Introduction

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

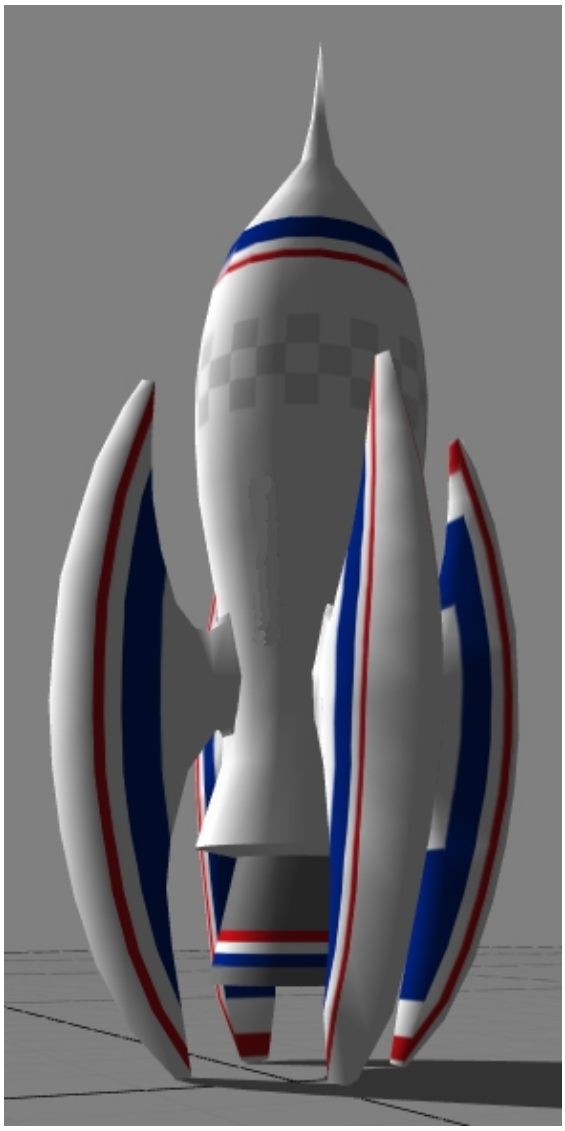
Table of Contents

Table of Contents

Table of Contents

Table of Contents

NVIDIA PhysX SDK 2.8 - Introduction



The NVIDIA PhysX SDK unites rigid body dynamics simulation and collision detection in a single easy-to-use package.

The rigid body dynamics component enables you to simulate objects with a high degree of realism. It makes use of physics concepts such as reference frames, position, velocity, acceleration, momentum, forces, rotational motion, energy, friction, impulse, collisions, constraints, and so on in order to give you a construction kit with which you can build many types of mechanical devices.

While this documentation will cover the features and usage of the SDK, because of the academically challenging nature of the subject matter, it is assumed that you have mastered the material covered in the mechanics part of an introductory physics course. If not, it is recommended that you get hold of a good book and familiarize yourself with the concepts listed above before continuing.

Installation

Setup

After downloading the NVIDIA PhysX SDK installer executable from the NVIDIA web site, follow the directions of the installer program. When complete, the directory structure described below will be created in the location you have specified.

Directory Structure

The SDK installer creates the following directory structure:

Directory	Description
(SDK Path)\Samples	SDK Samples
(SDK Path)\Bin	SDK DLL files and Sample Binaries
(SDK Path)\Graphics	Include and Library files for the graphics library used by the samples
(SDK Path)\SDKs\Cooking\include	Include files for the Cooking SDK
(SDK Path)\SDKs\Foundation\include	Basic/Foundation include files (e.g., math, utilities, etc.)
(SDK Path)\SDKs\NxCharacter\include	Include files for the character SDK
(SDK Path)\SDKs\Physics\include	Include files for the main physics SDK
(SDK Path)\SDKs\PhysXLoader\include	Include file for the loader library
(SDK Path)\SDKs\lib	Library files for each component of the NVIDIA PhysX SDK
(SDK Path)\SDKs\Docs	Documentation

There may be various other directories containing stand alone demos, demo scenes, and tools.

