any supported modeler, and how they can be combined to create various types of runtime rigid body objects.

#### 4.7.3.1 Rigid Body Properties

A Havok rigid body created in a modeler contains the following basic set of properties which govern the dynamic behavior of the exported body at runtime:

| | |
|---|---|
| Mass | The mass of a rigid body governs how the object will interact with other objects. If the mass is set to zero, the resulting rigid body will be fixed in space. |
| Friction | The coefficient of friction for the rigid body's surface. This affects its motion relative to surfaces it comes into contact with. Although this is actually a pair-wise coefficient, friction is specified on a per rigid body basis. When affecting the relative motion of two surfaces, the surfaces' friction coefficients are combined to produce a coefficient for the interaction. Usual values for Friction are between 0 and 1. Notice that values greater than 1, although physically invalid, are allowed. |
| Restitution | The coefficient of elasticity for the rigid body. This governs the effect that collisions have on the velocities of colliding rigid bodies. Like Friction, this is a pair-wise coefficient - when two bodies collide, their elasticity values are combined to produce a coefficient for the interaction. Usual values for Elasticity are between 0 and 1. Notice that values greater than 1, although physically invalid, are allowed. |
| Center of Mass | The center of mass of a rigid body defines its local balance point. This is normally calculated automatically. |
| Inertia Tensor Proportions | The inertia tensor is normally a representation of the mass of a rigid body and how that mass is distributed throughout the body. Since mass is provided as a distinct property of these rigid bodies, the inertia tensor in this case is used only to specify the proportional distribution of that mass. This is normally calculated automatically. |

Table 4.8: Basic Rigid Body Properties

In addition to these basic properties, there are advanced properties which most users will not want to alter from the default, but which are available for finer control if needed (see the relevant chapters in the Havok Physics or Havok Physics 2012 manual for more details on their effects):

| | |
|---|---|
| Max Linear Velocity | The maximum allowable linear velocity of the rigid body in m/s. |
| Max Angular Velocity | The maximum allowable angular velocity of the rigid body in degrees/simulation frame. |
| Linear Damping | The linear damping to be applied to the rigid body, in units of inverse seconds. The linear velocity of the body then decays exponentially in time with rate constant given by the damping value. |
| Angular Damping | The angular damping to be applied to the rigid body, in units of inverse seconds. The angular velocity of the body then decays exponentially in time with rate constant given by the damping value. |
| Allowed Penetration | The maximum allowed penetration depth for this object. |
| Quality Type | The quality type governs general features of the behavior of the rigid body on interaction with other bodies. "Fixed" or "Keyframed" will disable the mass control (since the body will have no mass). |
| Solver Deactivation | Allows you to enable an extra single object deactivation schema. That means the engine will try to deactivate single objects if those objects get very slow. This does not save CPU resources, though it can dramatically reduce slow movements in big stacks of objects. Options are "Off", "Low", "Medium", "High". |
| Deactivator Type | Determines which mechanism Havok will use to classify the rigid body as deactivated. Options are "Spatial", "Never". |
| Collision Filter Info | This (integer) value can be used by collision filters to identify the entity - for example, if a group collision filter is used, this value would specify the entity's collision group. |

Table 4.9: Advanced Rigid Body Properties

**Overriding the Center of Mass / Inertia Tensor**

The values of the "Center of Mass" and "Inertia Tensor" properties of a rigid body are normally determined automatically by the Create Bodies (or [Physics 2012] Create Rigid Bodies) filter. This is done by gathering the shape information related to a rigid body and integrating over its volume, which results in accurate and realistic values for objects with uniform density.

However, it may be desirable to specify a specific value for either of these properties from within the modeler. In each supported modeler, options are provided to override each of the center of mass and the inertia tensor. If 'override' is enabled, then the filter will use the value specified in the modeler instead of automatically calculating a value.

#### 4.7.3.2 Shape Properties

Shapes work together with a rigid body to define the collision behavior of (a part of) the body. A rigid body can use a single shape or it can use several shapes grouped together (these are known as *compound* rigid bodies).

| | |
|---|---|
| Shape Type | This specifies the type of shape which should be used for runtime collision detection. An appropriate runtime shape of this type is created by the Create Bodies filter, by determining the closest possible match to the object's geometry. |
| Extra Radius | Specifies an additional collision radius to be used by the shape - this can be useful in optimizing the simulation speed at runtime. The interpretation of this radius depends upon the Shape Type: For convex bodies this specifies the convex radius, for meshes this specifies the per-triangle radius. |

Table 4.10: Shape Properties

**Modeler vs. Simulation Shapes**

The Havok Content Tools allow you to specify a shape type to be created from a given a mesh in a modeler. That means that, for the physical simulation, you generally use a different representation for the object to the actual representation in the modeler. For example, you can simulate a car chassis as a box, a head as sphere, or a forearm as a capsule. The advantage of this is that a simpler representation (box, sphere, cylinder) will be simulated faster than the original mesh models could be, with no noticeable changes on behavior. Another advantage is that it is more memory-efficient to represent these simple objects (e.g. a sphere is just a center and a radius) than it is to represent a whole mesh. Therefore, it is important to carefully select a shape type for each shape in your scene.

The following list presents the different kinds of Havok shapes, ordered by their complexity (simpler shapes will be simulated faster):

1. Sphere (simplest)
2. Capsule
3. Box
4. Convex Hull / Cylinder
5. Mesh (most complex)

Notice that this is only an indication. Complexity of the simulation also depends on many other variables - whether the object is fixed or not, whether objects are in contact or proximity, relative size, etc..

Shapes used by run-time rigid bodies can not change throughout a simulation. If an object whose geometry changes over time is used to define a shape in the modeler, then only its geometry at the initial export frame is used to define the exported shape.

**Convex vs. Concave Objects**

A shape is defined as *convex* if, given any two points inside the object, you can always go in a straight line from one to the other without leaving the object. Convex objects include spheres, cylinders, and boxes. On the other hand, a donut or a cereal bowl are *concave*.

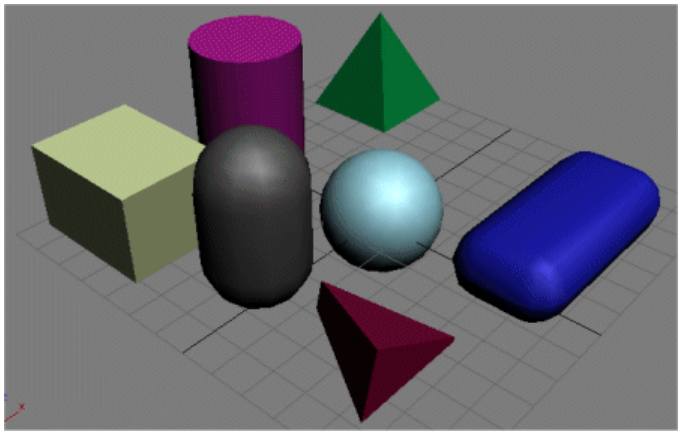


Figure 4.40: Examples of Convex Objects

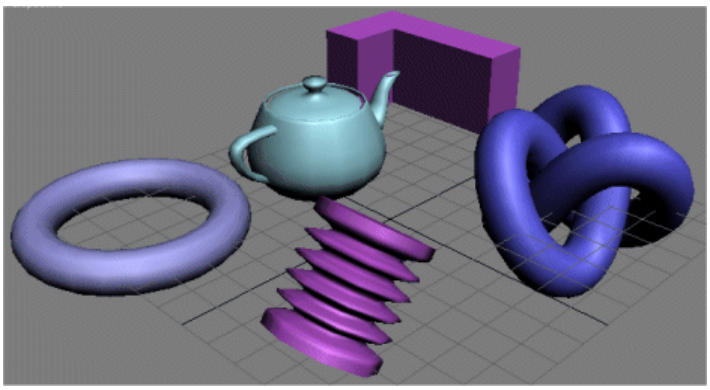


Figure 4.41: Examples of Concave Objects

Concave movable objects are inefficient for runtime simulation. The Physics filters can (and by default, will) construct convex shapes from any concave meshes. The only shape type which will replicate the polygonal surface of concave objects is the **Mesh** type.

In many cases, you can approximate a concave object by combining multiple convex objects. For example, the L-shaped object in the picture above could easily be simulated as a compound rigid body with two box shapes. Check the compound rigid bodies section for more details.

#### 4.7.3.3 Simple Rigid Bodies

Simple rigid bodies are those in which both rigid body and shape information are attached to the same object:

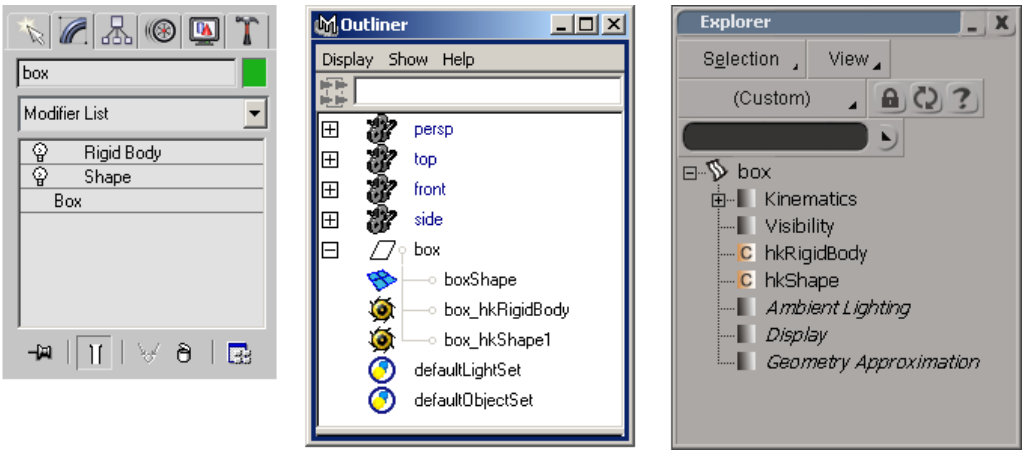
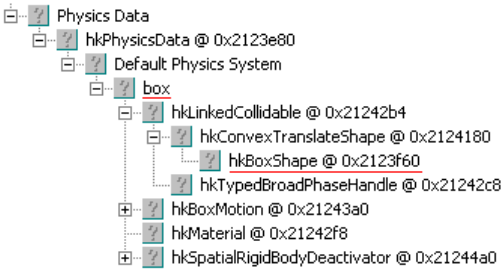



Figure 4.42: Simple Rigid Bodies

In this case the modelers have attached rigid body and shape information to the same object, named 'Box' - in 3ds Max they are attached as modifiers, in Maya, they are attached as nodes, and in XSI they are attached as parameter sets.

On export, the Physics filters will create a runtime rigid body object named 'Box', with a single box shape based on that of the box's mesh. This can be seen from the resulting XML (using a View XML filter):




**Note:**

The shape type of new shape modifiers/nodes/parameter sets is set automatically by the Havok Content Tools. In this case it has automatically been set to 'Box', since the shape information is attached to a box mesh in each modeler. Also, a non-zero mass has been specified in the rigid body information (or the exported body will always be fixed).

#### 4.7.3.4 Compound Rigid Bodies

Compound rigid bodies are those in which rigid body and shape information are attached to some object, and additional shape information is attached to some children of that object:

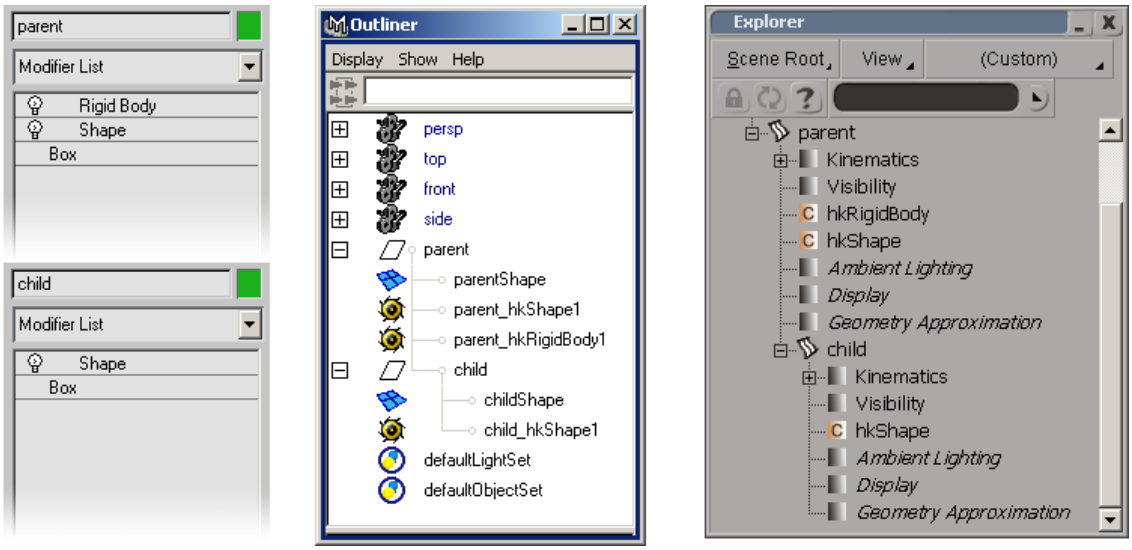
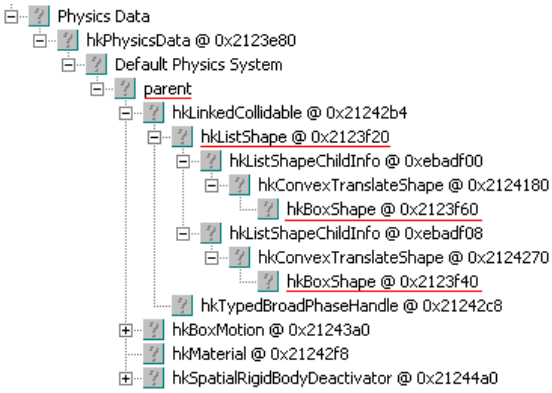



Figure 4.43: Compound Rigid Bodies

On export, the Physics filters will in this case create a single runtime rigid body object (called 'parent') which contains a compound shape made of the two child shapes:




One of the `hkpBoxShape`s above is generated from the parent mesh, while the other is generated from the child mesh.

#### 4.7.3.5 Rigid Bodies with Proxies

The final, and most unusual, type of rigid body is one in which only rigid body information is attached to some object, and only shape information is attached to some of its children:

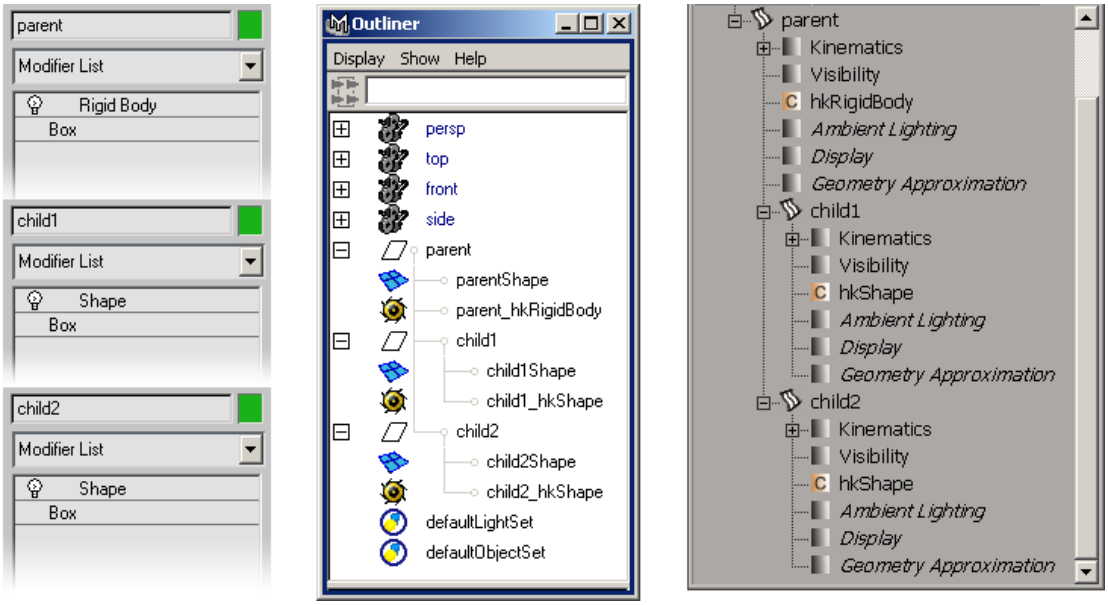
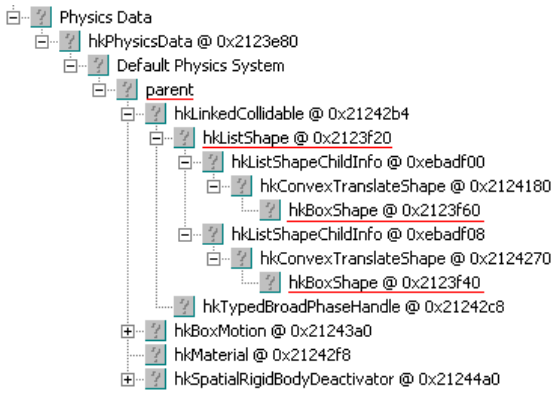



Figure 4.44: Rigid Bodies with Proxies

On export, the Physics filter will again create a single runtime rigid body object (called 'parent') which contains a compound shape made of the two child shapes:




In fact, the resulting Havok rigid body structure is the similar to that of the compound rigid body case. However, in this case both `hkpBoxShapes` are generated from the child meshes - no shape is generated from the parent mesh.

## 4.7.4 Physics : Constraint Concepts

### 4.7.4.1 Constraint Spaces

Setting up physical constraints between rigid bodies is the next logical step once rigid bodies have been created.

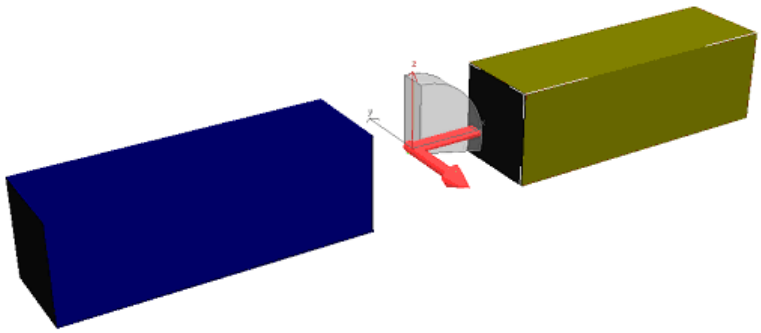
The most important concept associated with constraint setup in the Havok Content Tools is that of constraint spaces. In this section we will explain this concept and how is it used during constraint setup.

The Create Constraints filter is based on these concepts, and uses them to convert constraint definitions in a modeler into runtime constraint objects during the export process.

### Parent and Child Rigid Bodies

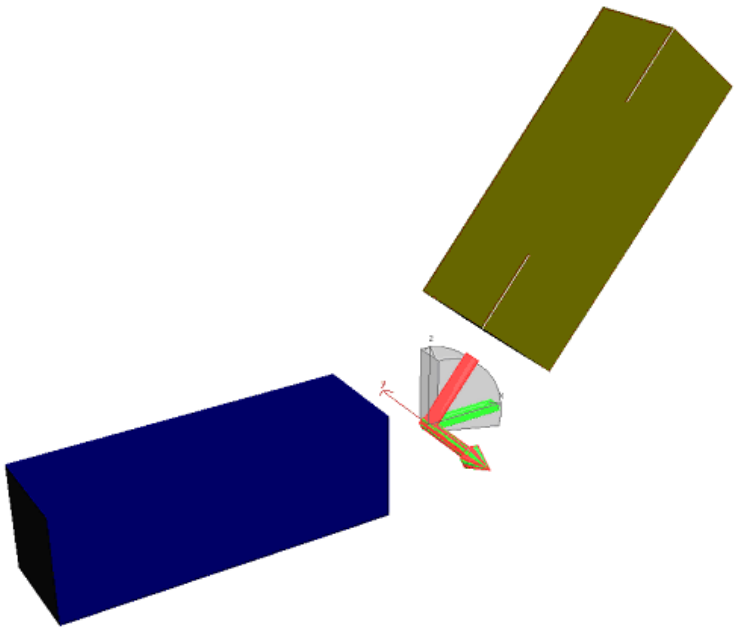
Constraints operate symmetrically: they affect two rigid bodies, and since movement is relative, it can be argued that there is no distinction of the roles of each rigid body. This is very clear, for example, in a spring connecting two rigid bodies.

However if we consider a hinge, we can begin to make a distinction:

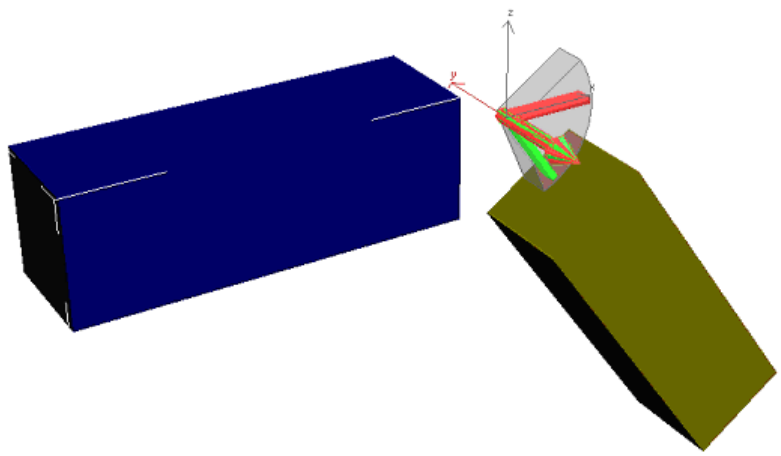


Given the configuration above where the shaded volume defines hinge limits, can the olive green box move 90 degrees up, or 90 degrees down? The answer is: it depends on the frame of reference of those limits.

If the limits are displayed based on the blue box and the red mark follows the olive green box, then the olive green box can move upwards:



However, if only the red mark follows the blue box and the limits follow the olive green, then the olive green box can move downwards:



This example illustrates the following idea:

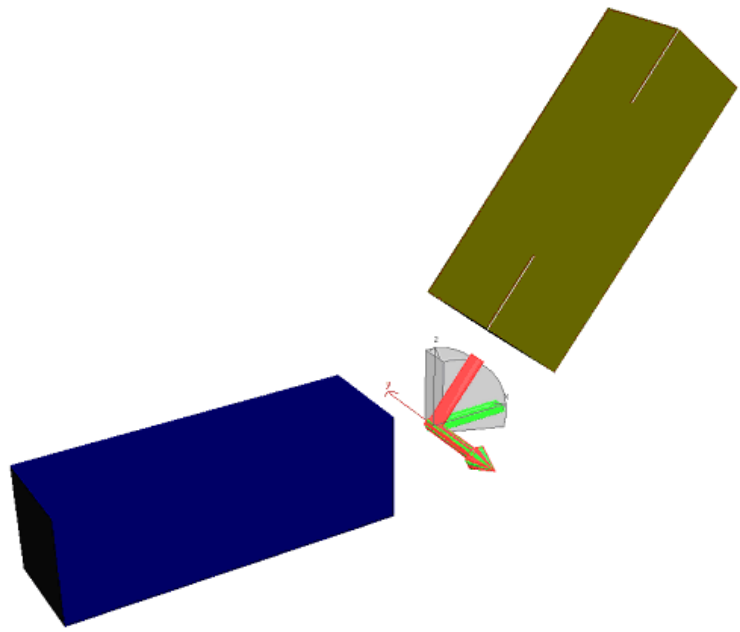
*When displaying information to represent constraint limits, the rigid bodies acquire a role in the constraint. One of them (which we denote as the parent) is associated with the spatial representation of the limits, and the other one (which we denote as the child) is associated with the free element that moves inside those limits.*

The reasoning behind naming the roles of the rigid bodies as *parent* and *child* is the analogy with bone hierarchies and forward kinematics. Movement of every joint in a hierarchy is usually defined relative to the parent bone of the joint: it is much more intuitive to define the limits based on the parent bone than on the child bone.

Consider, for example, a shoulder: you would usually define the joint as a rotation of the arm inside a cone that follows the torso, rather than the rotation of the torso inside a cone that follows the arm. They are both equivalent, but the first is much more intuitive.

**Parent and Child Constraint Spaces**

Closely related to the concept of parent and child roles for rigid bodies, there is also the concept of parent and child *spaces* associated with a constraint.



The space represented in green, which includes also the limits displayed in gray, is the *Parent Space* - this follows the parent rigid body at runtime. The space represented in red above is what we call the *Child Space* - this follows the child rigid body at runtime.

The important thing to remember is that a constraint operates by considering two spaces: Parent Space (which follows the Parent rigid body) and Child Space (which follows the Child rigid body). At run-time, the constraint will try to ensure that both spaces match (within the limits of each constraint) in world space. For example, a ball-and-socket constraint will ensure that the position of both spaces matches, but will ignore orientations. A hinge constraint will ensure the position and the direction of the main axis match, and will only allow some (possibly limited) rotation around that main axis.

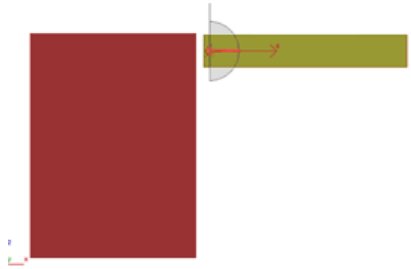
**Note:**
When a constraint constrains a rigid body to the world, the rigid body in question is always the child rigid body. The parent rigid body is considered to be a fixed rigid body (i.e., the parent space is fixed in world).

### Manipulating Constraint Spaces

When working with constraints, therefore, we have four spaces to consider:

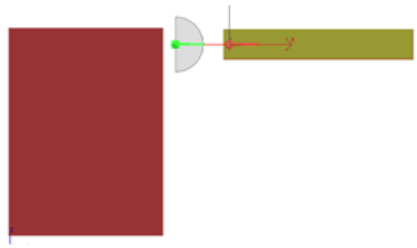
- **Parent Rigid Body Space** : The position and orientation of the Parent rigid body
- **Parent (Constraint) Space** : The position and orientation of the Parent Space of the constraint
- **Child Rigid Body Space** : The position and orientation of the Child rigid body
- **Child (Constraint) Space** : The position and orientation of the Child Space of the constraint

As we have seen, at runtime the parent and child constraint spaces always follow the relevant rigid body spaces. When setting up constraints in a modeler, however, we may want to limit how the spaces move relative to each other during manipulation to make setting up the constraints easier and more intuitive. Consider the following example:

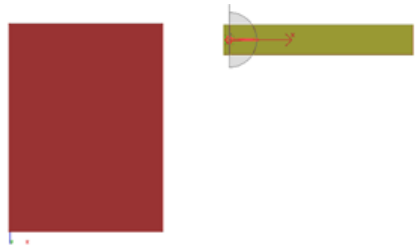


We have setup a hinge to connect the olive green box (arm) to the red box (torso). Notice that the two spaces of the constraints are aligned.

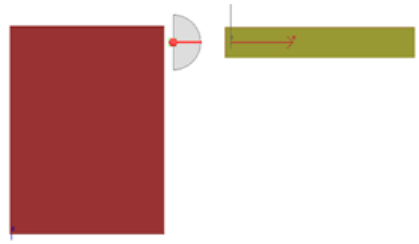
Imagine we now decide that we want the olive green box to be a little further away, so we move the olive green box to the right. If we keep each space relative to its associated rigid body, we'll get something like this:



Which basically means that now the two constraint spaces are not aligned, since they have different world positions. Unless we fix this, at runtime the constraint will snap in order to align the translation components of both spaces. In most cases, what we would have liked would have been for the whole constraint to move with the child rigid body:



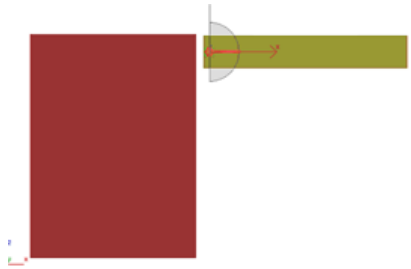
or instead for the whole constraint to remain where it was:



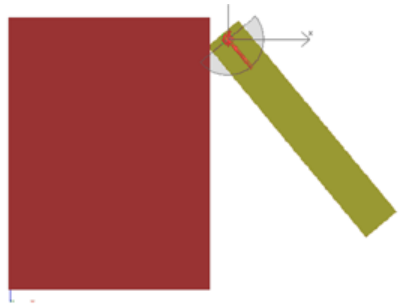
So, one general rule is that:

*Very often during constraint setup, we want both spaces of the constraint to move as a unit.*

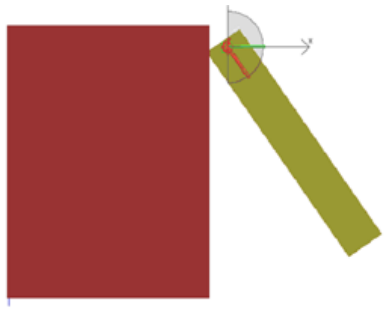
But notice that this not may always be the case. Consider the similar example where we start with the same setup as before:



Imagine now that we are happy with the constraint setup, but want our 'skeleton' to be in a different pose, maybe in order to match another skeleton or skin. In particular, we want the arms to be lowered down. If the two spaces of the constraints move as one, then what we would end up with is:



This would change the behavior of the constraint - the arms won't be able to go as high as before. So in this case, we would prefer both spaces to remain independent, so only the child space (in red) moves.



So, refining the rule above:

*Very often during constraint setup, we want both spaces of the constraint to move as a unit. But not always.*

In particular, for constraints like hinges and ball-and-sockets, we usually don't want the translation of the spaces to differ (because it will cause snapping) but for rotations we may. And for springs or prismatic constraints having differing positions for the two spaces is common.

There is another general rule regarding constraint spaces and the pivot of rigid bodies:

*Very often the child constraint space, in particular its position, is the same space as the child rigid body space.*

The reason for that is that when we create hierarchical systems of objects (like bones) in the modeler, we usually place the pivot points at the locations where the bones rotate relative to each other. When replicating the structure with rigid bodies, we usually place the constraints at those locations.

For example, in the case of the arm above, the most common location for the pivot of the olive green box in the modeler would be the location where we have placed the constraint.

**Constraint Manipulation in the Modelers**

The Havok Content Tools package provides tools for setting constraints and manipulating constraint spaces in each supported modeler. These tools include options to customize the behavior during constraint manipulation (as described above).

In each modeler, constraints are always associated with the *child* rigid body. There are many reasons for this:

- When representing a hierarchy with rigid bodies and constraints, there are N rigid bodies and N-1 constraints. This means there is a 1-to-1 association between rigid bodies and constraints, except for the root rigid body, where there are no constraints.
- This also reflects the close relationship between bones and joints in skeleton setups: wrist/hand, knee/calf, ankle/foot.
- As we have seen, the constraint spaces are usually located at the pivot location of the child rigid body.
- A constraint always has a child rigid body, while the parent rigid body is optional - it may be constrained to the world.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

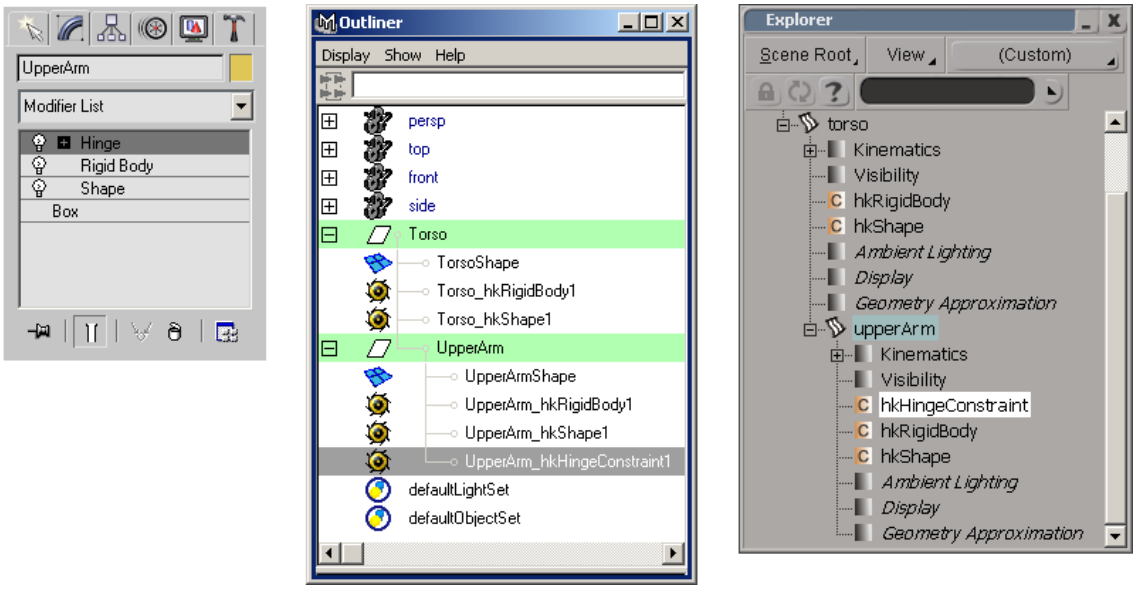



Figure 4.45: Constraints are always associated with the Child Rigid Body

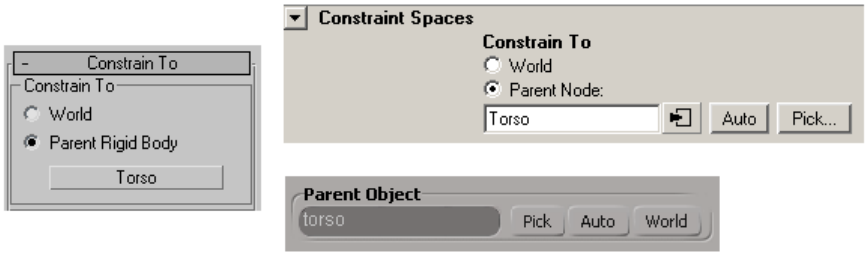



Figure 4.46: The (optional) Parent Rigid Body is a parameter of the constraint

### Manipulation Tools

Constraint spaces can always be specified numerically within each modeler (through a constraint's modifier/attribute editor/property panel). However it is much more intuitive to be able to interact with a constraint dynamically in the viewports. This behavior is provided by the manipulation tools. Each modeler has a specific implementation of the manipulation tool, since the capabilities and internal behavior of the modelers differ greatly:

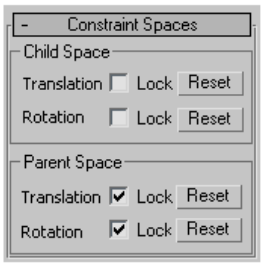
- In 3ds Max, Constraint Spaces are represented as subobjects of each constraint modifier, and can be manipulated using standard move and rotate tools.
- In Maya, Constraint Spaces are modified through manipulators associated with each constraint node. These manipulators can be accessed by directly selecting a constraint node and activating the 'manipulate' tool, or by indirectly selecting a constraint node (e.g., by selecting its parent transform) while the 'Havok Physics 2012' tool is active - this tool provides and quicker means to access and manipulate Havok nodes in general.
- In XSI, Constraint Spaces are modified through a custom setup, which is activated by the 'manipulate' buttons in a constraint parameter set's property panel. A temporary hidden null object is created in the scene, and is wired to the constraint space parameters. By translating/rotating this null object, an effective manipulation scheme is achieved. Once manipulation is complete, we

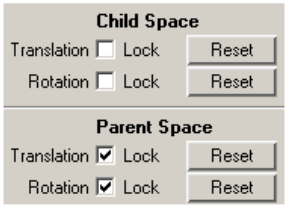
need to remove the temporary nodes - this occurs if the selection changes or if the 'end manipulate' button is pressed.

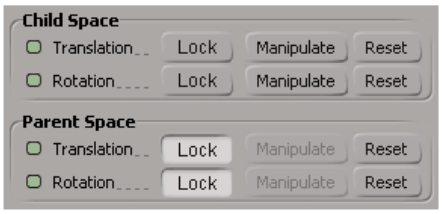
See the individual modeler documentation for more on each manipulation tool.

### Locking/Unlocking of Spaces

In order to allow different manipulation modes and specific behaviors of constraints how are modified within a scene (as described earlier), Havok constraints allow the user to lock and unlock the constraints spaces, both in terms of their translation and rotation:




### Child Space

- **Lock Translation**

  If ON, the translation of the child space is fixed and cannot be manipulated (by default, the child space is located at the pivot location of the child rigid body). If OFF, the translation component can be manipulated.

  The Child Space is always kept relative to (always follows) the Child Rigid Body.

- **Reset Translation**

  Pressing this button aligns the translation of the Child Space to that of the pivot of the child rigid body.

- **Lock Rotation**

  If ON, the orientation of the child space is fixed and cannot be manipulated (by default, this orientation matches the orientation of the child rigid body pivot). When OFF, the orientation of the child space can be modified/manipulated.

  The Child Space is always kept relative to (always follows) the Child Rigid Body.

- **Reset Rotation**

  Pressing this button will align the orientation of the Child Space to match the orientation of the child rigid body.

**Parent Space**

- **Lock Translation**

  If ON:

  - The translation of the Parent Space cannot be manipulated (by default is aligned with the Child Space).
  - The translation of the Parent Space is kept relative to the Child Space (i.e., both spaces will move together).

  If OFF:

  - The translation of the Parent Space can be manipulated.
  - The translation of the Parent Space is kept relative to the Parent Rigid Body (if there is any) or fixed in world (if there is no parent).

- **Reset Translation**

  Pressing this button will align the translation of the Parent Space to that of the Child Space

  **Note:**
  The exception being the Stiff Spring constraint, where the Parent Space will be aligned with the parent rigid body.

- **Lock Rotation**

  If ON:

  - The orientation of the Parent Space cannot be manipulated (by default is aligned with the Child Space).
  - The orientation of the Parent Space is kept relative to the Child Space (i.e., both spaces rotate together).

  If OFF:

  - The orientation of the Parent Space can be manipulated.
  - The orientation of the Parent Space is kept relative to the Parent Rigid Body (if there is any) or fixed in world (if there is no parent).

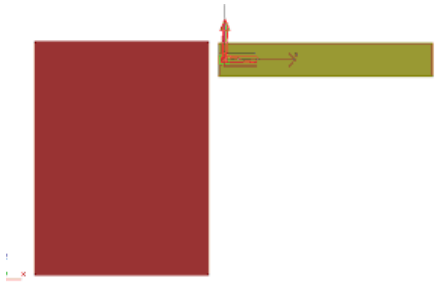
- **Reset Rotation**

  Pressing this button will align the orientation of the Parent Space to that of the Child Space.

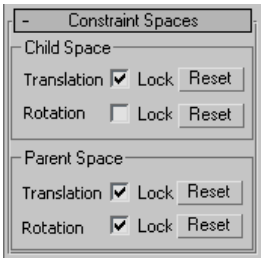
Notice how locking/unlocking the child space only affects whether manipulation is available or not; while in the case of the parent space it also affects the behavior of that space whenever the rigid bodies in the constraint move.

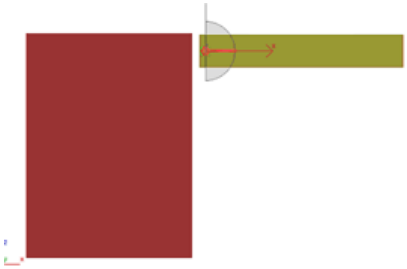
**Example**

In this case a constraint is created with the 'arm' as the child rigid body and the 'torso' as the parent rigid body:

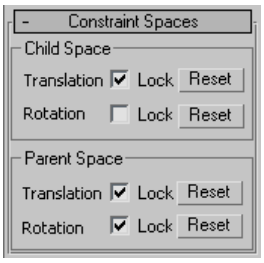


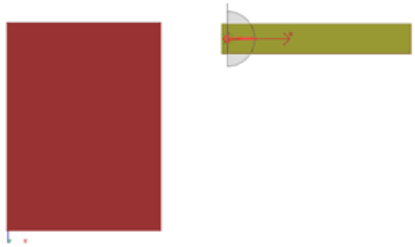
Creating the constraint by default aligns both parent and child spaces with the pivot of the child rigid body. We may want to orientate the whole constraint, for example to match the desired rotation axis for a hinge. We unlock rotation of the child space (so we can change it), and keep the parent space fully locked, so it follows the child space (and the constraint moves as one). Now when we rotate the child space, the parent space follows:



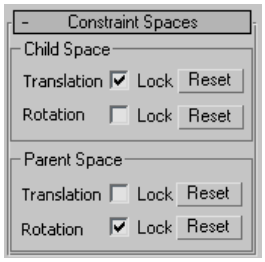



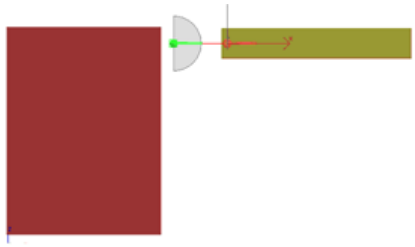
Imagine that we now want to move the arm to the right (or the torso to the left). By keeping the parent space translation locked, both spaces move/remain together with the child rigid body:



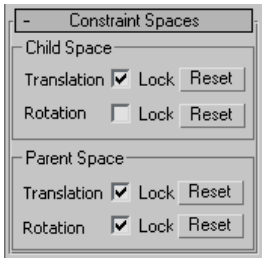


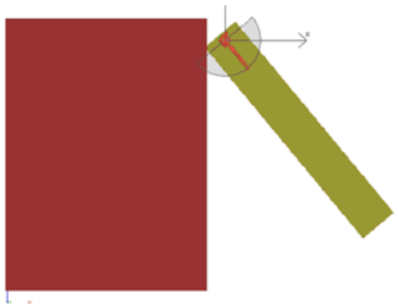
However, if the translation of the parent space is unlocked before the translation, the parent space moves/remains with the parent rigid body:



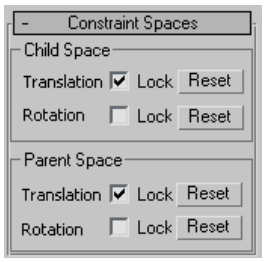



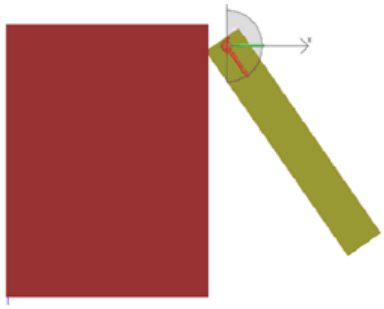
Consider now the case where we want to rotate the arm clockwise (or the torso counter-clockwise). In the first case the parent space rotation is locked, so both spaces rotate together:



However, if we unlock the parent space rotation, it remains relative to the parent rigid body and does not follow the child space:



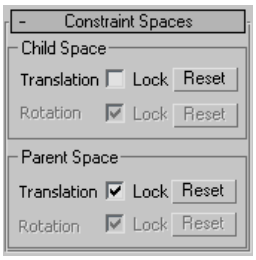
#### 4.7.4.2 Constraint Types

The Havok Content Tools provide several types of constraint for each supported modeler. The Create Constraints filter converts any of these constraints found in a scene into their runtime equivalents during the export process.

**The Ball and Socket Constraint**

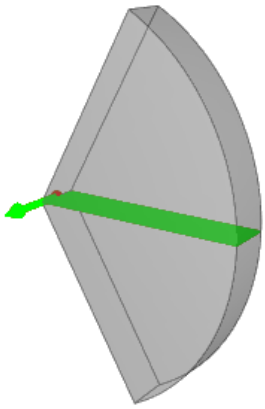
The ball and socket constraint is the simplest type of constraint provided by the Havok Content Tools. At runtime, this constrains the child space position in the child rigid body to the parent space position in the parent rigid body / world.

This constraint does not provide any additional properties beyond those provided by the common constraint space properties. Also, this constraint has no concept of rotation so the rotation elements of the constraint space properties are disabled/hidden:




**The Hinge Constraint**

The hinge constraint allows you to constrain two rigid bodies together relative to a position and axis in each body's space, or to constrain an axis in one body's space to an axis in world space. The constrained objects are then free to rotate about this axis. The allowed rotation between the attached objects may also be limited. This limit is defined with respect to an axis perpendicular to the hinge axis (the *zero-axis*) for each body. These sets of axes are defined by the child and parent constraint spaces.



The properties provided by the hinge constraint include the standard constraint space properties plus: the option to limit the angle of rotation; a frictional torque value for the rotation; a motor type to drive the rotation:

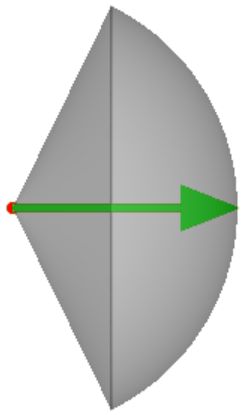
Some modeler's also provide a 'display size' property, which affects the display in the viewports only.

**The Rag Doll Constraint**

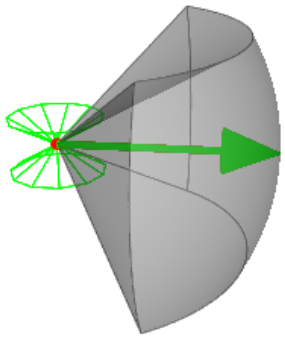
Rag doll constraints are useful for simulate complex joints, such as those found in some human and animal joints (shoulders, hips). These joints may require some freedom of rotation in more than one axis, as well as some ability to twist.

Like the other constraints we have seen so far, this one constrains the child space position in the child rigid body to the parent space position in the parent rigid body / world. However the rotational constraints have a somewhat complex parametrization which requires some additional explanation.

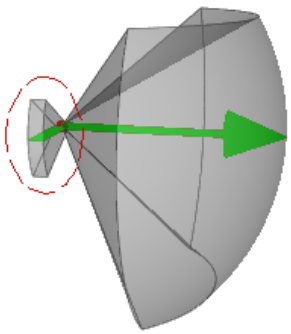
The first parameter is a 'cone angle', which is measured from the main axis of the parent space. This ranges from 0 to 180 degrees (no freedom to full freedom):



Two other angular parameters (which we call 'plane max' and 'plane min') are used to specify cones which subtract from the volume above. These cones are specified from axes which are perpendicular to the main axis, one from the positive axis and the other from the negative axis. Their effect is clear when a three dimensional view is taken:



The main axis of the child space is now constrained to be located within the shaded volume in the diagram above. We can also constrain the rotation of the child space around its main axis. This behaves the same way as the hinge constraint, and can be represented in a similar way - as an additional volume at the base of the rag doll constraint:

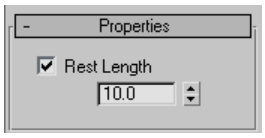


The properties provided by the rag doll constraint allow these various angular parameters to be edited, as well as a frictional torque value and a motor type. Options are also provided which give some control over the visual representation of the constraint in the modeler:

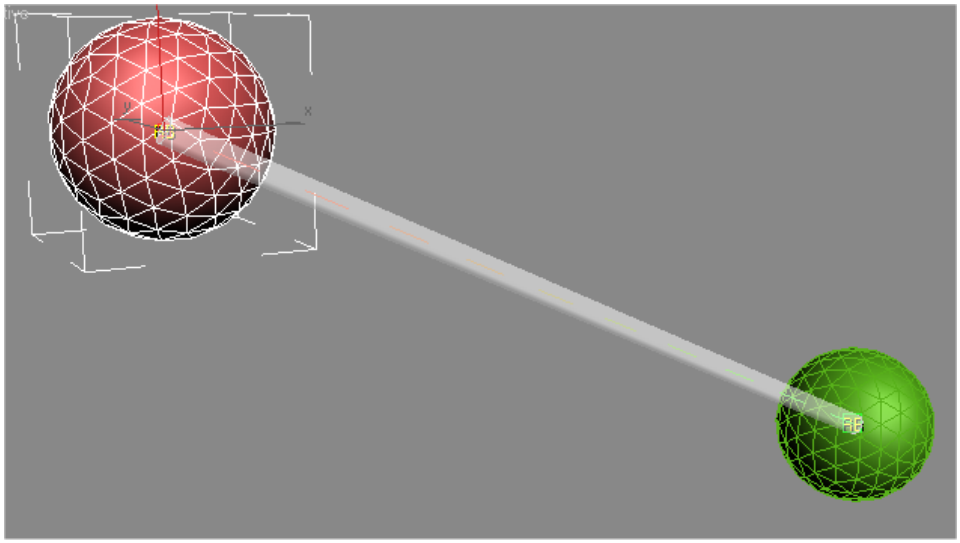
Again, some modeler's also provide a 'display size' property, which affects the display in the viewports only.

**The Stiff Spring Constraint**

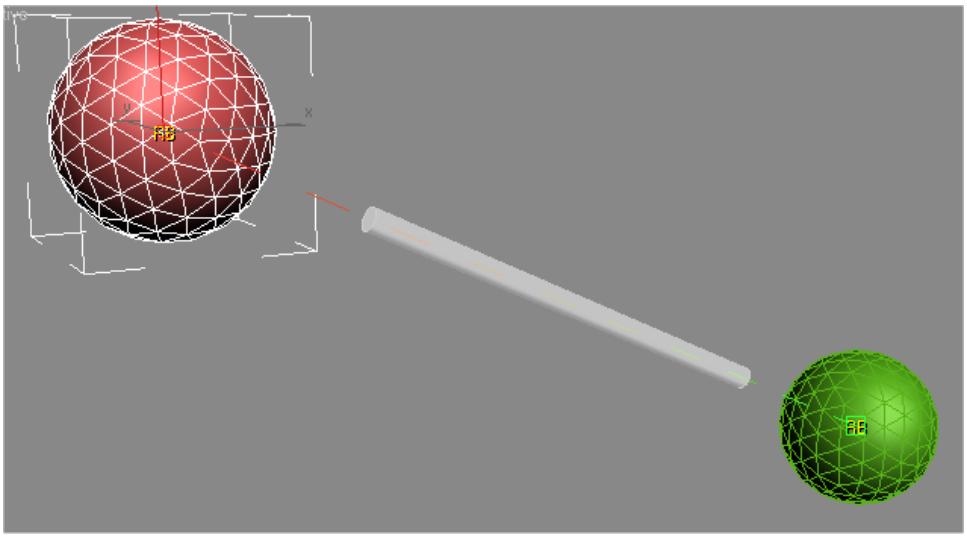
The stiff spring constraint constrains the child space position in the child rigid body to lie a set distance from the parent space position in the parent rigid body (as if they were connected by an inflexible rod). As in the ball and socket constraint, there is no constraint on the child or parent space rotation.




The properties provided by the stiff spring constraint include the standard constraint space properties plus an optional rest length parameter. If this parameter is disabled, the spaces are constrained by default to be separated by their current distance.



Specifying a rest length parameter overrides this default.

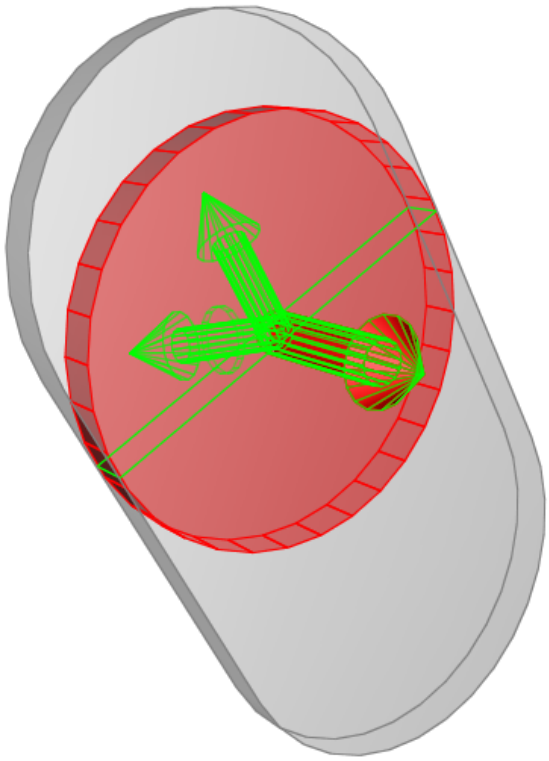


The rest length is displayed as a solid gray bar (of that length) in between the two spaces. A dashed line is also shown in between both spaces.

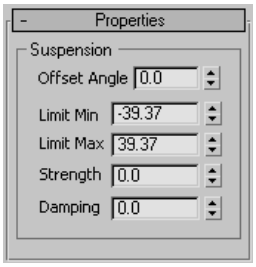
**The Wheel Constraint**

The wheel constraint constrains the child rigid body to rotate about a specified "spinning" axis relative to the parent body, and optionally to translate along a specified "suspension" axis for specified distances in each direction. The "steering" axis is by default aligned with the suspension axis, but may be offset relative to the suspension axis by a specified rotation angle about the spinning axis.

In the constraint display, the child space is represented as a wheel in the plane perpendicular to the spinning axis, indicated by an arrow. The suspension axis is indicated by an arrow with a perpendicular plane marking the zero point of the suspension limits. The limits are represented by a surface showing the volume swept out by the wheel as it moves along the suspension axis between the limits. The steering axis, which differs from the suspension axis if the offset angle is set to be non-zero, is denoted by an arrow with an annular ring around it.



The properties provided by the wheel constraint include the offset angle between the steering and suspension axis, the suspension limit minimum and maximum (negative and positive values respectively), and frictional torque and damping parameters.

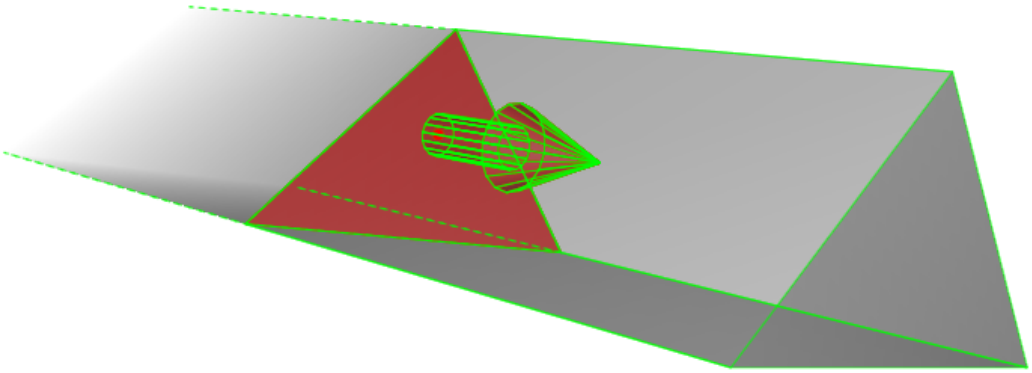



The 'display size' property in this case scales the size of the displayed wheel, plane and arrows while keeping the distance between the suspension limits constant.

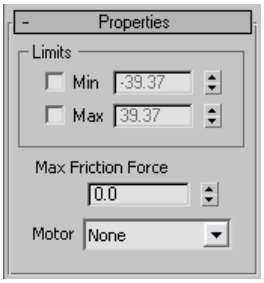
**The Prismatic Constraint**

The prismatic constraint constrains the child rigid body to move along a specified axis relative to the parent body by specified distances in each direction (or if not specified, by an unlimited distance).

In the constraint display, the child space is represented as a plane perpendicular to the main axis, indicated by an arrow. The limits are represented by a plane perpendicular to the main axis indicating the zero position, and a surface showing the boundary of the volume swept out by the plane as it moves along the prism axis between its defined limits. If the minimum or maximum limit is not enabled, the prism volume in that direction fades out.



The properties provided by the prismatic constraint include optional minimum and maximum limits (negative and positive values respectively), a friction value, and motor type.




The ' **Display Size** ' property in this case scales the size of the displayed prism and arrows while keeping the distance between the limits constant.

#### 4.7.4.3 Breakable Constraints

Any constraint can be made breakable if desired, by selecting the " **Breakable** " checkbox. The " **Threshold** " parameter specifies the force beyond which the constraint breaks.

### 4.7.5 Animation: Repositioning Animations and Motion Extraction

In order for animation blending to work well at runtime it is necessary that the animations being blended are 'similar' to avoid ambiguities about the path a bone should take between orientations or positions which are wildly different (such as a facing forward and a facing backward pose). In most cases this is guaranteed by the natural limits of motion of character limbs, but for the *root* bone in particular this may not be the case.

For example, the source of the animation may come from mocap data where the starting pose is displaced from the origin and rotated arbitrarily. Additionally all animations with 'motion' such as a walk-forward animation will over the course of the animation have large changes in displacement along one axis which needs to be handled when the animation loops.

In order to address these problems, the Havok Animation Content Tools provide functionality to reposition or realign animations, and also to split an animation into two components using Motion Extraction. Motion Extraction is motivated by the need to loop animations and also provides a coarse representation of motion suitable for physical simulation. Motion Extraction preserves the content of the original motion by storing extracted components in a separate channel. Repositioning or Reorienting the animation is motivated by the need to blend character poses together. Repositioning or Reorienting changes the underlying animation data irrecoverably. The details of these operations are described below.

#### 4.7.5.1 Repositioning and Reorienting Animations

You can modify the root transform of your animations to be repositioned or realigned with the world at two points in the filter pipeline:

- The Create Animations filter allows you to shift the entire animation during creation in each of the 3 world directions (X,Y or Z) so that these component of the root position are initially zero. This will preserve overall motion of the root, but ensure that the animation starts at the origin.

  This may be useful for modeler assets which have not been consistently 'centered' to the origin. You may wish, for example, to ensure that a character's root always starts directly over the origin (in X and Y) by checking the X and Y boxes here in the filter settings.

- The Rotate Animations filter allows you to reorient the entire animation so that the root aligns with a specified axis on the first frame. This will preserve overall motion of the root, but re-orient it in a given direction.

  This may be useful for or modeler assets which have not been consistently 'oriented' in their facing direction, similar to the above use-case. It may also be useful for cases where the *final* frame of the animation has known orientation, but the start frame is arbitrary, such as for example for a series of 'sit down' animations which approach a chair (at the origin say) from arbitrary positions and orientations.

**Note:**

In both these use cases the animation still contains all of the 'motion'. For example, if the animation is a walk-forward animation (not on-the-spot) then using either of the above functionalities will keep that motion though they will translate its start location or reorient its direction. This is in contrast to the Extract Motion filter, see below.

#### 4.7.5.2 Using Motion Extraction

The Extract Motion filter will extract specified components of the root track and hence will remove some of the 'motion', placing it alongside the animation track in a `hkaAnimatedReferenceFrame` object. In our walk-forward animation case, using the Extract Motion filter will result in the animation becoming an on-the-spot animation, with the forward motion pulled out into a separate `hkaAnimatedReferenceFrame`.

**Note:**
No information is lost - the component removed is always stored in its entirety in the m_extractedMotion member of the animation (as may be seen by examination using a View XML Filter) and recombining the extracted motion with the resultant animation at runtime will exactly produce the original animation.

**Orientation Conventions**

The Animation section of the Havok User Guide details the motivation, design and use of Motion Extraction from a runtime viewpoint. Because the conventions chosen in the modeler vary in the industry , the Havok Content Tools offer multiple options for motion extraction, however it is expected that conventions are consistent across all assets. For example, it is expected that all animations for a given character are modeled with the same 'facing' direction. If this is not the case, then the Rotate Animations filter may be used as a preprocess *before* motion extraction is applied, but it is not expected that motion extraction be used to 'reorient' an animation.

**Choice Of Sampling**

The Extract Motion filter will always remove motion from the *root* of the animation. Usually the motion you want to remove will be a 'simpler' version of the original motion. For example with the root of the animation being the pelvis of a walking character, the animated root motion will have a significant forward velocity with a small up and down velocity (due to foot lift and placement). The motion you will want to extract from that is likely a constant forward velocity only. This is illustrated in the first example in the Motion Extraction / Locomotion section of the User Guide. Here the extracted motion is 'simpler' in that it ignores the up/down component.

In general there are two ways that you can specify how the extracted motion is sampled from the scene:

- (Default) The Extract Motion filter allows you to *implicitly* specify this motion track by assuming you will synthesize the extracted motion *from the root itself*, but with only certain motions allowed in the final extracted motion. In this case you will normally filter out the (usually small) motions of the root you are not interested in by suitable choices in the 'Allowed Motion' check boxes, which allows you to create the 'simpler' motion track.

  This method is useful if the root of your hierarchy has a lot of detail in its movement (like the pelvis) but is close to the overall 'motion' you wish to extract.

- The Extract Motion filter allows you to *explicitly* specify this motion track by choosing the 'Use Alternative Reference' option. In this case you will normally have added a node to the scene which follows the exact motion you desire to be extracted from the root of the animation, and no further choices in the 'Allowed Motion' are necessary.

  This method is useful if you wish to have fine-grained control over the extracted motion, or if the motion of the character is complex (such as twisting about multiple axes, or with complex accelerations).

In both methods, you may further simplify the motion extracted by specifying how many samples are

used - the extracted motion will be assumed to be piecewise-linear (or constant velocity) between sample points.

### 4.7.6 Animation: Annotations

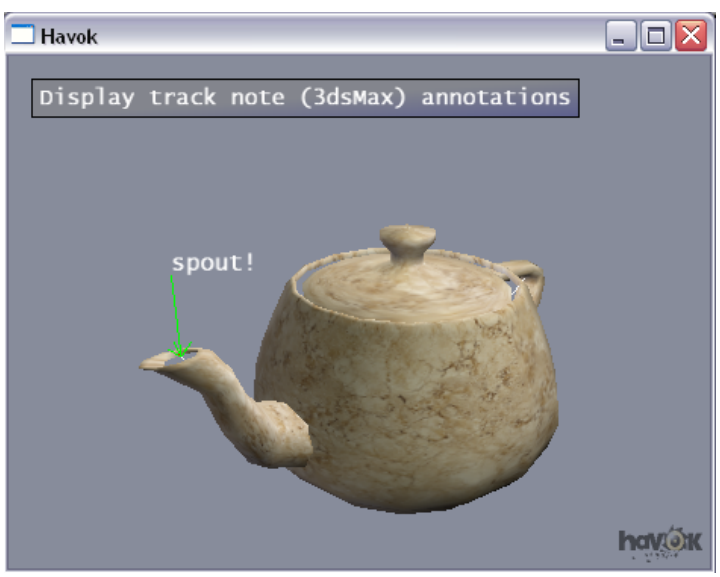
Havok Animation supports the concept of annotations. Annotations are animated string variables associated to bones. They are normally used to mark (annotate) particular frames of the animation that are interesting from the point of view of the application running the animation. Examples of usage would be:

- In a walk/run animation, annotate the frames where the foot is planted/lifted (to trigger sounds, or to do IK)
- Mark transition points between different animation clips
- Mark interesting poses in an animation (for example, Havok's GDC 2005 demo uses multiple annotations in a single "get up" animation in order to identify good poses from which to drive a rag doll before getting up.

The Havok Content Tools provide support for exporting annotation data from each supported modeler. See the Havok Content Tools documentation for your relevant modeler for more on the specific implementations.

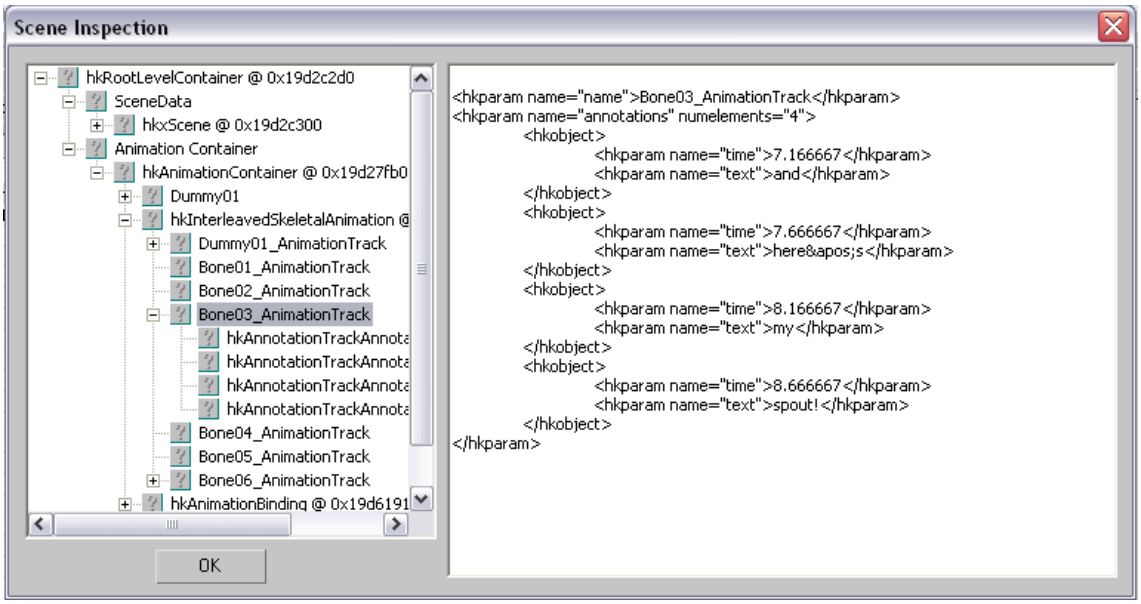
#### 4.7.6.1 Runtime Access to Annotations

The Annotation demo (Animation Api > Playback > Annotation), featuring an animated teapot which displays annotated strings, is a good starting point for learning about using annotations at run-time:




The `hkaAnnotationTrack` class stores annotation data. An `hkaAnnotationTrack` simply consists of a `time` and a `text` value, so multiple `hkaAnnotationTrack`s are used to represent an annotation as it

changes value, with each track containing the new value of the annotation and the time at which it changes. This can be seen in the following screenshot of the View XML filter:




A skeleton's annotated tracks can be accessed via the *hkaAnimatedSkeleton::BoneAnnotation* struct, which also stores the ID of the bone to which the annotation track belongs. The `getNumAnnotations()` and `getAnnotations()` methods in `hkaAnimatedSkeleton` analyze the specified timestep and return any relevant annotation data. The following code snippet from the Annotation demo shows how this is done:

```
// Grab any track note annotations that are present and valid for the time step
// Usually we would just query for this delta, but for demo purpose we extend this.
// This means the annotations are displayed on screen for longer
hkReal delay = 0.25f;

// Query to see how many annotation tracks we have.
const hkUint32 numAnnotations = m_skeletonInstance->getNumAnnotations( delta + delay );

// Create a temporary array and query for the annotations.
hkLocalArray<hkaAnimatedSkeleton::BoneAnnotation> annotations( numAnnotations );
annotations.setSize( numAnnotations );
m_skeletonInstance->getAnnotations( delta + delay, annotations.begin() );

// Advance the active animations
m_skeletonInstance->stepDeltaTime( delta );
```

### 4.7.7 Animation: Controlling Compression

There are several tweakable parameters available when using Havok's Animation compression pipeline (using the Quantized Compression, Spline Compression, Delta Compression and Wavelet Compression filters). The following guide provides a short explanation on how to choose these parameter values to balance size, quality and decompression speed.

Animation compression involves a trade off between size, quality, and to a lesser extent, speed. By

providing the user with a variety of compression options, Havok's Animation compression filters allow the user to tailor compression settings to their specific needs and requirements. The description and trade offs involved in choosing Spline and Quantized compression options are described in the Spline Compression Filter and Quantized Compression Filter sections, respectively. Controlling the output of the Wavelet and Delta Compression, which share a number of internal features, is described below.

**Warning:**
The final *file size* of the asset when written may not represent the minimal asset size in memory at runtime). Please see the section Common / Controlling File Size and Animation / Controlling File Size for how to produce files of minimal size.

#### 4.7.7.1 Controlling Quality: Wavelet and Delta Compression

The sections of the compression pipeline which may introduce error into the animation signal are:

- Truncation during track analysis
- Precision loss during quantization
- Wavelet coefficient truncation/thresholding
- 3-Component Quaternion Compression

**Track Analysis**

Track analysis attempts to remove static and clear degrees of freedom from your rig. You independently specify tolerances for position, rotation and scale. This analysis does not degrade gracefully; if you pick over-aggressive tolerances then the error is generally very obvious.

Symptoms of over-aggression are generally *low-frequency* error and include:

- Loss of movement in a limb
- A complete subskeleton appears rotated
- Collapsed bones (usually small bones collapse first e.g., fingers, toes and clavicle bones)
- Loss of root motion

To compensate, simply reduce the tolerances presented. Usually you can be more aggressive with positional tolerances than with rotational tolerances (the values compared for rotation are actually quaternion components; the tolerance does not represent an angular difference). It is a good idea to double-check the values displayed during track analysis to ensure that the automatic identification of degrees of freedom correspond to your expected rig setup. For example if none of the bones in your rig change scale you should expect to only see three dynamic position tracks for the root movement. All other dynamic tracks should correspond to rotation, for example:

*Bad: (Using the default Relative Position Tolerance = 0.01)*

```
[Track Analysis]
        Total DOFs 760
        Static position DOFs 201
        Static rotation DOFs 77
        Static scale DOFs 0
        Clear position DOFs 6
        Clear rotation DOFs 36
        Clear scale DOFs 228
        Dynamic position DOFs 21
        Dynamic rotation DOFs 191
        Dynamic scale DOFs 0
        Redundancy 3.584906
```

*Good: (In this case Relative Position Tolerance = 0.1)*

```
[Track Analysis]
        Total DOFs 760
        Static position DOFs 219
        Static rotation DOFs 99
        Static scale DOFs 0
        Clear position DOFs 6
        Clear rotation DOFs 36
        Clear scale DOFs 228
        Dynamic position DOFs 3
        Dynamic rotation DOFs 169
        Dynamic scale DOFs 0
        Redundancy 4.418605
```

As a rule of thumb, be as aggressive here as you can (larger tolerances), but do not stray far from the default values. In most situations the values here depend on the rig set up, so you can generally choose a good set of values for a given character and reuse them.

**Note:**
Rotation tolerance is usually far smaller than position tolerance and also far more critical as it is likely that most of your dynamic data will be rotational.

For more specific details on how the tolerances are used to analyze data see the section 'Track Analysis' in the Havok Animation manual.

### Quantization Bit Width

After track analysis all remaining degrees of freedom are transformed (using a delta or wavelet transformation) and then quantized. Quantization typically introduces more error than other stages.

Quantization error degrades more gracefully than track analysis, but is generally *high-frequency* (and thus may be more noticeable). Symptoms include:

- Jittering motion
- Shaking end effectors
- Popping motion

When choosing an appropriate quantization value pay close attention to the end effectors (hands and feet). Quantization error usually appears as shaking here and the obvious solution is to increase the number of bits used.

To combat quantization error it may be worth considering shortening chains in your skeleton. Currently error accumulates down the chains in hierarchies as all animation data is stored (and compressed) in local/bone space.

Also consider the range of values that needs to be quantized. Obviously the larger this range is the larger the error each bit represents. For example, if your root motion moves for `x=0` to `x=10000` and you are using 8-bit quantization you may see your character snap to regular positions. If the error is contained in the root motion, it is worth considering adding motion extraction to the pipeline.

Before quantization is performed, each degree of freedom is scanned to find the signal range for that track (min, max, offset and scale). These values are used for this track for the whole animation. For long animations it is often worth dissecting them into two or more shorter animations to improve both quality and compression. For example, if hands only move in the second half of the animation then splitting it in 2 will both improve the individual ranges for quantization and potentially find more redundancy during track analysis.

**Coefficient Truncation/Thresholding**

Wavelet compression adds an extra truncation/thresholding and encoding stage to the pipeline. Once wavelet transformation has taken place many of the values are zero or close to zero. The truncation/thresholding phase clamps a proportion of the total values to zero. The wavelet transformation itself is lossless. It is simply this truncation phase that introduces loss. This loss degrades the quality somewhat gracefully.

For wavelet compression to be effective, a heuristic for 'dropping'/discarding coefficients of the wavelet-transformed data must be picked. For versions of Havok pre Havok 5.0.0 this heuristic is to discard a fixed percentage (truncation fraction) of all (sorted) wavelet coefficients, which gives a roughly linear change in file size (a truncation ratio of 50% or 0.5 would half the size of a file with truncation ration of 0% or 0). This linearity is not exact because of the overhead of static data storage and the RLE used on the final data (hence it depends on bitwidth choice also). A better heuristic is to threshold the coefficients by an absolute value, and discard coefficients less than this value, or whose contribution to the final sample is less than a value. In this case the track analysis parameters can be used directly as the thresholding values, and the 'Compression' slider is no longer used. For the same quality of animation, this heuristic gives about 10-20% improvement in compression. We anticipate that this improvement should always be used in the future (hence the truncation algorithm is marked as 'Deprecated'). However we offer backward compatibility in both the Content Tools and the runtime. In the Content Tools, a checkbox for 'Use Truncation Ratio (Deprecated)' can be used.

**Note:**

If you are using 'truncation' the only other consideration is block size. The longer the input signal is, the better the wavelet transform will perform i.e., more values will be zero or close to zero. With our block encoding scheme the signal is presented to the wavelet transformer in blocks, each of which has 'block size' elements. As a rule-of-thumb the shorter the block size is the lower the truncation proportion should be; e.g. with a block size of 8 usually a truncation of 50 to 60 percent may be acceptable (this will set 4 or 5 of the 8 transformed values to zero!). With a larger block size, much higher proportions are possible with little visual loss.
A good example of this is cut-scene data. If there are no playback restrictions (see below) then the entire length of the animation can be wavelet transformed (block size = -1 or full) and a very high threshold used for the truncation proportion.

**3-Component Quaternion Compression**

In addition to the tolerances above, track analysis may optionally compress the rotation component of all tracks by using '3-component quaternion compression', reconstructing the 4th components from the other 3, and from Havok 5.0.0 onward does this by default. Since the remaining coefficients will be quantized (and in the case of a wavelet-compressed animation, wavelet-transformed before quantization), reconstruction *may* introduce some noticeable error if very aggressive compression settings are being used in conjunction with this algorithm. However, we do not expect that this option need be disabled for most use-cases. If you find problematic animations which exhibit noticeable error unless this heuristic is disabled, please contact Havok support.

#### 4.7.7.2 Controlling Decompression Speed and Memory

When setting block size for Wavelet and Delta compression, it is important to consider the runtime implications. An entire block must be decompressed together. A block size of 8 implies the decompression system must fully decompress 8 frames for each dynamic degree of freedom in your animation. Either you take the CPU hit for doing this or you pass a cache during sampling and cache the data.

See the section on caching in the Animation Runtime for more detail on how to choose an appropriate cache size.

**Note:**
If you are decompressing cut-scene data (compressed with block size=full) it is entirely possible to create a temporary cache for use while the cut-scene is playing back. This cache can be discarded when the cut-scene has finished playing.

See the Spline Compression Filter documentation for information on how settings affect the speed of decompression.

#### 4.7.7.3 Computing and Controlling Size

The following shows the statistics displayed at the end of compression:

```
[Compression]
      Original Size 125952 Compressed Size 5924 [1886 static/incompressible]
      Compression Ratio: 21.261311:1
```

This original size is computed from the storage requirement for a full 4x4 matrix (32 bit floats) per frame for each bone in the animation. (The example used here has 48 bones and 41 frames). The compressed size is the total in-memory requirement for the compressed animation. This data is comprised of a compressible part (the final output from the pipeline) and an incompressible part (data used for pipeline set up; e.g. the quantization ranges for each DOF, the track analysis mask and values, etc.). The value here is deliberately broken out as it will not change when you tweak either quantization bit size or truncation. This allows you to see how much further you can continue to compress (in this animation 4038 bytes are being used to represent the signal).

Also note that if you are using delta compression, the final size of the compressed animation doesn't reduce linearly with decreases in bit width. For example 16 bit delta compression is not twice the size of 8 bit. The reason here is that the block encoding is storing a full float at the start of each block. With a block size of 8 a full float is stored followed by 7 quantized values. Only the quantized values change as the bit width changes. The float storage remains constant. Similarly there is not a linear

relationship between the wavelet truncation coefficient and the final size of the compressed data. When data is transformed into the wavelet basis many values are already very small and after quantization go to zero. The truncation coefficient simply guarantees a proportion of the data is in fact zero - this allows the encoder to work well.

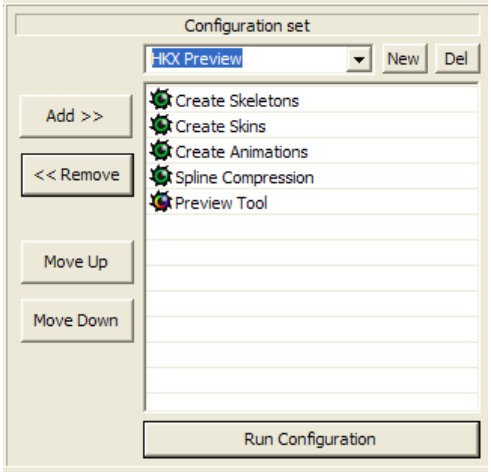
The spline compression filter does not output the number of static tracks, as this number may vary over time.

### 4.7.8 Animation: Previewing Compression

Compressed animations may be viewed in a variety of ways using the Preview Tool. Examples of using the Preview Tool Filter to preview compression results are described below.

#### 4.7.8.1 Previewing Compressed Animations

Compressed animations may be previewed creating a compressed animation (see the Controlling Compression section) and adding the Preview Tool to the end of the current filter stack. An example filter stack is shown below.



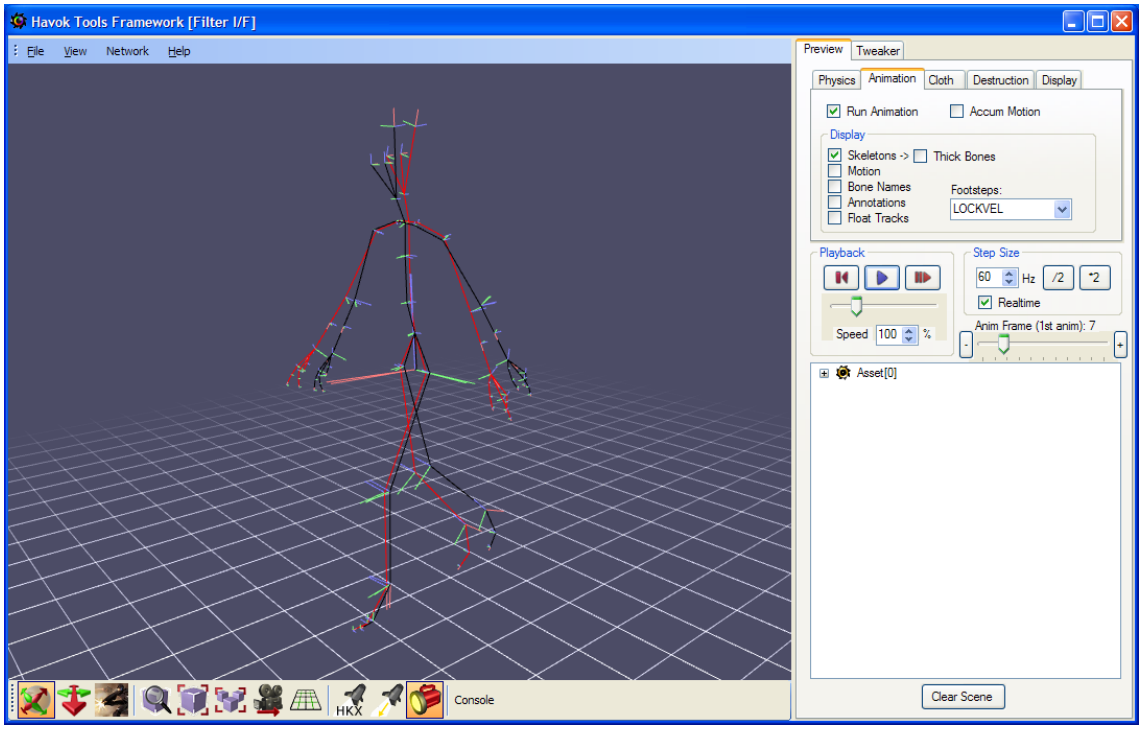

Users may choose to display either Meshes (skins) or Skeletons using the Preview Tool Menus. Users may preview additional compression settings by adjusting the appropriate compression filter settings and re-running the current filter stack configuration.

**Note:**
Preview windows do not need to be closed before re-running the filter stack. Users may keep several preview windows open to compare, side by side, the quality of different compression settings.

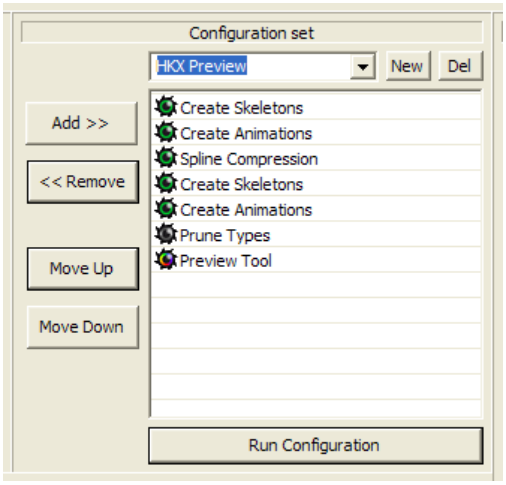
#### 4.7.8.2 Comparing Compressed and Uncompressed Animations

Compressed and uncompressed animations may be viewed simultaneously as Skeletons in the Preview Scene Filter window. This allows users to compare the quality of compressed animations versus the

original motion. The example screenshot shown below illustrates a compressed (red) and uncompressed (black) skeleton overlaid on top of one another. In this example quite lossy compression settings were used resulting in large visual differences between the two skeletons.

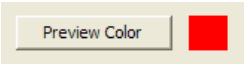



To preview compressed and uncompressed animations together, users should create a compressed animation as usual, then add an additional Create Skeletons Filter and a Create Animations Filter before the Preview Tool in the filter stack. An example filter stack illustrating this is shown below.



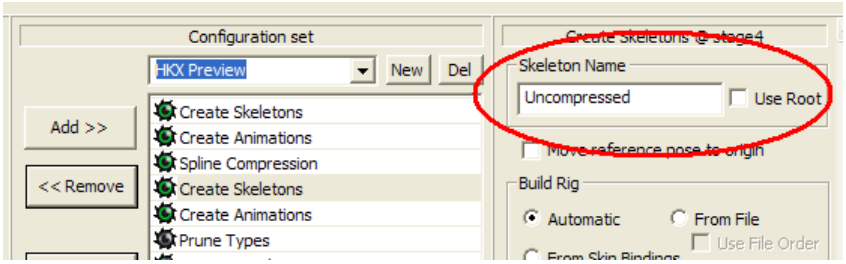

Each instance of the Create Animations Filter will add an instance of the uncompressed animation to the filter pipeline. In this example the uncompressed output of the first Create Animations Filter will be replaced by compressed output from the Spline Compression Filter; the second instance of the Create

Animations Filter will add the uncompressed animation back into the filter pipeline for display. Users can control the color of each skeleton using the Create Animations Filter. At the bottom of this filter is an option to choose a color, shown below. This color is applied to skeletons only, not to meshes (skins).




It is the user's responsibility to choose unique colors for the compressed and uncompressed animations. Note that the compression filters retain color of the animations which they compress. To view the skeletons, ensure that **Skeletons** are enabled. If skins are obstructing the view, they may need to be pruned out using the Prune Types Filter.

It is important that each skeleton in the scene have a unique name. When creating the additional skeleton, ensure it has a unique name as shown below.




**Note:**

If a preview of extracted motion is desired, an corresponding instance of the Extract Motion Filter needs to be added to the filter stack after each instance of the Create Animations Filter.

**Note:**

Users must remember to remove any added Create Skeletons Filter and Create Animations Filter instances before exporting the results to disk.

### 4.7.9 Animation: Controlling File Size

The final size of an animation asset file (written using the Platform Writer Filter) can be reduced by using the guidelines provided in Common: Controlling File Size. In addition to these, the following mistakes should be avoided in order to reduce the final size of the asset:

**Warning:**

Common mistake: Leaving the (modeler independent intermediate) scene data in any animation assets (unless you are exporting meshes).
Common mistake: Leaving meshes or skeletons in animation assets.
Common mistake: Leaving track names (not useful unless you intend to perform runtime binding) in animation assets.

### 4.7.10 Physics and Animation: Local Frames

A **local frame** is a named frame of reference that is attached to a rigid body or the bone of an animation skeleton. Local frames are useful for tagging rigid bodies or skeletal bones. For example, you may want to tag a weapon at the point it should be grabbed by a character, as shown here:

The three spheres labeled with "LF" represent local frames on their respective weapons. Each local frame has a name, such as "weapon grip", and a group. Then in the SDK you can look for the local frame named "weapon grip" and attach the weapon to the hand of a character using the position and orientation of the local frame relative to the parent rigid body. The group allows you to group together local frames using a common group name. This can be useful if you want to add clusters of local frames using the same local frame name. For example, you might want to specify where on the object the right hand and the left hand of a character can be used to pick up the object. These local frames could be named "leftHandGrab" and "rightHandGrab". But suppose that the object can be grabbed in several ways. You can use the group concept to group together each pair of "leftHandGrab" and "rightHandGrab" local frames. Then you can use the group tag to make sure to select a "leftHandGrab" and "rightHandGrab" that belong to the same group.

Each local frame is represented in the modeler as an object that should be parented to a rigid body, the bone of an animation skeleton, or another local frame. If you parent local frames to other local frames, they form a transformation hierarchy. Each local frame stores a position and orientation relative to its parent.

Local frames are exported in the Create Rigid Bodies filter (if parented to a rigid body) and the Create Skeletons filter (if parented to a skeletal bone).

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

## 4.8 The Preview Tool

### 4.8.1 Introduction

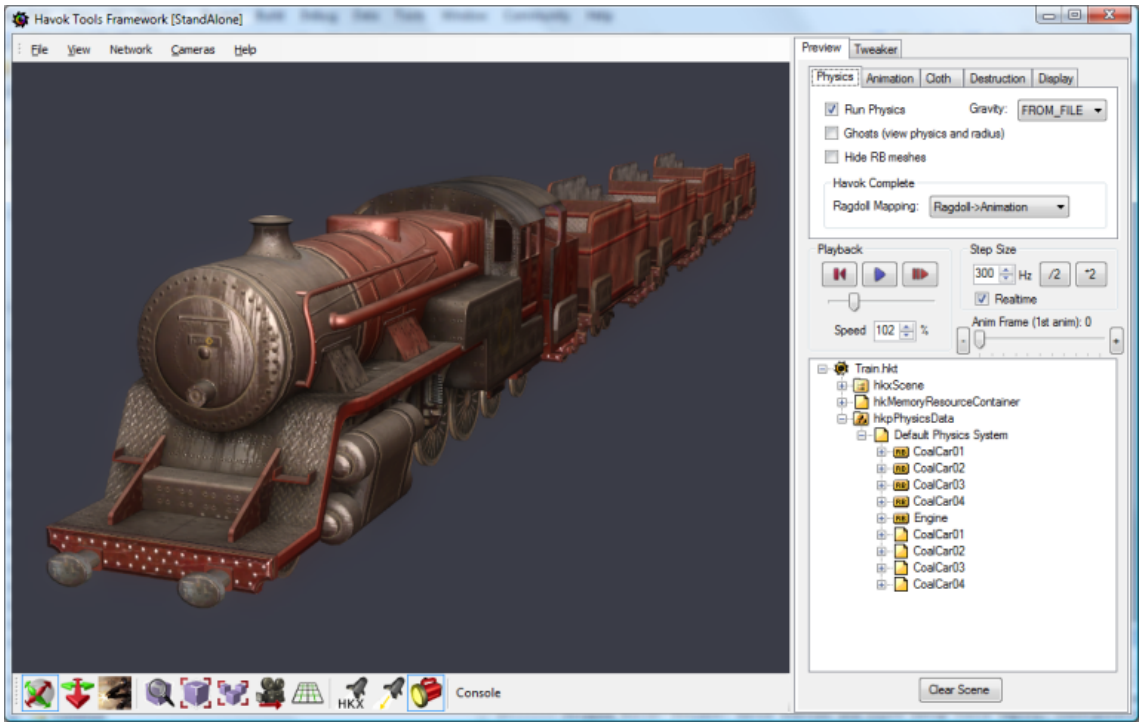



Figure 4.47: The Preview showing a train setup for a Physics 2012 simulation

The Preview Tool is a modular, plugin-based tool that supports dynamically altering the contents at runtime so that you can try different assets and configurations. It groups the contents per file loaded so if, for example, you have an animation per asset file you can enable or disable the animation by right clicking on that file in the Preview lower right tree view.

It requires .Net 2.0 and by default will look best on a graphics card that supports Shader Model 2 with DirectX 9, but will gracefully degrade to fixed function with only diffuse textures and so on.

#### 4.8.1.1 Ways to Launch the Preview

The Preview Tool itself is a ManagedC++ Control, that is installed in the Global Assembly Cache. As such it can be launched as part of any application that can make a Form to house it. We provide a standalone executable (`ToolStandAlone.exe`) and a Filter for the Havok Content Tools Filter pipeline.

**Note:**
If you want to add it to your own tools, see the `Havok.Tool.UI` Assembly that was installed on your computer, and the `Source\ContentTools\Common\Tool\StandAlone` client source code.

**Standalone Preview Tool**

The `ToolStandalone.exe` is the most straightforward way to use the Preview. It is a standalone exe-

cutable that just shows the `Havok.Tool.UI`, and has the following command line arguments:

- **-r d3d9,d3d9s,d3d10,ogl,ogls** : Override the registry settings and use this specific Renderer; **-r d3d9s** is the default DirectX 9 (' **s** ', so using only shaders).
- **filename filename .. filename** : Any argument without a ' **-** ' is assumed to be a filename to try and load. Use **"filename"** for paths with spaces. You can of course drag-and-drop in more once the Tool is running.

The standalone will check the same registry and environment variable settings as it would if run through a filter , so it should behave the same no matter how it is launched.

#### Preview Tool Filter

When you launch the filter manager, either from a modeler or from the standalone filter manager, you will see a 'Preview Tool' filter in the Graphics section. This is a filter that, just like the standalone tool, makes a window to house the `Havok.Tool.UI` and then all the rest of the operation is the same as normal. The only pre-launch options for the preview filter are the renderer choice again, through a combo dropdown in the specific filter options.

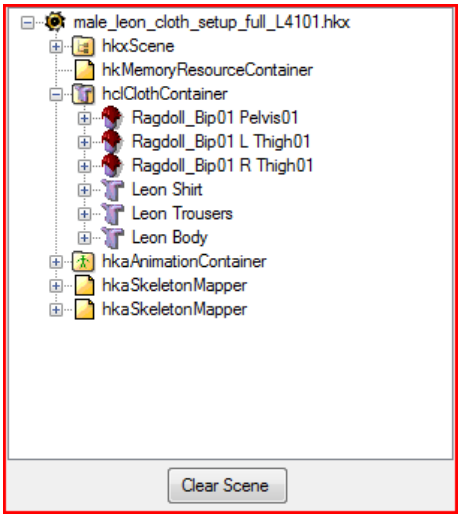
**Note:**
Multiple Preview Tool filters can be run simultaneously (multiple stages, and/or multiple runs). The only current issue with this is that they will share some of the same debug display instances and thus you may see some multiple display issues when using those functions.

#### Remote Update

With the 'Remote Update' filter, what will happen is that instead of launching a new preview thread, it will try to open a socket connection to a given address. For the Demo framework-based remote-update-capable demos, they assume port 25011. For the Preview Tool, use port 25010 by default. See the filter documentation for more details.

### 4.8.2 Assets

The Preview works on the concept that every file added to the scene is an asset that can be enabled, disabled, or removed. This will change over time to be more fine-grained (animation, rigid body, etc.) so that much more control will be offered. For now though it is simply file-based. When you add a file, or when you preview through the filter pipeline (where the current filter contents will be one asset), the tree view will appear in the lower right of the screen. The tree view lets you navigate the contents of the current scene in a similar way as the View XML filter does, but in addition to that it lets you select nodes of the scene by clicking on the corresponding `hkxNode` elements in the tree view.




To disable, enable, or remove an asset once it has been added, simply right click on the asset and select the option you want. Note for instance that if you have multiple animations enabled they will apply to all the skeletons they can, and so will blend together. Just use enable and disable to select the ones you want to test from your asset list. This assumes that multiple animations will be in multiple files for now.

### 4.8.3 Main Window Controls

The Preview Tool lets you configure various aspects of the UI and controls, by letting you set options for the camera, lights, rendering and the viewport grid. Most of the options are accessible from both the main menu and the toolbar (depicted below).

The icons, from left to right, are: Trackball camera, Fly Mode camera, First Person Mode camera (Destruction demos only), Focus selected, Zoom selected, Zoom extents all, Follow cam, grid options, Lights from file, Single light, Flash light, 1 Viewport, 2 Viewports, 4 Viewports, Error console .

If you hold down the **Space** bar while you move the mouse, the Preview will try to move the object. If Physics 2012 is not paused and the object has a physics shape (rigid or breakable body), then a type of spring will be attached that will pull the object around in the simulation. If the system is paused, or the object is neither of the above, then it will simply transform the object. Note that you can use this on physics shapes to make test scenes (add your assets, pause the sim, move the objects around to starting positions, and hit play again). For Cloth and Animation Skins for instance, once you resume simulation the display object will revert to simulation control and so the placed transform will be wiped for now. The same is true for breakable bodies that are constrained by flexible joints: they will snap back to the constrained position once you release the mouse. If you want to rotate the object instead of just translate it, hold down **Ctrl** as well as **Space** . Rotate will only work for paused or non-Physics 2012 or non-Cloth simulation items.

Below you can find a description of the controls divided by category.