Components Overview

The SDK is divided into the following components:

- Physics SDK - used to simulate rigid bodies, fluids, etc.
- Cooking SDK - allows meshes to be preprocessed into a form suitable for use with the physics SDK.
- Foundation SDK - a library of container, math and utility classes for use by the other SDKs.
- Character SDK - provides tools to simulate a game character.
- PhysXLoader - used to dynamically load the appropriate core physics SDK. The loader locates the correct DLL (taking into account version number, installation location, etc.).

Deployment

The core NVIDIA PhysX files (PhysXCore.dll, NxCooking.dll and the firmware binaries) are no longer intended for distribution within a game installer. Instead, a dedicated installer for developers to execute as part of the install process is provided. The installer will install the software runtime files and, if necessary, the NVIDIA PhysX hardware drivers.

Command line options:

PhysXInstaller_x.x.x.exe [/semiSilent] [/noLanguageSilent] [/noXFIRE] [/noSysTray] [/noHwDriver]

- /semiSilent - will not show the installation options page, only the language selection, license agreement page, and install files page
- /noLanguageSilent - will not show the installation options page, only the license agreement page and install files page
- /noXFIRE - will not install the XFIRE client (this will uncheck the option)
- /noSysTray - will not autostart the PhysX system tray icon (this will uncheck the option)
- /noHwDriver - will not install the NVIDIA PhysX Processor Device Driver (this will uncheck the option)

NOTE: This is not a completely silent PhysX installation due to the EULA agreement that the end user must check.

Compiler Setup

The example programs are supplied with project files for Microsoft Visual Studio VC++ 7 and 8 (Solution files). You will be able to open the files provided and issue a build command without additional configuration; however, the created example programs will need to access the SDK DLLs, which are located in the /BIN/win32 directory (this includes PhysXLoader.dll and NxCharacter.dll).

If you are using a compiler other than VC++ 7 or 8, or if you would like to create a new project using the SDK, set it up as follows:

1. Specify the '*SDKs\Foundation\include*', '*SDKs\Physics\include*', and '*SDKs\PhysXLoader\include*' as directories in which include files are located.
2. Identify the *PhysXLoader.lib* file in '*SDKs\lib\win32*' as a library to the linker.
3. Additionally, you may need to specify '*SDKs\Cooking\include*' and '*SDKs\NxCharacter\include*' plus its corresponding .lib file if you use features from these libraries.

When you have created an executable, make sure PhysXLoader.dll is available somewhere in the windows DLL search path (see the MSDN documentation entry, "LoadLibrary", for details). The easiest way to achieve this is to have the compiler write the produced executable into the /BIN/win32 directory.

An important compiler setting to take note of is the (reserved) stack size, you should ensure that this is sufficient to accommodate the scenes which you will be simulating. As a rough guideline, it should be proportional to the number of actors, number of joints in a chain and the complexity of your mesh objects. Plus a larger stack size is required in debug versus release mode. As an example 1Mb should be sufficient for about 1000 actors in a jointed chain in debug mode.

In addition, the NVIDIA PhysX driver installer must have been used to install the correct software/hardware drivers.

API Basics

Architecture

The SDK has an ANSI C++ programming interface. Internally the SDK is implemented as a hierarchy of classes. Each class that contains functionality that can be accessed by the user implements an interface. This interface is effectively a C++ abstract base class. In addition, some stateless utility functions are exported.