#### 4.8.3.1 Camera

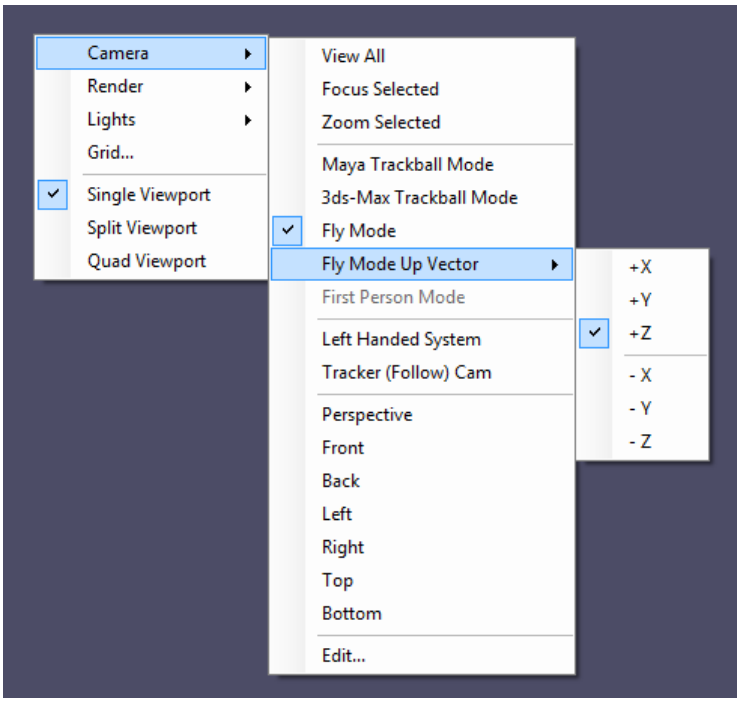
The graphics are the same as we use for demos, and thus so are the keys for interaction. The main keys are:

- **Fly Mode** (default on startup, the 'arrows' toolbar icon):
  - **Left mouse button** : Look.
  - **Middle wheel** : Dolly (so will move, in-out to the current point of interest. Speed of flymode is proportional to this distance).
  - **Alt + Left mouse button** : A more refined dolly.
  - **W , A , S , D** : Fly.
- **First Person Mode** (the 'gun' toolbar icon):
  - **Left mouse button** : Fire current gun.
  - **W , A , S , D** : Run around.
  - **Escape** : Relinquish mouse and switch back to Fly Mode. Select First Person Mode again to resume.
- **Space** : Select the object under the mouse pointer.

The up vector for Fly Mode can be set by accessing the Camera menu, and choosing an up vector appropriate for your scene. The settings for the up vector will be saved and automatically reloaded the next time the preview tool is accessed.

There are two variants of the trackball, which allow you to switch to Maya- or Max-style navigation. These options are exposed through right clicking with the mouse on the view you want to alter, and selecting the Camera menu. They will alter how the camera moves in relation to **Alt** , etc., and change button conventions. They do not enable any extra Max- or Maya-style keyboard shortcuts for camera functionality.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

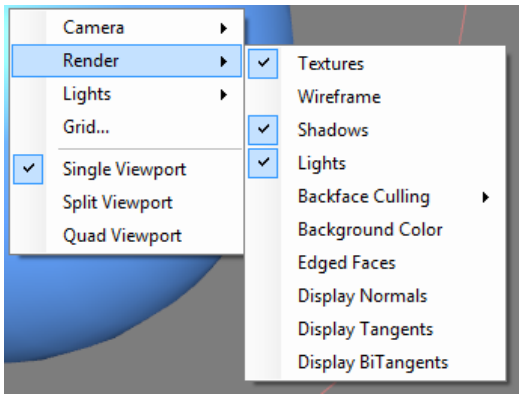



A lot of the movement speed is controlled by the distance to your point of interest (as we have no fixed scale as the scene could be any size), so setting a good starting camera can make navigating the scene easier. If you have a camera in your scene it will be used upon load, if there is none use the 'Focus on Selected', 'Zoom Selected' and 'Zoom All' buttons (in that order, left to right, in the toolbar) to get a good starting camera. If you scene contains more than one camera you can choose which one to use via the Cameras menu entry.

You can change the handedness of the display from the default Right Handed to a Left Handed system, by means of the handedness menu item. When you switch handedness, it will flip the perceived horizontal camera coordinates for the whole scene. It thus assumes that all you loaded assets are in the same space. It does not alter any coordinates, just the camera projection used to view them.

#### 4.8.3.2 Graphics

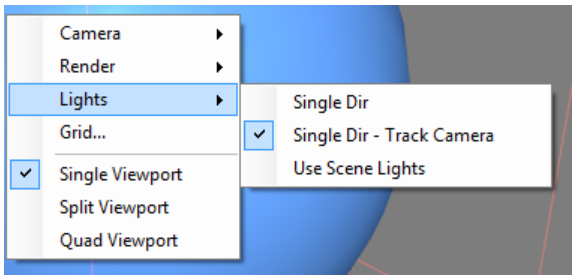
Render settings can be accessed through right clicking on the viewport you want to alter. They control the rendering parameters for the preview and let you activate or deactivate textures, wireframe, shadows, lights and edged faces rendering. They also let you set backface culling mode (none, clockwise or counter clockwise) and the color of the background.




Through this menu you can also enable the rendering of normals, tangents and bitangents. They are drawn in the viewport in blue, red, and green, respectively.

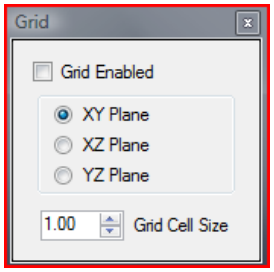
#### 4.8.3.3 Lights

If your scene has no lights, or you don't like the way they look, use the 'Single Dir' or 'Single Dir - Track Camera' light options. By default the UI will use the track camera version so it will always be like a flashlight. If you go from track to just single dir, it will use the last position of the track light, so that is a good way to place a single light for preview (track until you like the look, then revert to non tracked single to fix it in place).



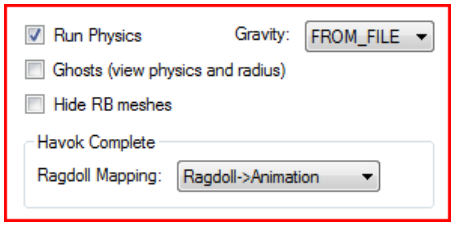

#### 4.8.3.4 Grid

By default the preview will show a grid aligned with the XY plane. You can enable or disable the grid, change its orientation and cell size using the grid dialog. To open the dialog, depicted below, click on the View > Grid menu entry.



### 4.8.4 Simulation Controls

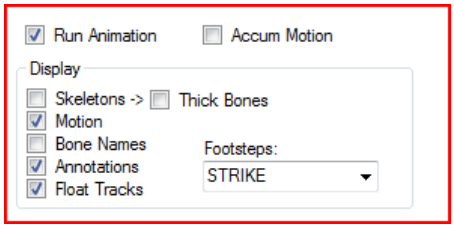
#### 4.8.4.1 Physics 2012




The Physics 2012 tab offers the following functionalities:

- **Run Physics 2012** : Toggles physics stepping
- **Gravity** : Allows you to select the desired gravity direction. It will default to -9.81 in the given direction (it assumes a value based on meters). For exact control, use a Create World filter before Preview.
- **Ghosts** : Show the underlying physics collision representation, including the collision radius.
- **Hide RB meshes** : Hide the graphics mesh associated with a Rigid Body or Breakable Body that came from the asset. If the Asset had no scene data, then the mesh should be the same as Ghost view, and so will not be hidden. If there was a graphics mesh for the RBs or BBs then they will be hidden, normally so you can see the Ghosts easier.
- **Ragdoll Mapping** : Selects how to treat any ragdolls that are found with animation mappers (Havok Complete feature).

#### 4.8.4.2 Animation




The Animation tab offers the following functionalities:

- **Run Animation** : Toggles animation stepping.
- **Accum Motion** : Toggles whether the animations will move the characters using the accumulated motion from the root bone (i.e., allow the characters to actually walk around).
- **Display Skeletons** : Toggles skeleton display. Thick Bones shows a solid geom bone representation that might be easier to see depending on your scales, etc.
- **Display Motion** : Display a line representing the future motion of the character.

- **Display Bone Names** : Displays labels beside each bone in the viewport.
- **Display Annotations** : During playback it will display a text label at the times and bones specified by annotations.
- **Display Float Tracks** : Will display float track curves.
- **Display Footsteps** : It will display results from footstep analysis.

#### 4.8.4.3 Display

The last tab is just the display options. Some of them are per object properties, so they show and change only the currently selected object, some are global and affect all objects in the scene.

Per-object properties are:

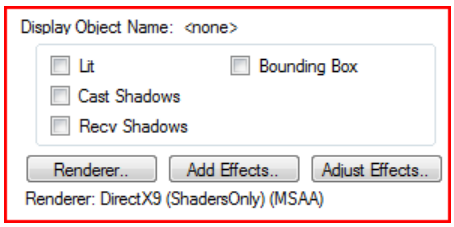
- **Lit** : Toggles lighting on or off. The effect will depend on the renderer you are using.
- **Cast Shadows** : Toggles the object's shadow caster state.
- **Recv Shadows** : Toggles the object's shadow receiver state.

  **Note:**
  Shadows must be enabled via View > Render > Shadows, which should work on any Shader Model 2 card, and have better shadows on Shader Model 3 cards.

Global properties are:

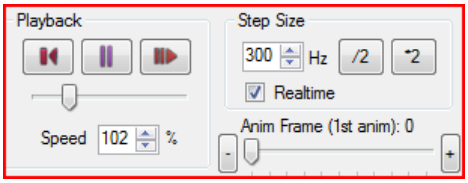
- **Bounding Box** : Toggles display of the object's bounding box.
- **Renderer...** : Lets you choose the renderer. You will need to restart the preview for the change to have effect. MSAA stands for Multisample Anti Aliasing, and is supported on most modern cards.
- **Add Effects...** : Lets you choose from a list of post rendering effects like Bloom, Screen Space Ambient Occlusion. This will only be enabled in the D3D9S and D3D10 renderers at the moment. Some post effects, such as DOF and SSAO require a second render target, which under D3D9 does not work with MSAA. Choose a non-MSAA D3D9 renderer, or use D3D10.

All objects will be marked as shadow casters unless they are large physics objects (which it will assume are just shadow receivers). For example to change your main landscape to be just a shadow receiver, select it and uncheck the Casts Shadows option. Since a single shadow map is used for the whole scene (from the first directional light), for shadows to work nicely you need to specify what in the scene is just a receiver and what is a caster. If you don't, it will try to spread the limit resolution of the shadow map perhaps too far and be too coarse.



#### 4.8.4.4 Playback and Stepping control

Playback controls allow you to rewind, pause and single-step your simulation and animation. You can also change the playback speed of the simulation/animation by dragging the speed slider or using the spinner control. This value is set to 100% by default.




Stepping controls allow you to set how the simulation is stepped. By default the simulation will be stepped at 60 Hz; you can edit this value with the step size spinner or use the **/2** or ***2** buttons to halve or double the step size.

If the time elapsed between two consecutive simulation steps is bigger than the step size, by default the simulation will try to keep up by performing a number of substeps. You can disable this behavior by unchecking the 'Realtime' checkbox. This way only one simulation step will be performed.

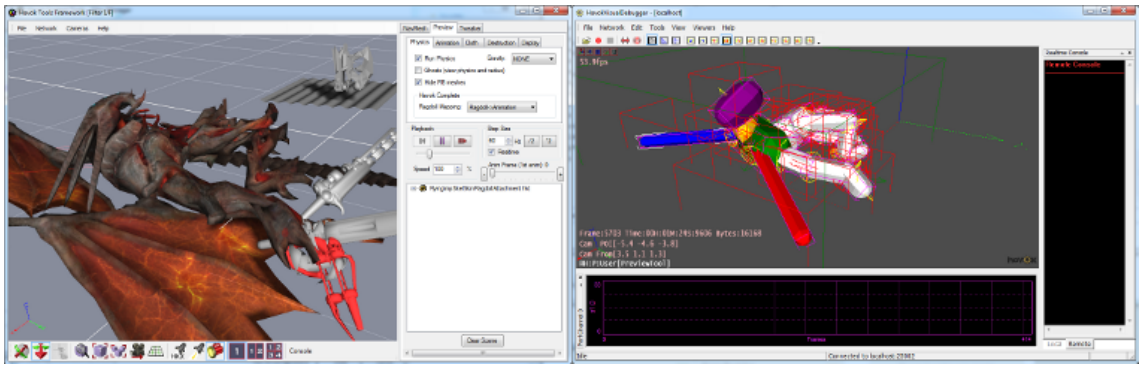
While simulating in realtime, if the simulation can not be stepped at the specified frequency a warning will be displayed in the lower left corner of the viewport, as depicted below. **This is no indication of the final runtime performance** (the preview tool has multiple overheads that wouldn't be present in a game); it's just an explanation on why the screen may refresh at a lower rate than expected.

### 4.8.5 Tweaker

All Havok serializable classes have reflection data that describes the object. As such it can be used to inspect and alter the data on-the-fly. The Tweaker tab is a simple implementation of that which will get more feature-rich in future. At the moment, it is a simple reflected edit of the currently selected object in the viewport. Not all object types currently report their class data, but expect this to grow into a generic tweaker for all object types in the future. For rigid bodies for instance, it will allow you to edit the data, and will remove and add the edited body on commit to refresh.

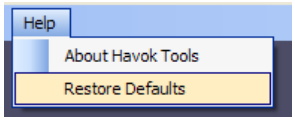
### 4.8.6 Debugging

Using the Visual Debugger it is possible to connect the the HCT Preview Window by connecting to port 25002 (this port number is displayed in the Error Console window). However, the HCT Preview Window is single-threaded and so is not representative of in-game performance.



### 4.8.7 Restoring defaults

Every time the preview is closed the current settings are saved to the registry. If there is something wrong with the simulation or the display happens after you start tweaking the settings but you can't spot which one of the settings that you changed is causing the problem, you can revert to defaults. To do so press Help > Restore Defaults in the main menu. Please note that this will restore all the settings.




## 4.9 Integrating with the Havok Content Tools

### 4.9.1 Introduction

The Havok filter pipeline provides a framework for generic asset processing. This framework consists of several components which are customizable and extensible to suit most user's requirements. This chapter provides technical details on how to extend and integrate with the various components of the pipeline.

### 4.9.2 Offline Processing

#### 4.9.2.1 .HKO Files

The filter configurations used to process an asset are always saved with their respective assets, but they can also be saved as separate `.HKO` files–check the Configurations section to see how to save and load these files. `.HKO` files are saved in XML form using Havok's serialization SDK, which means that they can be edited by hand.[6] An options file consists of a series of serialized objects (in `<hkobject>` tags) of different types. The order of these objects is important, as it controls the order of appearance of the configurations and their filters when they are displayed in the filter manager.

[6] Although .HKO files can easily be edited by hand (and autogenerated), be careful when doing so as there are limits to what the serialization framework and the filter manager can handle in terms of incorrect input.

```
<?xml version="1.0" encoding="utf-8"?>
<hkoptions>
    <hkobject class="hctConfigurationSetData">
        <hkparam name="filterManagerVersion">65537</hkparam>
        <hkparam name="activeConfiguration">0</hkparam>
    </hkobject>
    <hkobject class="hctConfigurationData">
        <hkparam name="configurationName">HKX Preview</hkparam>
        <hkparam name="numFilters">1</hkparam>
    </hkobject>
    <hkobject class="hctFilterData" name="PreviewScene">
        <hkparam name="id">2744629365</hkparam>
        <hkparam name="ver">66049</hkparam>
        <hkparam name="hasOptions">true</hkparam>
    </hkobject>
    <hkobject class="hkFilterPreviewOptions" name="PreviewScene">
        <hkparam name="renderer">RENDERER_DIRECTX9</hkparam>
        <hkparam name="hardwareSkinning">false</hkparam>
        <hkparam name="stepSimulation">true</hkparam>
        <hkparam name="passOnSimulationResult">true</hkparam>
    </hkobject>
</hkoptions>
```

The first object in the example is of type `hctConfigurationSetData`. This is a header object found once at the start of every options file. It contains the version of the Havok Filter Manager the options file was created with, and the index of the active configuration, which will be displayed in the Filter Manager when the options are loaded.

The next object, of type `hctConfigurationData`, contains the information for a specific configuration. It lists the configuration's name, and the number of filters it contains. When adding or removing filters by hand be sure to modify the `numFilters` value appropriately. `hctConfigurationData` objects are followed directly by XML describing the filters they contain, which is in turn followed by a `hctConfigurationData` object for the next configuration in the set.

Every filter is described by either one or two XML objects. The first, `hctFilterData`, is always present. It contains the filter's ID and version number, and a `hasOptions` element. If `hasOptions` is true, the `hctFilterData` is followed by a second XML object detailing the options saved with the filter. If it's false then the filter has no options, and the single `hctFilterData` object is an adequate description.

The filter options object in the example is of type `hkFilterPreviewOptions`. Each type of filter will have different options data saved against it. When modifying this information refer to the explanations of the individual filters in the Filter Pipeline section to see what the various values represent.

#### 4.9.2.2 Overriding and Upgrading Filter Settings

It is possible to automatically specify which filters to use when exporting an asset through the use of an HKO file. This can be useful in a number of situations. For example, you may want to:

- Ensure that exactly the same filter options are used whenever you export an asset.
- Change (upgrade) the filter settings of multiple assets.
- Export data using a specific filter set, but without modifying the original filter settings saved with the assets.

This is done by copying a HKO file containing the desired filter set to your Havok Content Tools installation directory ("`%ProgramFiles%\Havok\HavokContentTools`" by default) and renaming it appropriately. To create such a HKO file simply launch the Filter Manager, build up a configuration set, and select **File > Save Configuration** from the main menu.

**Overriding Filter Settings**

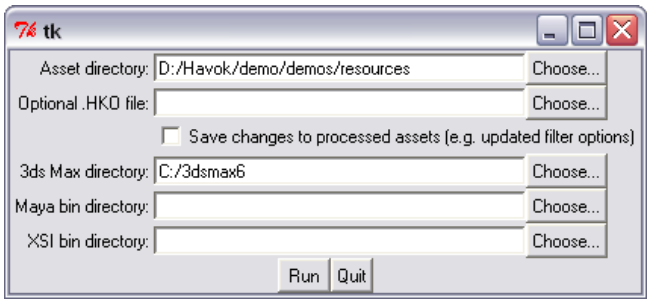
To temporarily override the filter settings put an options file called "`hkOverride.hko`" in your Havok Content Tools installation directory (same folder as "`hkfiltermanager.dll`"). If the filter manager finds such an override file when an asset is processed, it will use its filter set to process the data instead of the usual filter set. Override options can never be saved with their asset.

**Upgrading Filter Settings**

If you want to upgrade the filter settings of an asset, place an options file called "`hkUpgrade.hko`" in your Havok Content Tools installation directory (same folder as "`hkfiltermanager.dll`"). If the filter manager finds such an upgrade file when an asset is processed, it will be used to process it and will also replace whatever filter set the asset originally had (you will need to resave the scene to save those changes with the asset).

#### 4.9.2.3 Batch Processing Scripts

A Python script for batch processing assets is provided as part of the Havok SDK: this script can be found in "`tools/bin/batchprocess/processAssets.py`".[7]




This script will attempt to invoke 3ds Max, Maya and XSI in silent mode and will execute scripts on each to recursively examine all the asset files and folders inside the specified **asset directory** . Any compatible scenes will be loaded and processed in batch (silent mode) in turn.

It is also possible to specify a single .hko file containing specific filter setup options whch should be used to process each asset. If none is provided, then each asset will use whatever filter setup is present in its scene (provided there is no upgrade or override HKO set up).

If the " **Save Changes to Asset Files** " check box is selected, the asset files will be resaved after processing. This is useful you want to upgrade the filter settings of the asset.

The **installation directories** for each of 3ds Max, Maya and XSI are required so the script can find the suitable executables. If a modeler's application folder is included in the `PATH` environment variable then

[7] You will need Python 2.3 or later to execute this script. You can download Python free of charge from www.python.org

you may leave the installation directory blank (Maya usually adds itself to this environment variable on installation).

It is also possible to run this script silently by passing the above arguments via the command line. See the script source for argument usage.

#### 4.9.2.4 Command-Line Processing

The Filter Manager can be run in a non-interactive, command-line only mode. `hctStandAloneFilterManager.exe`, found in the Havok Content Tools installation directory, accepts the following command-line arguments:

| Argument | Purpose |
|---|---|
| `filenames` | A list of space-separated filenames to load into the Filter Manager |
| `-s` | Specify a configuration set file (typically .hko) |
| `-i` | Force the application to run in interactive mode |
| `-p` | Specify an asset path (no spaces allowed) |
| `-o` | Output path |

Table 4.11: Command-line arguments for the Stand Alone Filter Manager

### 4.9.3 The Havok Scene Data Format

In this section we will describe the format and structure of the information that is processed in the filter pipeline. Please note that some of the information in this section should be complemented by the *Serialization* chapters in this manual.

#### 4.9.3.1 Variants and Named Variants

Sometimes, when describing data, we need to have the ability of storing information the type of which is unknown at compile time. For example, a node in the scene may contain a mesh, a camera, a light, etc. While inheritance can help in some cases, in many cases we want to leave the door open for arbitrary types.

A Variant (`hkVariant`) object is defined as

```
struct HK_EXPORT_COMMON hkVariant
{
    //+hk.MemoryTracker(ignore=True)
    void* m_object;
    const hkClass* m_class;
};
```

That is, an arbitrary object described through an `hkClass` (check the *Serialization* documentation for details on `hkClass`). The `hkClass` object is required because otherwise there is no information available about the type or structure of the object pointed by `m_object`.

A Named Variant is nothing more than combination of a Variant and a name:

```
struct NamedVariant
{
    const char* m_name;
    const char* m_className;
    hkVariant m_variant;
};
```

**Note:**
The `NamedVariant` class contains cached information (the `className`) and thus access to its members is done through `getName()`, `getObject()` and `getClass()` methods as well as a `set()` method.

#### 4.9.3.2 Root Level Container

The information passed around through the filter pipeline is structured through the use of a Root Level Container object. An `hkRootLevelContainer` object is just a collection of Named Variants:

```
class hkRootLevelContainer
{
    [...]
        class NamedVariant* m_namedVariants;
        int m_numNamedVariants;
};
```

It also defined some methods to easily find object by name and type. Structuring the processed data in this way allows for different sets of data (scene, animation, physics) to be added by different filters to be grouped and found easily.

#### 4.9.3.3 Environment Data (hkxEnvironment)

An `hkxEnvironment` object is normally used during asset processing in order to store information which is relevant to the processing but may not be required to persist.

An `hkxEnvironment`, in a very similar fashion to an OS environment, stores a dictionary of string pairs (variable, value):

```
class hkxEnvironment
{
    public:
[...]
        hkResult setVariable (const char* name, const char* value);
        const char* getVariableValue (const char* name) const;
[...]
        hkString convertToString () const;
        hkResult interpretString(const hkString& str);
[...]

    private:
        struct Variable
        {
            const char* m_name;
            const char* m_value;
        };
        hkArray<struct Variable> m_variables;
};
```

The `hkxEnvironment` dictionary is case-insensitive when searching for a variable name; that is, a variable stored as "Output" will also be retrieved when looking for "output". Values are returned with the case they were last stored.

The class also provides methods to convert to and from a string representation. This is used by the 3ds Max, Maya and XSI export options to allow user to specify their own environment variables.

These scene exporter automatically add an `hkxEnvironment` object which contains some variables. In particular, these variables are added:

- `modeller`: The name and version of the modeler exporting the scene. For example "`3ds Max 8.0`".
- `assetPath`: The full path of the asset being exported. For example "`c:\scenes\myScene.max`".
- `assetFolder`: The folder where the asset being exported is stored. For example "`c:\scenes\`".
- `asset`: The name, without folder or extension, of the asset. For example "`myScene`".
- `selected`: The name of the first selected object, if any. For example "`sphere01`".

The exporters may also add other environment variables specified by the user. When processing an asset, the filter manager will also add another environment variable:

- `configuration`: The name of the configuration being executed. For example "`Default`"
- `infoString`: A string describing the filters executed before the current filter. Used, for example, by the Preview Tool filter (which displays that string in the viewport so the user can distinguish multiple preview windows).
- `slot`: A string with a number representing the position of the current filter in the processing.

Environment variables are useful when trying to communicate processing information between user, exporters and filters without modifying the actual content being exported. For example, the Platform Writer filter will pickup and substitute any variables used in its `filename` parameter.

#### 4.9.3.4 Scene Data (hkxScene)

When a scene is exported from a modeler like 3ds Max, Maya and XSI, there is usually one Named Variant, apart from the hkxEnvironment, in the Root Level Container: an `hkxScene` object.

As its name suggests, an `hkxScene` object provides a modeler-agnostic description of the scene (nodes, attributes, meshes, lights, etc.). The role of each of the 3ds Max, Maya and XSI Scene Exporters is to extract information from the scene in the modeler and convert it into the data stored by this object.

The `hkxScene` class and other classes used with it are defined in the `hkscenedata` project. An `hkxScene` object contains:

- General information about the asset: name of the asset and of the modeler, length (in seconds) of the animated data
- A scene graph of nodes (`hkxNode`): Nodes are described below.
- Arrays of lights, meshes, materials, textures and skin bindings: These objects are also referenced from the actual associated nodes in the scene.
- An *applied transform* matrix: This matrix is modified by the Scene Transform filter, and can be used for processing that depends on spatial data that is not explicit in the scene (and therefore is cannot transformed by the filter).

**Scene Nodes (hkxNode)**

Nodes (`hkxNode` objects) are stored in a tree structure, starting by a single root node in the `hkxScene` object. Thus, every `hkxNode` has an array (possibly empty) of child `hkxNodes`. Each node also contains:

- A name
- A pointer to the object (mesh, light, camera..) associated with the node. A Variant is used for storage since there are multiple types possible.
- An array of keyframes (4 x 4 column major matrices) representing the (animated) local transform of the node.
- An array of annotations associated with the node.
- A user properties buffer.
- A selection flag (set to "true" when the node was selected before export).
- An array of attribute groups, described below.

**Attribute Groups (hkxAttributeGroup)**

A Havok node (`hkxNode`) or material (`hkxMaterial`) can contain any number of attribute groups. Attribute groups are named groups of attributes, and as such they contain:

- A name: usually describes what the object/property its attributes describe. For example: "`hkpRigidBody`".

- An array of Havok attributes (`hkxAttribute`): described below

```
struct hkxAttributeGroup
{
    char* m_name;
    struct hkxAttribute* m_attributes;
    int m_numAttributes;
[....]
};
```

The class also defines multiple methods to easily find attributes (and their values) in the attribute group.

**Attributes (hkxAttribute)**

Since attributes can be of different types (string, ints, etc), the `hkxAttribute` class is very similar to a Named Variant:

```
struct hkxAttribute
{
    char* m_name;
    hkVariant m_value;
};
```

Although `m_value` is an `hkVariant`, and therefore it can reference any object, only certain types are expected. These are:

- `hkxAnimatedFloat`
- `hkxAnimatedMatrix`
- `hkxAnimatedQuaternion`
- `hkxAnimatedVector`
- `hkxSparselyAnimatedBool`
- `hkxSparselyAnimatedEnum`
- `hkxSparselyAnimatedInt`
- `hkxSparselyAnimatedString`

These classes store the animated values for the attribute (if the attribute is not animated, a single value is stored). The animation for floats, matrices, quaternions and vector values is stored as a sequence of regularly sampled values. For booleans, enums, integers and strings, the animation is stored sparsely as pairs `(time, value)`.

**Attribute Processing: Attribute Repository**

The way in which attributes are associated to nodes and materials during export depends on the choice of modeler (check the *Exporting Custom Data from 3ds Max*, *from Maya* and *from XSI* sections for details). Since different modelers support different sets of attribute types (for example, Enums are supported by

Maya but not 3ds Max), and ideally we want to have the same abilities in all modelers, it is useful to give the Scene Exporters some hints so that exported attributes can be transformed/complemented with additional information.

For example, since 3ds Max doesn't support enum (it implements them as simple integers), we can provide extra information for specific attributes, where we associate integer values to string representations. Then, during export, this information can be used to export the attribute as an enum.

Sometimes we also want to add extra semantics to attributes so they can be manipulated generically. For example, a float value can represent very different things: a percentage, a mass, a distance, etc., and depending on what it represents it should be processed in different ways. For example, if a float attribute represents a distance value, then it should be affected by the Scene Transform filter. But if it represents an absolute value (mass, for example), then it shouldn't.

All this extra information is stored in what is called an *attribute repository*: a set of XML files which contains sets of `hctAttributeDescription` objects, grouped in `hctAttributeGroupDescription` objects. This file is read by the scene exporters during the export process and is used to transform/complement the attributes extracted from the scene. For each attribute that may need to be transformed, it specifies:

- Whether the attribute depends on another (boolean) attribute. If the value of the other attribute is "false", then this attribute will be ignored (won't be exported)

  **Note:**
  This is still supported, but it's deprecated. From version 4.0, attributes (for example "*centerOfMass*") are automatically ignored if an integer/boolean attribute of the same name, preceded with with the word "*change*" (for example "*changeCenterOfMass*") is found and its value its set to 0/false.

- Whether the attribute should be converted (if necessary) to an specific type. If that type is an enum, then an enum definition (for the case where the attribute should be converted to enum) is also specified in order to assign string values to integer values.
- A scale value for float attributes (defaults to 1.0)–currently used to convert XSI angle attributes (stored in degrees) into radians.
- Flags specifying whether the attribute should be affected by scene transformations, and how.

The relevant files installed by the Havok Content Tools are inside the `AttributeProcessing` folder in the main installation folder. This folder contains subfolders ("`max`", "`maya`" and "`xsi`") with attribute processing information that is specific to each one modelers. Additional XML files (with their name starting with "`hct`") can also be added to any of these folders; the exporters will parse them and process attributes accordingly.[8]

**Attribute Selection: Exporting built-in properties**

To export some of the (huge variety of) built-in properties available in each modeler (for example the transparency of a particular material, the cone angle of a spotlight, and so on) one approach is to create custom attributes in the modeler with names beginning with "`hk`", and wire them to the desired property. This can be fairly cumbersome. A much more flexible alternative is available: a user-edited XML file which describes which built-in attributes should be exported. There is support for wildcarding, so that for example the entire set of properties associated with a material, light or camera say can be exported easily. In addition, it is possible to specify properties that should NOT be exported, to prevent bloating the exported scene.

[8] It also installs some files specific to the tutorial in this chapter.

During export the exporter searches for relevant XML files, inside the `attributeSelection` folder (in the main installation folder). It will also search for files inside the subfolder "`max`", "`maya`" and "`xsi`", depending on the modeler being used.

An example such file is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<hkpackfile classversion="2">

    <hksection name="__data__">

        <hkobject class="hctAttributeSelectionDatabase" name="#0000">
            <hkparam name="attributeAdditions" numelements="0">

                <!-- -->
                <!-- Note : If you add /uncomment any attribute selection, make sure you update the "
                  numelements" value above -->
                <!-- -->

                <!-- This example selects the "wirecolor" property of the "display" property set-->

                <hkobject>
                    <hkparam name="typeName">display</hkparam>
                    <hkparam name="subTypeName">*</hkparam>
                    <hkparam name="attributeNames" numelements="1">
                        <hkcstring>wirecol</hkcstring>
                    </hkparam>
                </hkobject>

                <!-- This would export all attributes (but commented out here) -->
                <!-- Note that file size will be greatly increased. -->

                <!--
                <hkobject>
                    <hkparam name="typeName">*</hkparam>
                    <hkparam name="subTypeName">*</hkparam>
                    <hkparam name="attributeNames" numelements="1">
                        <hkcstring>*</hkcstring>
                    </hkparam>
                </hkobject>
                -->

            </hkparam>

            <hkparam name="attributeRemovals" numelements="0">

                <!-- Attributes that should NOT be exported can be added here as well - the format is the
                  same as for additions -->

            </hkparam>
        </hkobject>
    </hksection>
</hkpackfile>
```

Notice that the XML file is nothing more than an XML-serialized version of an object of type `hctAttributeSelectionDatabase` (defined in `hctSceneExport/AttributeSelection/hctAttributeSelection.h`), containing two arrays (`attributeAdditions` and `attributeRemovals`) of objects of type `hctAttributeSelection`.

The attribute *additions* section specifies which attributes from the scene graph should be added to the exported scene, and the section for attribute *removals* specifies which attributes should be removed

(removals are interpreted after additions, so they can be used to complement each other). Note that the `numelements` values in the `<hkparam>` tags must be updated by hand as new entries are added.

Each one of these sections are arrays of entries of type `hctAttributeSelection`, which contain three members. These members are interpreted differently depending on the modeler being used:

| Member | 3ds Max Interpretation | Maya Interpretation | XSI Interpretation |
|---|---|---|---|
| `typeName` (string) | Type name of the Modifier, Material, Custom Attribute or Base Object. For example, "*GeoSphere*" for geospheres. | Type name of the Maya Node. For example, "*PointLight*" for point lights. | Type name of the XSI Property (parameter set). For example, "Display" for the Display parameter set, "customparamset" for any User-Defined parameter set (like Havok's). |
| `subTypeName` (string) | Name of the Parameter Block containing the parameter. This name may or not be visible in the UI. | Not used | Name of the XSI Property (parameter set) object. This is useful in order to distinguish between user-defined parameter sets (since they all have the same type name) |
| `attributeNames` (array of strings) | Internal (MAXScript) names of the parameters. | Names of the Maya Attributes. | Names of the XSI parameters. |

Table 4.12: Attribute Selection for each modeler

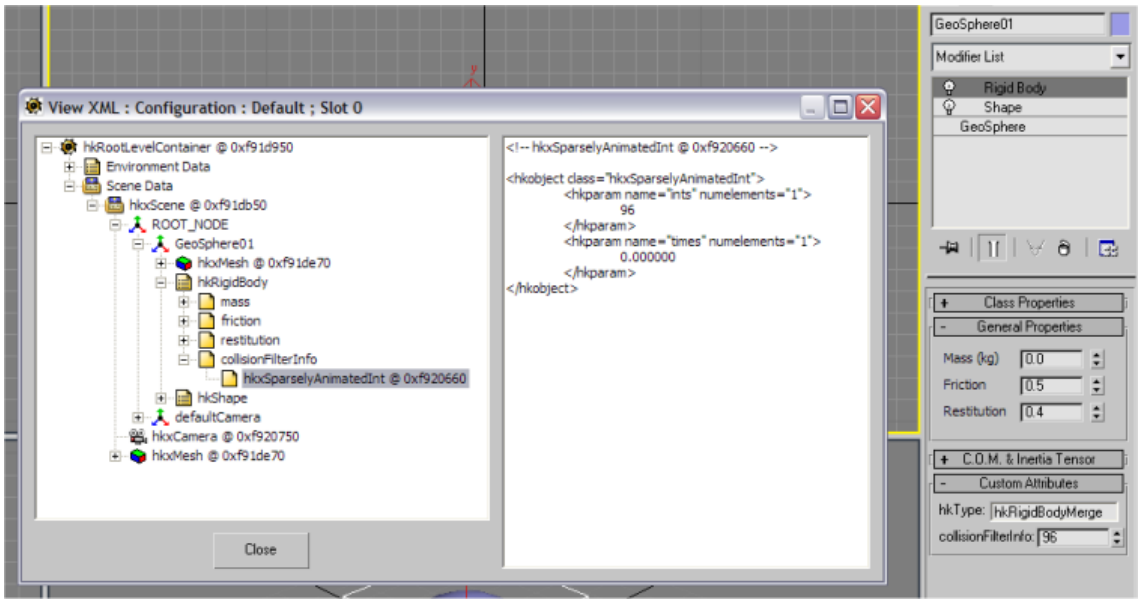
For all the string entries, an asterisk can be used as a wildcard (by itself or at the end of a name).

### More Attribute Processing

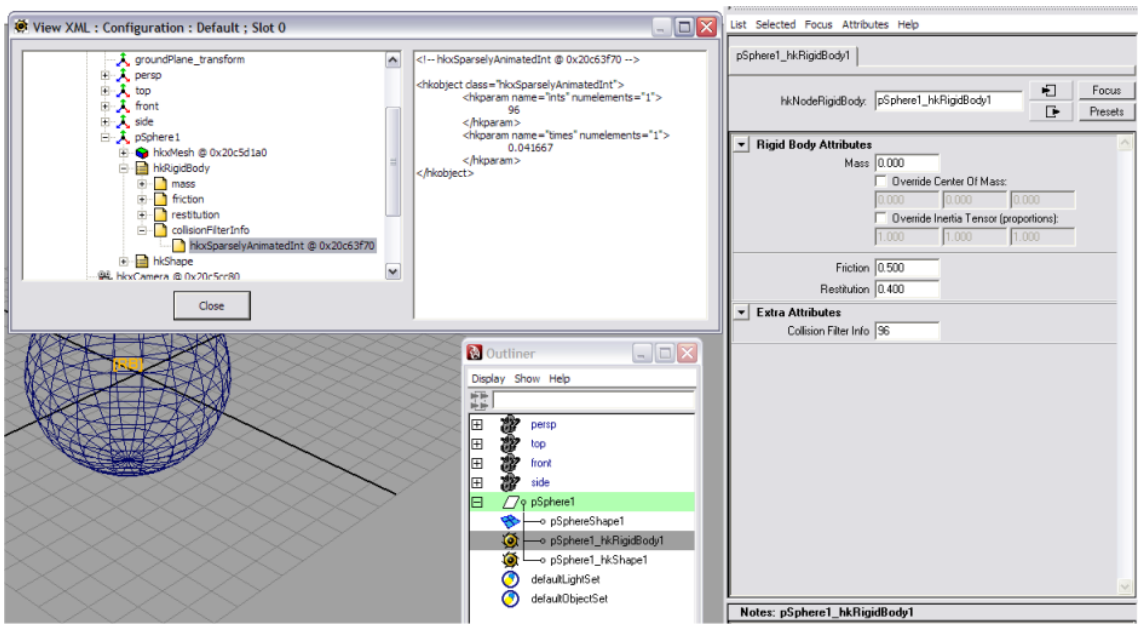
There is also more attribute processing common to all scene exporters which is not driven by XML files/`hkxAttributeDescriptor` objects. In particular:

- Attribute groups named "`hkXXXXXMerge`" will be merged into any previous attribute group of name "`hkXXXXX`" .

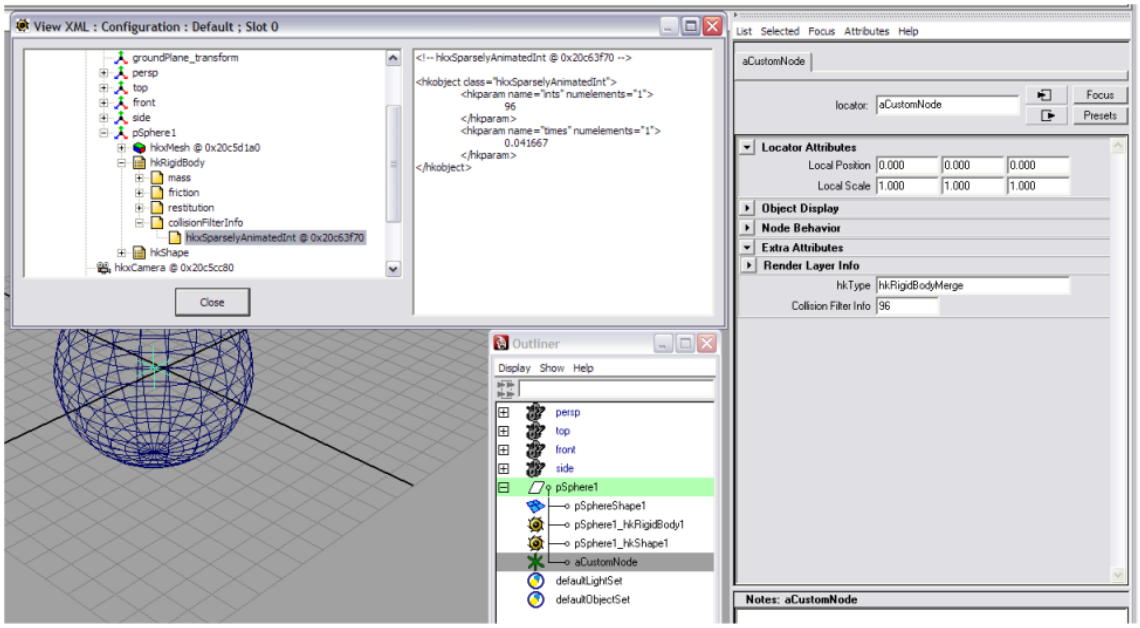
  This allows extending attribute groups already defined with more attributes. For example, in 3ds Max, you can add extra attributes to the `hkpRigidBody` attribute group of an object by adding a Custom Attribute group named `hkRigidBodyMerge` (`hkType == "hkRigidBodyMerge"`) to the `hkpRigidBody` modifier, rather than modifying the source of the `hkpRigidBody` modifier or creating a completely new modifier / custom attribute:



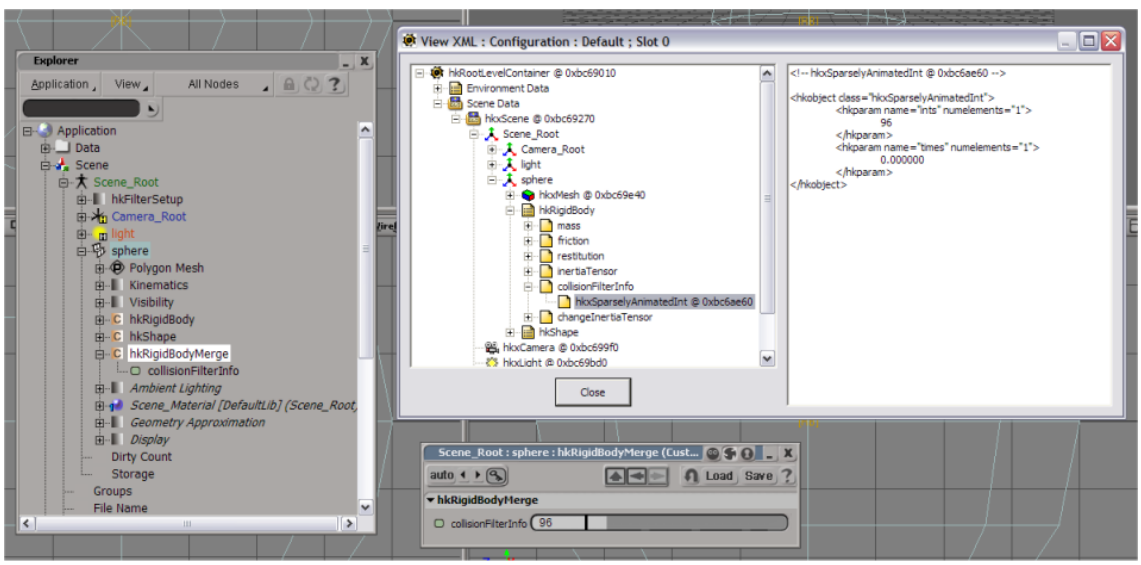

In Maya, adding user-defined attributes to a Havok node will automatically add them to the same attribute group (the exporter doesn't distinguish between user-defined and built-in attributes).




But if you add the extra attributes to a different node (which would usually create two attribute groups) you can ensure the attribute groups are merged by following the same mechanism.




In XSI you can add custom parameters to an existing parameter set; but they usually won't show in the UI since the original parameter set is not aware of them. Therefore adding a new parameter set is the common option. As with 3ds Max and Maya, you can ensure the attributes are merged into a single group by naming the second group with the "Merge" suffix:




Of course, the attribute will be ignored by the filters unless it's an attribute they support; you may want to extend or create new filters to interpret any newly exported attributes.

- Attribute pairs with names *xxxx* + *changeXxxx* (where *xxxx* can be any name, and *changeXxxx* is a boolean or integer attribute) are treated as one: if the *changeXxxx* attribute is set to false or 0, the *xxxx* attribute is ignored. The *changeXxxx* attribute is always ignored (not exported).

  This is used extensively for the physics tools in order to provide optional properties that, if not set, it will be calculated during processing (by a filter) and not in the modeler. For example, the center of mass and the inertia tensor of rigid bodies are processed in this way.

- Float attribute triplets named *xxxx_X*, *xxxx_Y* and *xxxx_Z* will be converted to a single vector attribute (named *xxxx*) with those three components. This is used in particular formodelers that don't have built-in support for vector attributes.
- Pairs of attributes with names *xxxx_Rotation* and *xxxx_Translation* (where *xxxx_Rotation* is a quaternion attribute and *xxxx_Translation* is a vector attribute) will be merged into a single attribute (named *xxxx*) of type matrix, built from those rotation and translation components. Again, this is used for platforms that don't have built-in support for matrix attributes.

Since this functionality is common to all scene exporters, it is implemented in the `hksceneexport` library. In particular, the `hctAttributeProcessingUtil` (in the `hksceneexport` library) utility class is used (check the Reference Manual for details).

#### 4.9.3.5 Physics Data (hknpPhysicsSceneData)

This type is found in the Root Level Container when processing using Physics Filters. It is automatically created by the "Create Bodies" filter. It contains bodies and constraints grouped into Physics Systems (`hknpPhysicsSystemData` objects); check the *Havok Physics User Guide* for more information about Physics data. It also (optionally) contains information about the physical world (`hknpWorldCinfo`) if the "Create World" filter was used.

#### 4.9.3.6 Physics 2012 Data (hkpPhysicsData)

This type is also found in the Root Level Container when processing using Physics 2012 Filters. It contains rigid bodies and constraints grouped into Physics 2012 Systems (`hkpPhysicsSystem` objects); check the *Havok Physics 2012 SDK User Guide* for more information about Physics 2012 data. It also (optionally) contains information about the physical world (`hkpWorldCinfo`).

```
class hkpPhysicsData
{
    [.....]
        class hkpWorldCinfo* m_worldCinfo;

        hkArray<class hkpPhysicsSystem*> m_systems;
};
```

#### 4.9.3.7 Animation Container (hkaAnimationContainer)

Another commonly used type found in the Root Level Container is an Animation Container (`hkaAnimationContainer`). This container is used by the Animation Filters to access and store objects related to animation. The class itself just contains arrays of basic animation types:

```
class hkaAnimationContainer
{
        class hkaSkeleton** m_skeletons;
        hkInt32 m_numSkeletons;

        class hkaAnimation** m_animations;
        hkInt32 m_numAnimations;

        class hkaAnimationBinding** m_bindings;
        hkInt32 m_numBindings;

        class hkaBoneAttachment** m_attachments;
        hkInt32 m_numAttachments;

        class hkaMeshBinding** m_skins;
        hkInt32 m_numSkins;
};
```

#### 4.9.3.8 Other Root-Level Data

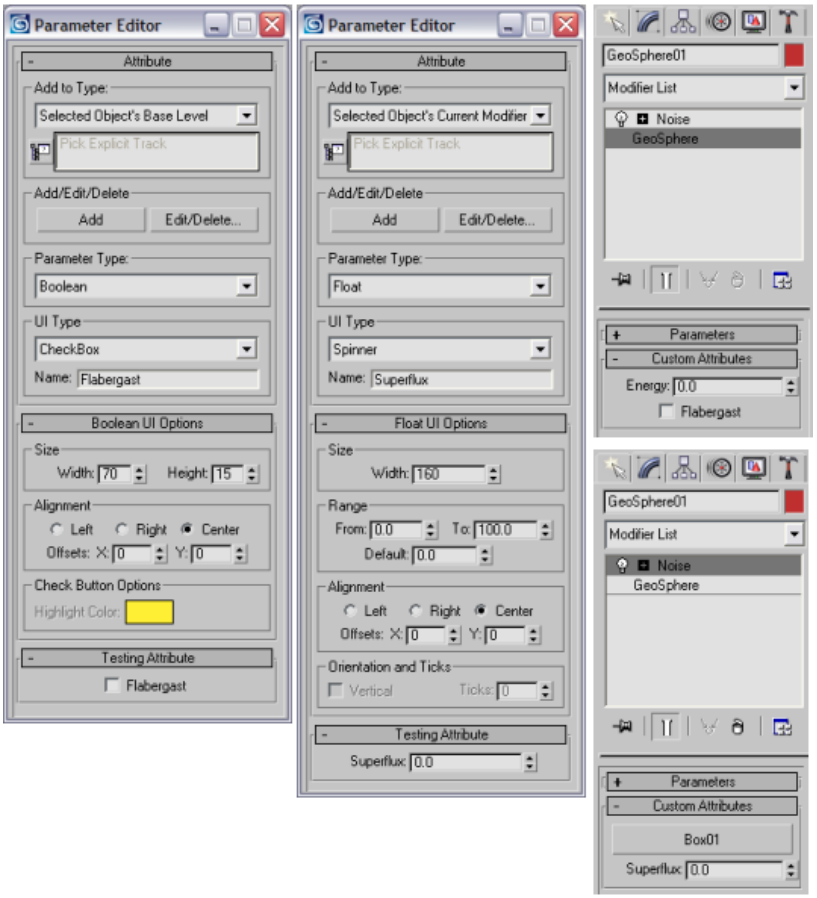
Some asset data may be stored outside scene data, animation container or physics data objects. For example, ragdoll instances created by the Create Rag Doll filter, or skeleton mappings created by the Create Mapping filter are stored using their own custom Named Variants in the Root Level Container.

### 4.9.4 Extending the 3ds Max Tools

#### 4.9.4.1 Exporting Custom Data

The best way to export custom data from 3ds Max is by exporting extra attributes associated with Havok nodes. This is how the Havok Physics 2012 Tools add extra rigid-body and constraint information to nodes (which is later interpreted by filters). There are two ways to associate attributes to nodes so they are exported:

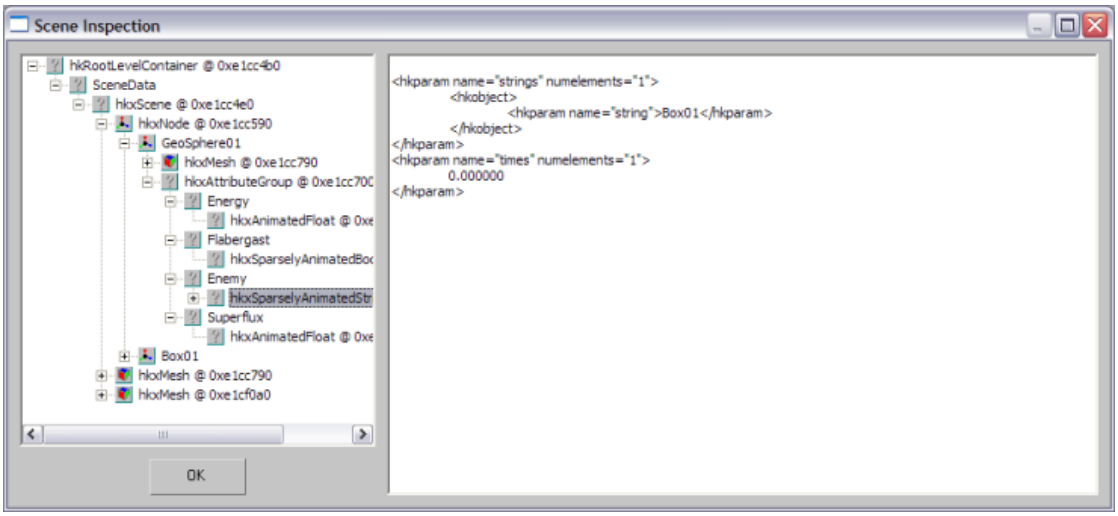
- **3ds Max Custom Attributes** : The easiest way; you can do this through the 3ds Max menus or MAXScript. Custom Attributes associated to an object or modifier will always be exported as Havok attributes associated with the node linked to the object/modifier. Custom Attributes added to a 3ds Max material will be exported as Havok attributes of the associated `hkxMaterial`.




- **Custom Modifier/Objects** : If a parameter block's name starts with "`hk`" then any parameters within that block will be exported as attributes associated with the Havok node or material. Therefore, defining and applying a custom object, material or modifier with a parameter block named "`hk...`" is another way to associate extra attributes to a node or material (this is how attributes used by the Physics 2012 Tools–within the modifiers–are exported).

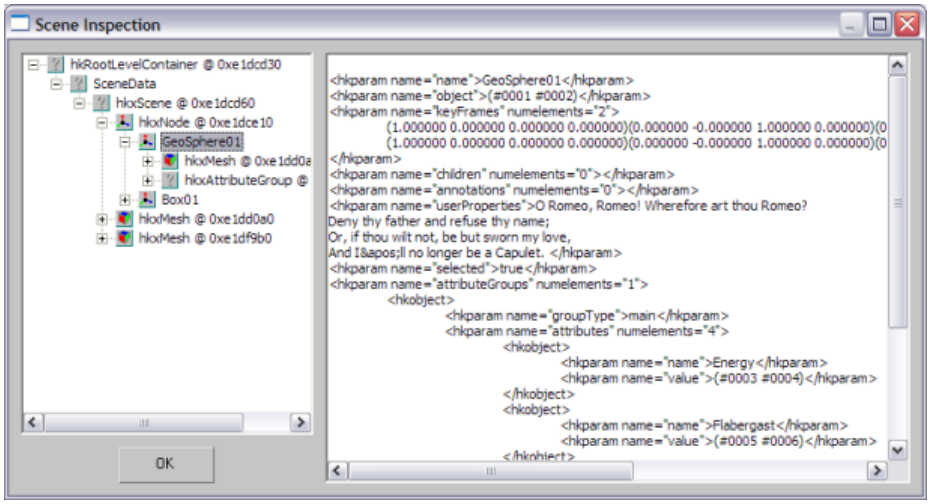
- **Attribute Selection** : You can also export built-in or custom attributes associated with nodes or materials (regardless on how they are named) by using attribute selection XML files.

In all cases, the name of the attribute group holding the exported attributes will be taken from the name of the parameter block (usually "main" for Custom Attributes created through 3ds Max UI). However if an "`hkType`" string parameter is found then its value will be used instead to name the attribute group.




### User Properties

The 3ds Max Scene Exporter also supports *User-Defined Properties*. They are exported as a single string (`m_userProperties`) associated with the Havok node (`hkxNode`).




#### 4.9.4.2 MAXScript Access

All properties and methods of the Scene Exporter and the Physics 2012 Tools for 3ds Max are exposed to MAXScript.

**Discovering Properties and Interfaces**

3ds Max uses two different mechanisms to expose objects to MAXScript: properties and interfaces.

Most non-static objects (modifiers, helpers, etc) expose their UI parameters through properties. You can query MAXScript for all the properties of an object by using the `showProperties` method:

```
ragdollModifier = \$.modifiers[#Rag_Doll]
Havok_Rag_Doll_Constraint:Rag Doll

showProperties ragdollModifier
  .childSpaceTranslation (Child_Space_Translation) : point3
  .childSpaceRotation (Child_Space_Rotation) : matrix3
  .parentSpaceTranslation (Parent_Space_Translation) : point3
  .parentSpaceRotation (Parent_Space_Rotation) : matrix3
  .constrainTo (Constrain_To) : integer
  .parent : node
  .childSpaceTranslationLocked (Child_Space_Translation_Locked) : boolean
  .childSpaceRotationLocked (Child_Space_Rotation_Locked) : boolean
  .parentSpaceTranslationLocked (Parent_Space_Translation_Locked) : boolean
  .parentSpaceRotationLocked (Parent_Space_Rotation_Locked) : boolean
  .coneAngle (Cone_Angle) : angle
  .planeAngleMin (Plane_Angle_Min) : angle
  .planeAngleMax (Plane_Angle_Max) : angle
  .twistMin (Twist_Angle_Min) : angle
  .twistMax (Twist_Angle_Max) : angle
  .maxFrictionTorque (Maximum_Friction_Torque) : float
  .motorType (Motor_Type) : integer
  .displayTwistLimits (Display_Twist_Limits) : boolean
  .displayPlaneLimits (Display_Plane_Limits) : boolean
false
```

You can easily query or change a property in an object by using the dot (.) notation:

```
ragdollModifier.twistMin
-10.0
ragdollModifier.twistMin = -30.0
-30.0
```

Interfaces are used for exposing methods and non-UI (derived) parameters. They are also used by static objects like utilities, where properties cannot be used.

You can query MAXScript for the interfaces available for an object by using the `showInterfaces` method:

```
ragdollModifier = \$.modifiers[#Rag_Doll]
Havok_Rag_Doll_Constraint:Rag Doll

showInterfaces ragdollModifier
  Interface: hkConstraintModifierInterface
   Properties:
    .childSpaceInWorld : matrix3 : Read|Write
    .parentSpaceInWorld : matrix3 : Read|Write
   Methods:
    <boolean>resetChildSpaceRotation()
    <boolean>resetChildSpaceTranslation()
    <boolean>resetParentSpaceRotation()
    <boolean>resetParentSpaceTranslation()
   Actions:
OK
```

As you can see, properties and interfaces complement each other.

### MAXScript Access to the Scene Export Utility

The Scene Export Utility is accesible through a static MAXScript object named `hkSceneExportUtility`. All the properties and methods available are exposed to MAXScript through the use of interfaces:

```
showInterfaces hkSceneExportUtility
  Interface: hkSceneExportUtilityInterface
   Properties:
    .useOptionsFile : boolean : Read|Write
    .optionsFile : string : Read|Write
    .environmentVariables : string : Read|Write
    .exportVisibleOnly : boolean : Read|Write
    .exportSelectedOnly : boolean : Read|Write
    .exportMeshes : boolean : Read|Write
    .exportMaterials : boolean : Read|Write
    .exportAttributes : boolean : Read|Write
    .exportAnnotations : boolean : Read|Write
    .exportLights : boolean : Read|Write
    .exportCameras : boolean : Read|Write
    .addDefaultCamera : boolean : Read|Write
    .animatedDataExport : enum : Read|Write
       animatedDataExport enums: {#current|#currentRange|#specificRange}
    .animationStart : time : Read|Write
    .animationEnd : time : Read|Write
   Methods:
    <boolean>exportScene <boolean>batchMode
    <string>getVersionString()
    <integer>getVersionNumber()
   Actions:
  Interface: hctSceneExportUtilityActions
   Properties:
   Methods:
    <boolean>doExport()  -- Action Interface
   Actions:
    Category: Havok_Scene_Export_Utility; Action: Export_Scene; Shortcut: -- none defined --
OK
```

The properties exposed are those described in the 3ds Max Scene Export Options section and visible in the UI, with the addition of two methods:

- `exportScene batchMode`: Exports and process the scene. If `batchMode` is TRUE, the scene is exported and processed silently (no dialogs are shown).
- `getVersionString()`: Returns a string describing the current version of the Havok Content Tools/3ds Max Scene Exporter (for example "Build 4.0.0.3 Beta")
- `getVersionNumber()`: Returns an integer value representing the current build of the Havok Content Tools. The number is built in hexadecimal, using two digits for each of the four components of the version number (major, minor, point, build). For example 4.0.0.3 = 0x04000003 = 67108867 in decimal.
- `doExport()`: Equivalent to pressing the **Export** button , or calling "`exportScene false`".

**Example**

```
-- Change some export options
if (hkSceneExportUtility.animatedDataExport != #specificRange) then
(
    hkSceneExportUtility.animatedDataExport = #specificRange
    hkSceneExportUtility.animationStart = 10f
    hkSceneExportUtility.animationEnd = 70f
)

-- Use filter and options in HKO file
hkSceneExportUtility.useOptionsFile = true
hkSceneExportUtility.optionsFile = "c:\\data\\compressionOptions.hko"

-- Export in batch mode
hkSceneExportUtility.exportScene true
```

### MAXScript Access to Other Havok Objects in 3ds Max

The following table gives the names of some of the havok objects exposed through MAXScript. Given an instance of them, you can discover their properties and/or interfaces through the methods described above.

| Name | Type | Description |
|---|---|---|
| `hkSceneExportUtility` | Static Object (Utility) | Described above. Exposes functionality and options in the 3ds Max Scene Exporter |
| `hkToolbarGUP` | Static Object (GUP) | Exposes functionality in the Havok Content Tools toolbar. |
| `hkRigidBodyModifier` | Modifier Type | Rigid Body Modifier |
| `hkShapeModifier` | Modifier Type | Shape Modifier |
| `hkBallAndSocketConstraintModifier` | Modifier Type | Ball-And-Socket Contraint Modifier |
| `hkHingeConstraintModifier` | Modifier Type | Hinge Constraint Modifier |
| `hkRagdollConstraintModifier` | Modifier Type | Ragdoll Constraint Modifier |
| `hkStiffSpringConstraintModifier` | Modifier Type | Stiff Spring Constraint Modifier |
| `hkPrismaticConstraintModifier` | Modifier Type | Prismatic Constraint Modifier |
| `hkWheelConstraintModifier` | Modifier Type | Wheel Constraint Modifier |
| `hkBoneProxyCA` | Custom Attribute Type | Used by the Rag Doll Toolbox to keep bone-proxy associations |
| `hkChainProxyCA` | Custom Attribute Type | Used by the Rag Doll Toolbox to keep chain-proxy associations |

Table 4.13: Havok Objects and Types exposed to MAXScript

### Examples

```
-- Export
hkSceneExportUtility.doExport()
true

-- Add a button to the toolbar and dock it
hkToolbarGUP.addMacro "Modifiers" "Taper"
OK
hkToolbarGUP.dock()
OK

-- Add a shape modifier to the selection
modPanel.addModToSelection (hkShapeModifier ())
OK
```

### Customizing the 3ds Max Rag Doll Toolbox

The Rag Doll Toolbox has been implemented in MAXScript. In order to facilitate customization, some of the behaviours and default options for this tool have been isolated in a single MAXScript file, `hvkTools_Customizable.ms`, which can be found in the "`3dsmax/scripts/havokPhysics/ragdollTools`" folder.

Among the customizable behaviours exposed in that file are:

- Default values for all the options (bone scale, axis, templates folder, etc.).
- Default rigid body properties (mass, restitution, friction) for proxy rigid bodies.
- Default naming methods for new proxies and joints.

#### 4.9.4.3 C++ Access

It is also possible to interface with the 3ds Max-based Havok Content Tools component through the use of C++. Full source is provided for all the tools (subject to the type of license agreement), but in order to reduce the amount of dependencies and facilitate C++ integration, full abstract interfaces to the Havok objects are exposed in a single, headers-only, project: `hkmaxfpinterfaces` (availability of this project is subject to the type of license agreement).

This project contains:

- C++ Interfaces (abstract classes) for all Havok types in 3ds Max.
- Class IDs and Function Publishing Interface IDs for all those types.
- Parameter Block IDs and Parameter IDs for IParamBlock2-based parameters.
- Other global IDs and Enums.

By using the headers in this project you can interface with all the Havok objects in 3ds Max without statically linking with any libraries. For details on how to use Function Publishing Interfaces, ClassIDs and Parameter Blocks in 3ds Max, please check the 3ds Max SDK documentation.

**Example**

```
#include <ContentTools/Max/MaxFpInterfaces/Export/hctSceneExportUtilityInterface.h>

void exportCurrentFrameInBatchMode()
{
  hctSceneExportUtilityInterface* theUtility =
    (hctSceneExportUtilityInterface*) GetInterface ( UTILITY_CLASS_ID,
                                                     HK_SCENE_EXPORT_UTILITY_CLASS_ID,
                                                     HK_SCENE_EXPORT_UTILITY_FPINTERFACE_ID);

  theUtility->iSetAnimatedDataExport (ADE_CURRENT_FRAME);
  theUtility->iExportScene (TRUE);
}
```

### 4.9.5 Extending the Maya Tools

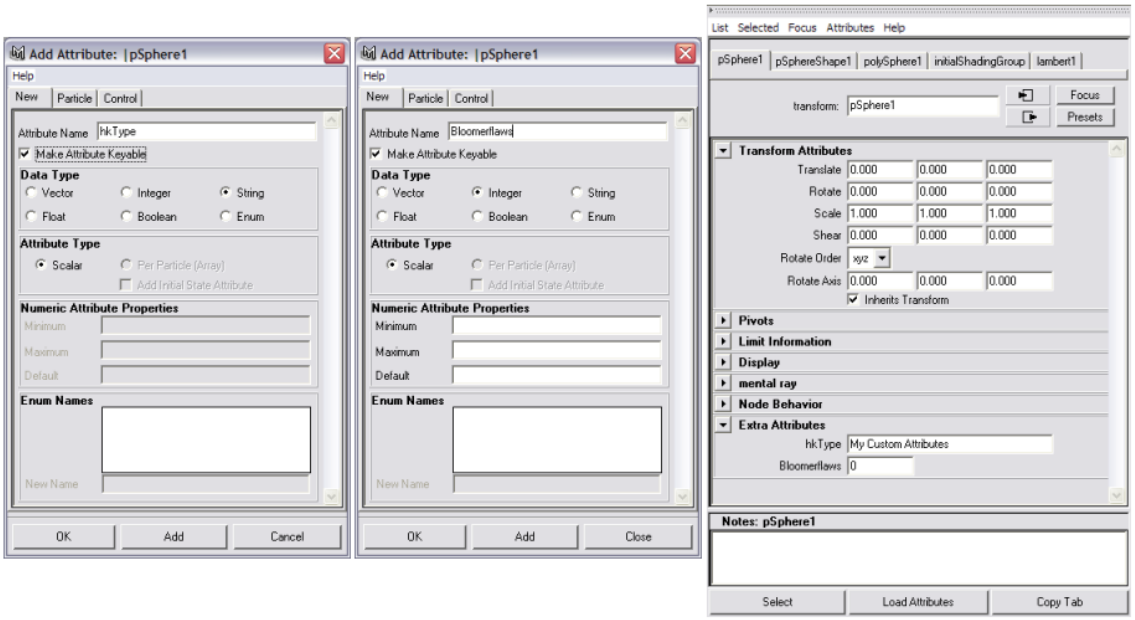
#### 4.9.5.1 Exporting Custom Data

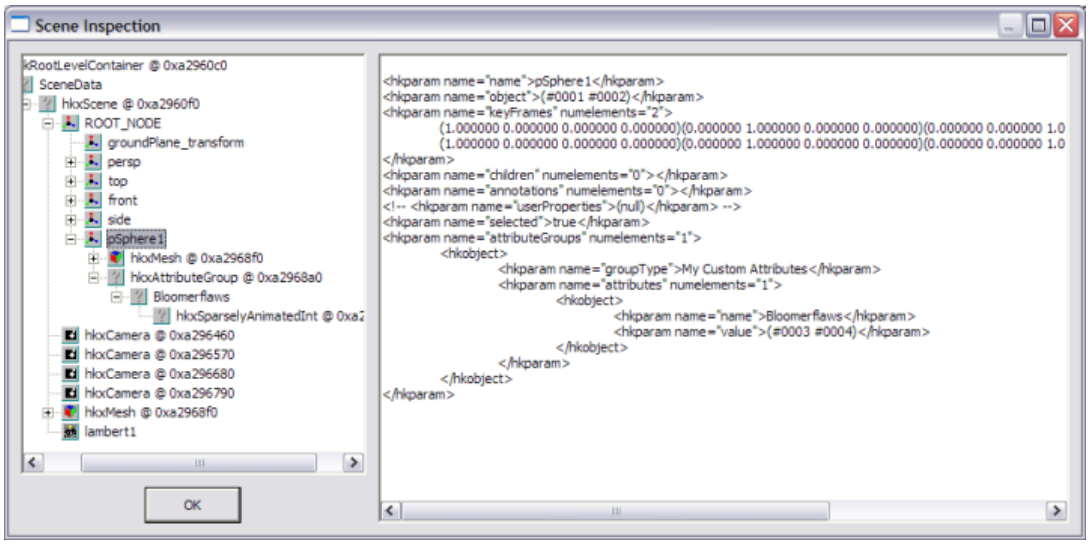
The best way to attach custom data to nodes within Maya is through the use of custom attributes. Since every Maya node has a multitude of built-in attributes as well as any additional custom attributes, exporting all of them would bloat the scene data considerably. Instead, the following method for tagging and exporting groups of attributes is used by the Havok Exporter:

1. Given a transform node which is being exported, gather a set of 'interesting' nodes containing:
   - the transform node itself
   - all of its non-transform child nodes
   - any other nodes which are connected from the transform node's `message` attribute to an attribute named `hkMessage`.
2. For each of these interesting nodes, step through all of the visible attributes looking for string attributes named `hkType*` (e.g. `hkType`, `hkTypeFoo`, `hkTypeBar`). The exporter interprets such an attribute as the beginning of an exportable group, so for every one of these it finds, a new attribute group is created in the exported HKX node. The name of the group is taken from the string value of the attribute, and each subsequent visible attribute in the current Maya node gets exported as a member of the group.

Once custom attributes have been exported as part of a HKX node in the scene data, they are then available to be used in the filter pipeline. Generally some custom filter work will be required to take that exported data and process it further, depending on what that data is intended for.

For example: any custom DAG node which contains at least an `hkType*` attribute, and is a child of a transform node, will have its subsequent attributes exported as a group within the transform's HKX node. This is exactly how the rigid body and constraint information generated by the Havok Physics 2012 Tools for Maya gets exported. The physics filters then take that data and use it to runtime physics objects.




### Attribute Selection

In addition to the above, version 4.5 introduces the ability to export explicitly selected attributes by using attribute selection XML files.

### 4.9.5.2 MEL Access

A large amount of the Maya-based component of the Havok Content Tools have been written using MEL. Those components (commands, nodes and attributes) written in C++ are automatically exposed to MEL by Maya. Therefore, using MEL is the best option for extending and integrating with the Havok Tools in Maya.

### MEL Access to the Scene Exporter

The Maya Scene Exporter can be invoked through MEL using the `hkCmdExportScene` command or the `hkProcExporter_export()` procedure:

- The **hkCmdExportScene** MEL command invokes the export process. This command can take multiple flags, which explicitly set the different export options.
- The `hkProcExporter_export()` MEL procedure does not take any parameters. It invokes the `hkCmdExportScene` command with flags reflecting the current export options in the scene. Calling this procedure is equivalent to pressing the **Export** button in Maya.

### The hkCmdExportScene command

The following table show the flags available on the **hkCmdExportScene** command. Notice how they reflect the options exposed by the export options dialog.

| Flag | Short Flag | Argument Type | Default |
|---|---|---|---|
| `-exportMeshes` | `-ms` | boolean | `true` |
| `-exportMaterials` | `-mt` | boolean | `true` |
| `-exportAttributes` | `-att` | boolean | `true` |
| `-exportAnnotations` | `-ann` | boolean | `true` |
| `-exportLights` | `-l` | boolean | `true` |
| `-exportCameras` | `-c` | boolean | `true` |
| `-visibleOnly` | `-v` | no argument | flag not present |
| `-selectedOnly` | `-s` | no argument | flag not present |
| `-startTime` | `-t1` | time (i.e., frame number) | `playbackOptions -q -minTime;` |
| `-endTime` | `-t2` | time (i.e., frame number) | `playbackOptions -q -maxTime;` |
| `-optionsFile` | `-o` | string | flag not present |
| `-useRotatePivot` | `-rp` | boolean | true |
| `-doNotSplitVertices` | `-nsv` | boolean | false |
| `-batchMode` | `-b` | no argument | flag not present |
| `-environmentVariables` | `-ev` | string | empty |

The `batchMode` flag is only available to **hkCmdExportScene** ; it is not exposed by the export options dialog. If this flag is specified, the command will export and process the scene silently, without opening any dialogs. It is most useful for batch processing assets (in conjuntion with the -o flag, for example).

```
// Export with the current options
hkProcExport_export();

// Export a specific frame range
hkCmdExportScene -t1 20 -t2 35;

// Export the current frame only
hkCmdExportScene -t1 ‘currentTime -q‘ -t2 ‘currentTime -q‘;

// Export silently using an options file
hkCmdExportScene -o "c:/data/compressionOptions.hko" -b;
```

Note that MEL ignores any invalid arguments without any warnings, so if the command is not behaving as expected make sure that the arguments are spelled the same as in the list above and do not contain any whitespace.

### The hkCmdGetVersion command

The Havok Scene Exporter for Maya also defines a command to query the current build and version of the Havok Content Tools: hctCmdGetVersion

| Flag | Short Flag | Argument Type | Default |
|---|---|---|---|
| `-string` | `-s` | no argument | flag not present |

If the `-string` (`-s`) flag is specified, the command will return a string representation of the current version and build. If the flag is not present, an integer representation will be returned. The number is constructed in hexadecimal, using two digits for each of the 4 components (major, minor, point, build) of the version number. For example, version 4.0.0.3 would be represented as 0x04000003, which in decimal is 67108867.

```
hkCmdGetVersion;
// Result: 67108867 //

hkCmdGetVersion -string;
// Result: 4.0.0.3 (4.0.0 Beta) //
```

### MEL Access to other Havok Objects

Apart from the `hkCmdExportScene` command, the main Havok objects in Maya are nodes and can therefore be fully accessed through their attributes. The following table enumerates the names of the Havok nodes in Maya:

| Name | Description |
|---|---|
| `hkNodeRigidBody` | Holds Rigid Body data |
| `hkNodeRigidBodyManip` | Manipulator for the Rigid Body Node |
| `hkNodeShape` | Holds Shape data |
| `hkNodeShapeManip` | Manipulator for the Shape Node |
| `hkNodeBallAndSocketConstraint` | Holds Ball-And-Socket Constraint Data |
| `hkNodeBallAndSocketConstraintManip` | Manipulator for the Ball-And-Socket Constraint Node |
| `hkNodeHingeConstraint` | Holds Hinge Constraint Data |
| `hkNodeHingeConstraintManip` | Manipulator for the Hinge Constraint Node |
| `hkNodeRagDollConstraint` | Holds Ragdoll Constraint Data |
| `hkNodeRagDollConstraintManip` | Manipulator for the Ragdoll Constraint |
| `hkNodeStiffSpringConstraint` | Holds Stiff Spring Constraint Data |
| `hkNodeStiffSpringConstraintManip` | Manipulator for the Stiff Spring Constraint |
| `hkNodePrismaticConstraint` | Holds Prismatic Constraint Data |
| `hkNodePrismaticConstraintManip` | Manipulator for the Prismatic Constraint |
| `hkNodeWheelConstraint` | Holds Wheel Constraint Data |
| `hkNodeWheelConstraintManip` | Manipulator for the Wheel Constraint |
| `hkNodeOptions` | Filter options are stored with the scene by using this node. |
| `hkNodeCapsule` | Capsule Mesh Node |
| `hkNodeDecomposition` | Used internally to decompose matrices into S,R,T components. |

Table 4.14: Havok Nodes in Maya

### Customizing the Maya Rag Doll Toolbox

The Maya Rag Doll Toolbox has been implemented in MEL. In order to facilitate customization, some of the behaviours and default options for this tool have been isolated in a single MEL file, `hkRagDollTools_Custom.mel`, which can be found in the `"scripts/ragdollTools"` folder.

Among the customizable behaviours exposed in that file are:

- Default values for each of the rag doll toolbox options (bone scale, templates folder, etc)

- Default rigid body properties (mass, restitution, friction) for proxy rigid bodies
- Default naming methods for new proxies and constraints.

#### 4.9.5.3 C++ Access

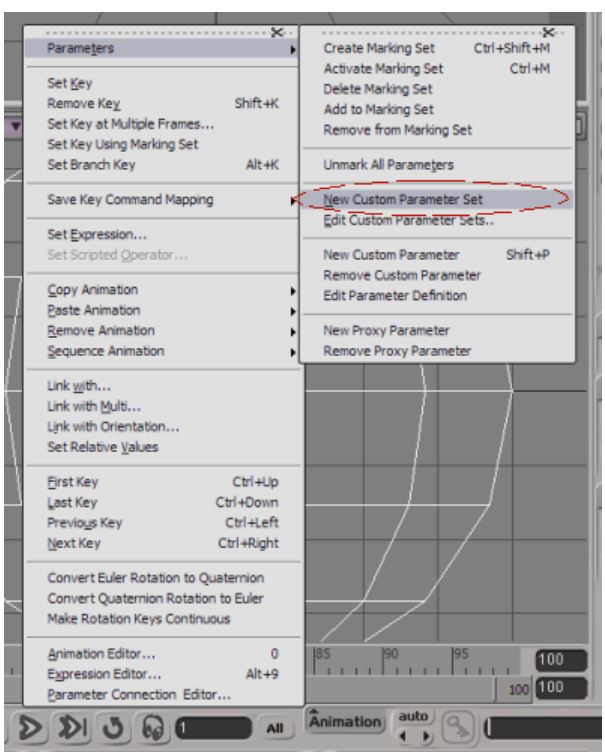
Most communication between Maya plugins is done through connections in Maya's DG, and therefore there is very little need for C++ exposure of plug-ins. For those requiring C++ access, full source is provided for the Havok Content Tools. The "`hctMayaNodeIds.h`" header file, in the "`hkmayasceneexport`" project, contains a full reference of the IDs associated with individual Havok nodes (availability of this file depends on the type of license agreement).

### 4.9.6 Extending the XSI Tools

#### 4.9.6.1 Exporting Custom Data

The best way to export custom data from XSI is by exporting extra attributes associated with Havok nodes or materials. This is how the XSI Physics 2012 Tools add extra rigid body and constraint information to nodes (which is later interpreted by filters).

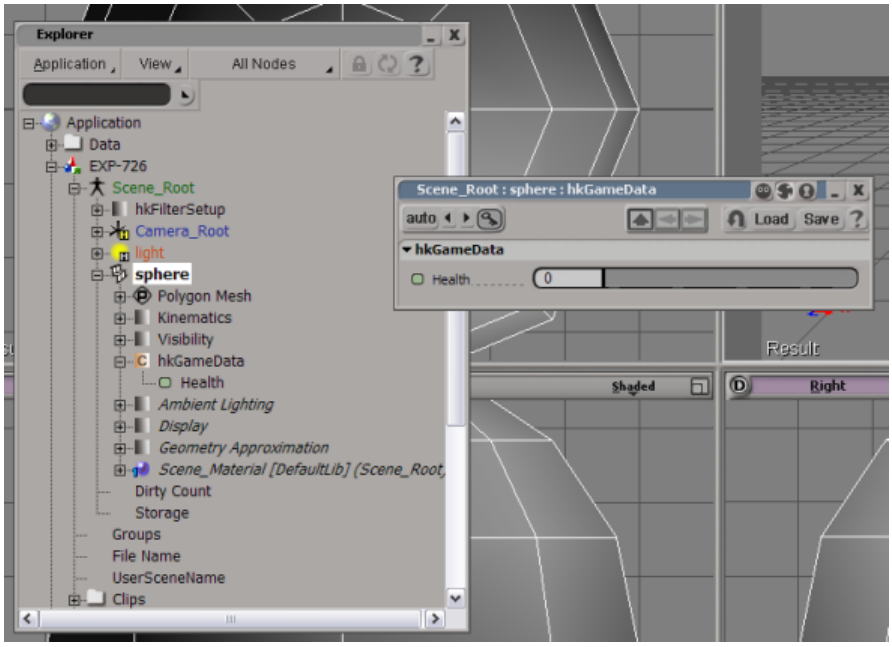
In XSI, scene nodes and materials store custom information in *Custom Parameter Sets* (also called *Properties*), which contain many individual parameters. Users and third-party tools can define and add their own custom parameters sets to store custom data. Using the XSI UI, you can easily create your own by using the **New Custom Parameter Set** and **New Custom Parameter** commands in the **Animation** menu:

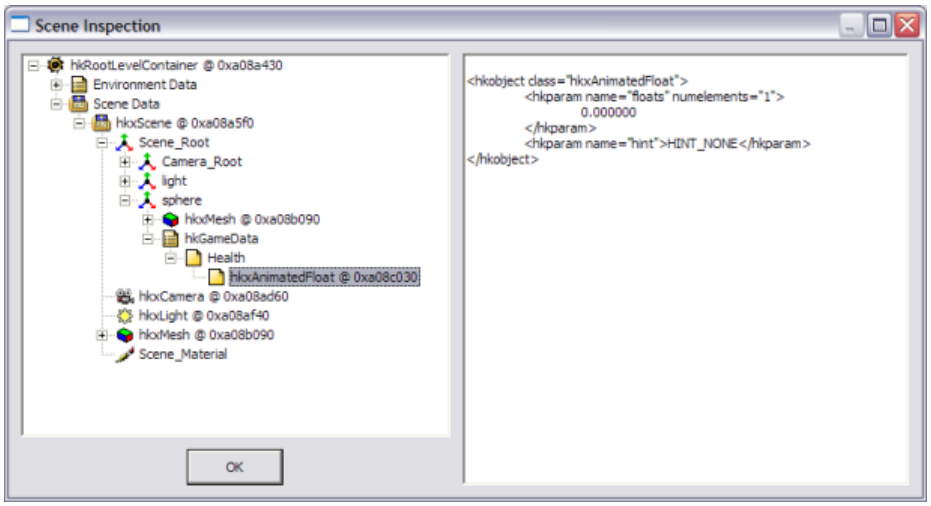


Since custom parameter sets are used extensively for many purposes, the Havok XSI Scene Exporter will export custom parameter sets that meet certain criteria:

- Either the type or the name of the custom parameter set must start with "`hk`"
- The custom parameter is not be an annotation (which are exported separately)

If the above criteria are met, the custom parameter set will be exported as an attribute group. For custom parameter sets which are registered XSI plugins, the name of the exported attribute group will be taken from the type of the parameter set. For custom parameter sets defined using XSI UI, the name is taken from the actual name of the parameter set (since its type is always "*customparamset*").




### Attribute Selection

In addition to the above, version 4.5 introduces the ability to export explicitly selected attributes (either in custom or built-in parameter sets) by using attribute selection XML files.

#### 4.9.6.2 Scripting and C++ Access

XSI doesn't have its own scripting language; its API supports multiple languages: C++, VBScript, JScript, Perlscript and Python.

The functionality implemented by the Havok Tools in XSI can be split into two sets

- Data: The majority of our data is stored in XSI using custom-defined parameter sets (also called properties) and some operators. XSI provides built-in functionality to access these objects to all supported languages.
- Actions: Actions are exposed to XSI through the use of Commands. Once a command is defined, it is immediately available through all supported languages.

**Parameter Sets**

Havok data is stored in custom parameter sets. It is very easy to access data from custom parameter sets; the easiest way is to use XSI's **GetValue** and **SetValue** commands (available in any language):

```
// JScript example
a = GetValue ("sphere.hkpRigidBody.mass");

' VBScript example
a = GetValue ("sphere.hkpRigidBody.mass");

# Python example
a = Application.GetValue ("sphere.hkpRigidBody.mass")

# PerlScript example
\$a = \$Application->GetValue("sphere.hkpRigidBody.mass");
```

In the case of C++, you can also use those two commands through the `Application::ExecuteCommand()` method, although this is a little convoluted:

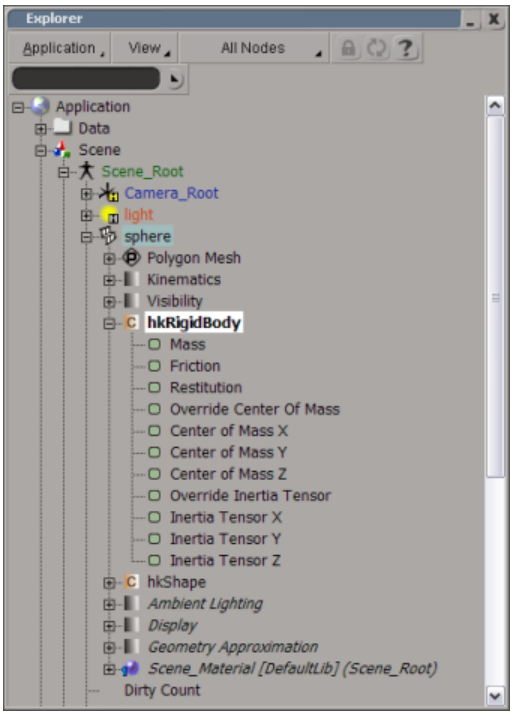
```
// C++ example using commands
CValueArray args(1);
args[0] = L"sphere.hkpRigidBody.mass";
CValue retval;
const CStatus status = Application().ExecuteCommand( L"GetValue", args, retval );
const double a = (double) retval;
```

XSI's API can also work with nodes, parameters and parameter sets as actual objects; this makes code, particularly in C++, easier to work with:

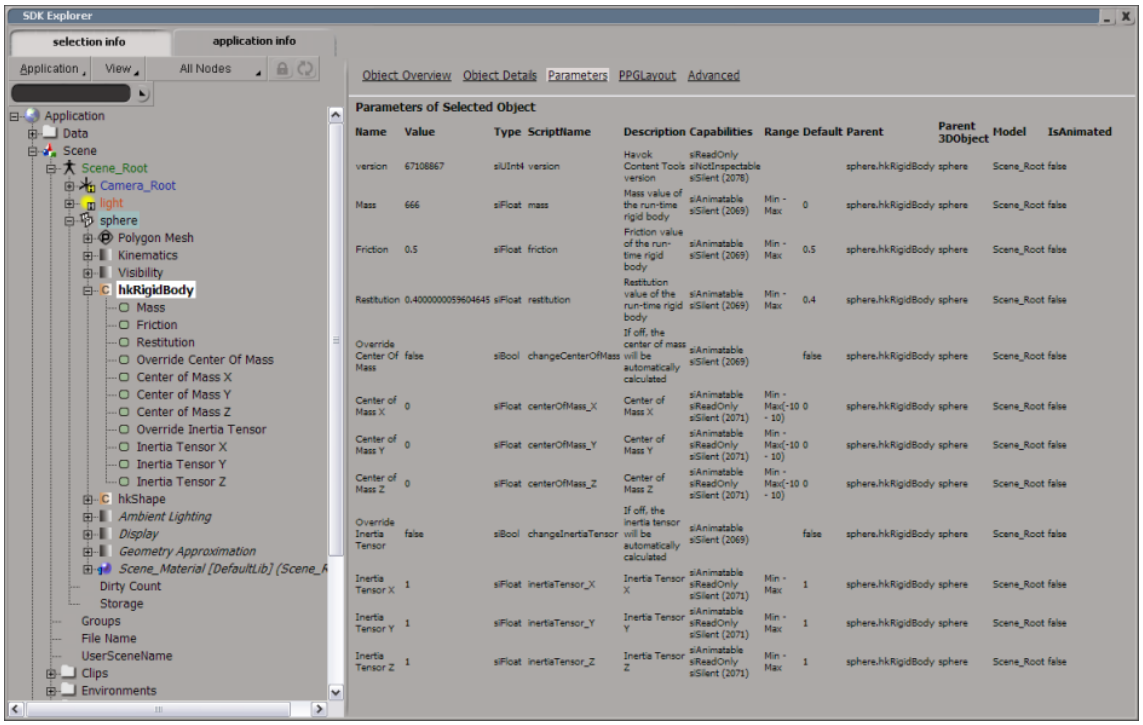
```
//
// C++ Example using API objects
//

// Access the parameter directly and query for its value
{
    CRef paramRef; paramRef.Set(L"sphere.hkRigidBody.mass");
    Parameter massParam(paramRef);
    const double a = (double) massParam.GetValue();
}

// Access the property and query the value one of its parameters
{
    CRef propRef; propRef.Set(L"sphere.hkRigidBody");
    Property rigidBodyProp(propRef);
    const double a = (double) rigidBodyProp.GetParameterValue(L"mass");
}

// Access the object, find the property, get the parameter, then query its value
{
    CRef objectRef; objectRef.Set(L"Sphere");
    X3DObject sphereObject (objectRef);

    CRef propRef;
    sphereObject.GetProperties().Find(L"hkRigidBody", propRef);

    Property rigidBodyProp (propRef);
    Parameter massParam = rigidBodyProp.GetParameter(L"mass");

    const double a = (double) massParam.GetValue();
}
```

For more information about this **CRef** , **X3DObject** , **Property** , **CustomProperty** and **Parameter** classes please check the XSI SDK Guide.

As you can see the names and type names of properties and parameters are used by XSI to find and identify them in the scene. These names are presented in the XSI UI; for example you can see them in the **XSI Scene Explorer** (if you have the " **All Nodes** " option selected):




An detailed description of properties and their parameters can be found using **XSI's SDK Explorer** view:




In the following table gives a short description of the Havok data types (custom properties and operators) used in XSI:

| Name | Description |
|---|---|
| `hkExportOptions` | Parented to the root of the scene, contains the Scene Exporter Options |
| `hkRigidBody` | Rigid body parameter set |
| `hkShape` | Shape parameter set |
| `hkBallAndSocketConstraint` | Ball-And-Socket Constraint parameter set |
| `hkHingeConstraint` | Hinge Constraint parameter set |
| `hkRagDollConstraint` | Rag Doll Constraint parameter set |
| `hkStiffSpringConstraint` | Stiff Spring Constraint parameter set |
| `hkPrismaticConstraint` | Prismatic Constraint parameter set |
| `hkWheelConstraint` | Wheel Constraint parameter set |
| `hkPhysicsPreferences` | Stored with the application, contains the Physics preferences |
| `hkRagDollToolboxOptions` | Parented to the root of the scene, contains the Rag Doll Toolbox Options |
| `hkRagDollProxy` | [Internal] Stores the association between a proxy created using the Rag Doll Toolbox and the original bone. |

Table 4.15: Havok Parameter Sets

### Commands

Plugin functionality is exposed to XSI using commands. All Havok commands in XSI are named with the prefix "`hkCmd...`" in order to distinguish them from Havok Parameter Sets.

Commands are very easy to execute through any of the languages supported by XSI. For example, the **hkCmdExportScene** command invokes scene export, using batch mode if its "*batchMode* parameter is set to true. So, in order to export the scene in batch mode we would use:

```
// JScript example
hkCmdExportScene (true);