Conventions

Each of the interface classes defines a set of methods. The following is a list of coding conventions:

- All classes are defined in a header file with the same file name as the class name.
- Types and classes start with a capital letter.
- Interface classes (all classes visible to the user) are prefixed with "Nx".
- Methods and variables start with a lowercase letter.
- Return values and parameters in places where a NULL value is acceptable (usually indicating "default" or "no object") are coded with pointer (*) syntax. Thus, the user should check for possible NULL values after calling methods that return a pointer.
- Return values and parameters in places where a NULL value is unacceptable are coded with reference (&) syntax. In this case, neither the user nor the SDK checks for NULL values, which do not exist according to C++. While the user may force a NULL value into the SDK by dereferencing a NULL pointer, this is obviously a bad idea. If you are uncertain about the possibility of pointer being NULL, assert it before passing the dereferenced pointer into the SDK to avoid a blowup.
- When the SDK needs the user to implement some functionality (for example, memory management) the user needs to implement an interface defined for this purpose. This scheme is used instead of C callback functions, but with the same purpose.

Data Types

In order to provide a certain degree of portability, the SDK makes use of size specific typedefs, defined in the Foundation\NxSimpleTypes.h.

The SDK's classes NxVec3, NxMat33, NxMat34, and NxQuat represent a 3-vector, a 3x3 matrix, a 3x4 matrix, and a quaternion respectively. They are currently configured to use single precision (32 bit) floating point scalars. These are the types the user should use when interfacing with the SDK.

The math classes also provide a large selection of type and format conversion methods for easy interfacing with your own math classes. Because of the way inline code is compiled, you may be able to add your own casting operators to these classes without the need to recompile the SDK DLL or make it a dependency of your math classes.

Units

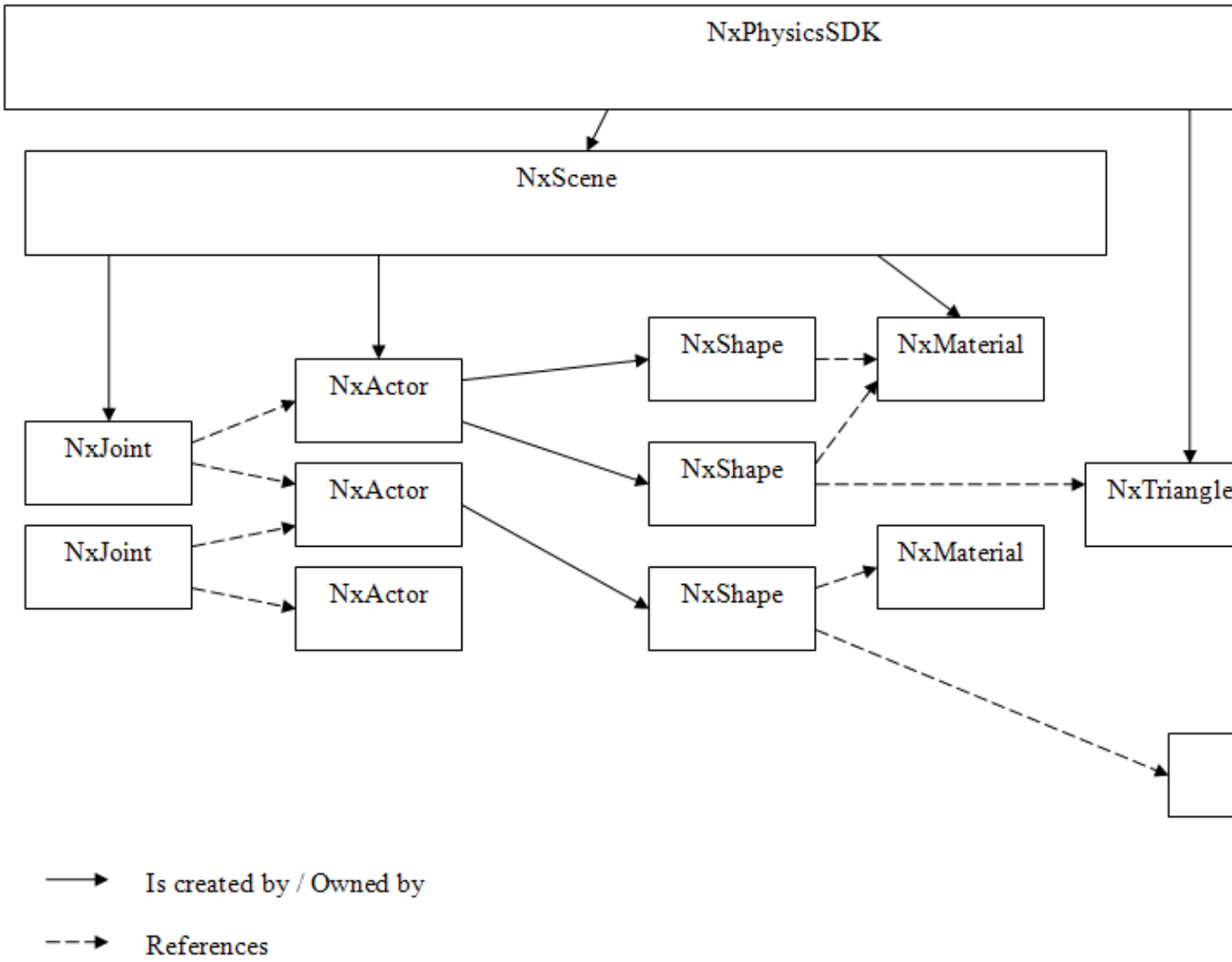
The SDK does not need to use any particular real world units. However, it is important to define a convention when it comes to units in your application's artwork pipeline. The SDK uses unitless numbers to measure three types of basic quantities: mass, length, and time. You can define these quantities to be in any units you want. (In our demos we use kilograms for mass, meters for length, and seconds for time.) The units of derived quantities follow from these basic units. For example, velocity is always distance/time, so in our case this is meters/seconds. Other such derived units apply to force, impulse, etc.