' VBScript example
hkCmdExportScene true

# Python example
Application.hkCmdExportScene(1)

# PerlScript example
\$Application->hkCmdExportScene(1);

// C++ Example
CValueArray args(1);
args[0] = true;
CValue retval;
const CStatus status = Application().ExecuteCommand( L"hkCmdExportScene", args, retval );
```

### Export-related Commands

The following table shows the names and usage of commands related to the XSI Scene Exporter:

| Command | Description |
|---|---|
| **hkCmdExportScene** *[batchMode]* | Exports the scene. If *batchMode* is set to true, it will export in batch (silent) mode, with no UI shown. By default, *batchMode* is false. |
| **hkCmdShowExportOptions** | Opens the Export Options dialog. |
| **hkCmdGetVersionNumber** | Returns a number representing the build of the Havok Content Tools. This number is built in hexadecimal, using two digits for each component of the version. For example, build 4.0.0.3 is represented as 0x04000003, which in decimal is 67108867. |
| **hkCmdGetVersionString** | Returns a string describing the build of the Havok Content Tools. For example: "*4.0.0.3 (4.0.0 Beta)*" |
| **hkCmdShowHavokToolbar** | Ensures the XSI Havok Toolbar is shown in the UI |

Table 4.16: Havok XSI Scene Exporter commands

### Physics-related Commands

The following table shows the names and usage of commands related to the XSI Physics Tools. Many of them are equivalent to an associated menu or toolbar option. Some of them take an optional *selection* parameter; by default, this parameter is set to the list of object currently selected in XSI.

| Command | Description |
|---|---|
| **hkCmdCreateRigidBodies** *[selection]* | Creates rigid bodies from the selected 3D object(s). |
| **hkCmdCreateCompoundRigidBody** *[selection]* | Creates a compound rigid body from the selected 3D objects. |
| **hkCmdCreateRigidBodyWithProxies** *[selection]* | Creates a rigid body with proxy shapes from the selected 3D objects. |
| **hkCmdCreateCapsule** *[name] [parent]* | Creates a capsule object. If *name* is specified, the object is created with that name. If *parent* is specified, the object is created as a child of that parent. |
| **hkCmdCreateBallAndSocketConstraint** *[selection]* | Creates a ball-and-socket constraint on the selected rigid body(s). |
| **hkCmdCreateHingeConstraint** *[selection]* | Creates a hinge constraint on the selected rigid body(s). |
| **hkCmdCreateRagDollConstraint** *[selection]* | Creates a rag doll constraint on the selected rigid body(s). |
| **hkCmdCreateStiffSpringConstraint** *[selection]* | Creates a stiff spring constraint on the selected rigid body(s). |
| **hkCmdCreatePrismaticConstraint** *[selection]* | Creates a prismatic constraint on the selected rigid body(s). |
| **hkCmdCreateWheelConstraint** *[selection]* | Creates a wheel constraint on the selected rigid body(s). |
| **hkCmdShowRagDollToolBox** | Opens the Rag Doll Tool Box. |
| **hkCmdShowPhysicsPreferences** | Opens a property editor for the Physics Preferences. |
| **hkCmdSelectRigidBodies** *[selection]* | Given the current selection, or a list of objects, it focus the selection on the rigid body parameter sets associated to them.[a] |
| **hkCmdSelectShapes** *[selection]* | Given the current selection, or a list of objects, it focus the selection on the shape parameter sets associated to them. |
| **hkCmdSelectConstraints** *[selection]* | Given the current selection, or a list of objects, it focus the selection on the constraint parameter sets associated to them. |
| **hkCmdCreateConstraint** *[selection] [constraintType]* | Used internally, it creates a constraint of the given type. |
| **hkCmdCreateEmptyRigidBodies** *[selection]* | (Advanced) Adds a rigid body parameter set to the selected objects, without adding any shape parameter sets. |
| **hkCmdCreateShapes** *[selection]* | (Advanced) Adds a shape parameter set to all the selected objects, without adding any rigid body parameter sets. |
| **hkCmdRemoveHavokProperties** *[selection]* | Removes Havok-specific properties on the given objects. |
| **hkCmdManipulateConstraint** | For internal use only. |
| **hkCmdManipulateRigidBody** | For internal use only. |
| **hkCmdRagDollToolBoxEvent** | For internal use only. |

Table 4.17: Havok XSI Physics 2012 Tools commands

[a] These commands can, for example, be associated to a keyboard shortcut. That way, it is possible to quickly access and edit specific properties of a scene object. The Havok Content Tools for XSI install a Keyboard Mapping (named "*Havok XSI Key Map*") which associates **CTRL+ALT + R/C/S** to the Select Rigid Bodies/Constraints/Shapes commands.

### Other Havok Objects

Although Custom Parameter Sets and Commands make up most of the Havok functionality in XSI, there are also some other Havok objects registered with XSI. Check the XSI SDK Guide for details on how to use these types of objects.

| Name | Type | Description |
|---|---|---|
| **hkOpCapsuleObject** | Operator | The Havok Capsule Operator, used to generate a polygonal capsule mesh. |
| **hkCustomDisplayRagDollToolbox** | Custom Display | The Rag Doll Toolbox. |
| **hkDisplayCallbackPhysics** | Display Callback | [Internal]. Used to display all Havok information in the viewports. |
| **Havok_RBs** | Filter | A selection filter that will select only those objects in the scene with Rigid Body Parameter Sets. |
| **hkMenuExport** | Menu | The **Havok** menu. |

Table 4.18: Other Havok objects in XSI

### 4.9.7 Writing your own Filters

#### 4.9.7.1 Overview

In some cases, you may want to extend the Havok Content Tools export pipeline by adding you own filters. Some examples of situations where you may want to do this are:

- You want to transform some mesh data to your own format, different from the formats available in the Havok SDK. You would then create a filter that replaces Havok meshes with your own custom meshes
- You want to manipulate the information on some objects, for example appending a user-specified prefix to all bones. You can then create a new filter takes a prefix from the user, and then looks for bones in the scene, modifying their name accordingly.
- You want to add extra information to the file based on the scene; for example: based on the user properties of some nodes in the scene, create specific game objects that you can load at runtime.
- You have implemented a specific compression algorithm that works particularly well for your game and you want the exported animations to be compressed using this algorithm.

Basically, if you have algorithm that you'd rather apply to the assets before export instead of doing so after import, then writing a filter is a sensible solution. The following sections will guide you on how to do so.

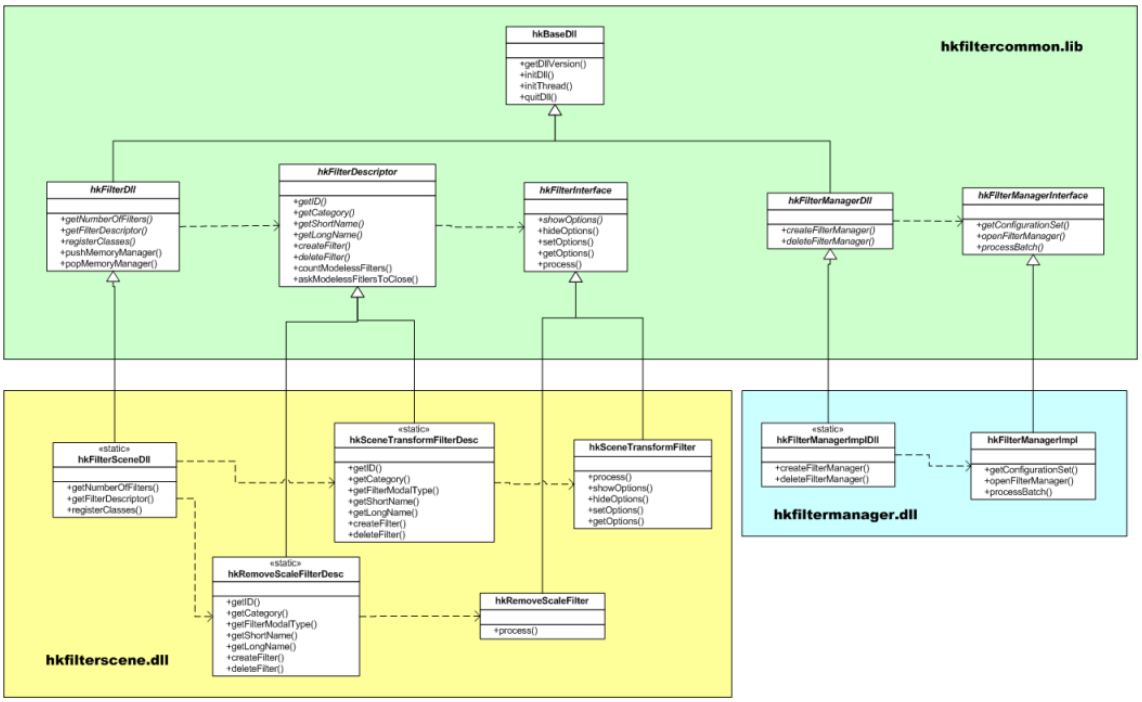



Figure 4.48: Some of the main classes involved in filter processing

The figure above shows an example of some classes involved in filter processing. The base classes are all defined in the library `hkfiltercommon` (shown in green in the picture). This library is fully documented in the Reference Manual. These classes define interfaces and default implementations for DLL interfaces, filter managers and filters. Both filters (in yellow) and the filter manager implementation (in blue) use these classes.

In the following sections we'll be discussing these classes and interfaces.

#### 4.9.7.2 Handling DLLs: hctBaseDll, hctFilterDll and hctFilterManagerDll

The filter toolchain is based on DLLs: the Filter Manager DLL is responsible for loading all of the other DLLs (the filters) and executing them when the user requests so. The Filter Manager DLL itself gets loaded by the actual application (3ds Max, Maya and XSI scene exporters, or the standalone filter manager).

Having the filter manager and filters as DLLs has several advantages: adding a new filter to the toolchain is as easy as building a new DLL and dropping it into the correct folder (more about this next). Also, the same toolchain is used by all modelers/applications without the need to build specialized tools for each modeler/application.

If you have worked with DLLs before you'll be aware that there are some issues that need to handled carefully:

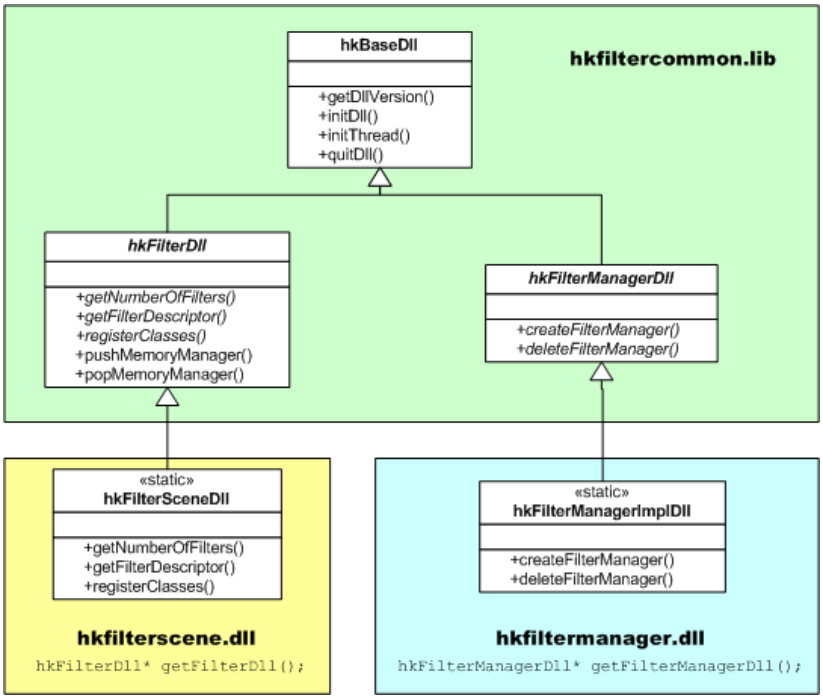
- Versioning: It is important to ensure that the DLL that is loaded matches the version expected by the loading process; otherwise unexpected errors may happen, as memory corruption and crashes.
- Memory management: As each DLL has its own copy static data, any class or algorithm that relies

on static data or singletons need to ensure all DLLs are synchronized and use the same singletons. Examples of static data are error handlers, memory managers, etc.

- Interface: Extending/modifying functionality in DLLs, if not done carefully, may convert the DLL interface into a bit of a mess, with a mixture of many, old and new, exported functions growing over time.

In order to simplify and reuse the logic to handle the above, the `hctBaseDll` class is used. This class defines functionality required for versioning and memory management, and groups all required methods into a single interface (so rather than exporting many functions, DLLs export a single function returning an instance of this class).

Two types of DLLs are used by the Havok filter processing: the filter manager DLL and multiple filter DLLs. Associated with each, there are the classes `hctFilterDll` and `hctFilterManagerDll`:




Each of these two classes extend the interface in `hctBaseDll` with methods required by DLLs implementing filters and the filter manager:

- `hctFilterDll`: Adds interface methods to access a virtual array of filter descriptors (`hctFilterDescriptor` objects; we'll talk about these later). This class also must register all classes created by any of the filters it implements. Finally, it defines (and provides default implementations) methods to handle temporary replacement of memory manager.
- `hctFilterManagerDll`: Adds interface methods in order to create and destroy a filter manager (`hkFilterManager` object).

Each DLL contains a single, static instance of a subclass of either `hctFilterDll` (for filter DLLs) or `hctFilterManagerDll` (for the filter manager DLL). This instance must be returned by the only exported function in the DLL, **getFilterDll()** or **getFilterManagerDll()** (depending on the type of

DLL). In the example shown, these static objects are instances of the classes `hctFilterSceneDll` (for `hkfilterscene.dll`) and `hctFilterManagerImplDll` (for `hkfiltermanager.dll`).

### Placing DLLs

Since the filter manager DLL is usually loaded by the scene exporters (or the standalone filter manager) and the filter DLLs are automatically loaded by the filter manager itself, you don't usually need to worry about loading DLLs; you just need to place your filter DLL in the right place.

The filter manager and the filter DLLs are kept together in a particular file structure so they can be found by the scene exporters. This file structure is automatically created in the specified installation directory by the Havok Content Tools installer. However, if you are developing your own filters, extending the filter manager, or would simply like separate installations of the filter manager for each of your modelers then you'll need to know the following:

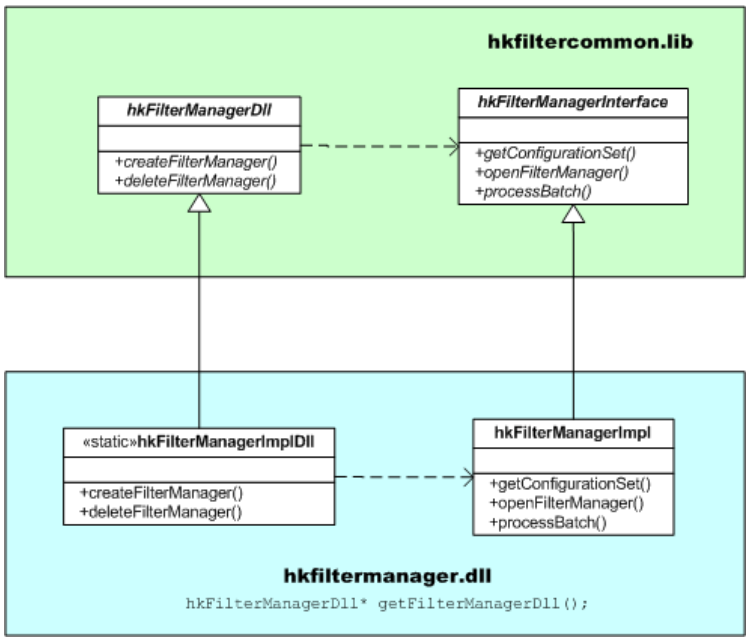
- The filter manager looks for filter DLLs in a "`.\filters`" subdirectory, relative to itself. For example, if the filter manager DLL is at "`c:\havokfilters\hkfiltermanager.dll`", it will look for filter DLLs in "`c:\havokfilters\filters\`". Any new or modified filter DLLs should be kept here.
- The installer also creates the registry entry "`HKEY_CURRENT_USER\Software\Havok\hkFilters\`", which stores (among other things) the location(s) of the filter manager. You'll need to modify the values here if you have multiple instances of the filter manager.

This is how the various applications look for the filter manager DLL:

| | |
|---|---|
| Standalone Filter Manager | The standalone filter manager expects to find the filter manager DLL in the same directory as the `hkStandaloneFilterManager.exe` executable. If it does not find the filter manager there, it checks the **FilterPath** registry value, which points to the filter manager's default location. |
| 3ds Max Scene Exporter | The 3ds Max scene exporter first looks for a "`havok`" folder inside the installation directory (for example, "`c:\3dsmax8\havok`"). If the filter manager dll is not found, it then checks for the `MaxPath` registry value containing the filter manager location. If it does not find the filter manager there, `FilterPath`, the default location, is used. If you have a separate installation of the filter manager for 3ds Max simply enter its location in the `MaxPath` registry value and the 3ds Max scene exporter will find it. |
| XSI Scene Exporter | The XSI scene exporter first looks for a "`havok`" folder inside the installation directory (for example, "`c:\softimage\xsi5.0\havok`"). If the filter manager dll is not found, it then checks for an `XsiPath` registry value containing the filter manager location. If it does not find the filter manager there, `FilterPath`, the default location, is used. If you have a separate installation of the filter manager for XSI simply enter its location in the `MaxPath` registry value and the XSI scene exporter will find it. |
| Maya Scene Exporter | The Maya scene exporter first looks for a "`havok`" folder inside the location of the havok module (for example, if the maya exporter is located in "`c:\mayaTools\plugins\`", then it will look for the filter manager DLL in a folder called "`c:\mayaTools\havok\`"). If not found, the Maya exporter will look in the folder specified by the `MayaPath` registry value. If it does not find the filter manager there, `FilterPath`, the default location, is used. If you have a separate installation of the filter manager for Maya simply enter its location in the `MayaPath` registry value and the Maya exporter will find it. |

Finally, there is an environment variable, `HAVOK_TOOLS_ROOT`, which you can set up to point to the filter manager and that overrides any registry settings . Note that this variable isn't created by the installer, so is not used by default for an installation. If you set it, it will use that path first before any registry settings. Clients typically use this one a per Process basis to pick a different Havok export pipeline version.

#### 4.9.7.3 The Filter Manager: hctFilterManagerInterface




Both filters and exporters can access the filter manager through an `hctFilterManagerInterface` object.

There is only one implementation of this class (`hctFilterManagerImpl`), but all external access is done through the interface class. We use an interface class not because we expect multiple implementations of it, but rather because we want to decouple the methods from their implementation, so methods are implemented in the DLL rather than statically linked to the caller (filter or exporter).

You can check the Reference Manual for details on the many methods defined in this interface.

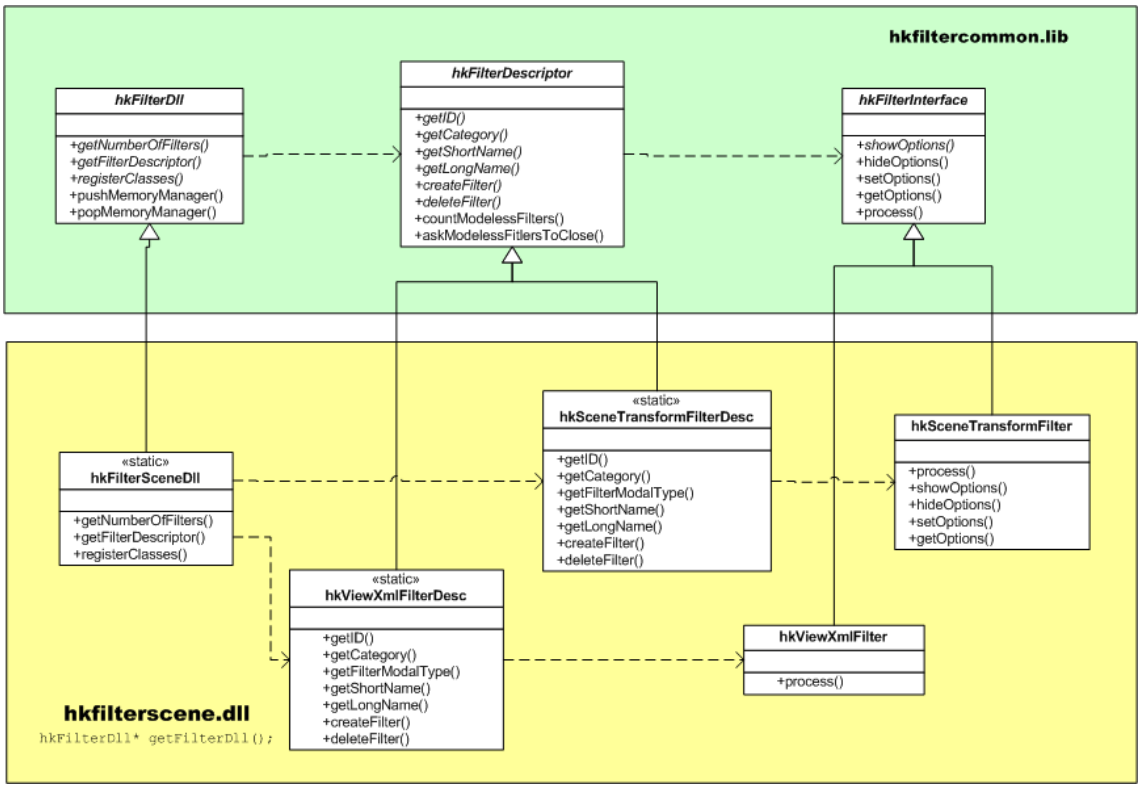
The `hkfiltermanager`(`.dll`) project provides the implementation class.

**The hksceneexport library**

In order to isolate some of the common, reusable functionality shared by the scene exporters and the standalone filter manager (loading the filters, triggering the processing, etc.), this functionality has been put in a library, `hksceneexport.lib`. This library is also fully documented in the Reference Manual.

#### 4.9.7.4 Filters: hctFilterDescriptor, hctFilterInterface

Multiple filters are described and implemented in a single DLL. In this example we show two of the many filters implemented in `hkfilterscene`:




### hctFilterDll

As we saw earlier, the `hctFilterDll` object exposed by each filter DLL provides access to a virtual array of `hctFilterDescriptor` objects through the **hctFilterDll::getNumberOfFilters()** and **hctFilterDll::getFilterDescriptor()** interfaces. Filters also register all classes they create through the **hctFilterDll::registerClasses()** interface.

### hctFilterDescriptor

This interface class defines methods to create, destroy and describe (name, category, id...) filters. These descriptors are used by the filter manager in order to present the list of available filters to the user without the need of creating them. Therefore, a single static instance is used for each filter available, returned through the **hctFilterDll::getFilterDescriptor()** method.

### hctFilterInterface

This is the interface class to the actual filters that process the content. Each filter type, therefore, must override this class and implement methods to:

- Retrieve/Set option data (if any)
- Open/Close its options UI (if any)
- Process the contents

The **hctFilterInterface::process()** method is therefore the most important method for each filter.

#### 4.9.7.5 Modeless Filters: hctModelessFilterDll, hctModelessFilterDescriptor, hctModelessFilter

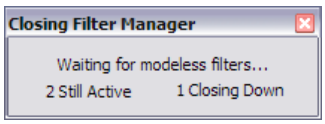
While most filters modify the contents they process (and therefore the next filter must wait for the filter to finish processing before continuing), some filters (like the View XML Filter and the Preview Tool Filter) make a copy of it and allow processing to continue while the filter remains active. These are what we call *modeless filters*.

Creating modeless filters has some extra implications:

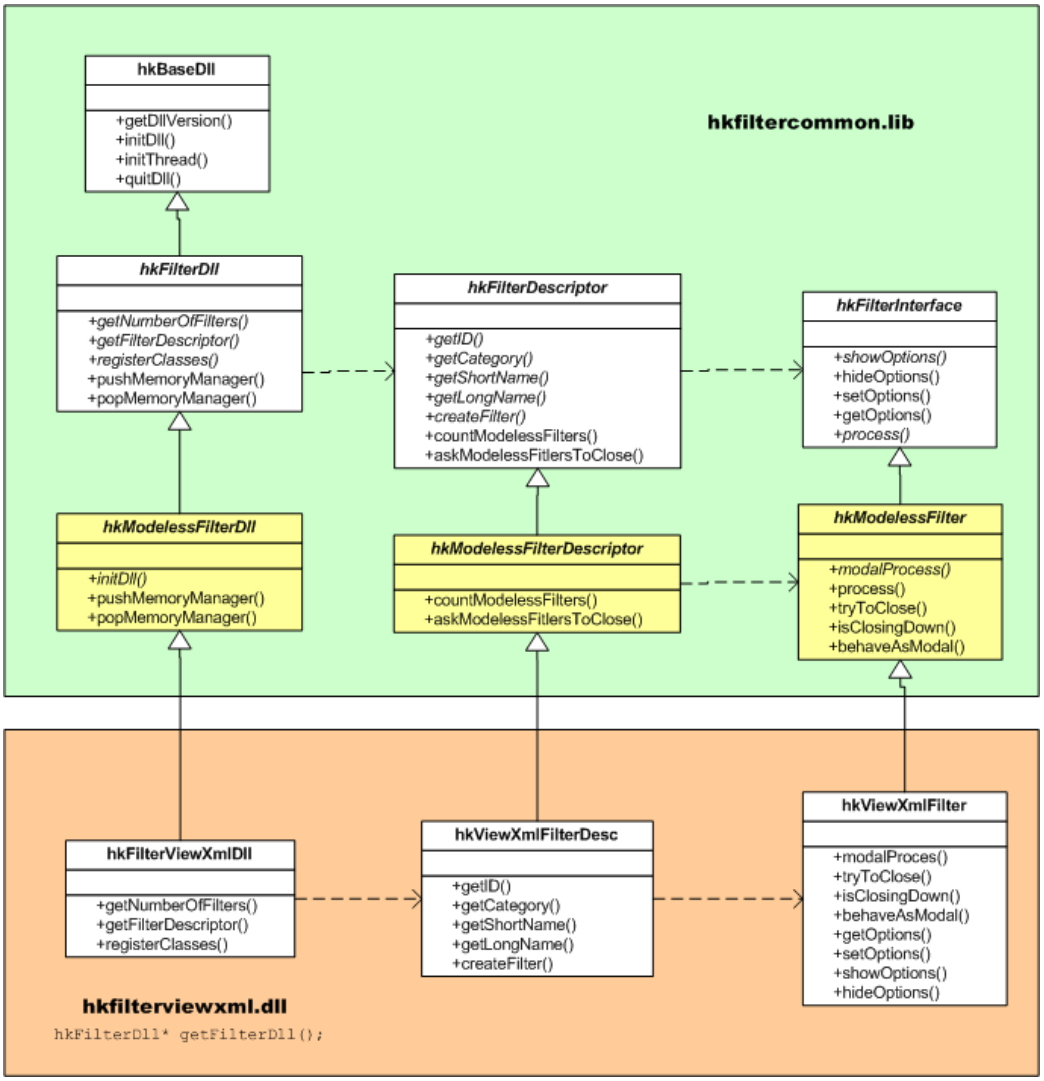
- In order to keep the filter active while processing continues, a new thread has to be created for the new active filter. This thread (and its associated memory, etc) needs to be properly created, managed and destroyed.
- A private copy (clone) of the contents has to be created for the filter to use (since the "main" contents will be modified by other filters).
- The memory manager of a DLL containing modeless filters cannot be replaced before and after processing since the memory may still be used by an active filter. So memory initialization and cleanup can only be done on DLL load and unload (for modal filters it is done before and after each call to `hctFilterInterface::process()`).

  **Important:**
  This means that *modeless and modal filters cannot be mixed in the same DLL* (since they manipulate their memory manager in different ways).

- We need to keep track of active instances of each modeless filter, and since each one runs in a different thread, we need to ensure we avoid race conditions and deadlocks.
- When the user closes the filter manager, active instances of modeless filters should "try to close" and report their progress on closing down (they may not be able to close, or they may require some time to do so).




In order to simplify the development of modeless filters, and since the functionality required to implement the above is independent of the actual processing done by each filter, specialized versions of `hctFilterDll`, `hctFilterDescriptor` and `hctFilterInterface` are provided (highlighted in yellow in the diagram below):




- `hctModelessFilterDll`

  It overrides the methods `pushMemoryManager()` and `popMemoryManager()` from `hctFilterDll` with empty implementation. For modeless filters it is no longer possible to replace the memory manager on the fly (another thread may be using it). It also overrides `initDll()` from `hctBaseDll` since, due again to memory managent, error messages are handled differently (they are not reported to the filter manager window).

  If your DLL implements modeless filters (remember, though, that you cannot mix modeless and modal filters in the same dll), you should inherit from this class rather than `hctFilterDll`.

- `hctModelessFilterDescriptor`

  This class provides implementations for keeping track of active instances of modeless filters, as well as asking them to close and to report their state.

- `hctModelessFilter`

  This class overrides `process()` from `hctFilterInterface`; it implements it by making a copy of the contents to process, creating a thread and then invoking `processModal()`. Filters inheriting from `hctModelessFilter` must therefore implement the `processModal()` method instead of `process()`.

  Code inside the `processModal()` method can act as if the filter was modal (since it operates on a private copy of the data and it runs in a separate thread).

By using these classes, implementing a modeless filter is a very easy task, almost identical to implementing a modal filter, with the only differences being:

- Modeless filters cannot be mixed with modal filters in the same DLL
- Implementations must inherit from `hctModelessFilterDll`, `hctModelessFilterDescriptor` and `hctModelessFilter`.
- The filter must implement `hctModelessFilter::processModal()` instead of `hctFilterInterface::process()`.

#### 4.9.7.6 Examples

Check out the "Extending the Toolchain" tutorial for an step-by-step example on how to extend the tools by exporting extra data from 3ds Max, Maya and XSI and creating a custom filter.

### 4.9.8 Customizing your Installation Method

The Havok Content Tools installer adds most of its files to a central location. Extra plugin files used by 3ds Max, Maya and Softimage are installed to other locations depending on the installation options chosen by the user. The installer also sets up registry keys and registers assemblies with the Global Assembly Cache (GAC).

#### 4.9.8.1 Steps taken by the Installer

The Havok Content Tools installer installs a required set of core files, plus optional sets of files for whichever modeler versions are present.

**Core Files, Registry Keys and GAC**

The core files are those which provide the Havok Filter Pipeline, specifically the filter manager DLL, the individual filter DLLs, and the attribute description XML files. The standalone filter manager, a set of batch processing scripts and some help files are also installed alongside the core files.

These files are installed into any standalone path ("`%ProgramFiles%\Havok\HavokContentTools`" by default), forming a particular folder structure.

Several registry keys and values are then set up under "`[HKEY_CURRENT_USER\Software\Havok]`". The most important of these is the "`hkFilters\FilterPath`" string value, containing the path to the installed filter manager DLL. This value is read by the scene exporters whenever they need to locate and load the filter manager.

The installer also registers assemblies with the Global Assembly Cache (GAC).

**3ds Max Files**

The optional 3ds Max files provide the tools used by 3ds Max to create and export Havok content. These consist of two plugins (one for export, one for physics tools) and a set of associated Maxscripts, icons, and tutorial scenes.

These files are installed directly into a 3ds Max application directory (chosen during installation), adding themselves to various subfolders (`\plugins`, `\UI`, `\scripts`). On startup, 3ds Max will automatically find these files and load the plugins.

**Maya Files**

The optional Maya files provide the tools used by Maya to create and export Havok content. These consist of two plugins (one for export, one for physics tools) and a set of associated MEL scripts, icons, and tutorial scenes.

These files are installed into any standalone path (chosen during installation), forming a particular folder structure: that of a Maya "*Module*".

The installer then instructs Maya to use this module by creating the following 'pointer' file: `"%CommonFiles%\Alias Shared\modules\maya\<x.x>\HavokContentTools.txt"`. The content of this simple text file is of the following form:

```
+ HavokContentTools <x.x> <module path>
```

On startup, Maya should locate this file and add the contents of the corresponding module to its list of available plugins. The plugins may then need to be manually loaded from within Maya's plugin manager.

**Softimage Files**

The optional Softimage files provide the tools used by Softimage to create and export Havok content. These consist of two plugins (one for export, one for physics tools) and a set of associated SPDL files, layouts, toolbars, presets, and tutorial scenes.

These files are installed into any standalone path (chosen during installation), forming a particular folder structure: that of an Softimage "*Workgroup*".

On startup, Softimage may automatically load the workgroup or it may have to be manually loaded: if it was installed to Softimage *factory* or *user* path then it will automatically loaded; otherwise it should be loaded by connecting to it from within Softimage's plugin manager.

#### 4.9.8.2 Creating a Custom Installation

The best way to install the Havok Content Tools across a network, or to prepare them for manual redistribution (e.g. in a zip file), is to

1. Remove any previous installations
2. Install the Content Tools to a temporary location
3. Review installed components and prepare for network installation
4. Ensure the required prerequisites are present on the target machines
5. Deploy the custom installation

The following instructions explain how to perform these steps in more detail:

**Remove any previous installations**

If a previous version of the Havok Content Tools is already installed on the computer used for the temporary installation, the tools should first be uninstalled using Windows Add/Remove programs. This should remove all traces of any previous installations from the computer, but to check that everything has been removed:

- Open the directory where the Havok Content Tools were installed and remove any leftover files and directories e.g. filter log files, etc. If the Havok Managed Tools were installed previously there may be a file called `HavokClothToolsInstall.msi` or `HavokPreviewAndClothSetupToolsInstall.msi` in the installation directory. If so, delete it.
- Open the Windows registry and delete the `HKEY_CURRENT_USER\Software\Havok` registry key, if it exists.
- Open the Global Assembly Cache (`C:\Windows\assembly`). If any Havok assemblies are still registered in the GAC (they're called "Havok.something.something"), right-click on them and select uninstall.
- Explore any Maya installation directories and if a file called `modules/HavokContentTools.txt` exists then delete it.

NOTE: All these steps should be performed automatically by uninstalling via Windows Add/Remove Programs, but if an installation has been customised or modified in any way (or in rare cases where the original installation process was particularly non-standard) there may be some files left over. This is why these extra steps are recommended.

**Install the Content Tools to a temporary location**

Run the Havok Content Tools installer, choosing the products, modelers etc. that correspond to your required setup until the 'Choose Install Location' page is reached. At this point choose a temporary directory (and make a note of it) e.g.

- `C:\Program Files\Havok -> D:\temp\Havok`

If any Max plugins were selected for installation choose a temporary directory for each of those, e.g.

- `C:\Program Files\Autodesk\3ds Max 2010 -> D:\temp\3ds_Max_2010`
- `C:\Program Files\Autodesk\3ds Max 2010 -> D:\temp\3ds_Max_2011`
- `C:\Program Files\Autodesk\3ds Max 2010 -> D:\temp\3ds_Max_2012`

If any Maya plugins were selected for installation a single directory needs to be chosen for all three. This directory is the same as the installation for the core files by default, so it's okay to choose the temporary location used for the core files i.e.

- `C:\Program Files\Havok -> D:\temp\Havok`

If any Softimage plugins were selected for installation the suggested directory is the Addons subdirectory of the corresponding Softimage installation. It's a good idea to use this structure in the temporary location e.g.

- `C:\users\userNameHere\Softimage\XSI_7.0\Addons` -> `D:\temp\XSI_7.0\Addons`
- `C:\users\userNameHere\Softimage\Softimage_7.5\Addons` -> `D:\temp\Softimage_7.5\Addons`
- `C:\users\userNameHere\Autodesk\Softimage_2010\Addons` -> `D:\temp\Softimage_2010\Addons`
- `C:\users\userNameHere\Autodesk\Softimage_2011\Addons` -> `D:\temp\Softimage_2011\Addons`
- `C:\users\userNameHere\Autodesk\Softimage_2012\Addons` -> `D:\temp\Softimage_2012\Addons`

Finally click 'Install'. All the required files will be installed to the specified temporary locations. The installation process is now complete.

**Review installed components and prepare for network installation**

**Installed Files**

The layout of the installed Havok Content Tools files should look something like the following (which assumes `D:\temp` was the temporary installation location)

```
D:\temp
    +---3ds_Max_2010
    +---3ds_Max_2011
    +---3ds_Max_2012
    +---Havok
    |    +---HavokContentTools
    |    +---HavokMayaModules
    |        +---2009
    |        +---2010
    |        +---2011
    |        +---2012
    +---XSI_7.0
    |    +---Addons
    +---Softimage_7.5
    |    +---Addons
    +---Softimage_2010
    |    +---Addons
    +---Softimage_2011
    |    +---Addons
    +---Softimage_2012
         +---Addons
```

NOTE: Depending on what modelers are chosen, a different set of 3ds Max/Maya/Softimage directories may be present.

Each of the directories in `D:\temp` above needs to go in a different place on the end user's machine, i.e.:

- `D:\temp\3ds_Max_2010` -> <User's 3ds Max 2010 install directory>
- `D:\temp\3ds_Max_2011` -> <User's 3ds Max 2011 install directory>
- `D:\temp\3ds_Max_2012` -> <User's 3ds Max 2012 install directory>

- `D:\temp\Havok` -> <Chosen location for the Havok Content Tools on the user's machine>
- `D:\temp\XSI_7.0` -> <User's XSI 7.0 install directory>
- `D:\temp\Softimage_7.5` -> <User's Softimage 7.5 install directory>
- `D:\temp\Softimage_2010` -> <User's Softimage 2010 install directory>
- `D:\temp\Softimage_2011` -> <User's Softimage 2011 install directory>
- `D:\temp\Softimage_2012` -> <User's Softimage 2012 install directory>

Prepare all these files for the chosen network installation method (e.g. put them in separate zip files for deployment in different places, or put them all in a single large zip file that matches the layout of the end user's machine, etc.). As there are a number of different ways to deploy files across a network, it's up to the user to decide exactly how to do this.

**Hook up installed plugins to the 3ds Max, Maya and Softimage modelers**

The 3ds Max, Maya and Softimage modelers need to know how to load any plugins that are installed. Some extra steps may be needed to do this. Figure out which steps are needed and how to perform them on the end user's machine.

**3ds Max**

As long as the plugin files are copied to the 3ds Max installation directory, no extra steps are needed. The 3ds Max modeler should find and load the Havok Content Tools plugins automatically.

**Maya**

A 'pointer' file is needed to help Maya find the Havok Content Tools plugins. It should be created here

`<MAYA_INSTALLATION_DIRECTORY>\modules\HavokContentTools.txt`

The file should contain the following line

```
+ HavokContentTools <MAYA_VERSION> <PATH_TO_CONTENT_TOOLS><LOCATION_OF_PLUGIN_FILES>
```

For example, assuming default Maya install locations these files should be created

- `C:\Program Files\Autodesk\Maya2010\modules\HavokContentTools.txt`
- `C:\Program Files\Autodesk\Maya2011\modules\HavokContentTools.txt`
- `C:\Program Files\Autodesk\Maya2012\modules\HavokContentTools.txt`

And each should contain one of the following lines

```
+ HavokContentTools 2010 <PATH_TO_CONTENT_TOOLS>\Havok\HavokMayaModules\2010
+ HavokContentTools 2011 <PATH_TO_CONTENT_TOOLS>\Havok\HavokMayaModules\2011
+ HavokContentTools 2012 <PATH_TO_CONTENT_TOOLS>\Havok\HavokMayaModules\2012
```

The Havok Content Tools installer also copies files to the following directories:

```
<PATH_TO_MAYA_INSTALLATION>\scripts\
<PATH_TO_MAYA_INSTALLATION>\plugins\
<PATH_TO_MAYA_INSTALLATION>\icons\
```

The Havok Content Tools expects this directory structure when loading the plugins in Maya.

**Softimage**

If the Softimage plugin files are put in the Addons subdirectory of the Softimage installation then Softimage will load them automatically. If they're put somewhere else they need to be connected manually from within the Softimage modeler.

**Registry Keys**

A number of registry keys and values will have been created by the temporary installation. Open the registry editor, right-click on the `HKEY_CURRENT_USER\Software\Havok` key, and save it as a `.reg` file.

Right-click on the saved `.reg` file and select 'edit', which opens it in a text editor. Update the paths in the `.reg` file to the corresponding location on the end user's computer i.e.

- `D:\\temp\\3ds Max_2010 -> <User's 3ds Max 2010 install directory>`
- `D:\\temp\\3ds_Max_2011 -> <User's 3ds Max 2011 install directory>`
- `D:\\temp\\3ds_Max_2012 -> <User's 3ds Max 2012 install directory>`
- `D:\\temp\\Havok -> <Chosen location for the Havok Content Tools on the user's machine>`
- `D:\\temp\\XSI_7.0 -> <User's XSI 7.0 install directory>`
- `D:\\temp\\Softimage_7.5 -> <User's Softimage 7.5 install directory>`
- `D:\\temp\\Softimage_2010 -> <User's Softimage 2010 install directory>`
- `D:\\temp\\Softimage_2011 -> <User's Softimage 2011 install directory>`
- `D:\\temp\\Softimage_2012 -> <User's Softimage 2012 install directory>`

**Global Assembly Cache (GAC)**

If there are any DLLs in the `HavokContentTools\tools\gac` directory (there usually will be) they need to be registered with the Global Assembly Cache on the end user's machine. This is done using the Global Assembly Cache Tool (`gacutil.exe`), which should be present if they have the Microsoft.NET Framework 2.0 installed, which is itself a prerequisite for the Havok Content Tools.

`gacutil.exe` can be installed in a number of locations e.g. `C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin`, `C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin` etc.

The following command installs assemblies to the GAC:

```
gacutil.exe /i <assembly name>
```

For example:

```
gacutil.exe /i <PATH_TO_HAVOK_CONTENT_TOOLS>\HavokContentTools\tools\gac\Havok.Cloth.Interfaces.dll
gacutil.exe /i <PATH_TO_HAVOK_CONTENT_TOOLS>\HavokContentTools\tools\gac\Havok.Cloth.Setup.dll
gacutil.exe /i <PATH_TO_HAVOK_CONTENT_TOOLS>\HavokContentTools\tools\gac\Havok.Graphics.Bridge.dll
gacutil.exe /i <PATH_TO_HAVOK_CONTENT_TOOLS>\HavokContentTools\tools\gac\Havok.Graphics.dll
gacutil.exe /i <PATH_TO_HAVOK_CONTENT_TOOLS>\HavokContentTools\tools\gac\Havok.Tool.Interfaces.dll
gacutil.exe /i <PATH_TO_HAVOK_CONTENT_TOOLS>\HavokContentTools\tools\gac\Havok.Tool.UI.dll
```

NOTE: For the registration step to work it needs to be executed from wherever `gacutil.exe` is located on the user's machine. Alternatively the location of `gacutil.exe` can be added to their `PATH` environment variable. Another alternative is to execute the command from one of the Visual Studio command prompts, as they automatically know where to find `gacutil.exe`. More information on the Global Assembly Cache Tool can be found on MSDN.

Decide how this step will be performed on the end user's computer, and prepare the necessary scripts, batch files, etc.

**Ensure the required prerequisites are present on the target machines**

In order for the Content Tools to run the following prerequisites need to be present on the end-user's machine:

- The .NET Framework 3.5 Service Pack 1:

  http://msdn.microsoft.com/en-us/netframework/cc378097.aspx
- Microsoft Visual C++ 2008 Service Pack 1 Redistributable Package ATL Security Update:

  http://www.microsoft.com/downloads/details.aspx?familyid=2051A0C1-C9B5-4B0A-A8F5-770A549FD78C
- DirectX End-User Runtimes (June 2010):

  http://www.microsoft.com/downloads/en/details.aspx?FamilyID=3B170B25-ABAB-4BC3-AE91-50CEB6D8FA8D

**Deploy the custom installation**

Perform each of the following steps:

- Deploy all the files to the correct locations on the end user's computer.
- Perform any steps needed to hook the Havok plugins up to the 3ds Max/Maya/Softimage modelers (there may not be any).
- Update the end user's registry (the modified `.reg` file simply needs to be executed).
- Register any Havok assemblies with the end user's Global Assembly Cache (using `gacutil.exe`).

### 4.9.9 Tutorial: Extending the Toolchain

While the most common parameters of rigid bodies, shape types and constraints are already exposed and processed by the Havok Content Tools (and more will be supported in the future), you may find the need to extend this coverage into particular functionality of the Havok SDK or your custom objects. This tutorial provides an step-by-step guide on how to extend both the modelling tools and filter processing in order to handle a new type of object. You may want to become familiar with the concepts and classes discussed in this chapter before working on this tutorial, as most of them will be used during it.

#### 4.9.9.1 Introduction

For this tutorial, we will consider the example of setting up and creating a `"phantom"` runtime object. A phantom is a shape in the scene which receives events whenever a rigid bodies enter or exit its volume. We will also create some configurable actions which will be applied to any rigid bodies which move within the phantom volume.

We would like to be able to use this added functionality with any supported modeler (3ds Max/Maya/XSI). Due to the design of the toolchain, this is a relatively simple process. We can separate the required work into two distinct areas:

1. We need a way to flag objects in a scene as phantoms, and to specify any number of additonal attributes which determine the runtime behaviour of the phantom and/or the actions.

   This will be done within the modeler, through the use of 'custom attributes'. As described earlier in this chapter, each of the supported modelers provides a way in which custom data may be attached to any object in the scene (3ds Max, Maya, XSI). This data will automatically be detected and exported by the Havok Scene Exporters. Very little work is therefore required within the modeler, and this work can easily be automated/scripted.

2. We need a way to interpret the exported phantom information, and convert it to a runtime phantom with associatied actions.

   This will be done within the filter pipeline, by creating a custom filter. This filter will look for any objects which contain our custom phantom data, and create appropriate runtime objects in the processed scene. Most of the work will therefore be performed by the filter, which is modeler-independent.

Before getting started, lets consider the structure of an `hkpPhantom` object. Observe that an `hkpShape` is required. Therefore when our filter wants to create a phantom based on a scene object, the first thing it will need to do is create an `hkpShape` representing the object's mesh. One solution would be to have the filter analyze the mesh and determine a good shape based on that. However, the Havok Content Tools already provide the ability to create `hkpShape` objects as part of rigid bodies, and provide a filter to interpret that information (the 'Create Rigid Bodies' filter). These tools already contain all of logic required to create `hkpShape` objects (including compound shapes), so it is much more beneficial to re-use this existing logic. In fact, we can just wrap a rigid body's shape with an `hkpPhantomCallbackShape` object, allowing us to also re-use any rigid body information (e.g. transform).

Therefore the solution which we will demonstrate in this tutorial is as follows:

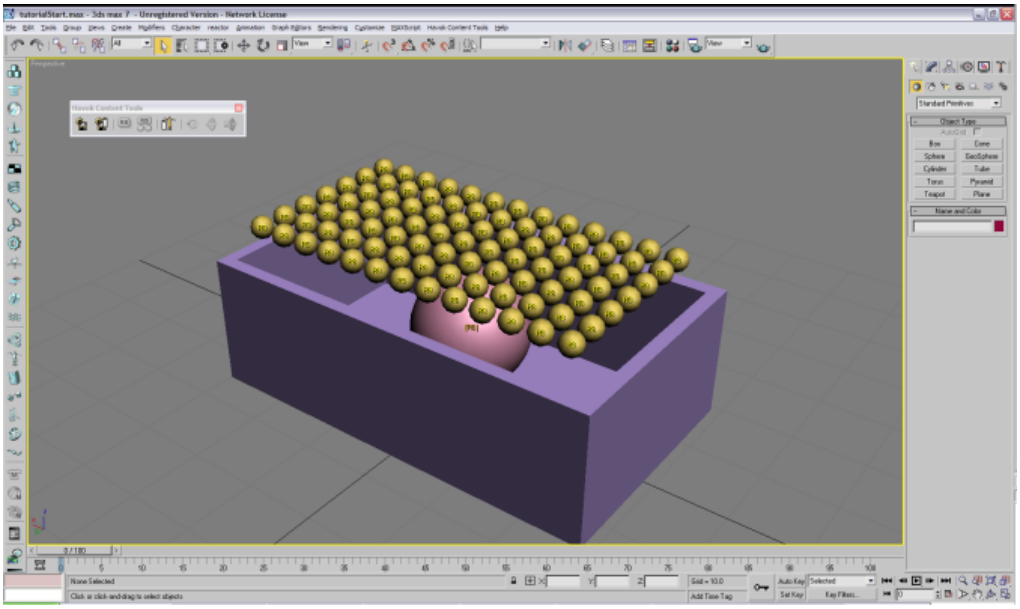
1. When we want to create a phantom from an object in the modeler, we will create a rigid body for it as per usual (which includes shape information). We will then add an **extra** attribute group to the

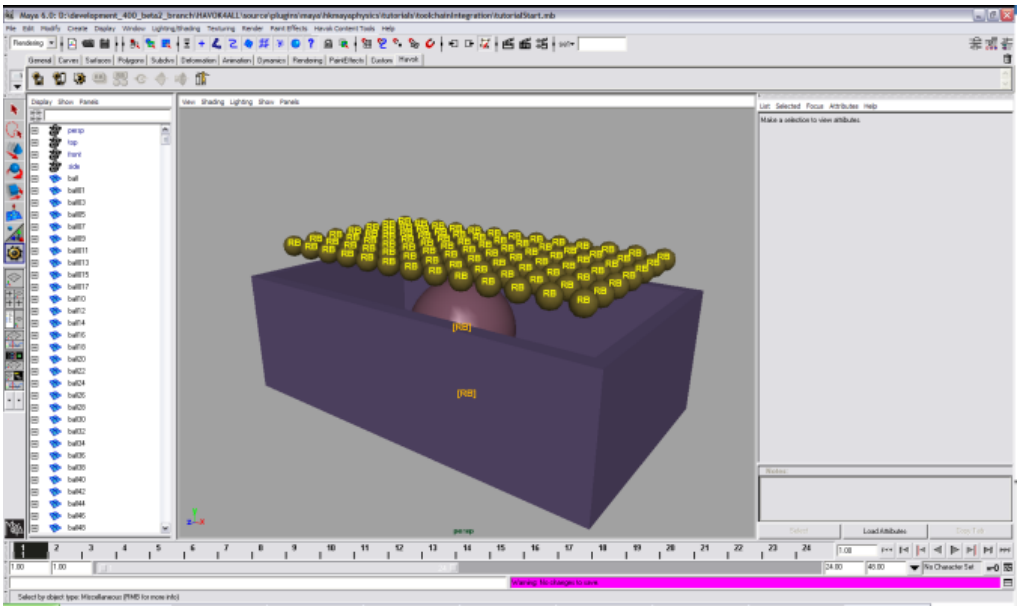
object specifiying phantom information. This attribute group will be called '*hkPhantomAction*'. We will also specify some additional processing rules for these attributes through the use of 'attribute description' XML files.

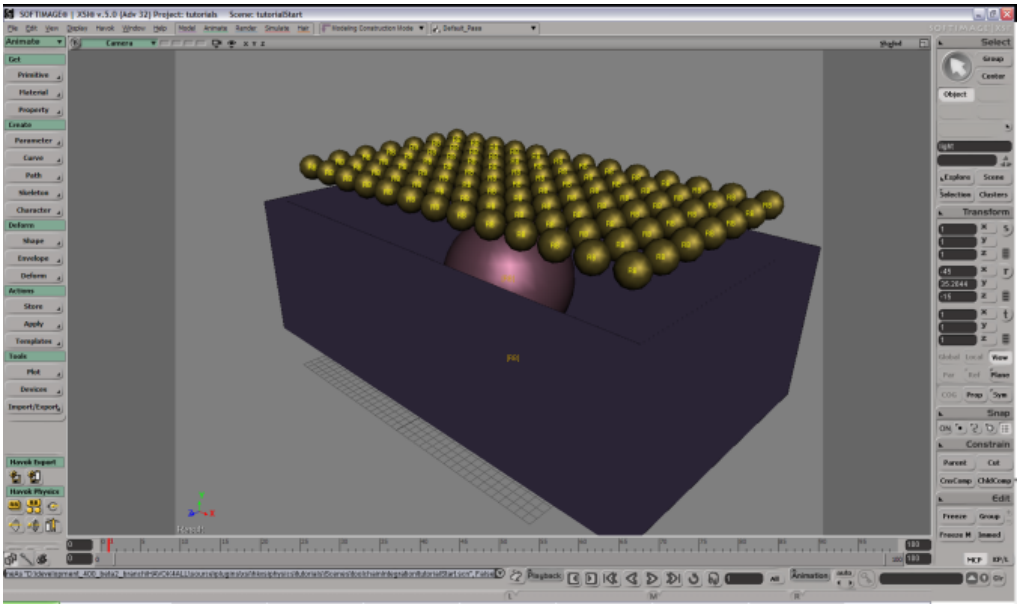
2. After exporting the data, we will add a 'Create Rigid Bodies' filter to the filter configuration; this will create a `hkpRigidBody` object, which includes a `hkpShape` object. Following that, we will add our custom filter; this will search for our specific attribute group in the scene and wrap each corresponding rigid body's `hkpShape` object with an `hkpPhantomCallbackShape` object. The values of the attributes which we specified in the modeler will be used to customise the behaviour of the actions which will act on any rigid bodies entering the phantom volume. We will call this filter "*convertToPhantomAction*".

#### 4.9.9.2 Starting up

We will use a very simple scene to test our toolchain extensions. Load the tutorial file `toolchainIntegration/tutorialStart` (`.max`/`.mb`/`.scn`):







This very simple scene contains an open box and multiple spheres. The large sphere in the middle (*phantom*) is what we want to use as our phantom object; we want any spheres which enter that volume to be affected by our magic action.

#### 4.9.9.3 Creating and Exporting the Attribute Group

The first step in the process is to create a rigid body within a modeler and add our custom attribute group to it, so it can be picked up later on by our custom filter. We would like to add our atributes as a group by the name of " **hkPhantomAction** "; this could be any name, but it should be descriptive of

the group contents. We would like to add the following attributes to this group:[9]

| | |
|---|---|
| action | A STRING value, specifying a type of action which should be applied to the phantom once it is created by the filter (from a selection of "wind", "attract", "deflect"). |
| direction | A VECTOR value, specifiying some direction which may be used by the action. |
| strength | A FLOAT value, specifiying some power value which may be used by the action. |

In the following subsection we will see how to add these attributes to our object using each specific modeler. We will then also see how to solve the problem of handling differences between modelers regarding attribute types and ranges by using an attribute repository.

### Adding the Attributes

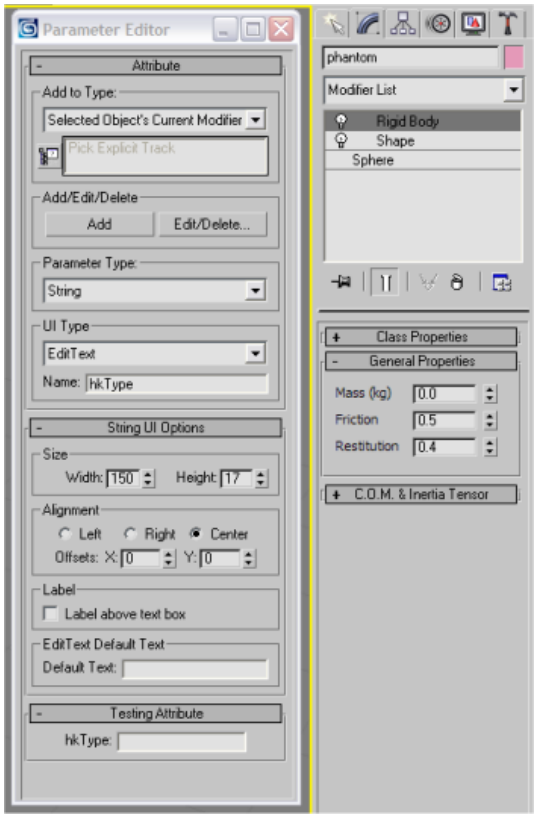
Our phantom object (named "phantom") already has the rigid body and shape attributes (added by `hkpRigidBody` and `hkpShape` modifiers/nodes/custom parameter sets); we now want to also add our new attributes. There are multiple ways to specify extra attributes to be exported from each modeler. In this tutorial, we will use the simplest one, which uses the modeler's UI to add them.

**Note:**
This has the advantage of not requiring any knowledge of scripting or plugin development, so it suits this tutorial. It is also possible, however, to add attributes by many other means, including scripting and creating custom nodes/modifiers/property sets using C++ or other supported languages. Attributes created this way have the advantage of being easier to customize, to reuse and to extend, but you'll need some knowledge of the modeler's scripting language and/or SDK in order to do so.

### 3ds Max

Make sure the rigid body modifier of our *phantom* object (the big sphere) is selected, then click on **Animation** > **Parameter Editor...** from the main menu. This opens a dialog which allows us to define and add custom attributes:

[9] These attributes were chosen for this tutorial so that we can demonstrate how to create and process various attribute types. Feel free to use whatever you come up with.




We want our set of attributes to be exported as a named group. The name of the default custom attribute container in 3ds Max in '*main*' (this can be changed using scripting), and as such any attributes we add using the UI would be exported under a group called 'main' by default. Since we would like to use a specific group name, begin by adding the following 'special' atrribute:

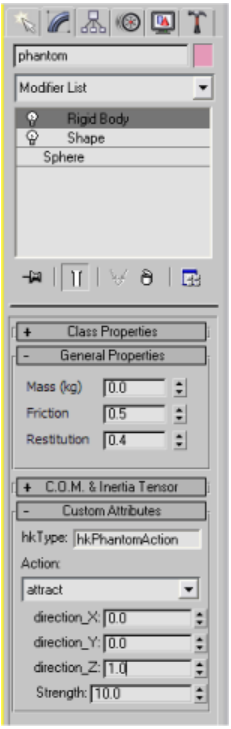
- A **String** named "`hkType`".

  When the exporter finds an attribute with this name (*hkType*), it uses its value to replace the name of the exported attribute group; in this case give it a value of "`hkPhantomAction`".

Next add the 'real' attributes in turn:

- An **Array** named "`action`", with the following elements: "`wind`", "`attract`", "`deflect`".
- Three individual **WorldUnits** named "`direction_X`", "`direction_Y`", and "`direction_Z`".
- A single **Float** named "`strength`".

Which should leave you with the following UI:

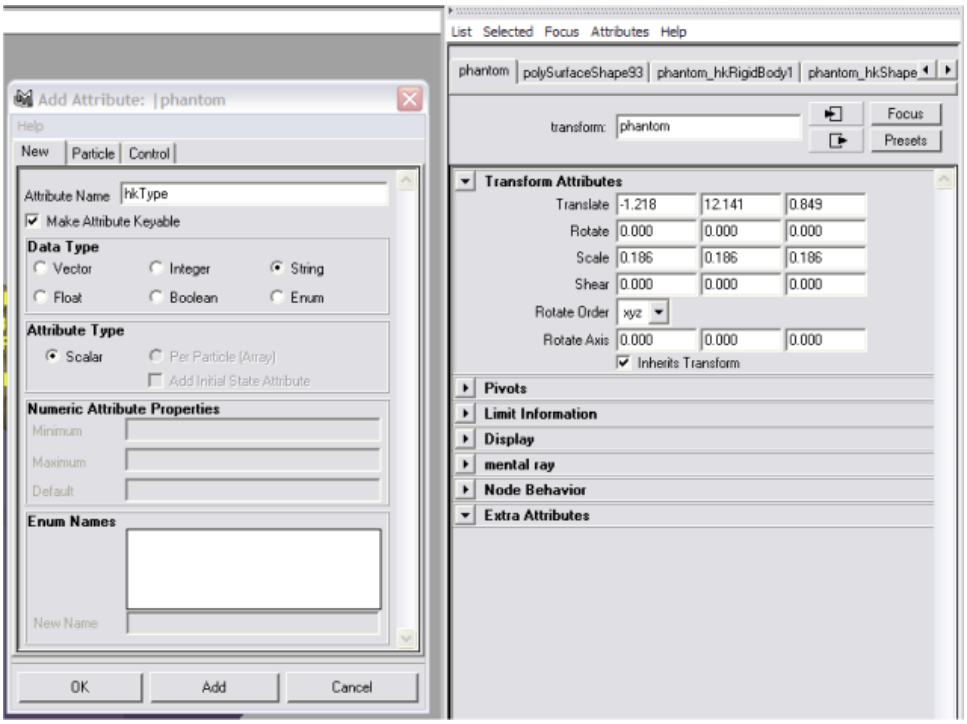



Set the following values:

- **hkType** : *hkPhantomAction*
- **Action** : *attract*
- **Direction** : *0,0,1*
- **Strength** : *10*

### Maya

Make sure the *phantom* node (that is, the transform node named "*phantom*") is selected, then click on **Attributes** > **Add Attribute...** from the attribute editor panel. This opens a dialog which allows us to define and add new attributes:




We want our attributes to be exported as a named attribute group. By default in Maya, any attributes we add will be in the same group as all of the existing attributes of a node. Therefore we require some way to specifiy a new group. This is done by first adding the following 'special' atrribute:

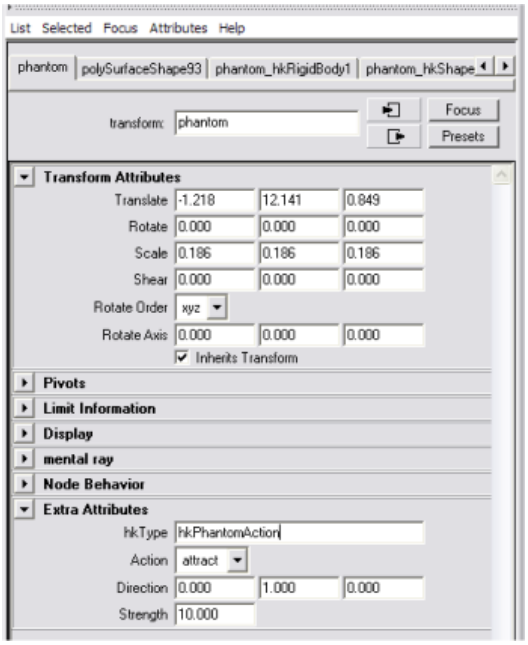
- A **String** attribute named "`hkType`".

  When the Maya scene exporter finds a string attribute with this name, it starts a new attribute group named with the value of this attribute. In this case give it a value of "`hkPhantomAction`".

Next add the 'real' attributes in turn:

- An **Enum** named "`action`", with the following elements: "`wind`", "`attract`", "`deflect`".
- A **Vector** named "`direction`".
- A **Float** named "`strength`".

Which should leave you with the following UI (you may need to expand the ' **Extra Attributes** ' rollout):

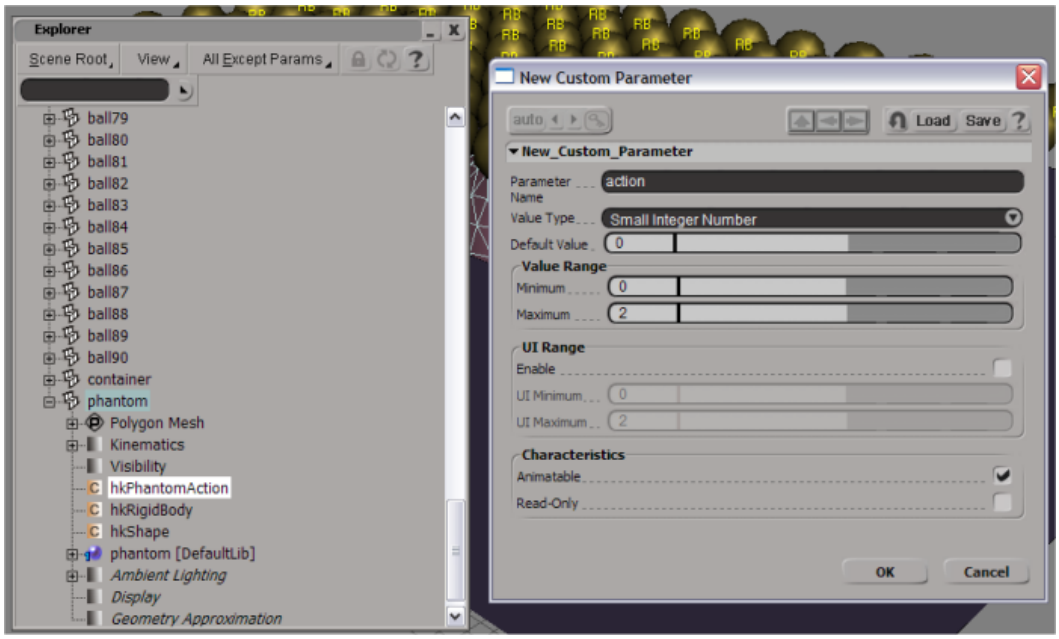



Set the following values for the attributes:

- **hkType** : *hkPhantomAction*
- **Action** : *attract*
- **Direction** : *0,1,0*
- **Strength** : *10*

### XSI

Making sure our phantom object is selected, click on **Animate** > **Parameter** > **New Custom Parameter Set** from the main menu. Enter "`hkPhantomAction`" as the name of the new parameter set. Now, with the new parameter set selected, click on **Animate** > **Parameter** > **New Custom Parameter Set** from the main menu (or press SHIFT+P). This opens a dialog which allows us to define and add a new custom parameter/attribute:




Add the following attributes in turn:

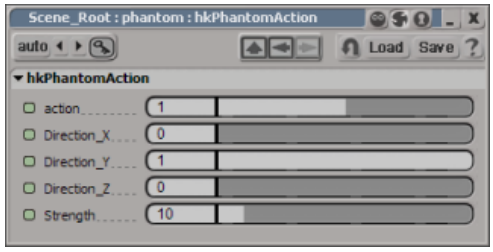
- An **Small Integer Number** named "`action`", with a range of zero to two (representing the three action types: "wind", "attract", "deflect").

  **Note:**

  There is no way to define enum attributes in XSI; we will use integers instead. Integers can be represented as strings in property editors by writing custom PPG code, however for this tutorial we just 'know' that the integers values 0,1,2 represent "wind", "attract" and "deflect".

- Three individual **Floating Point Number** s named "`direction_X`", "`direction_Y`", and "`direction_Z`".
- A **Floating Point Number** named "`strength`".

This should leave you with the following PPG for the custom parameter set:




Set the following values for the attributes:

- **Action** : *1* (attract)
- **Direction** : *0,1,0*

- **Strength** : *10*

### Pre-processing Attributes

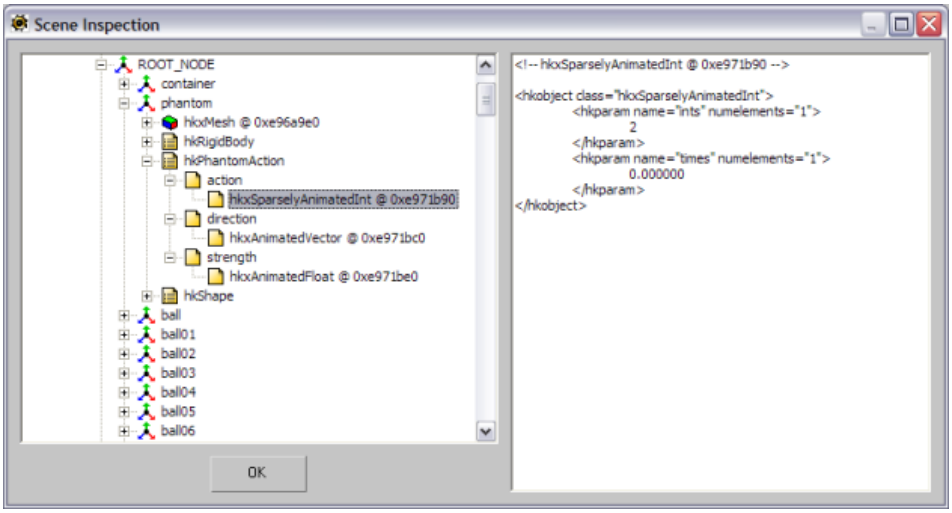
Each modeler has different means and capabilities to store and represent attributes. For example, some modelers (XSI and to some extent 3ds Max) do not have support for enumerated values, some others can't represent vector attributes with a single attribute, etc.

Since we want to build our filters in a modeler-independent way, the scene exporters provide the ability to pre-process the attributes before they are passed to the filters. Some processing is always done (for example merging X,Y,Z float attributes into a single vector attribute) while some other is driven by the user through XML files (an attribute repository): for example, integer -> string conversions or marking a float as "scalable" (world units) values.

We will need to drive some preprocessing of our attributes from each modeler in order to match our original `hkPhantomAction` attribute group definition (a string, a vector and a float). The XML files for each modeler are automatically installed for you; for the purpose of this tutorial, you may want to temporarily remove them to see their effect. The files can be found in the `max`, `maya`, or `xsi` folders inside the main installation folder (there the filter manager DLL is placed): for example, `C:\Program Files\Havok\HavokContentTools\AttributeProcessing\Max\hctMaxTutorialAttributeData.xml`.

### 3ds Max

Without adding anything to the attribute repository (you can test this by temporarily removing the `AttributeProcessing/Max/hctMaxTutorialAttributeData.xml` file) this would be the output associated to our *phantom* node:




Notice how the X, Y and Z values of *direction* have been merged into a single vector value; this is always done automatically. Also, if you examine the "*direction*" attribute, you see that it has the flag **HINT_TRANSFORM_AND_SCALE** . This is because we created the X, Y and Z attributes in 3ds Max as "world units" and therefore the exporter assumes that the value should be fully transformed by any scene transformations.

Notice however how the *action* parameter is being exported as an integer (with value = 2 for the second value, "*attract*"): 3ds Max doesn't keep track of the value-name association so the exporter cannot see

the name, it only sees the integer value.

Here is where the attribute repository comes into play: we can tell the exporter to apply changes to attributes by using a max-specific XML file (the `attributeProcessing/max/hctMaxTutorialAttributeData.xml` we temporarily removed; place it back now). Let's see the contents of this file:

```
<?xml version="1.0" encoding="utf-8"?>
<hkpackfile classversion="2">

    <hksection name="__data__">

        <hkobject class="hctAttributeDescriptionDatabase" name="#0000">
            <hkparam name="groupDescriptions" numelements="1">

                <hkobject>
                    <hkparam name="name">hkPhantomAction</hkparam>
                    <hkparam name="attributeDescriptions" numelements="2">
                        <hkobject>
                            <hkparam name="name">action</hkparam>
                            <hkparam name="forcedType">FORCE_STRING</hkparam>
                            <hkparam name="enum">#0001</hkparam>
                            <hkparam name="hint">HINT_NONE</hkparam>
                        </hkobject>
                        <hkobject>
                            <hkparam name="name">direction</hkparam>
                            <hkparam name="hint">HINT_TRANSFORM</hkparam>
                        </hkobject>
                    </hkparam>
                </hkobject>

            </hkparam>
        </hkobject>

        <hkobject class="hkClassEnum" name="#0001">
            <hkparam name="name">hkShapeTypeEnum</hkparam>
            <hkparam name="items" numelements="3">
                <hkobject>
                    <hkparam name="value">1</hkparam>
                    <hkparam name="name">wind</hkparam>
                </hkobject>
                <hkobject>
                    <hkparam name="value">2</hkparam>
                    <hkparam name="name">attract</hkparam>
                </hkobject>
                <hkobject>
                    <hkparam name="value">3</hkparam>
                    <hkparam name="name">deflect</hkparam>
                </hkobject>
            </hkparam>
        </hkobject>

    </hksection>

</hkpackfile>
```

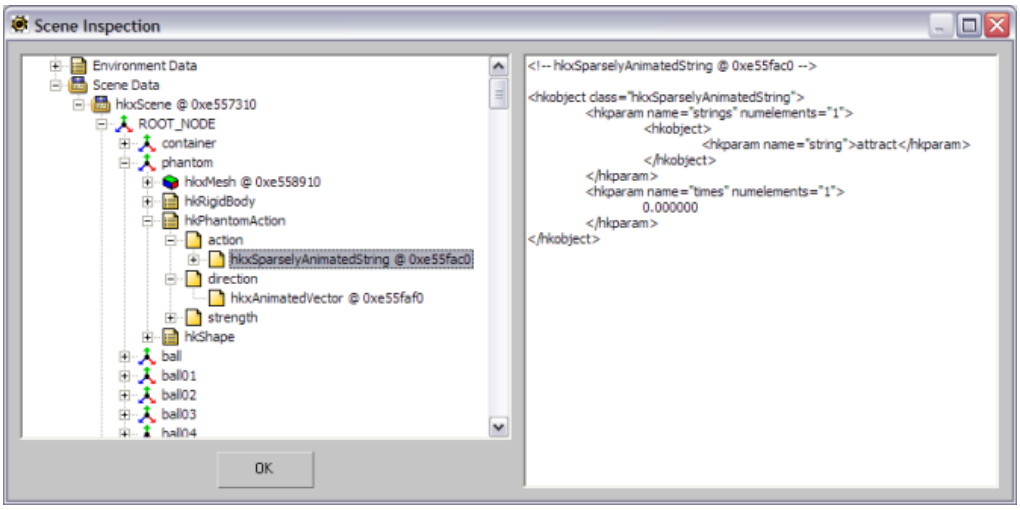
The file contains an attribute description database (`hctAttributeDescriptionDatabase`) object, with a single attribute group description (`hctAttributeGroupDescription`) for *hkPhantomAction*. We only have two attribute descriptions (`hctAttributeDescription`).

- One for our *action* attribute, specifying that we want to force its type to be a string (

**FORCE_STRING** ). We also provide, in order to do the conversion, a `hkClassEnum` object that maps values with names (1,2,3 for *wind*, *attract* and *deflect*).

- One for the *direction* attribute–3ds Max assumed it should be transformed and scaled (as it could represent a position)–but since we know it represents a direction, we don't want it to scale with the scene. Therefore we set the flag **HINT_TRANSFORM** .

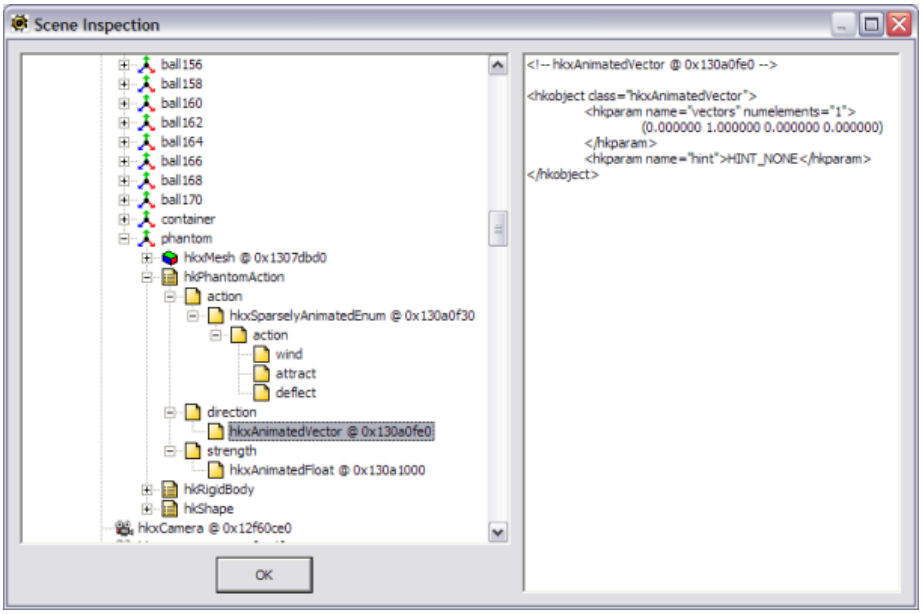
With this file in place, the exported attributes now look like this:




The attributes now match what our filter expects, so we are done!

**Maya**

Without adding anything to the attribute repository (you can test this by temporarily removing the `AttributeProcessing/Maya/hctMayaTutorialAttributeData.xml` file) this would be the output associated to our *phantom* node:




Notice that our action attribute is exported as an enum; Maya has built-in support for enumerated attributes. Our filter expects a string. It is very easy to convert from an enum to an string: you just take convert the value to its associated name (notice how the valueâĘŤname association is already exported from Maya).

The direction value is exported as a vector. That's also what we want. However, Maya doesn't know what the vector represents; it could be a color, a position, a direction, etc. We'd like to specify that it represents a direction; that is, that it should be transformed (rotated), but not scaled, by a scene transformation.

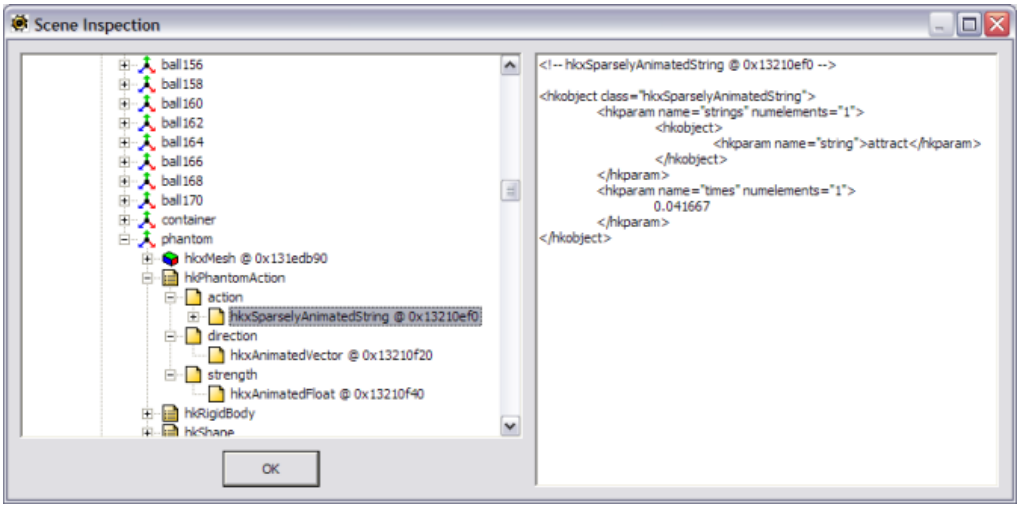
Here is where the attribute repository comes into play: we can tell the exporter to apply changes to attributes by using a maya-specific XML file (the `AttributeProcessing/Maya/hctMayaTutorialAttributeData.xml` we temporarily removed; place it back now). Let's see the contents of this file:

```
<?xml version="1.0" encoding="utf-8"?>
<hkpackfile classversion="2">

    <hksection name="__data__">

        <hkobject class="hctAttributeDescriptionDatabase" name="#0000">
            <hkparam name="groupDescriptions" numelements="1">

                <hkobject>
                    <hkparam name="name">hkPhantomAction</hkparam>
                    <hkparam name="attributeDescriptions" numelements="2">
                        <hkobject>
                            <hkparam name="name">action</hkparam>
                            <hkparam name="forcedType">FORCE_STRING</hkparam>
                        </hkobject>
                        <hkobject>
                            <hkparam name="name">direction</hkparam>
                            <hkparam name="hint">HINT_TRANSFORM</hkparam>
                        </hkobject>
                    </hkparam>
                </hkobject>

            </hkparam>
        </hkobject>

    </hksection>

</hkpackfile>
```

The file contains an attribute description database (`hctAttributeDescriptionDatabase`) object, with a single attribute group description (`hctAttributeGroupDescription`) for *hkPhantomAction*. We only have two attribute descriptions (`hctAttributeDescription`).

- For the action attribute, we specify that we want to convert it to an string attribute using **FORCE_STRING** . Notice that the original enum attribute can easily be converted to an string attribute; we don't need to specify any value-to-name conversion since it was already contained in the enum attribute.
- For the direction attribute, we add the flag **HINT_TRANSFORM** to note that this vector should rotated, but not scaled, by any scene transformations applied.

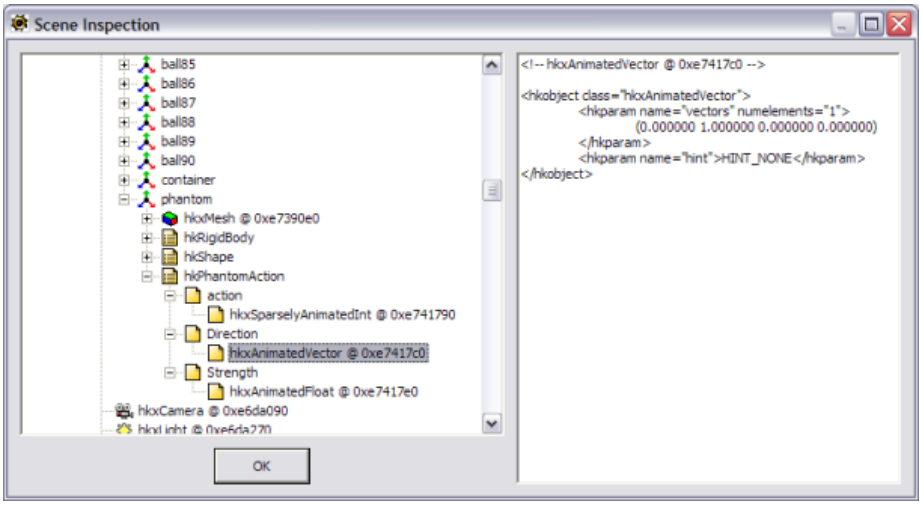
With this file in place, the exported attributes now look like this:




The attributes now match what our filter expects.

### XSI

Without adding anything to the attribute repository (you can test this by temporarily removing the `Xsi/hctXsiTutorialAttributeData.xml` file) this would be the output associated to our *phantom* node:




Notice that our action attribute is exported as an integer (with value = 1 for the second value, "attract"), since XSI doesn't support enumerated attributes

Notice also how our three X, Y and Z attributes for the direction were automatically merged into a single vector attribute. That's good. However, XSI doesn't know what the vector represents; it could be a color, a position, a direction, etc. We'd like to specify that it represents a direction, that is, that it should be transformed (rotated), but not scaled, by a scene transformation.

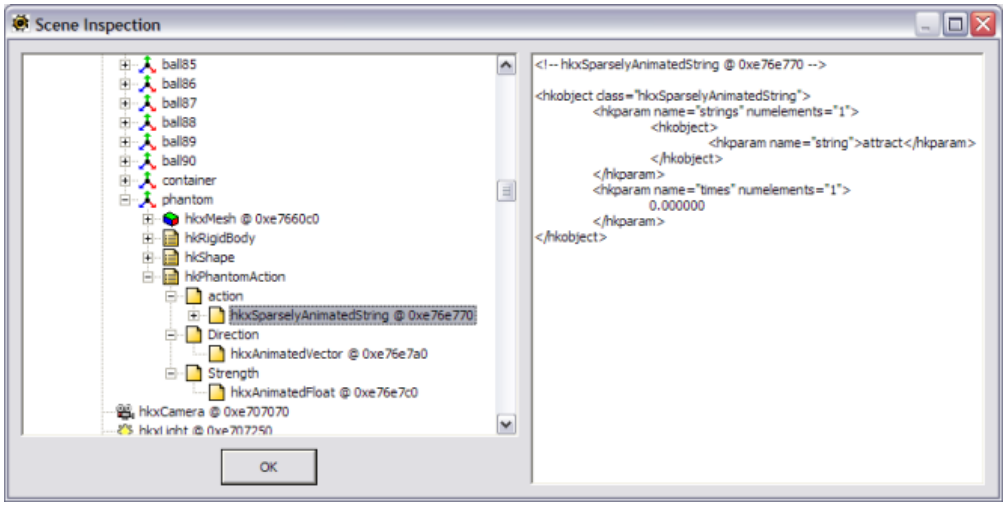
Here is where the attribute repository comes into play: we can tell the exporter to apply changes to attributes by using an XSI-specific XML file (the `attributeProcessing/xsi/hctXsiTutorialAttributeData.xml` we temporarily removed; place it back now). Let's see the contents of this file:

```
<?xml version="1.0" encoding="utf-8"?>
<hkpackfile classversion="2">

    <hksection name="__data__">

        <hkobject class="hctAttributeDescriptionDatabase" name="#0000">
            <hkparam name="groupDescriptions" numelements="1">

                <hkobject>
                    <hkparam name="name">hkPhantomAction</hkparam>
                    <hkparam name="attributeDescriptions" numelements="2">
                        <hkobject>
                            <hkparam name="name">action</hkparam>
                            <hkparam name="forcedType">FORCE_STRING</hkparam>
                            <hkparam name="enum">#0001</hkparam>
                            <hkparam name="hint">HINT_NONE</hkparam>
                        </hkobject>
                        <hkobject>
                            <hkparam name="name">direction</hkparam>
                            <hkparam name="hint">HINT_TRANSFORM</hkparam>
                        </hkobject>
                    </hkparam>
                </hkobject>

            </hkparam>
        </hkobject>

        <hkobject class="hkClassEnum" name="#0001">
            <hkparam name="name">hkShapeTypeEnum</hkparam>
            <hkparam name="items" numelements="3">
                <hkobject>
                    <hkparam name="value">0</hkparam>
                    <hkparam name="name">wind</hkparam>
                </hkobject>
                <hkobject>
                    <hkparam name="value">1</hkparam>
                    <hkparam name="name">attract</hkparam>
                </hkobject>
                <hkobject>
                    <hkparam name="value">2</hkparam>
                    <hkparam name="name">deflect</hkparam>
                </hkobject>
            </hkparam>
        </hkobject>

    </hksection>

</hkpackfile>
```

The file contains an attribute description database (`hctAttributeDescriptionDatabase`) object, with a single attribute group description (`hctAttributeGroupDescription`) for *hkPhantomAction*. We only have two attribute descriptions (`hctAttributeDescription`).

- One for our *action* attribute, specifying that we want to force its type to be a string ( **FORCE_STRING** ). We also provide, in order to do the conversion, a `hkClassEnum` object that maps values with names (0,1,2 for *wind*, *attract* and *deflect*).
- For the direction attribute, we add the flag **HINT_TRANSFORM** to note that this vector should rotated, but not scaled, by any scene transformations applied.

With this file in place, the exported attributes now look like this:




The attributes now match what our filter expects.

#### 4.9.9.4 Writing the Filter

Now that we have the data exported from the modeler, we need to create a filter to interpret that data and create our phantom object. We have called the DLL containing this filter `hkfiltertutorial` and the filter itself "*Convert to Phantom Action*". Depending on your type of product license, you may find the source of this `hkfiltertutorial` project in your `include\plugins\common\hkfiltertutorial` folder.

**Setting up the project**

A single DLL can contain multiple filters, but in our case we will create a DLL for just a single filter (alternatively, we could add this filter to an existing filter DLL, for example `hkfilterphysics`). We will use the convention of placing the filter-independent files in the root folder of `hkfiltertutorial`, while we'll place the files specific to the "*Convert To Phantom Action*" filter inside the `hkfiltertutorial/converttophantomaction` folder.

If you start the Visual Studio project from scratch, create a DLL project with Multithreaded DLL configurations (that's the runtime library used by the exporters and filters). In particular, you may want to create the following configurations:

- `Release DLL`: Uses release multithreaded DLL runtime libraries, with full optimizations and no debug information. This configuration is useful to build the final DLL to ship to users.

Notice that the configuration uses Release Multithreaded DLL runtime libraries; this is because that's the runtime library used by the application (3ds Max, Maya, XSI) and mismatching them would cause instability.

Add the following two files to the project:

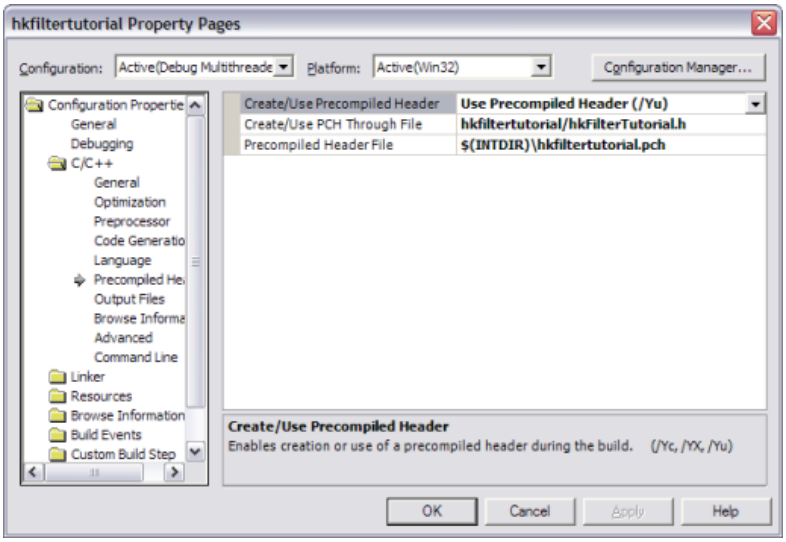
#### 4.9.9.5 Example: hctFilterTutorial.h

```
#ifndef HAVOK_FILTER_TUTORIAL__H
#define HAVOK_FILTER_TUTORIAL__H

#include <ContentTools/Common/Filters/Common/hctFilterCommon.h>

#include <ContentTools/Common/Filters/FilterTutorial/resource.h>

#endif // HAVOK_FILTER_TUTORIAL__H
```

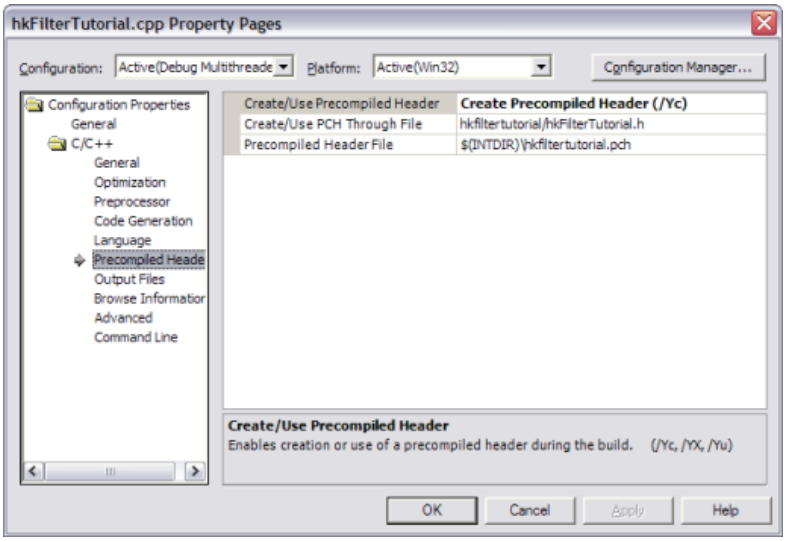
#### 4.9.9.6 Example: hctFilterTutorial.cpp

```
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>

// Keycode
#include <Common/Base/keycode.cxx>

// DLL main
HINSTANCE hInstance;
static BOOL CommonControlsInitialized = FALSE;

BOOL WINAPI DllMain(HINSTANCE hinstDLL, ULONG fdwReason, LPVOID lpvReserved)
{
    switch ( fdwReason )
    {
        case DLL_PROCESS_ATTACH:
        {
            hInstance = hinstDLL;                // Hang on to this DLL's instance handle.

            if (!CommonControlsInitialized)
            {

                CommonControlsInitialized = TRUE;
                InitCommonControls();            // Initialize Win32 controls
            }
            break;
        }

        case DLL_PROCESS_DETACH:
        {
            if (CommonControlsInitialized)
            {
                CommonControlsInitialized = FALSE;
            }
        }

        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
            break;
    }
    return TRUE;
}
```

We will use the `hctFilterTutorial.h` header filer as our precompiled header, since it contains very common includes and will speed up our compile times:




Then set `hkfiltertutorial.cpp` as the one to create that precompiled header:




The file `hkfiltertutorial.cpp` contains the `DllMain()` function, which here is only used to initialize the common controls library and to store a handle to the DLL instance.

### Defining our custom phantom shape and actions

Before we continue with more filter and dll specific classes and files, let's first define our custom phantom shape. Since it is specific to our filter, we'll place the files inside the `converttophantomaction` folder:

#### 4.9.9.7 Example: convertophantomaction/MyPhantomShape.h

```
#ifndef INC_MYPHANTOM_SHAPE_H
#define INC_MYPHANTOM_SHAPE_H

#include <Physics2012/Collide/Shape/Misc/PhantomCallback/hkpPhantomCallbackShape.h>

/// meta information
extern const class hkClass MyPhantomShapeClass;

// Our phantom call back shape will be notified when an object enters or exits the shape
// We use those event to dynamically add / remove an action.
class MyPhantomShape : public hkpPhantomCallbackShape
{
    //+vtable(1)
    public:

        HK_DECLARE_REFLECTION();

        MyPhantomShape();
        MyPhantomShape( hkFinishLoadedObjectFlag flag ) : hkpPhantomCallbackShape(flag) {}

        // hkpPhantom interface implementation
        /*virtual*/ void phantomEnterEvent( const hkpCollidable* phantomColl, const hkpCollidable*
          otherColl, const hkpCollisionInput& env );
        /*virtual*/ void phantomLeaveEvent( const hkpCollidable* phantomColl, const hkpCollidable*
          otherColl );

    public:

        enum ActionType
        {
            ACTION_WIND,
            ACTION_ATTRACT,
            ACTION_DEFLECT
        };

        hkEnum<ActionType, hkInt8>    m_actionType;
        hkVector4                        m_direction;
        hkReal                            m_strength;
};


#endif //INC_MYPHANTOM_SHAPE_H
```

Check the Havok Physics 2012 SDK for details on `hkpPhantomCallbackShape`. We basically want to override the two event handlers (enter and leave) so an action is applied to the object when it enters our shape and removed when it leaves. The implementation can be found in `MyPhantomShape.cpp`.

#### 4.9.9.8 Example: convertophantomaction/MyPhantomShape.cpp

```
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>
#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/MyPhantomShape.h>

#include <Physics2012/Dynamics/Action/hkpBinaryAction.h>
#include <Physics2012/Dynamics/Entity/hkpRigidBody.h>


//
// Actions
//

// Apply a constant force to the rigid body
// (Use a binary action so that it maintains a ref to the phantom)
class MyWindAction : public hkpBinaryAction
{
public:

    MyWindAction( hkpRigidBody* phantom, hkpRigidBody* rb, const hkVector4& force ) : hkpBinaryAction(
      phantom,rb), m_force(force) {}
    MyWindAction( hkFinishLoadedObjectFlag flag ) : hkpBinaryAction(flag) {}

    /*virtual*/ void applyAction( const hkStepInfo& stepInfo )
    {
        hkpRigidBody* rb = (hkpRigidBody*)getEntityB();
        rb->applyForce( stepInfo.m_deltaTime, m_force );
    }

    /*virtual*/ MyWindAction* clone( const hkArray<hkpEntity*>& newEntities, const hkArray<hkpPhantom*>&
      newPhantoms ) const
    {
        return HK_NULL;
    }

    hkVector4 m_force;
};

// Apply a 'point' force to the rigid body
class MyAttractAction : public hkpBinaryAction
{
public:

    MyAttractAction( hkpRigidBody* phantom, hkpRigidBody* rb, float strength ) : hkpBinaryAction(phantom,
      rb), m_strength(strength) {}
    MyAttractAction( hkFinishLoadedObjectFlag flag ) : hkpBinaryAction(flag) {}

    /*virtual*/ void applyAction( const hkStepInfo& stepInfo )
    {
        hkVector4 force;
        force.setSub4( getEntityA()->getMotion()->getPosition(), getEntityB()->getMotion()->getPosition()
          );
        float distance = force.length3();
        force.mul4( m_strength / ( ( distance * distance ) + HK_REAL_EPSILON ) );

        hkpRigidBody* rb = (hkpRigidBody*)getEntityB();
        rb->applyForce( stepInfo.m_deltaTime, force );
    }

    /*virtual*/ MyAttractAction* clone( const hkArray<hkpEntity*>& newEntities, const hkArray<hkpPhantom*>
      & newPhantoms ) const
    {
        return HK_NULL;
    }
```

. . . continued from above:

```
    float m_strength;
};



//
// hkpPhantom interface implementation
//

MyPhantomShape::MyPhantomShape()
{
    m_actionType = ACTION_WIND;
    m_direction.setZero4();
    m_strength = 0.0f;
}

/*virtual*/ void MyPhantomShape::phantomEnterEvent( const hkpCollidable* phantomColl, const hkpCollidable*
  otherColl, const hkpCollisionInput& env )
{
    hkpRigidBody* phantom = hkGetRigidBody( phantomColl );
    hkpRigidBody* rb = hkGetRigidBody( otherColl );
    if( !phantom || !rb )
    {
        return;
    }

    // Create the appropriate action
    hkpAction* action = HK_NULL;
    switch( m_actionType )
    {
        case ACTION_WIND:
            {
                hkVector4 force;
                force.setMul4( m_strength, m_direction );
                action = new MyWindAction( phantom, rb, force );
            }
            break;
        case ACTION_ATTRACT:
            {
                action = new MyAttractAction( phantom, rb, m_strength );
            }
            break;
        case ACTION_DEFLECT:
            {
                // use the attract action with a negative strength
                action = new MyAttractAction( phantom, rb, -m_strength );
            }
            break;
    }

    // Add it to the world
    if( action )
    {
        rb->getWorld()->addAction( action );
        action->removeReference();
    }
}
```

...continued from above:

```
/*virtual*/ void MyPhantomShape::phantomLeaveEvent( const hkpCollidable* phantomColl, const hkpCollidable*
  otherColl )
{
    hkpRigidBody* phantom = hkGetRigidBody( phantomColl );
    hkpRigidBody* rb = hkGetRigidBody( otherColl );
    if( !phantom || !rb )
    {
        return;
    }

    // Find and remove the appropriate action
    int numActions = phantom->getNumActions();
    for( int i=numActions-1; i>=0; --i )
    {
        hkpAction* action = phantom->getAction(i);

        if (action)
        {
            hkArray<hkpEntity*> entities;
            action->getEntities( entities );
            for( int j=0; j<entities.getSize(); ++j )
            {
                if( entities[j] == rb )
                {
                    action->getWorld()->removeAction( action );
                }
            }
        }
    }
}
```

In this CPP we define three custom binary actions (`MyWindAction`, `MyAttractAction` and `MyDeflectAction`) that apply different types of forces to the associated objects.

**Note:**
For simplicity we didn't properly implement the `clone()` method of those actions, but you should do so if you wanted your action to be cloneable.

Our phantom shape (`MyPhantomShape`) implements the `phantomEnterEvent()` method by creating one of the three actions (depending on the `m_actionType` member) and adding it to the world. The `phantomLeaveEvent()` method, in turn, removes those actions.

### Making the Phantom Shape serializable

You may have noticed the **HK_DECLARE_REFLECTION();** macro in `MyPhantomShape.h`. This macro is used by serialization, it tells our scripts that this class should have an associated `hkClass` and `hkTypeInfo`.

Since our filter will create objects of this new type (`MyPhantomShape`) that will be serialized, we need to declare an `hkClass` associated with it (please check the *Serialization* documentation for details). If you run the `generateReflections.py` script (please check the *Serialization > Using the Infrastructure* documentation for details), the following CPP will be generated; add it to the project:

#### 4.9.9.9 Example: convertophantomaction/Classes/convertToPhantomActionReflections.cpp (autogenerated)

```
// WARNING: THIS FILE IS GENERATED. EDITS WILL BE LOST.
// Generated from 'ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/MyPhantomShape.h'
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>
#include <Common/Base/Reflection/hkClass.h>
#include <Common/Base/Reflection/hkInternalClassMember.h>
#include <Common/Base/Reflection/hkTypeInfo.h>
#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/MyPhantomShape.h>
#define True true
#define False false



//
// Enum MyPhantomShape::ActionType
//
static const hkInternalClassEnumItem MyPhantomShapeActionTypeEnumItems[] =
{
    {0, "ACTION_WIND"},
    {1, "ACTION_ATTRACT"},
    {2, "ACTION_DEFLECT"},
};
static const hkInternalClassEnum MyPhantomShapeEnums[] = {
    {"ActionType", MyPhantomShapeActionTypeEnumItems, 3, HK_NULL, 0 }
};
extern const hkClassEnum* MyPhantomShapeActionTypeEnum = reinterpret_cast<const hkClassEnum*>(&
  MyPhantomShapeEnums[0]);

//
// Class MyPhantomShape
//
static const hkInternalClassMember MyPhantomShapeClass_Members[] =
{
    { "actionType", HK_NULL, MyPhantomShapeActionTypeEnum, hkClassMember::TYPE_ENUM, hkClassMember::
      TYPE_INT8, 0, 0, HK_OFFSET_OF(MyPhantomShape,m_actionType), HK_NULL },
    { "direction", HK_NULL, HK_NULL, hkClassMember::TYPE_VECTOR4, hkClassMember::TYPE_VOID, 0, 0,
      HK_OFFSET_OF(MyPhantomShape,m_direction), HK_NULL },
    { "strength", HK_NULL, HK_NULL, hkClassMember::TYPE_REAL, hkClassMember::TYPE_VOID, 0, 0, HK_OFFSET_OF
      (MyPhantomShape,m_strength), HK_NULL }
};
extern const hkClass hkpPhantomCallbackShapeClass;

extern const hkClass MyPhantomShapeClass;
const hkClass MyPhantomShapeClass(
    "MyPhantomShape",
    &hkpPhantomCallbackShapeClass, // parent
    sizeof(MyPhantomShape),
    HK_NULL,
    0, // interfaces
    reinterpret_cast<const hkClassEnum*>(MyPhantomShapeEnums),
    1, // enums
    reinterpret_cast<const hkClassMember*>(MyPhantomShapeClass_Members),
    HK_COUNT_OF(MyPhantomShapeClass_Members),
    HK_NULL, // defaults
    HK_NULL, // attributes
    0, // flags
    0 // version
    );
HK_REFLECTION_DEFINE_VIRTUAL(MyPhantomShape);
```

We will see later on how to register this class. Notice that, since the three actions (`MyWindAction`,

`MyReflectAction` and `MyAttractAction`) are only runtime actions (that is, are created on the fly and never serialized) we don't need to create or register any `hkClass` for them.

### Creating our Convert To Phantom Action filter

Let's now create our filter. Filter (`hctFilterInterface`) classes are always paired with filter descriptors (`hctFilterDescriptor`). There is only one descriptor object per class, giving information about the type of filter (its name, category, etc.) while each filter object represent an individual filter in the filter manager.

#### 4.9.9.10 Example: convertophantomaction/hctConvertToPhantomActionFilter.h

```
#ifndef HAVOK_FILTER_CONVERT_TO_PHANTOM_ACTION_H
#define HAVOK_FILTER_CONVERT_TO_PHANTOM_ACTION_H

#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionOptions.h>

/*
** This filter is part of the "Toolchain Integration" tutorial of the Havok Content Tools.
** Check the Havok Content Tools documentation for details
*/
class hctConvertToPhantomActionFilter : public hctFilterInterface
{
    public:

        HK_DECLARE_CLASS_ALLOCATOR(HK_MEMORY_CLASS_EXPORT);

        // Constructor - takes a pointer to the filter manager that created the filter
        hctConvertToPhantomActionFilter(const hctFilterManagerInterface* owner);

        // Destructor
        /*virtual*/ ~hctConvertToPhantomActionFilter();

        // The main method in any filter, it does the job
        /*virtual*/ void process( class hkRootLevelContainer& data, bool batchMode );

        // Asks the filter to show its UI for options
        /*virtual*/ HWND showOptions(HWND owner);

        // Asks the fitler to close the options UI (and store any changes)
        /*virtual*/ void hideOptions();

        // Storage of option data
        /*virtual*/ void setOptions(const void* optionData, int optionDataSize, unsigned int version);
        /*virtual*/ int getOptionsSize() const;
        /*virtual*/ void getOptions(void* optionData) const;

    public:

        // The options dialog window
        HWND m_optionsDialog;

        // Called by other methods, it makes sure the stored options (m_options) match what the values in
          the UI
        void updateOptions();

        // The options associated with this filter
        hctConvertToPhantomActionOptions m_options;

        // A buffer for storing the options in XML form.
        mutable hkArray<char> m_optionsBuf;

};

// The filter descriptor is used by the filter manager to query information about the filter (name,
  category, etc)
// as well as to create individual instances.
class hctConvertToPhantomActionFilterDesc : public hctFilterDescriptor
{
    public:

        /*virtual*/ unsigned int getID() const { return 0x9376be9a; }
        /*virtual*/ FilterCategory getCategory() const { return HK_CATEGORY_USER; }
```

...continued from above:

```
        /*virtual*/ FilterBehaviour getFilterBehaviour() const { return HK_DATA_MUTATES_INPLACE; }
        /*virtual*/ const char* getShortName() const { return "Convert To Phantom Action"; }
        /*virtual*/ const char* getLongName() const { return "[TUTORIAL] Converts specific rigid bodies to
          phantom actions. Must be placed after the Create Rigid Bodies filter. "; }
        /*virtual*/ unsigned int getFilterVersion() const { return HCT_FILTER_VERSION(1,0,0); }
        /*virtual*/ hctFilterInterface* createFilter(const class hctFilterManagerInterface* owner) const {
          return new hctConvertToPhantomActionFilter(owner); }

        /*virtual*/ HavokComponentMask getRequiredHavokComponents () const { return HK_COMPONENT_PHYSICS;
          }
};

extern hctConvertToPhantomActionFilterDesc g_convertToPhantomActionDesc;

#endif // HAVOK_FILTER_CONVERT_TO_PHANTOM_ACTION_H
```

We've marked virtual methods defined in parent classes (`hctFilterInterface` and `hctFilterDescriptor`) and overriden here with the **/*virtual*/** comment in order to distinguish them from other methods.

Let's start with the filter descriptor, `hctConvertToPhantomActionFilterDesc`. As you can see (and as its name implies) one its responsibilities is to describe the filter (check the documentation for hctFilterDescriptor for details):

- It gives it a unique ID
- It assigns a category (the tab in the filter manager). For user-defined filter, we recommend you use the **HK_USER** category.
- It specifies how does the filter operate with data; in our case, it changes it.
- It returns a short name and a long name (description) for the filter
- It defines what Havok components are required to support the output of this filter. In this case, the Havok Physics 2012 component is required.
- It creates individual instances of the filter.

The only instance of the descriptor (`g_convertToPhantomActionDesc`) is declared here (and defined in `hkConvertToPhantom.cpp`)

Let's look at the filter, `hkConvertToPhantonActionFilter`. The most important method is `process()`. We will look at the other methods (related to the display and storage of filter options) later on when discussing filter options.

Let's see how to implement this `process()` method:

#### 4.9.9.11 Example: convertophantomaction/hctConvertToPhantomActionFilter.cpp

```
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>

#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionFilter.h>

#include <Physics2012/Utilities/Serialize/hkpPhysicsData.h>

#include <Common/SceneData/Graph/hkxNode.h>
#include <Common/SceneData/Attributes/hkxAttributeGroup.h>

#include <Common/Base/Math/Util/hkMathUtil.h>

#include <Physics2012/Dynamics/Entity/hkpRigidBody.h>
#include <Physics2012/Dynamics/World/hkpPhysics 2012System.h>

#include <Common/Serialize/Util/hkBuiltinTypeRegistry.h>
#include <Common/Base/Reflection/hkClass.h>
#include <Common/Base/Reflection/hkTypeInfo.h>

#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/MyPhantomShape.h>
#include <Physics2012/Collide/Shape/Misc/Bv/hkpBvShape.h>

// The only instance of our filter descriptor
hctConvertToPhantomActionFilterDesc g_convertToPhantomActionDesc;

hctConvertToPhantomActionFilter::hctConvertToPhantomActionFilter(const hctFilterManagerInterface* owner)
:    hctFilterInterface (owner)
{
    m_options.m_removeMeshes = false;
    m_optionsDialog = HK_NULL;
}

hctConvertToPhantomActionFilter::~hctConvertToPhantomActionFilter()
{

}

void hctConvertToPhantomActionFilter::process( hkRootLevelContainer& data, bool batchMode )
{
    // Find and hkxScene and a hkPhysics 2012 Data objects in the root level container
    hkxScene* scenePtr = reinterpret_cast<hkxScene*>( data.findObjectByType( hkxSceneClass.getName() ) );
    if (scenePtr == HK_NULL || (scenePtr->m_rootNode == HK_NULL) )
    {
        HK_WARN_ALWAYS(0xabbaa5f0, "No scene data (or scene data root hkxNode) found. Can't continue.");
        return;
    }
    hkpPhysicsData* physicsPtr = reinterpret_cast<hkpPhysicsData*>( data.findObjectByType(
      hkpPhysicsDataClass.getName() ) );
    if (physicsPtr == HK_NULL)
    {
        HK_WARN_ALWAYS(0xabbaa3d0, "No physics data found, you need to use Create Rigid Bodies before this
          filter or have bodies already in the input data.");
        return;
    }

    // Keep track of how many phantoms we created so we can reported
    int numConverted = 0;

    // Search for rigid bodies in the scene
    for (int psi=0; psi<physicsPtr->getPhysicsSystems().getSize(); psi++)
    {
        const hkpPhysicsSystem* psystem = physicsPtr->getPhysicsSystems()[psi];
```

. . . continued from above:

```
for (int rbi=0; rbi<psystem->getRigidBodies().getSize(); rbi++)
{
    hkpRigidBody* rbody = psystem->getRigidBodies()[rbi];
    const char* rbName = rbody->getName();

    // Require an associated node in the scene
    hkxNode* rbNode = (rbName)? scenePtr->findNodeByName(rbName): HK_NULL;
    if( !rbNode )
    {
        continue;
    }

    // Require an 'hkPhantomAction' attribute group
    const hkxAttributeGroup* attrGroup = rbNode->findAttributeGroupByName("hkPhantomAction");
    if (!attrGroup)
    {
        continue;
    }

    // Create our phantom shape
    MyPhantomShape* myPhantomShape = new MyPhantomShape();
    {
        // Set action type (required)
        const char* actionTypeStr = HK_NULL;
        attrGroup->getStringValue( "action", true, actionTypeStr );
        if( actionTypeStr )
        {
            if( hkString::strCasecmp( actionTypeStr, "wind" ) == 0 )
            {
                myPhantomShape->m_actionType = MyPhantomShape::ACTION_WIND;
            }
            else if( hkString::strCasecmp( actionTypeStr, "attract" ) == 0 )
            {
                myPhantomShape->m_actionType = MyPhantomShape::ACTION_ATTRACT;
            }
            else if( hkString::strCasecmp( actionTypeStr, "deflect" ) == 0 )

            {
                myPhantomShape->m_actionType = MyPhantomShape::ACTION_DEFLECT;
            }
            else
            {
                HK_WARN_ALWAYS(0xabbad3b4, "Unknow action type ("<<actionTypeStr<<").");
            }
        }
        else
        {
            HK_WARN_ALWAYS(0xabba9834, "Can't fine \"action\" attribute");
        }

        // Set other attributes (optional)
        attrGroup->getVectorValue( "direction", false, myPhantomShape->m_direction );
        attrGroup->getFloatValue( "strength", false, myPhantomShape->m_strength );

        // Useful warnings
        if ((myPhantomShape->m_actionType == MyPhantomShape::ACTION_WIND) && (myPhantomShape->
          m_direction.lengthSquared3()==0.0f))
        {
            HK_WARN_ALWAYS(0xabbadf82, "Wind direction is invalid - action will have no effect");
```

. . . continued from above:

```
                }
                if (myPhantomShape->m_strength==0.0f)
                {
                    HK_WARN_ALWAYS(0xabbacc83, "Strength is 0 - action will have no effect");
                }

            }

            // Set the phantom as a new bounding shape for the body
            {
                const hkpShape* oldShape = rbody->getCollidable()->getShape();

                hkpBvShape* bvShape = new hkpBvShape( oldShape, myPhantomShape );
                myPhantomShape->removeReference();

                rbody->setShape( bvShape );
                bvShape->removeReference();
            }

            // Remove meshes if the user chose to do so
            if (m_options.m_removeMeshes && (hkString::strCmp(rbNode->m_object.m_class->getName(), "
              hkxMesh")==0) )
            {
                rbNode->m_object.m_object = HK_NULL;
            }

            HK_REPORT("Converted rigid body \""<<rbName<<"\" to phantom action");
            numConverted++;
        }
    }

    // Give a warning if the filter didn't do anything useful
    if (numConverted==0)
    {
        HK_WARN_ALWAYS(0xabba7632, "No rigid bodies converted to phantom action.");
    }
    else
    {
        HK_REPORT("Converted "<<numConverted<<" rigid bodies.");
    }
}
```

The method takes an `hkRootLevelContainer` object (*data*), and a boolean (`batchMode`) flag. The *batchMode* flag tells the filter whether the user is processing in batch mode or not. When running in batch mode, filters should not open any UI or block the processing. Our filter won't do any of both, so we can safely ignore this flag.

The `hkRootLevelContainer` object (data) contains the contents that our filters works with and modifies. We start by looking for scene data (an `hkxScene` object; so we can look at the attributes) and physics data (an `hkpPhysicsData` object; so we can look at rigid bodies) inside it.

Once those two are found, we look at all rigid bodies (`hkpRigidBody`) objects in our physic data, and find the matching `hkxNode` in the scene (by comparing names). If that node has an `hkPhantomAction` attribute group, then we have a candidate to be processed. Otherwise we ignore it.

If we are succesful (we have an `hkpRigidBody` with an `hkPhantomShape` attribute group), we create our phantom shape (`MyPhantomShape`) and use methods in `hkxAttributeGroup` to query for attribute values and pass them to the phantom shape. We then wrap the original shape and the phantom with an

`hkpBvShape` (check the Havok Dynamics chapter for details).

The last step optionally removes the association node -> mesh (so the rigid body mesh won't be displayed in the Preview Tool filter). We will discuss how this option can be added to the filter in the following section (*Adding options to the filter*).

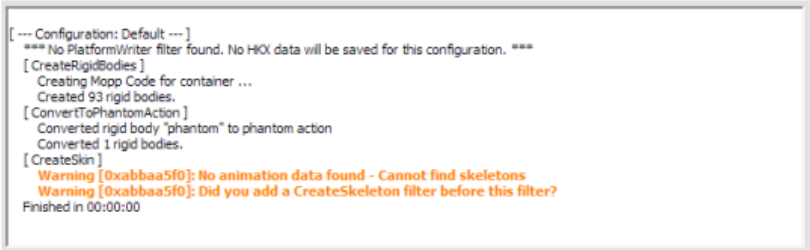
### Warnings and Reporting

When processing assets, it is very useful to report the activity of the filter and raise any warnings and errors that may be detected. In our example above, we raise warnings whenever processing fails or has no effect. We also report the name of any rigid body converted, as well as a total at the end of processing.

You can use the following macros to report progress and warnings:

- **HK_REPORT (text)** : Reports some information
- **HK_REPORT_SECTION_BEGIN (id, name)** : Starts a section (following **HK_REPORT** calls will be nested in this group)
- **HK_REPORT_SECTION_END ()** : Finishes a section
- **HK_WARN_ALWAYS (id, text)** : Reports a warning. The ID can be used to identify individual warnings. For toolchain warnings, we use the range `0xabba0000`–`0xabbaffff` as a convention.
- **HK_WARN (id, text)** : Reports a warning **in debug only** .

You can also use other SDK error reporting macros, like **HK_ASSERT** and **HK_ERROR** . These macros, however, will abort execution of the filter. For filter processing, it is usually a better and safer approach to detect, warn, and recover gracefully from unexpected situations rather than using assertions.




### Adding options to the filter

Let's now take a look at how to add options to our filter. In our case, we are going to give the option to remove the association of a rigid body node to its original mesh whenever it is converted to a phantom (so the phantom won't be displayed by the Preview Tool filter).

We will call this option "*removeMeshes*"; we saw in the previous section how to use it during the `process()` method. Let's see now how to store this option with the filter and how to display it in the filter manager.

### Storing options with a filter

Each filter gets an opportunity to save and retrieve option data, which is stored with the filter configuration (and usually saved with the asset, or into an .HKO file). Since options need to be serialized, we'll start by putting our option in a struct:

#### 4.9.9.12 Example: convertophantomaction/hctConvertToPhantomActionOptions.h

```
#ifndef HKCONVERTOPHANTOMOPTIONS__H
#define HKCONVERTOPHANTOMOPTIONS__H

/// hkConvertToPhantomOptions meta information
extern const class hkClass hctConvertToPhantomActionOptionsClass;

/// Options associated to our Convert To Phantom Action filter.
struct hctConvertToPhantomActionOptions
{
    HK_DECLARE_NONVIRTUAL_CLASS_ALLOCATOR( HK_MEMORY_CLASS_EXPORT, hctConvertToPhantomActionOptions );
    HK_DECLARE_REFLECTION();

    /// Option to remove associated meshes
    hkBool m_removeMeshes;
};

#endif // HKCONVERTOPHANTOMOPTIONS__H
```

As with our custom action, adding the **HK__DECLARE__REFLECTION()** macro flags this class so the serialization scripts will generate an associated `hkClass` object in a `.cpp`. If you run the `generateReflections.py` script (please check the *Serialization > Using the Infrastructure* documentation for details), the following CPP will be generated; add it to the project too:

#### 4.9.9.13 Example: convertophantomaction/Classes/hctConvertToPhantomActionReflections.cpp

```
// WARNING: THIS FILE IS GENERATED. EDITS WILL BE LOST.
// Generated from 'ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionOptions.h'
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>
#include <Common/Base/Reflection/hkClass.h>
#include <Common/Base/Reflection/hkInternalClassMember.h>
#include <Common/Base/Reflection/hkTypeInfo.h>
#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionOptions.h>
#define True true
#define False false



//
// Class hctConvertToPhantomActionOptions
//
static const hkInternalClassMember hctConvertToPhantomActionOptionsClass_Members[] =
{
    { "removeMeshes", HK_NULL, HK_NULL, hkClassMember::TYPE_BOOL, hkClassMember::TYPE_VOID, 0, 0,
      HK_OFFSET_OF(hctConvertToPhantomActionOptions,m_removeMeshes), HK_NULL }
};
extern const hkClass hctConvertToPhantomActionOptionsClass;
const hkClass hctConvertToPhantomActionOptionsClass(
    "hctConvertToPhantomActionOptions",
    HK_NULL, // parent
    sizeof(hctConvertToPhantomActionOptions),
    HK_NULL,
    0, // interfaces
    HK_NULL,
    0, // enums
    reinterpret_cast<const hkClassMember*>(hctConvertToPhantomActionOptionsClass_Members),
    HK_COUNT_OF(hctConvertToPhantomActionOptionsClass_Members),
    HK_NULL, // defaults
    HK_NULL, // attributes
    0, // flags
    0 // version
    );
HK_REFLECTION_DEFINE_SIMPLE(hctConvertToPhantomActionOptions);
```

Notice in `hctConvertToPhantomActionFilter.h` how an instance of this `hctConvertToPhantomActionOptions` class (`m_options`) is stored in the filter, initialized in the constructor and used during processing. In order to store these options with the filter, three methods must be implemented:

- `int getOptionsSize ()`: Must return the size (in bytes) of the serialized XML representing the options
- `void getOptions (buffer)`: Must filled the passed buffer with the XML representing the options
- `void setOptions (buffer, size, version)`: The filter can read the given XML options from the buffer. The version parameter can be used to detect older versions of the options (and upgrade them accordingly).

Let's check how to implement these methods:

#### 4.9.9.14 Example: convertophantomaction/hkConvertToPhantomAction_Options.cpp

```
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>

#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionFilter.h>

void hctConvertToPhantomActionFilter::setOptions(const void* optionData, int optionDataSize, unsigned int
  version )
{
    // Get the options from the XML data.
    if ( hctFilterUtils::readOptionsXml( optionData, optionDataSize, m_optionsBuf,
      hctConvertToPhantomActionOptionsClass ) == HK_SUCCESS )
    {
        hctConvertToPhantomActionOptions* options = reinterpret_cast<hctConvertToPhantomActionOptions*>(
          m_optionsBuf.begin() );

        m_options.m_removeMeshes = options->m_removeMeshes;
    }
    else
    {
        HK_WARN_ALWAYS( 0xabba482b, "The XML for the " << g_convertToPhantomActionDesc.getShortName() << "
          option data could not be loaded." );
        return;
    }
}

int hctConvertToPhantomActionFilter::getOptionsSize() const
{
    // We write the options to a temporary buffer and return the size of it. The buffer itself will be
      used
    // later on by getOptions().
    hctFilterUtils::writeOptionsXml( hctConvertToPhantomActionOptionsClass, &m_options, m_optionsBuf,
      g_convertToPhantomActionDesc.getShortName() );
    return m_optionsBuf.getSize();
}

void hctConvertToPhantomActionFilter::getOptions(void* optionData) const
{
    // Get options is always called after getOptionsSize() - so we reuse the temporary buffer we used in
      getOptionsSize()
    hkString::memCpy( optionData, m_optionsBuf.begin(), m_optionsBuf.getSize() );
```

In order to read the options, we use the utility `hctFilterUtils::readOptionsXML()`, and then copy the temporary data in the buffer to our stored options, `m_options`.

For retrieving the options, we serialize them into a temporary buffer, returning its size in `getOptionsSize()` (which always called first) and its contents in `getOptions()`.

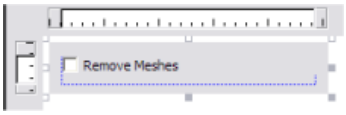
**Displaying Options**

Now that we know how to use, store and load our options, let's see how to show them in the filter manager UI. Two methods in `hctFilterInterface` are used for this:

- `HWND showOptions (HWND owner)`: Must open the filter options window inside the given window control, and it must return the dialog handle of the new created window.
- `void hideOptions()`: This is called whenever option editing must be finished (when closing the filter manager or when executing filters). The filter must make sure the stored options match the

displayed options.

We'll start by creating the dialog ( **IDD_CONVERT_TO_PHANTOM_ACTION_DIALOG** ) to show our options. In order to fit in the right side pane of the filter manager, the width of the dialog must be 153 pixels, and it has to be of *Child* style, with no border. We add a check box control for the "*Remove Meshes*" option, which we name **IDC_CB_RemoveMeshes** .




Let's see know how to show and hide this dialog:

#### 4.9.9.15 Example: convertophantomaction/hkConvertToPhantomAction_Dialog.cpp

```
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>

#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionFilter.h>

extern HINSTANCE hInstance;

BOOL CALLBACK _convertToPhantomActionDialogProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    // We store a pointer to the filter associated with this dialog using Get/SetWindowLongPtr()
    hctConvertToPhantomActionFilter* filter = reinterpret_cast<hctConvertToPhantomActionFilter*> ( (
      hkUlong) GetWindowLongPtr(hWnd,GWLP_USERDATA)) ;

    switch(message)
    {
        case WM_INITDIALOG:
            {
                filter = (hctConvertToPhantomActionFilter*)lParam;
                SetWindowLongPtr(hWnd, GWLP_USERDATA, (LONG)lParam); // so that it can be retrieved later

                CheckDlgButton(hWnd, IDC_CB_RemoveMeshes, filter->m_options.m_removeMeshes);

                return TRUE; // did handle it
            }
    }
    return FALSE; //didn't handle it
}

void hctConvertToPhantomActionFilter::updateOptions()
{
    // Ensure the options we store match the options shown in the UI
    if (m_optionsDialog)
    {
        m_options.m_removeMeshes = IsDlgButtonChecked(m_optionsDialog, IDC_CB_RemoveMeshes) == TRUE;
    }
}


HWND hctConvertToPhantomActionFilter::showOptions(HWND owner)
{
    if (m_optionsDialog)
    {
        hideOptions();
    }

    m_optionsDialog = CreateDialogParam(hInstance, MAKEINTRESOURCE(IDD_CONVERT_TO_PHANTOM_ACTION_DIALOG),
        owner, _convertToPhantomActionDialogProc, (LPARAM) this );

    return m_optionsDialog;
}

void hctConvertToPhantomActionFilter::hideOptions()
{
    // Update any changes before we close UI
    updateOptions();

    if (m_optionsDialog)
    {
        DestroyWindow(m_optionsDialog);
    }

    m_optionsDialog = NULL;
```

. . . continued from above:

```
}
```

We use `CreateDialogParam()` and pass a pointer to the filter so the dialog proc (`convertToPhantomActionDialogProc`) can properly access the filter. Notice how, when closing the dialog, we make sure that whatever changes the user did in the UI are reflected in the stored options.

**The filter DLL**

Now that we have created our filter and its descriptor, we need to expose them to the DLL so the filter manager can detect and load them. We do so by inheriting a custom class from `hctFilterDll` (see handling dlls for details):

#### 4.9.9.16 Example: hctFilterTutorialDll.h

```
#ifndef HAVOK_FILTER_TUTORIAL_DLL_H
#define HAVOK_FILTER_TUTORIAL_DLL_H

#include <ContentTools/Common/Filters/Common/Filter/hctFilterDll.h>

// This class describes our filter DLL : It provides access to the filter descriptor
// for each filter implemented in the DLL, and registers all classes possibly created in those
// filters.
class hctFilterTutorialDll : public hctFilterDll
{
    public:

        hctFilterTutorialDll (HMODULE dllModule);

        // Must return how many types of filters are implemented in this DLL
        /*virtual*/ int getNumberOfFilters() const;

        // For each filter implemented in this DLL, we must return a filter descriptor
        /*virtual*/ hctFilterDescriptor* getFilterDescriptor (int index) const;

        // There are some statics in the Physics 2012 SDK that we want to make sure are initialized
        /*virtual*/ void initDll ( const hkMemoryInitUtil::SyncInfo& baseSystemInfo, hkError* errorStream
          );
};


// Function exported by the DLL
__declspec( dllexport ) hctFilterDll* getFilterDll (HMODULE dllModule);

#endif // HAVOK_FILTER_TUTORIAL_DLL_H
```

We need to override `getNumberOfFilters()` and `getFilterDescriptor()`, since they are pure virtual methods in `hctFilterDll`, and we need to return our own filter descriptor. We also override `registerClasses()` since we must register any serializable classes possibly created by our filters, and `initDll()` in order to initialize some static data used by Havok Physics 2012.

Notice also how we declare `getFilterDll()` as a function exported by this DLL. This is the only function exported by the DLL.

#### 4.9.9.17 Example: hctFilterTutorialDll.cpp

```
#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorial.h>

#include <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorialDll.h>

#include <Common/Serialize/Util/hkBuiltinTypeRegistry.h>


#include <Physics2012/Collide/Util/Welding/hkpWeldingUtility.h>
#include <Physics2012/Dynamics/World/hkpWorldCinfo.h>


// Filters we want to expose in this dll:
#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/
  hctConvertToPhantomActionFilter.h>

__declspec( dllexport ) hctFilterDll* getFilterDll (HMODULE dllModule)
{
    static hctFilterTutorialDll gFilterAssetDll (dllModule);

    return &gFilterAssetDll;
}


hctFilterTutorialDll::hctFilterTutorialDll (HMODULE dllModule) : hctFilterDll (dllModule)
{

}

/*virtual*/ int hctFilterTutorialDll::getNumberOfFilters() const
{
    return 1;
}

/*virtual*/ hctFilterDescriptor* hctFilterTutorialDll::getFilterDescriptor (int index) const
{
    static hctFilterDescriptor* m_filterDescs[] =
    {
        &g_convertToPhantomActionDesc,
    };

    return m_filterDescs[index];
}

/*virtual*/ void hctFilterTutorialDll::initDll ( const hkMemoryInitUtil::SyncInfo& baseSystemInfo, hkError
  * errorStream )
{
    hctFilterDll::initDll(baseSystemInfo, errorStream);

    if (isInitialized())
    {
        hkpWorldCinfo defaults;
        hkpWeldingUtility::initWeldingTable(defaults.m_snapCollisionToConvexEdgeThreshold,
          defaults.m_snapCollisionToConcaveEdgeThreshold);
    }
}

// REGISTER ALL COMPLETE CLASSES
#define HK_CLASSES_FILE <ContentTools/Common/Filters/FilterTutorial/hctFilterTutorialClasses.h>
#include <Common/Serialize/Util/hkBuiltinTypeRegistry.cxx>
```

The implementation of the DLL-exported method `getFilterDll()` just returns a pointer to an statically

defined instance of `hctFilterTutorialDll`, and `getFilterDescriptor()` just returns a pointer to the static descriptor declared in `convertophantomaction/hctConvertToPhantomActionFilter.h`.

The `hctFilterTutorialClasses.h` file contains all the classes that need to be registered. Notice how we also register our custom shape class, `MyPhantomShape`.[10]

```
// hctFilterTutorialClasses.h
#include <Physics2012/Collide/Classes/hkpCollideClasses.h>
#include <Physics2012/Dynamics/Classes/hkpDynamicsClasses.h>
#include <Physics2012/ConstraintSolver/Classes/hkpConstraintSolverClasses.h>
#include <Physics2012/Internal/Classes/hkpInternalClasses.h>
#include <Physics2012/Utilities/Classes/hkpUtilitiesClasses.h>
// Our phantom class
HK_CLASS(HK_EXPORT_MYGAME,MyPhantomShape)
```

Finally, we create an `hkfiltertutorial.def` file with our DLL exports:

#### 4.9.9.18 Example: hctFilterTutorial.def

```
LIBRARY hctFilterTutorial
EXPORTS
    getFilterDll    @1

SECTIONS
    .data READ WRITE
```

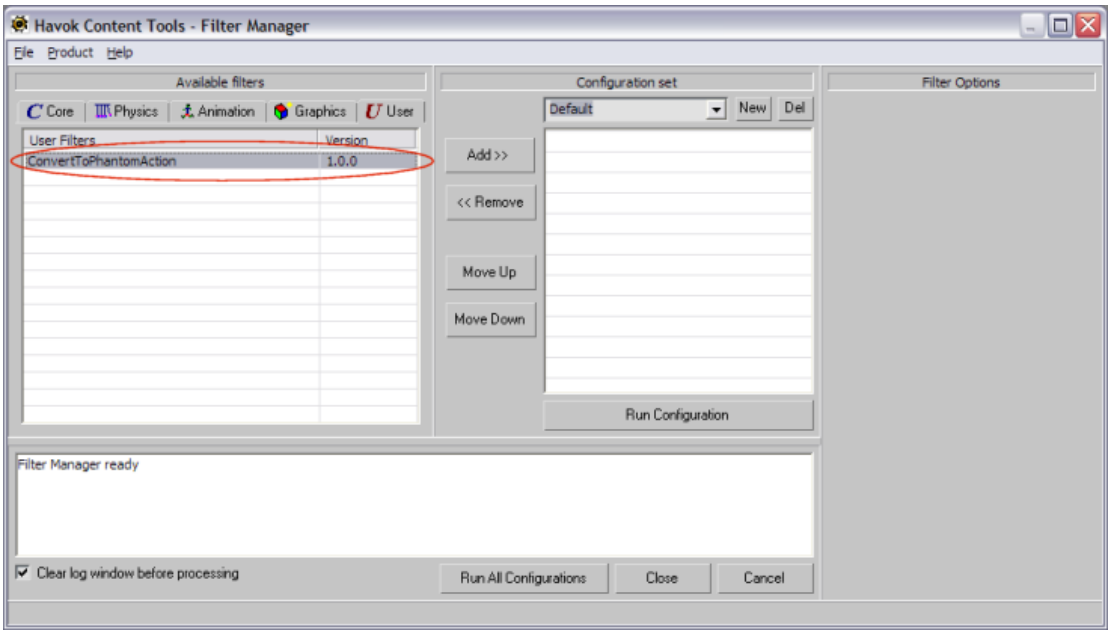
And we are done!

**Placing the filter DLL**

Make sure you place our new filter DLL in the right folder: look for a "`filters`" folder inside the install folder of the Havok Content Tools (usually "`%ProgramFiles%/havok/HavokContentTools/filters`") and place it there[11]. If you've done that properly, the filter should appear in the right category ("*User*" in our case)

[10] It is very important not to miss any created class, and there is no overhead with registering classes that are actually not created. So in this case we register all hkCollide classes (since we create some of them, like hkpBvShape) even though there will be many classes which we don't create.

[11] The DLL is actually installed by the Havok Content Tools installer so you can check it out before writing it yourself. Feel free to replace it with your custom one.

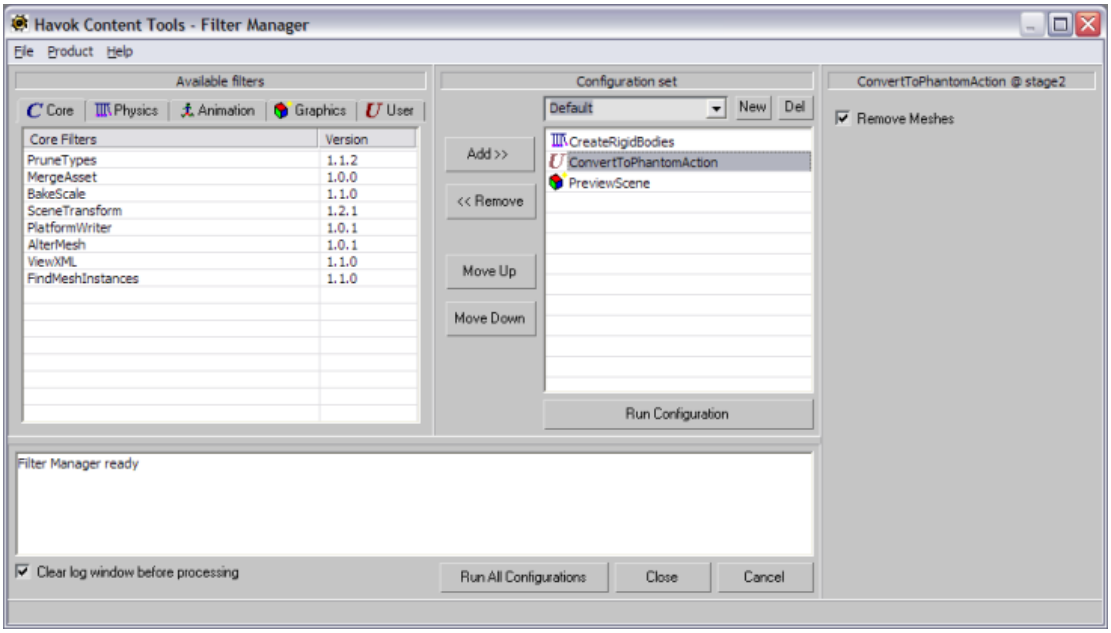



#### 4.9.9.19 Processing

Now that we have the content created and the filter ready, let's set up the processing to create our phantom action in the scene we started with.

Open the filter manager and add the following filters: *Create Rigid Bodies* (*Physics 2012* Category) , *Convert To Phantom Action* (*User*) and *Preview Tool* (*Graphics*):

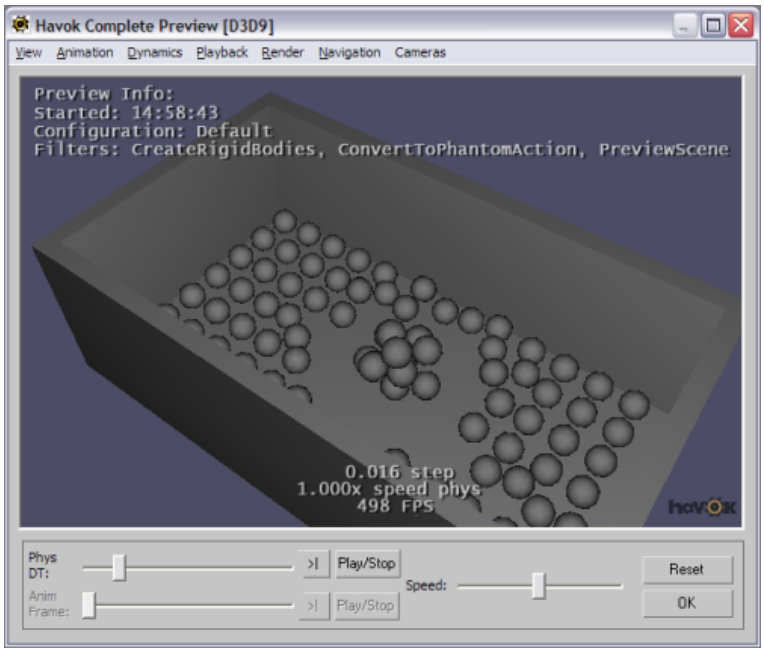
In the *Convert To Phantom Action* filter, check the " **Remove Meshes** " options.



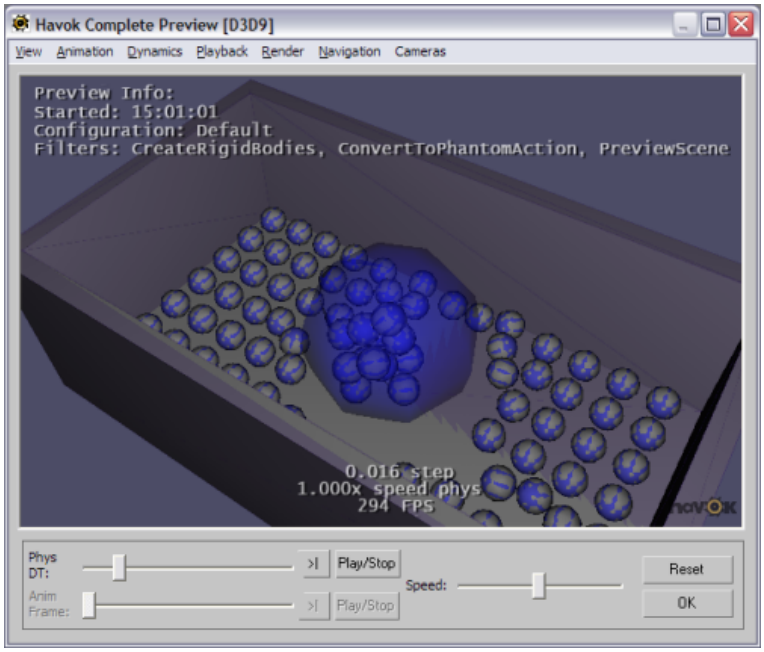
In order for the Preview Tool filter to run, it needs to be rebuilt with our custom shape class `MyPhantomShape` registered with the `PreviewAssetManager` class (in `hctPreviewAsset.cpp`).

```
// Provides the implementation for the phantomEnterEvent and the phantomLeaveEvent methods
#include <ContentTools/Common/Filters/FilterTutorial/ConvertToPhantomAction/MyPhantomShape.cpp>
// Contains the associated hkClass and hkTypeInfo for the MyPhantomShape class
#include <ContentTools/Common/Filters/FilterTutorial/Classes/hctFilterTutorialReflectedClasses.cpp>

extern const hkClass MyPhantomShapeClass;
extern const hkTypeInfo MyPhantomShapeTypeInfo;