As with any numeric software, and especially because the SDK uses only single precision floating point numbers by default, it is important to keep the numbers within a range of relatively high precision. What exactly this range is depends on your particular needs regarding simulation accuracy.

API Reference

- NxVec3
- NxMat33
- NxMat34
- NxQuat

Architecture Diagram



API Reference

- NxPhysicsSDK
- NxScene
- NxJoint
- NxActor
- NxShape
- NxMaterial
- NxTriangleMesh
- NxConvexMesh

Math Classes

The math functionality in the SDK consists of the classes NxMath, NxVec3, NxQuat, NxMat33, NxMat34, and some floating point unit control functions and macros in the NxFPU.h file.

The NxMath class contains wrappers for the standard C floating point scalar math functions, such as sin() or fabs(), and defines some common mathematical constants, such as Pi.

NxVec3 is a three element vector. NxMat33 is a 3x3 matrix, most often used to express rotations, but sometimes used to express inertia tensors. However, the SDK does not permit scaling or any other affine transformation that could be expressed with such a matrix. NxQuat is a quaternion class and provides a more succinct way to express rotations.

NxMat34 is a complete affine transformation composed of a rotation matrix and a translation vector.

Linear transformations are written in this guide as well as in comments as H = [R, t] where R is the rotation matrix and t is the translation vector. The transformation of a point p to p' is written as follows:

```
p' = H p = R p + t
```

The product of two transformations A= [aR at] and B = [bR bt] is shown below:

```
A B = [aR bR aR bt + at]
```

Matrix element storage is meant to be opaque to the user, but may be retrieved from NxMat33 using the get/setRow/ColumnMajor() methods which will execute the appropriate transformation if needed. Left or right handedness is not a concern of the SDK, though coordinate frames are visualized in the documentation as right handed. The easiest way to send a transform matrix to OpenGL is like this:

```
NxMat34 m;//PhysX matrix
float glmat[16];//OpenGL matrix

m.getColumnMajor44(glmat);//copy to glmat
glMultMatrixf(glmat);//send to OpenGL
```

NOTE: While Direct3D matrices are technically row major, they have an additional semantic difference to OpenGL matrices which cancels this out; therefore, use this same code for Direct3D.

API Reference

- NxMath
- NxMat33
- NxMat34
- NxQuat
- NxVec3

Simple Shapes

The SDK defines a set of shape classes that provide simple functionality. They are not to be confused with the Physics SDK's shape classes, which in part build on these classes and provide more complex internal functionality. The simple shapes are as follows:

- NxBounds3 - Axis Aligned Bounding Box (AABB)
- NxBox - Oriented Bounding Box (OBB)
- NxCapsule - Capsule (Line segment with a distance)
- NxPlane - Parametric Plane
- NxRay - Infinite Ray/Line
- NxSegment - Finite Ray/Line (i.e., has two end points)
- NxSphere - Sphere, Point, and Radius

API Reference

- NxBounds3
- NxBox
- NxCapsule
- NxPlane
- NxRay
- NxSegment
- NxSphere

SDK Initialization

To become familiar with the SDK and how to use it, below is a walk through of the SampleBoxes demo application source code. It can be found in the file /SampleBoxes/src/NxBoxes.cpp.

The symbols of the SDK may be included in the user's relevant source files through the following header:

```
#include "NxPhysics.h"
```

If your application uses windows.h included before the SDK, you must suppress its definition of the macros min and max, as these are common C++ method names in the SDK. To do so, use the following:

```
#define NOMINMAX
#include <windows.h>
```

Start up the Physics SDK with the following:

```
gPhysicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION, &myAllocator, &myOutputStream);

if(!gPhysicsSDK)Error("Wrong SDK DLL version?");
```

The first argument is the version number defined in the SDK headers. This makes certain that the include files you are using define the same version of the classes that the DLL contains. If there is a version mismatch, an error is reported and NULL is returned.

The second argument is an optional memory allocation object, and may be NULL. See Memory Management for more information.

The third argument is an optional output stream object, and may be NULL. See Error Reporting for more information.

When the call is successful, you will have a pointer to an NxPhysicsSDK interface.

When you are finished using the SDK, you must release it. Releasing the SDK, with the method shown below, automatically releases all of the SDK objects you created:

```
gPhysicsSDK->release();
```

If NxCreatePhysicsSDK is called more than once, the same actual object is returned (as it is a singleton class). However, since there is a reference counter for the singleton, you must match each call with a call to release(). The allocator argument will be ignored for any calls past the first, but the output stream will be changed.

API Reference

- NxPhysicsSDK

Casting and Instancing

Instances of class interfaces cannot be created with the C++ new operator because they are pure virtual classes. The instance of the first class created, NxPhysicsSDK, is returned by the NxCreatePhysicsSDK function, discussed in the SDK Initialization section.

Many SDK interfaces contain create* methods that return instances of other types. These methods are used to create SDK objects. To avoid overly long parameter lists and receive aid in error checking, the SDK uses descriptor classes to pass creation parameters. The descriptor is a temporary variable only needed for the duration of the creation call. You can allocate it as a stack variable. Below we will create a scene as an example. For this we need a scene descriptor:

```
NxSceneDesc sceneDesc;
```

The scene descriptor initializes itself to some default values which will not suffice, in most cases, to create an object. The user may, therefore, override these parameters:

```
sceneDesc.gravity.set(0,-9.8f,0);
```

To create the SDK object, a reference to the scene descriptor is passed to the create method:

```
gScene = gPhysicsSDK->createScene(sceneDesc);
if (!gScene) Error("Can't create scene!");
```

NOTE: It is important that the user check the return values of all create methods, because if the descriptor object contains illegal parameters, the creation will fail.

After you are finished using an object and want to get rid of it, you must destroy it. Once again, you may not use the standard C++ delete operator. It would lead to memory corruption. Instead, you must ask the object that created your instance to delete it for you. This is done using a method called release*:

```
gPhysicsSDK->releaseScene(*gScene);
gScene = 0;      //the object is now deleted,
                 //so don't use the pointer anymore!
```

After you release an object, do not use that pointer or reference again. While you may notice some similarities between this convention and that of COM (as it is used in DirectX™), there is an important difference: The up casting and down casting operators (described below) do not count as a COM QueryInterface(), so no release() needs to be called for them. In other words, here you release an object, and not a pointer. When calling release, the object pointed to is guaranteed to be deleted, and no reference counting is used.