// Registers the class (should be called in the PreviewAssetManager constructor)
hkBuiltinTypeRegistry::getInstance().addType(&MyPhantomShapeTypeInfo, &MyPhantomShapeClass);
```

If you press the **Run Configuration** button, the filters will be executed and the Preview Tool filter will open a preview window. Press **Play** to start the simulation:




Notice how the spheres entering the volume of our phantom are now attracted to its center, and how the mesh for the phantom is not displayed. You can use the rigid body ghosts display to view the phantom shape. Click on **View** > **Physics 2012 Ghosts** :




And we are done! Feel free to experiment with the different action types and different parameter values.

#### 4.9.9.20 Going further

In this tutorial we've focused on the basic ideas behind extending the toolchain both on the modeler side and the filter processing side. Here are some other suggestions and improved ways to extend the toolchain

**More powerful attribute definitions**

In our tutorial we showed how to attach extra attributes to nodes using each of the modeler's UI. This had the advantage of not requiring any knowledge about scripting or plugin development. However, in order to provide more powerful and reusable attribute definitions and UI, you will need to resort to each of the modeler-specific ways to define types. We will mention a few of them:

**3ds Max**

- In 3ds Max you can create Custom Attribute Definitions using the MAXScript language (and to some extent, C++). This has the advantatge of being able to customize the UI behaviour, handle versioning, and reuse the attribute definition across multiple objects.
- Modifiers are also inspected during export and exported as attribute groups, so you can use custom modifiers to add attributes to objects. Modifiers are the most powerful option; they have the advantage of providing callbacks for display and functionality for manipulation. The Havok Content Tools for 3ds Max use custom modifiers for their rigid body, shape and constraint attributes.

Please check the exporting custom data from 3ds Max for details on how the scene is explored and attributes exported in 3ds Max. Check the 3ds Max SDK and MAXScript documentation for details on how to create Custom Attributes and modifiers.

**Maya**

- In Maya you can use MEL to automate the creation of custom attributes, the disadvantage being that you have little control on how they are displayed and versioned.
- The most powerful option in Maya is to create custom nodes (shape or locators) using C++; the exporter will export them (if named and connected properly) as attribute groups associated with the node. Among the advantages are the fact that you can customize the attribute UI (using templates), versioning, and, in case of locators, you can also have a custom display in the viewports. The Havok Content Tools for Maya use custom nodes for their rigid body, shape and constraint attributes.

Please check the exporting custom data from Maya for details on how the scene is explored and attributes exported in Maya. Check the Maya documentation for details on how to use MEL and the C++ SDK to create attributes, custom nodes and locators.

**XSI**

- You can automate the attribute creation process by using scripting to add the attributes; this, however, still doesn't provide much flexibility regarding UI behaviour.
- You can create Custom Properties (Parameter Sets) with self-installing plugins, using any of the languages (C++, VBScript, JScript, Python, etc.) supported by XSI. Doing this has the advantage that you can customize the layout and event handling of the property pages (PPG) when working with those properties. The Havok Content Tools for XSI use Custom Properties for their rigid body, shape and constraint attributes.

Please check the exporting custom data from XSI for details on how the scene is explored and attributes exported in XSI. Check the XSI SDK documentation for details on how to create attributes and custom properties.

**Extending current types**

If you want to extend an already defined attribute group with a custom parameter, you can either:

- Modify the original definition of the attribute, or
- Attach an extra attribute group named as "*xxxxMerge*", where "*xxxx*" is the name of the original one: the exporters will merge the new attributes into the original group.

In order to handle the extra attribute during processing, you can either extend the original filter to handle the new attribute or create a new filter that takes the output from the previous filter and then interprets the new attribute (very similar to what we have done in this tutorial).

Sometimes it may be easier to extend the toolchain with transformations rather than new types. Consider the example of our tutorial: rather than creating a completely new "`hkPhantomShapeAction`" type and filter process, we reused the data and processing for rigid bodies, and then applied a "convert rigid body to phantom" process. Similarly, if you are considering adding a new type to the toolchain, it may be worth considering the option of treating the new type as a "mutation" of a previous type.

**Other examples**

This tutorial showed how to extend the toolchain with a custom action but the same mechanism is applicable to other toolchain extensions. Here are some examples:

**Using user-defined "materials" to override rigid body properties**

If you wanted to override the mass, friction and restitution of rigid bodies by a value based on a choice of predefined "materials", the steps would be very similar to those explained in this tutorial:

- Define a custom attribute group (`hkCustomMaterial` for example) with a Material attribute (an enum) to the modeler, adding it to those rigid bodies you want to apply the material to
- Create a custom filter (*Apply Custom Material* for example ) that, whenever this attribute group is found, it overrides the mass, friction and restitution of the rigid body with values stored in a "material table". The object's mass, for example, could be calculated from the material's density and the rigid body's volume.

The "material table" could be statically defined in the filter, exposed to filter options, it could be based on external text files, etc.

**Replicating a rigid body at multiple marker points**

If you wanted the ability to specify arbitrary points in space to represent locations where a rigid body should be replicated, the steps would again be very similar:

- Define a custom attribute group (`hkRigidBodyClone` for example) with a RigidBodyName attribute (a string[12]), and apply it to dummy objects in the scene representing the location where the rigid body should be cloned
- Create a custom filter (*Clone Rigid Bodies* for example) that detects nodes in the scene with that attribute group, searches for a rigid body with the specified name, and clones it at the location of the node.

## 4.10 Artist Best Practices

### 4.10.1 Introduction

This chapter contains advice and information on best practices for artists when setting up assets, both in terms of optimal behavior and runtime performance.

### 4.10.2 [Physics] Collision Detection Optimizations

We will explore how to make your levels more efficient for use with Havok. The first task is to understand the general operations of the Havok collision detection system, and what issues in a scene can lead to poor performance.

[12] Or a "node reference" if the modeler supports that type (3ds Max for example). Node references are converted to strings during export.

First, let's take a quick look at the different levels of collision detection that happen in the simulation world. Havok breaks collision detection up into three phases: broadphase, midphase and narrowphase.

#### 4.10.2.1 Collision Level: Broadphase

**Key Points** :

- Merge all fixed rigid bodies into a single rigid body wherever possible.
- Keep active dynamic rigid bodies to a minimum.

The broadphase is the highest level collision detection system, and it is used to quickly identify pairs of bodies that could potentially be colliding. Each rigid body is represented by a box (an AABB to be more precise, which is a box aligned to the world axes, i.e. never allowed to rotate).

For the most part, you do not need to understand the inner workings of the broadphase. Simply remember that having fewer rigid bodies in the world is better because the broadphase is less cluttered. While this is true for both fixed and moving rigid bodies, when designing the level you should focus more on limiting the number of **active** moving rigid bodies in the scene.

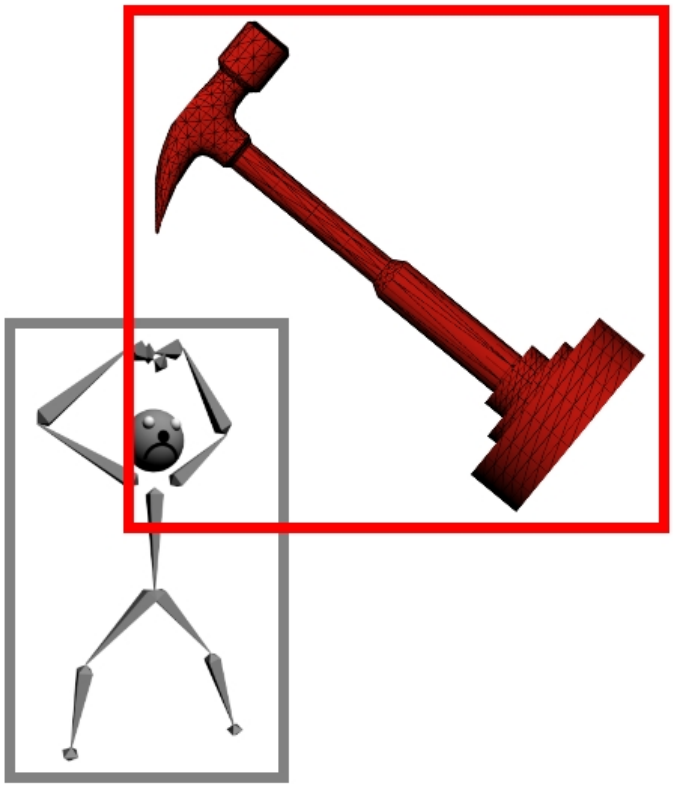


Figure 4.49: Broad-phase collisions (using AABBs)

For each pair of rigid bodies identified in the broadphase, we take a closer look to see if they are actually colliding. This is called the *narrowphase* collision detection: objects are compared on a shape by shape basis (a rigid body being formed of one or more primitive shapes). For interactions of simple rigid bodies

(formed of a single Primitive Shape: Sphere, Capsule, Box, Convex Hull, Cylinder or Triangle) only one comparison is carried out.

For rigid bodies formed of a **compound shape** (i.e. a collection of shapes), each individual shape is equivalent collision-wise to a separate simple rigid body. Similarly for a mesh shape (a shape formed of a triangle list), each triangle involved in a *narrowphase* collision is equivalent to (if not more expensive than) a whole separate simple rigid body.

Before discussing the *narrowphase*, we'll explore the *midphase*, which aims to reduce the number of individual shape collision comparisons that must be performed when colliding compound shapes.

#### 4.10.2.2 Collision Level: Midphase (Physics 2012 Only)

**Key Point** :

- Wrap all Compound Shapes (i.e. shape collections) in a MOPP structure.

In Physics 2012 Havok uses a separate collision detection step (the midphase) to minimize the cost associated with complex shapes collisions (such as triangle landscapes). In general, any shape collection of more than a few objects should be wrapped up in a MOPP. This is especially applicable for triangle landscapes. When wrapped in a MOPP, a rigid body can be quickly tested for potential collisions against other rigid bodies.

If potential collisions are identified, then only the relevant shapes in this body will be further tested. Consider the example above with the character and the hammer. Without a MOPP around the hammer, every individual triangle forming the hammer would be tested against each shape of the character. With a MOPP, none of the triangles would be tested.

If a portion of the hammer was actually colliding with the character's arm, only the triangles "near" the arm would be tested further for collisions.

MOPPs are primarily intended for large fixed landscapes, but it is applicable to dynamic shape collections as well. However, we also advise against the use of complex moving shape collections, and would recommend investigating if a convex hull would better suit the collision quality needs for such an object.

Please refer to the documentation on the Create Rigid Bodies (Physics 2012) filter for information on how to automatically create MOPPs for shape collections (also known as compound shapes).

#### 4.10.2.3 Collision Level: Narrowphase

**Key Points** :

- Avoid unnecessary details in your collision shapes whenever possible.
- Keep the potential number of collision calculations low by using large triangles in the collision landscape and by maintaining shape collections that are as simple as possible.
- Remove all physical detail in regions where physical objects and characters can't enter.

The final collision detection step (narrowphase) is carried out between the colliding objects' basic parts: capsule vs. triangle, capsule vs. capsule, box vs. box, box vs. triangle, etc. Take the example of a table made of a collision list of 3 boxes and let's consider the situation where the broadphase box (AABB) for that table overlaps with every rigid body of a 12 body ragdoll. We'll then obtain 36 collision calculations taking place every frame the bodies are moving, which is not insignificant.

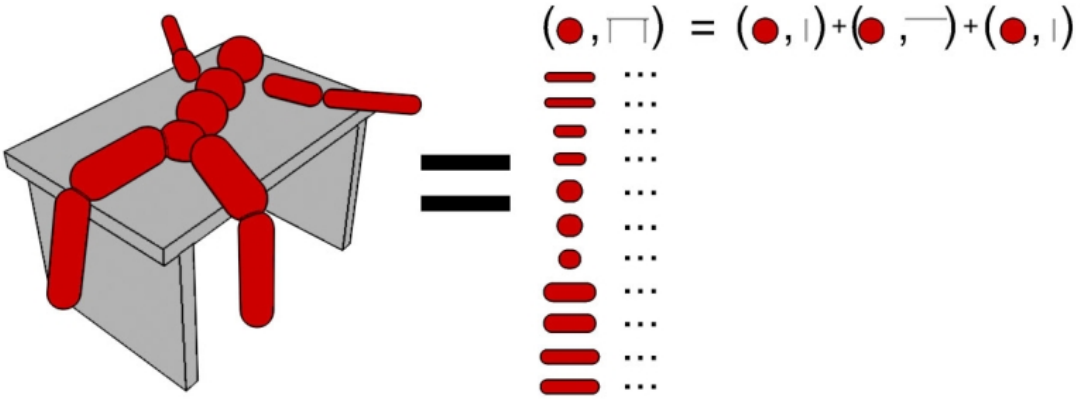


Figure 4.50: Ragdoll colliding with a table.

Now consider the case where the same ragdoll has been draped across a highly triangulated statue in a level. The bounding box for each shape in the ragdoll might overlap 50 triangles (and possibly even more). This works out to 50 * 12 collisions = 600 collision calculations for the ragdoll colliding with the statue.

The ragdoll might seem like an example that people would expect to be costly but imagine the statue has 400 triangles in total and a player walks up against the statue so that the capsule's bounding box (A.I.s and Player Character are usually simulated using capsules) overlaps most of the triangles in the statue. That's 400 collision calculations (or more) per frame for that character just because the character walked close to the statue – the same cost as calculating collisions for 400 capsules falling on one large triangle!

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

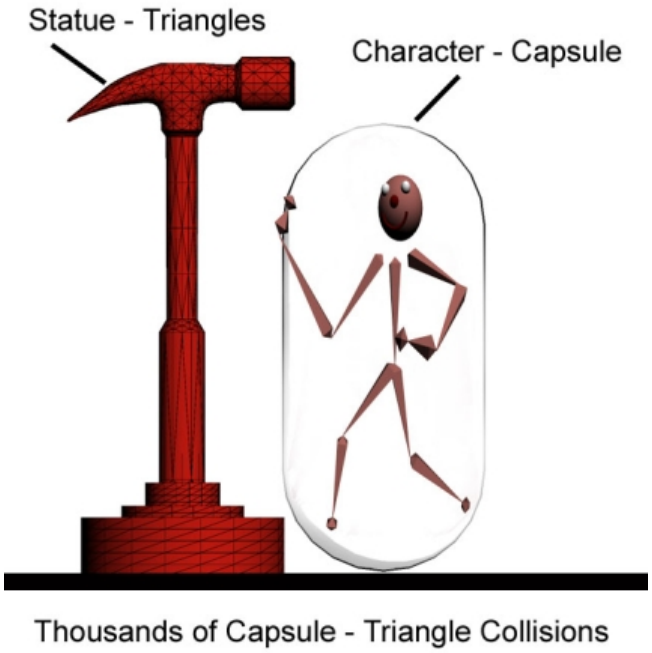



Figure 4.51: Collisions between a capsule and a large triangle list.

During the narrowphase, final collision calculations are carried out at the most basic shape (e.g. triangle) level. This means that when you're calculating potentially expensive areas of narrowphase collision detection, consider each fixed triangle, box, capsule, etc. in the scene as a potential collision object.

You want to avoid scenarios where a moving object or the character could collide against multiple fixed rigid bodies in the scene.

It's hard to let go of detail but in most cases it really won't affect the physical feel of the game. If you can make a chair with two simple boxes, then do so. How many objects in the scene are small enough or would last long enough to actually get in between the legs of the chair? Always ask yourself: is this amount of detail necessary to maintain realistic behavior. Always convert detailed triangle areas to the coarsest possible representation that would still perform the basic requirements of collision detection.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

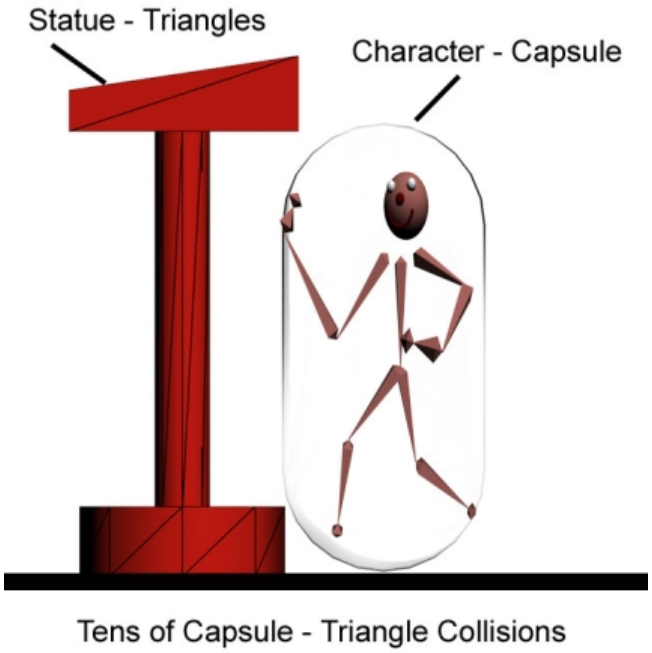



Figure 4.52: Collisions between a capsule and a simpler triangle list.

#### 4.10.2.4 The Big Collision Win

**No More Triangles!**

We've already discussed how each triangle is equivalent to one collision shape. Havok supports many other types of collision primitives. If these could be used to represent regions of your scene there is an opportunity for large CPU cost savings.

The main advantage to using these alternative objects is that certain objects can be represented using far fewer pieces than if they were constructed using triangles. Consider the previous example of the statue. Even with the representation optimized, the statue will contain nearly 100 triangles; at best a character might overlap only twenty of these but imagine instead if the statue was made up of a cylinder, a capsule and a box, the same collisions can be evaluated using only three collision calculations.

Confidential Information of Havok. ©Copyright 1999-2014 Havok. All Rights Reserved. 

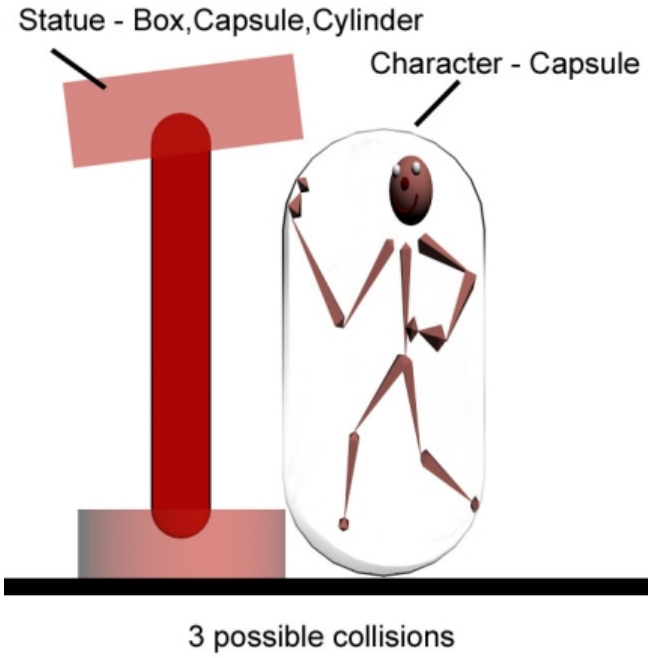



Figure 4.53: Collisions between a capsule and a list of simple shapes.

If you have a rectangular floor formed of triangles, we highly recommend using a single large box for its collision representation instead.

There are other benefits to replacing triangles with simple collision shapes. Triangles are thin and have no concept of "inside" or "outside" in terms of collision. When they are used to enclose volumes they can trap and hold moving bodies. Additionally, if an object sufficiently penetrates a triangle the proper collision response can be to push the object through to the other side. There are various ways this can happen and various methods to prevent it but it is best to simply avoid these situations all together.

Boxes, spheres, capsules, cylinders and convex hulls define collision volumes. The thicker the shape the less likely it is that moving rigid bodies will penetrate it sufficiently for them to pass through.

Consider the following floor made of triangles with a small box falling towards it. When the floor is made of triangles and the box is falling downwards, even though the triangles have some thickness the box only has to pass half way through the triangles in one step before it will be pushed out the bottom of the triangles and probably out of the world.

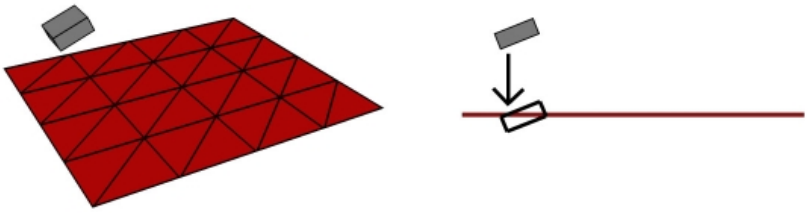


Figure 4.54: A box colliding against a triangle floor.

Now if we replace the floor triangles with a thick box:

Figure 4.55: A box colliding against a box floor.

The moving box is less likely to move through the floor box in one step before the collision detection catches it and the system solves the collision and pushes the box out correctly.

Consider the scenario where a rigid body is activated at runtime and added to the world. What happens if this new rigid body is penetrating the landscape triangles (when it is added)? The object gets pushed in between the floor triangles which can cause jitter, or even worse: an object falling through the landscape.

Figure 4.56: A box being pushed through a triangular floor.

If the floor is instead simulated using a fixed box then, on addition, the collision will be handled correctly and the newly added rigid body will quickly be pushed out correctly into the world.

Figure 4.57: A box being pushed back out of a box floor.

These volumes can be also be used to prevent constrained systems of rigid bodies (e.g. ragdolls) achieving undesirable poses. For example a seat modeled as part of the triangulated landscape could hold constrained bodies (upper arms / forearms) apart. If the seat were modeled as a box then nothing could pass through into the space under the seat.

That being said, it's not always possible to replace triangles with collision volume shapes. The need to model accurate holes in floors, to model curved surfaces and to store per-triangle collision properties that can be changed on the fly means that triangles are at times the simplest option.

#### 4.10.2.5 Summary and General Tips

To recap, here are the best ways to optimize your scenes and rigid bodies for performance:

- **Minimize the number of rigid bodies** in the scene (to reduce broadphase cost).
- **Minimize the complexity of rigid bodies** (to reduce narrowphase cost).
- Do not use visual geometry for the collision representation. In almost all cases, it is simply overkill. Use simple collision shapes to broadly define the collision representation.
- Use simpler geometry representations wherever possible. Minimize the number of triangles whenever you can.
- Favor simple shapes over convex hulls and triangles/meshes (see the collision cost table below).
- Avoid overlapping or redundant shapes. In particular, shapes fully contained within another are simply wasted collision processing time.
- Avoid using shapes that are too small, especially in collision list shapes. Small shapes have an increased risk of passing through triangle shapes, and getting "stuck" in a landscape.
- Keep the numbers of objects in constrained systems to a minimum. All shapes within a constrained system are simulated together.

| Performance Cost | Shape Type |
|---|---|
| Cheapest | Sphere |
| | Capsule |
| | Box |
| | Triangle |
| | Cylinder |
| | Convex Hull |
| Most Expensive | Meshes/Triangle List |

Table 4.19: Object Collision Cost - Rule of Thumb

**Avoid Using Concave Shapes**

Use convex shapes whenever possible. It is usually better to represent a concave object as either a list of convex shapes or a convex hull. Havok provides two very useful tools for this purpose:

- **Convex Decomposition Tool** : Concave meshes should always be simplified for use in Havok. In many cases (such as the Hammer object above), it can easily be approximated with a set of simple shapes. This is the recommended approach. In situations where greater detail is required, Havok has provided the convex decomposition tool. This tool will take in any concave object and produce a set of convex hulls which approximate the shape within some tolerance. For more information please refer to the documentation for the convex decomposition tool.

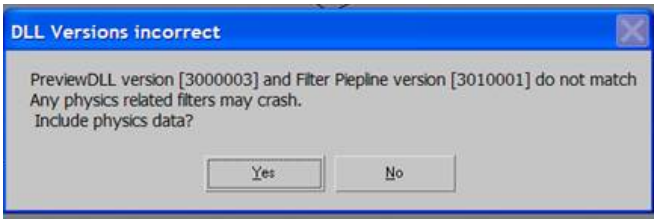
  Please note that even after decomposing objects, we highly recommend combining these individual shapes into a single rigid body (using a Compound Shape) whenever possible.
- **Convex Hull Tool** : Great speed improvements can be gained by avoiding the use of Meshes whenever possible. For convex meshes, this can easily be obtained by changing the type of the shape to **convex hull** . Internally, this will automatically generate a convex hull for any object. A convex hull is always preferred to a compound shape (shape collection) in terms of performance. You can also generate an explicit convex hull using the Convex Hull Tool. This will allow you to customize the hull to your needs. For more information please refer to the convex hull tool documentation for 3DS Max, Maya or XSI.

## 4.11 Troubleshooting

### 4.11.1 Troubleshooting: Processing Assets

#### 4.11.1.1 "DLL Versions Incorrect" dialog appears during export

If when exporting the following dialog appears:




This means that you have the Legacy Physics 2012 Exporter installed and the current scene has physics content created with it. The Legacy Physics 2012 Exporter creates its run-time content by using a DLL (`hkpreview.dll`) which contains the latest Havok runtime. The Scene Exporters for max and maya are able to detect the presence of the Legacy Physics 2012 Exporter and will attempt to merge the physics content created with it with the exported scene. However, in order for this to work properly, it is very important that the (`hkpreview`) DLL is built with the same Havok SDK version as the filters and filter manager.

Usually the installer for the Havok Content Tools will replace the `hkpreview` DLL with the proper version. But if, for example, you installed the Legacy Physics 2012 Exporters after the Havok Content Tools, an incompatible version may be installed. You can either:

- Reinstall the Havok Content Tools so they update the `hkpreview` DLL, or
- Copy the `hkpreview.dll` manually from the Havok SDK installation (in the "`tools`" folder) into the proper location for your modeller. The `ReadMe.txt` file located in the same directory contains directions for 3ds Max and Maya.

You can also ignore any legacy physics content by switching off the " **Invoke Legacy Physics 2012 Exporter** " option in the Scene Exporter Options (check the related documentation for the 3ds Max and Maya Scene Exporters)

#### 4.11.1.2 Filter Manager doesn't appear after exporting the scene

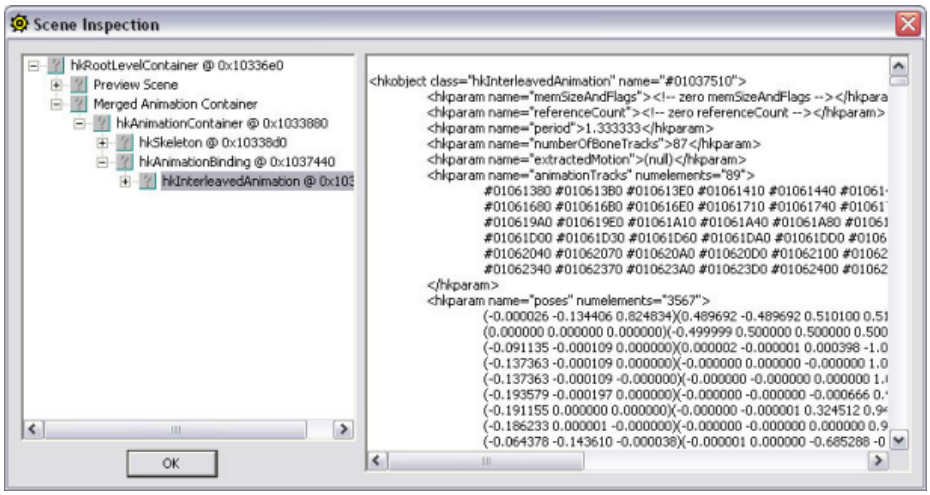
If the Filter Manager dialog does not appear then it's likely that processing has failed during export.

- If you have built a custom version of the export utilities then use normal modeller plugin debugging procedures.
- If you have not modified the export utilities then send the original asset (`.max`, `.mb` file) to support for analysis.

#### 4.11.1.3 Asset is processed, but does not load at run-time

If the asset is processed successfully through the filter pipeline, but it does not load at runtime:

- Ensure the correct versions of the filters and filter manager are installed. Version information for filters is listed beside each filter in the available filters window. Version information for the filter manager can be found in the **Help** > **About** menu.
- Check the contents of the asset using the standalone filter manager:
  - Launch the standalone filter manager and load the required asset. As you load assets the title bar in the Loading dialog box changes to reflect the number of files successfully loaded. If you are attempting to load files check the output console window. A warning is shown for each specified file which fails to load. Failure usually occurs because with the file is simply incorrect (i.e., does not contain Havok data) or is the incorrect version.
  - Once the application has launched you will be presented with the filter manager. Include a View XML filter in the stack and use this to browse the asset contents. Ensure that the expected data is contained within the asset and has not been pruned. The image below shows an example of a scene being investigated




#### 4.11.1.4 Create Bodies filter doesn't create any rigid bodies

Make sure you have added rigid body and shape information to the objects (check the Physics Tools for 3ds Max, Maya or XSI for details on how to do so).

Physics information is stored using node attributes, and collision information is created from shape nodes by looking at the meshes associated with nodes. Ensure that the " **Export Attributes** " and " **Export Meshes** " options are selected in the Scene Export Options (for 3ds Max, Maya or XSI).

#### 4.11.1.5 Can't see anything in the Preview Tool filter

The most common cause for the Preview Tool filter not showing anything is problems with the cameras.

Particularly in 3ds Max, clipping planes may be causing the issue. Try changing the clipping planes through the **Cameras** > **Adjust Camera...** option in the preview.

Ensure that the **View** menu in the preview has enough options switched on, such as **Display Meshes** and/or **Rigid Body Ghosts** .

Problems may also be caused by graphics drivers. Most card manufacturers upgrade their drivers regularly; try downloading the latests version, particularly if you are using Havok FX.

#### 4.11.1.6 The Preview Tool filter doesn't use my chosen renderer

Currently it is not possible to have multiple preview windows open at the same time using different renderers. Each preview that is opened will always use the same renderer as that of any previews which are already open. If the chosen renderer is different to that of any which are already open, then the window title will indicate that the renderer in use is different to the desired one, eg: "`Havok Preview [ D3D9 \textbf{- forced} ]`".

#### 4.11.1.7 Crash in the Preview Tool filter

If you experience a crash in the Preview Tool filter, it may be caused by a configuration-specific problem. Try upgrading your graphics card drivers, and switching to a different renderer (OpenGL/Direct X) in order to fix or workaround the issue (but do not hesitate to report it).

#### 4.11.1.8 Filter configuration changes don't seem to be stored

If your changes to the filter configurations in the filter manager don't seem to be stored/saved when you close the filter manager, ensure that you are not overriding those changes by having the " **Use HKO options file** " option selected in your scene exporter (3ds Max/Maya/XSI) options or by having an override/upgrade HKO file present.

#### 4.11.1.9 Can't (properly) export NURBS skin

The Havok Content Tools do not support skinned NURBS; the NURBS surface may be exported as (converted to) a static mesh but the bone-skin binding information will not be exported. This is because skinning a NURBS surface modifies control points while skinning a polygonal mesh modifies vertices. Therefore it is not possible to accurately convert a NURBS skin-binding to a polygonal skin-binding (in general, `meshSkin(nurbsToMesh(nurbs),t) != nurbsToMesh(nurbsSkin(nurbs.t))`).

#### 4.11.1.10 "No bone influences in skin [<name>]" warning

This warning is raised in the 3ds Max scene exporter when a mesh with a skin modifier is exported, but no bone influences are found. This can be caused by no bones being associated to the skin modifier, or by all the bone weights (influences) being zero.

Ensure you have properly bound the skin to the bones; moving the bones should deform the skin mesh.

#### 4.11.1.11 Skin not updated in the Preview Tool filter

- Ensure that you have correctly used the Create Skeleton, Create Skin and Create Animation filters before the Preview Tool filter.
- Multiple mesh binding mappings are only supported by hardware skinning. If you are limiting the number of bones in the Create Skin filter, this will likely have caused the mesh binding to be split into multiple mappings. Enable **Hardware Skinning** in the Preview Tool filter options (DirectX only) or disable the limit on the number of bones in the Create Skin filter.
- Skinned NURBS surfaces are not supported. Use polygonal meshes instead.

#### 4.11.1.12 Can't get Extract Motion filter to work

The most common user errors when using the Extract Motion filter are:

- **Wrong coordinate system** : Make sure you have selected the proper coordinate system in the filter options (the **Up/Down** and the **Forward/Backwards** direction)
- **Non-linear motion** : Make sure you are using the right amount of samples. Using 2 samples only (constant velocity) will ignore any non-linear component of the motion; switch to **Same as Animation** in order to extract non-linear motion.

Use the **View** > **Extracted Motion** in the Preview Tool filter to visualize the extracted motion and ensure you get the expected results.

#### 4.11.1.13 Textures do not match original texture in the modeller

This may be caused by the original texture being transformed inside the modeller. When exporting bitmap textures, the scene exporters will just export a reference to the file. Therefore, information about extra transformations applied to the texture (color correction, cropping, scaling or UV transformation) is not exported. In order to get a faithful match, you can:

- Bake the texture transformation into the texture bitmap file (so no transformations are done by the modeller)
- Use UV transformations on the mesh rather than on the texture

#### 4.11.1.14 Textures do not match original texture in the modeller

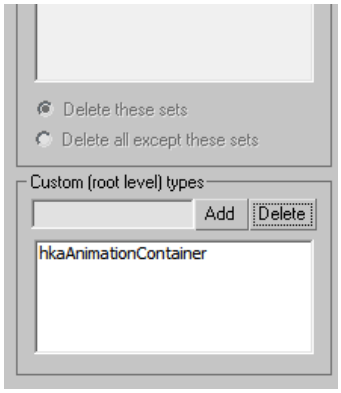
This may be caused by the original texture being transformed inside the modeller. When exporting bitmap textures, the scene exporters will just export a reference to the file. Therefore, information about extra transformations applied to the texture (color correction, cropping, scaling or UV transformation) is not exported. In order to get a faithful match, you can:

- Bake the texture transformation into the texture bitmap file (so no transformations are done by the modeller)
- Use UV transformations on the mesh rather than on the texture

#### 4.11.1.15 Removing product-specific data before export

Sometimes may want to remove all data specific to a particular product before export–for example, in the cases where you haven't licensed that product.

You can use the Prune Types filter in order to do so. Product-specific data is stored at the top level of the processed contents, and each product uses a different type to store its data.




Use the **Custom (root level) types** section of the filter to specify those product-specific containers you want to remove. The following table gives you a list of the container types used by each product:

| | |
|---|---|
| Havok Animation | `hkaAnimationContainer` |
| Havok Physics | `hknpPhysicsSceneData` |
| Havok Physics 2012 | `hkpPhysicsData` |

Table 4.20: Container Types used by each Havok Product

**Note:**
The `hkaAnimationContainer` can also be removed by enabling the **All Animation Data** checkbox in the Prune Types filter.

#### 4.11.1.16 Other problems processing assets

The View XML filter can be a powerful debugging tool: it will help you spot any problems with the contents of the scene and the results of each filter. Place it at different stages of your filter processing and examine the data at each stage.

Processing assets will also generate a processing log, where information about asset processing, as well as any warnings or errors found, is reported. Ensure that the contents of the processing log matches your expectations.

```
Setting up Filter Manager...
Setup Complete!

[ --- Configuration: Default --- ]
   *** No PlatformWriter filter found. No HKX data will be saved for this configuration. ***
   [ SceneTransform ]
      Processed all nodes in the scene.
      Processed 7 meshes.
      Processed 4 cameras.
   [ CreateRigidBodies ]
      Creating Mopp Code for pCylinder1 ...
      Created 7 rigid bodies.
   [ CreateRigidBodies ]
      Warning [0xabbaa5f0]: Some physics data found already, have you already created rigid bodies?
      Creating Mopp Code for pCylinder1 ...
      Created 7 rigid bodies.
   [ CreateConstraints ]
      Created 2 constraints.
   [ PreviewScene ]
   Finished in 00:00:00
```

If you are still running into problems processing assets and the cause of the problem is unclear, use the Support Data snapshot utility to create a set of data files suitable for later analysis

- Open the problem asset and launch the filter manager in the usual way.
- Once the manager has launched set up the filters so the problem is reproduced.
- From the help menu choose **Support Data** > **Save** .
- You will be prompted for a filename and 2 files will be created with the extension `support.hkx` and `support.hko` respectively.
- Send these file to support for analysis.
- NOTE: Before sending the files you can check their validity using the stand alone previewer.
  - Launch the previewer and load the `support.hkx` file
  - Once the filter manager has loaded use the **File** > **Load Configuration** option to load the options from the `support.hko` file.

## 4.11.2 Troubleshooting: 3ds Max Tools

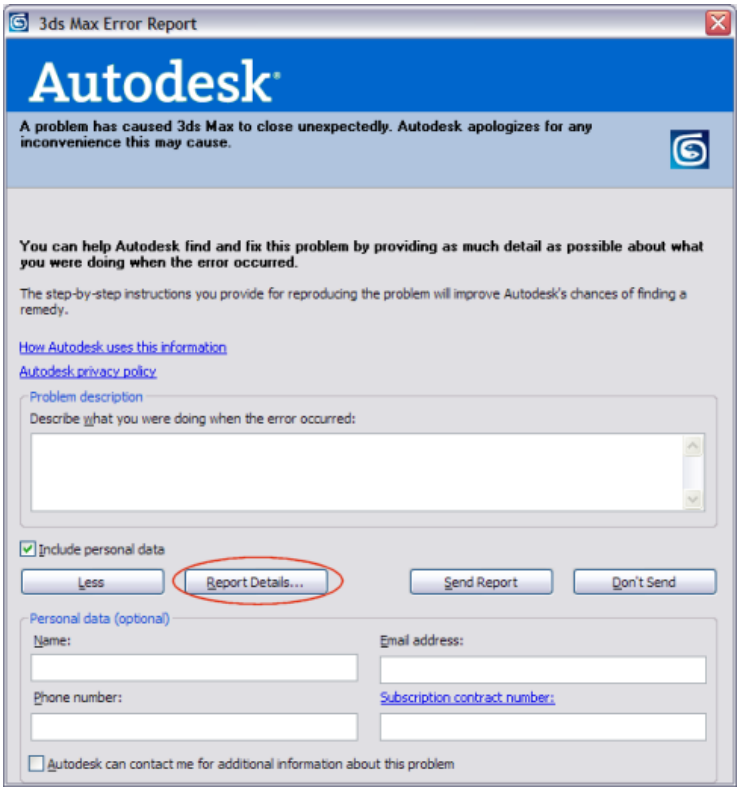
### 4.11.2.1 MAXScript Exceptions in general

When the Havok Content Tools are invoked through the **Havok Content Tools** menu or toolbar, they are called from MAXScript code. When a crash happens inside the tools, MAXScript reports an exception and tries to recover - most times, unsucessfully. Therefore, anytime you see a MAXScript exception raised when executing an operation with the Havok Content Tools (usually during export) the most likely cause is a crash.

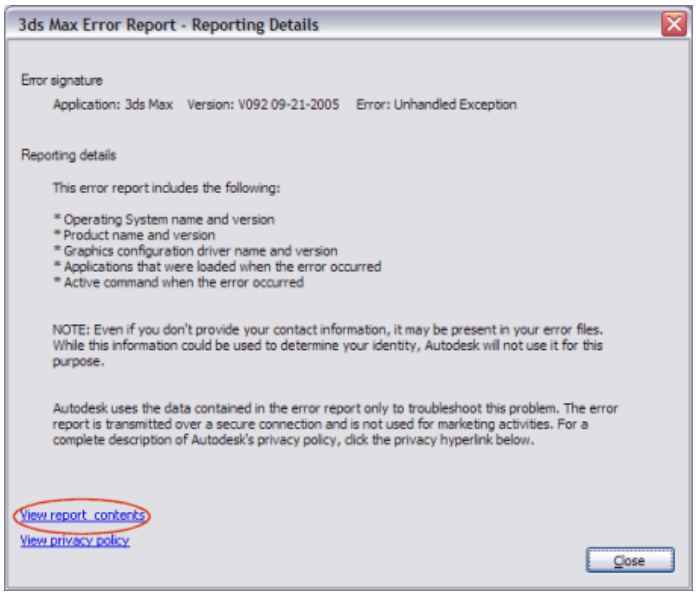
If you export by first opening the Scene Export Utility ( ), and then clicking on the **Export Scene** button, MAXScript is not used and the exception should be replaced by a full crash, which is more useful as it allows you to send us the crash information (see Reporting Crashes below)

### 4.11.2.2 Reporting Crashes

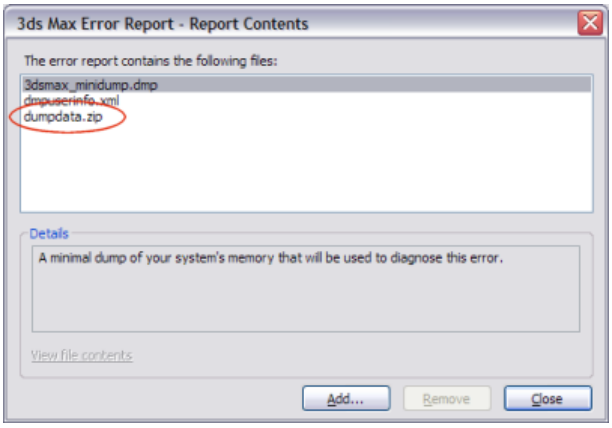
Should you experience crashes during export which are not solved by any of the items in this troubleshooting section you can send us the crash information reported by 3ds Max (if you are getting a MAXScript exception please check the section above in order to generate a full-blown crash). When the " **3ds Max Error Report** " window appears...




...instead of sending it to Autodesk, please click on the **Report Details...** button and then on the **View Report Contents** link.




This will list the files that would be sent to Autodesk. This includes `dumpdata.bin` (hovering the mouse over the file name will show its location in your hard drive).




Please include this file when reporting the issue to Havok support (no need to send it to Autodesk), as well as the asset you were trying to process. This will help us reproduce your problems.

#### 4.11.2.3 Can't find Havok's HKX format in the File > Export... dialog

The 3ds Max Scene Exporter is implemented as a 3ds Max Utility object and not as a file exporter (since it exports the scene to the filter pipeline rather than directly to a file), so it no longer appears as a choice of format/extension in the 3ds Max **File > Export...** dialog. Check the 3ds Max Scene Exporter section for details.

#### 4.11.2.4 I can't make a rigid body from an object

You cannot create a rigid body from an object if:

- The object is already a rigid body
- The object doesn't have a mesh representation (dummies, curves or helpers)
- The object is a Character Studio bone. Use the Rag Doll Toolbox to create proxies instead.

#### 4.11.2.5 Can't see the Havok Content Tools toolbar/menu

If cannot see neither the **Havok Content Tools** toolbar nor the menu, then the Havok Content Tools have not been properly installed for your version of 3ds Max. Try reinstalling them.

If you can't see the **Havok Content Tools** toolbar, but you can see the **Havok Content Tools** menu, click on the menu option **Show Toolbar** . If that still doesn't work, then the problem may be that the toolbar is placed outside the visible desktop area. Try docking it by executing the following MAXScript command:

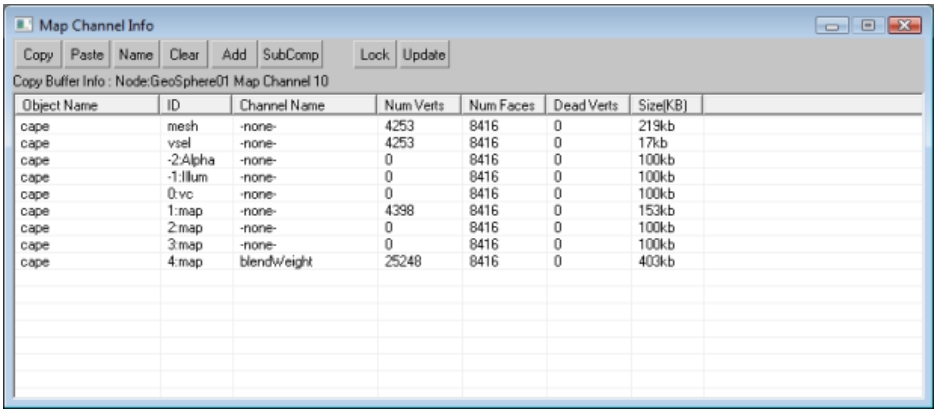
```
hctToolbarGUP.dock()
```

#### 4.11.2.6 Uninstalling the Havok Content Tools does not remove the Havok Content Tools menu

3ds Max remembers the configuration of menus between sessions, so uninstalling the Havok Content Tools may leave an empty **Havok Content Tools** menu in 3ds Max. You can remove it through the **Customize** > **Customize User Interface...** menu option in 3ds Max.

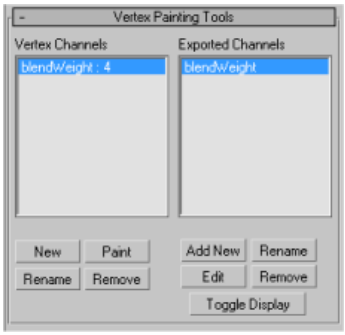
#### 4.11.2.7 Can't see some channels in the Mesh Channel Toolbox

The Mesh Channel Toolbox lists only named vertex color channels. In 3ds Max, channels are named using the **Map Channel Info** dialog, accessible through the **Tools** > **Channel Info...** menu item:




| Object Name | ID | Channel Name | Num Verts | Num Faces | Dead Verts | Size(KB) |
|---|---|---|---|---|---|---|
| cape | mesh | -none- | 4253 | 8416 | 0 | 219kb |
| cape | vsel | -none- | 4253 | 8416 | 0 | 17kb |
| cape | -2:Alpha | -none- | 0 | 8416 | 0 | 100kb |
| cape | -1:Illum | -none- | 0 | 8416 | 0 | 100kb |
| cape | 0:vc | -none- | 0 | 8416 | 0 | 100kb |
| cape | 1:map | -none- | 4398 | 8416 | 0 | 153kb |
| cape | 2:map | -none- | 0 | 8416 | 0 | 100kb |
| cape | 3:map | -none- | 0 | 8416 | 0 | 100kb |
| cape | 4:map | blendWeight | 25248 | 8416 | 0 | 403kb |

The " **Channel Name** " column shows the name of the channel, which is the name that appears in the **Vertex Channel** list box in the Mesh Channel Toolbox. The **ID** of the channel also appears beside it.




If your channel doesn't appear in the list of vertex channels, it is likely that it has no name. Go into the **Map Channel Info** utility and give it a name.

Modifiers like the **Vertex Paint** modifier or the Export Channel modifier reference channels by their ID. If you decide to move a vertex channel to a different ID, make sure you:

- Name the new vertex channel (using the Map Channel Info utility)

- Set the ID of any associated **Vertex Paint** modifier to the new ID
- Set the ID of any associated Export Channel modifier to the new ID

#### 4.11.2.8 Can't create selections with the Mesh Channel Toolbox

In general the Toolbox lets you create and work on vertex and triangle selections that are created with Max modifiers or when the current object is an Editable Mesh or Editable Poly. But not all modifiers are supported as some of them don't keep their selections when saving to disc and reloading. Here follow the lists of modifiers you can use for vertex and triangle selections:

- For vertex selections use Mesh Select, Poly Select and Edit Mesh modifiers or just convert your object to Editable Mesh or Editable Poly.
- For triangle selections use Mesh Select and Edit Mesh modifiers or convert your object to Editable Mesh.

#### 4.11.2.9 Mesh Channel Toolbox does not display all vertex and triangle selections

Vertex and triangle selections created with unsupported modifiers will not be listed in the Toolbox. Please see the section about supported modifiers for more details.

#### 4.11.2.10 Vertex painting color feedback is not visible on mesh while in paint tool

If there are Export Channel modifiers higher in the modifier stack than the vertex paint modifier, they may need to be disabled while painting to see color feedback.

### 4.11.3 Troubleshooting: Maya Tools

#### 4.11.3.1 Errors of the form "Node [Xxx] has no attribute [Xxx]" on loading a scene

When moving from one version of the Havok Content Tools to another, some attributes of our custom node types may be added or removed. In the case that they are removed, any scenes which were saved with these attributes present may complain when loaded into a version where they are not present. Maya currently does not provide any way for plugins to deal with this scenario, and instead reports it as an error. By examining the script window, you can see which attributes are missing and determine whether it is something to worry about.

Normally it is safe to ignore these errors. When the scene is resaved the errors will no longer appear.

#### 4.11.3.2 Can't find Havok's HKX format in the File > Export... dialog

The Maya Scene Exporter is implemented as a Maya command and not as a file exporter (since it exports the scene to the filter pipeline rather than directly to a file), so it no longer appears as a choice

of format/extension in Maya's **File** > **Export...** dialog. Check the Maya Scene Exporter section for details.

#### 4.11.3.3 Can't see the Havok Content Tools menu or the Havok shelf

Make sure you have installed the Havok Content Tools for the version of Maya you are using, and the plug-ins have been properly loaded in Maya.

#### 4.11.3.4 Havok Shelf is Empty

Maya remembers the configuration of shelves during sessions, so uninstalling the Havok Content Tools may leave an empty Havok shelf in Maya. You can remove it through the **Window** > **Settings/Preferences** > **Shelves...** menu option in Maya.

#### 4.11.3.5 Can't see Havok rigid bodies and/or constraints in the viewports

In order to see Havok content (rigid bodies, constraints, etc.) in the viewports, the Havok Physics Tool must be active. In addition, double-clicking on the tool icon in Maya's toolbox provides customisation options for this tool. It is possible to disable the display of rigid bodies and/or constraints on a group basis, so make sure that the content you require is not hidden in this way.

Any Havok node which is directly selected (via the outliner for example) will always be displayed in the viewports, regardless of the tool settings.

#### 4.11.3.6 Exported node transforms don't match the original transforms in Maya

By default the Maya scene exporter will override the node transforms in order to match the location of the rotate pivot point. This is required by the Maya Physics tools, so changing this option is not recommended if you are creating physics content. The " **Include Rotate Pivot in Transforms** " option can be found in the Maya Scene Exporter Options.

#### 4.11.3.7 Shape Type defaults to Convex Hull although object is a sphere/box/capsule/cylinder

The Havok Maya tools will try to detect the type of a mesh by checking its construction history. For example, if a *"polySphere"* node is found, the shape type will be flagged as "Sphere" by default.

If the construction history is lost (for example, by using the **Duplicate** tool) then the Havok tools no longer have direct knowledge of the mesh type and will default to "*Hull*". Therefore you will need to manually set the **Shape Type** parameter for those objects.

For duplication, you can use the " **Duplicate input graph** " option with the **Duplicate Special** tool, instead of the **Duplicate** tool; this will keep the construction history of the duplicated node.

#### 4.11.3.8 Adding Havok nodes breaks UV Texture editor "UV Sets" menu

Maya gets confused if the first child shape of a transform node is not a mesh. A workaround is to select the mesh itself (rather than its parent transform node) before going into the UV editor.

#### 4.11.3.9 Adding Havok nodes breaks "Assign Material" context menu

This is due to a bug in Maya: Maya is checking for the number of renderable objects on the selected node, and is expecting to receive an `int`, but is instead receiving an `int[]` array, and thus failing to build the menu. This has been fixed in Maya 8.5.

#### 4.11.3.10 Vertex painting color feedback is not visible on mesh while in paint tool

Check that the viewport menu item **Renderer** > " **High Quality Rendering** " is not checked (the paint tool color feedback is turned off in Maya while in the High Quality viewport rendering mode). Also, note that it may be necessary to hide other meshes in the scene before painting on the selected mesh becomes active.

#### 4.11.3.11 Paint Attributes Tool warns about missing attributes

Creating a painting channel registers the corresponding attribute name as paintable in the user preferences (see the Maya user prefs MEL file, usually in `Documents and Settings\user\My Documents\maya\2008\prefs\userPrefs.mel`), and this list of paintable attributes tends to accrete. If a paintable attribute is not present on the mesh (or even in the scene) the tool warns that it cannot find that attribute. Unfortunately we cannot clear the list of paintable attributes in our tools since this may interfere with your own preferences for paintable attributes. If the warnings are a nuisance they can be avoided by periodically editing the `userPrefs.mel` file and removing the "makePaintable" commands corresponding to attributes that are not present in the scene.

### 4.11.4 Troubleshooting: XSI Tools

#### 4.11.4.1 Physics 2012 preferences are not persisted across different scenes

This may occur if the XSI tools are installed into the 'factory' addons folder. A bug in XSI means that preferences saved inside this folder are not detected by XSI, and as such they cannot be persisted across different scenes. If this case is detected, any preferences are instead stored as a node in the scene, and a warning is given to the user. Any new scene will use the default preference values.

#### 4.11.4.2 Rag Doll Toolbox disappears/gets replaced by white background

This is due to a bug in XSI. If a custom display host becomes hidden (by minimizing the window or the application, by switching to a diffferent tab in a custom relational view, etc.) then the custom display host may lose the value of the attribute which specifies the particular custom display to use. To retrieve the Rag Doll Toolbox, simply right click on the white background and select 'Havok Rag Doll Toolbox'.

There are also some redraw issues in XSI which may cause parts of the toolbox to be erased. Simply activate/move the toolbox to force a full redraw.

#### 4.11.4.3 Exported node hierarchy doesn't match the original XSI hierarchy

By default the XSI exporter will reparent the exported nodes during export to ensure that chain end effectors are always children of the last link of the chain. This option, " **Reparent Chain Effectors** ", can changed in the XSI Scene Exporter Options.

#### 4.11.4.4 Exported node transforms don't match the original transforms in XSI

By default the XSI exporter will override the translation component of the node transforms in order to match the location of the pivot point. This is required by the XSI Physics tools, so changing this option is not recommended if you are creating physics content. The " **Include Pivot in Transforms** " option can be found in the XSI Scene Exporter Options.

### 4.11.5 Troubleshooting: Toolchain Integration

#### 4.11.5.1 Can't see my custom filter in the filter manager

- Make sure you've placed it in the right location.
- Check the filter log window for any warnings raised by the filter manager.
- Ensure your custom filter is built with the same sdk version as the filter manager
- Make sure you defined and exported the necessary DLL functions