NOTE: If you release an object which you have used to create other objects, these other owned objects will be released automatically. For example, if you release the SDK, all the created scenes will be released.

Up Casting

Up casting is the operation of converting a derived class reference (or subclass) to a super class reference.

In versions of the SDK prior to 2.1.2 the SDK did not support interface inheritance and thus related classes did not form real hierarchies. This is no longer the case; it is possible to use the C++ built in inheritance mechanism to implicitly cast from subtypes to super types.

Down Casting

Down casting is the opposite of up casting. Here you are trying to convert a reference from a subclass to one of its potential derived classes. Unlike up casting, downcasts cannot be validated at compile time, so there is a possibility that it will fail if called on the wrong type of object. For example, an NxShape reference pointing to an NxSphereShape object will successfully cast to an NxSphereShape reference, but the cast will fail when you try to cast it to an NxBoxShape. The example below shows the syntax of attempting downcasts in the SDK:

```
//this is still the sphere object created above.
NxShape * shape = shapePointers[0];

//get back the sphere pointer
NxSphereShape * sphere = shape->isSphere();

//try to get a box shape pointer:
NxBoxShape * p = shape->isBox();
//failure! here p will be set to 0.
```

NOTE: A failed downcast is not an error as such, so no error is reported other than returning a NULL pointer. See the API reference regarding the specific class hierarchy, and which classes provide which up and downcast methods.

Memory Management

When initializing the SDK, it is possible to specify a user defined class for memory management. This mechanism is not particularly useful for PC development, but is meant to be used on systems with special memory management requirements. Either way, it allows the user to monitor the memory usage of the SDK, which is important in many cases.

To have the SDK use your own memory allocation methods instead of the standard C malloc/realloc/free, you need to define a class derived from NxUserAllocator, and define all of its pure virtual methods. These methods are basically the C runtime malloc/realloc/free, so on your first try you could just call these methods. An example implementation is shown below. The SDK does not guarantee any particular usage patterns that allow you to make a truly specialized allocator. In many cases, the SDK will allocate a chunk of memory with your allocator at startup, and then internally manage this block for its internal allocation needs.

The allocator that you define should be passed as the optional second argument to the NxCreatePhysicsSDK() method on startup. After you create the SDK, there is no way to change or remove the allocator.

NOTE: No method from the SDK should be called within the allocator.

Example

```
class MyAllocator : public NxUserAllocator
{
public:

    void * malloc(NxU32 size)
    {
        return ::malloc(size);
    }

    void * mallocDEBUG(NxU32 size,const char *fileName, int line)
    {
        return ::_malloc_dbg(size,_NORMAL_BLOCK, fileName, line);
    }

    void * realloc(void * memory, NxU32 size)
    {
        return ::realloc(memory,size);
    }

    void free(void * memory)
    {
        ::free(memory);
    }

} myAllocator;

gPhysicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION, &myAllocator, 0);
```

Threading

The user should ensure that the allocator they provide is thread safe, since it is called in the context of the physics processing thread(s) in addition to the user thread. In general, the OS provided/C library memory allocation routines are thread safe unless the user is linking to the single thread specific C runtime library (in which case additional thread synchronization is required).

API Reference

- NxUserAllocator

Debug Rendering

The debug renderer mechanism is intended to let the SDK communicate potential problems to the user in a visual way, rather than just textual error messages. This is very useful as many problems arise due to the incorrect spatial layout of objects within this software. For example, a car will not roll if its wheels' axes are set up to be pointing in the wrong direction; however, since this configuration of the axes is not strictly illegal, and thus will not produce an error message, the problem may not be immediately obvious to the user.

The SDK can help to quickly identify this problem by letting the user visualize dozens of settings with just a few lines of code. Because the SDK does not know or care what sort of graphics API you are using, you have to render the geometry it generates.

Note: Objects in compartments cannot currently be visualized.

The data is a container holding a list of points, lines, and triangles. Below is a complete implementation using the OpenGL graphics library. It reads the data out of the NxDebugRenderable and sends it to OpenGL.

```
...

//Do this after advancing the simulation.

const NxDebugRenderable *dbgRenderable=gScene->getDebugRenderable();

renderData(dbgRenderable);
...

void renderData(const NxDebugRenderable& data) const
{
    glLineWidth(1.0f);

    // Render points
    {
        NxU32 NbPoints = data.getNbPoints();
        const NxDebugPoint* Points = data.getPoints();

        glBegin(GL_POINTS);

        while(NbPoints--)
        {
            setupColor(Points->color);
            glVertex3fv(&Points->p.x);
            Points++;
        }
        glEnd();
    }
    // Render lines
    {
        NxU32 NbLines = data.getNbLines();
        const NxDebugLine* Lines = data.getLines();

        glBegin(GL_LINES);

        while(NbLines--)
        {
            setupColor(Lines->color);
            glVertex3fv(&Lines->p0.x);
            glVertex3fv(&Lines->p1.x);
            Lines++;
        }
        glEnd();
```

```
        }
        // Render triangles
        {
            NxU32 NbTris = data.getNbTriangles();
            const NxDebugTriangle* Triangles = data.getTriangles();

            glBegin(GL_TRIANGLES);

            while(NbTris--)
            {
                setupColor(Triangles->color);
                glVertex3fv(&Triangles->p0.x);
                glVertex3fv(&Triangles->p1.x);
                glVertex3fv(&Triangles->p2.x);
                Triangles++;
            }
            glEnd();
        }
    }

    void setupColor(NxU32 color) const
    {
        NxF32 Blue= NxF32((color)&0xff)/255.0f;
        NxF32 Green= NxF32((color>>8)&0xff)/255.0f;
        NxF32 Red= NxF32((color>>16)&0xff)/255.0f;

        glColor3f(Red, Green, Blue);
    }
```

It is assumed that the OpenGL matrices are already set up so that the geometry rendered will be transformed as if defined in world space.

Next we need to have the Physics SDK use our drawing code to visualize something. Suppose your objects are moving in an odd way; you suspect the inertia tensors might be set wrong. You decide that it&"temp0015.html">SDK Parameters section.

API Reference

- NxPhysicsSDK
- NxScene
- NxDebugRenderable

User Data

When you create a shape, it is advised to associate some application specific data with the shape object. This way, if one of your methods receives, for example, an NxShape pointer, it can easily retrieve the associated data. Use the public variable userData to do this. For example:

```
struct MyData
{
    int shapeColor;
    bool dangerous;
};
```

```
NxShape * myShape = ...;
MyData * data = new MyData;
data->shapeColor = 3;
data->dangerous = true;


myShape->userData = data;


//then you can define:
bool isShapeDangerous(NxShape & s)
{
return ((MyData *)s.userData)->dangerous;
}
```

This works with most SDK types, not just NxShape.

Object Names

For convenience, the SDK now provides a way to store a name with each actor, shape, and joint. This name can be accessed either via the object descriptor's name field before instancing, or by using the setName() and getName() methods of the class. The name string is neither copied nor used by the SDK, it merely stores the pointer, just like the userData field.

API Reference

- NxShape
- NxActor
- NxJoint

User Defined Classes

As mentioned in other sections such as Memory Management, the SDK requires (in most cases) the user to implement some functionality. In particular, the user may implement the following classes:

Class Interface	To Implement Functionality
NxUserAllocator	Memory Management
NxUserOutputStream	Error Reporting
NxUserContactReport	Contact Information
NxUserNotify	Misc. events, including joint breakage
NxUserRaycastReport	Ray Casting Information
NxUserTriggerReport	Trigger Information
NxUserEntityReport	The user passes this to a number of collision detection routines to receive results
NxStream	Data Serialization

Do not modify the SDK state with any set()-type method from a user class method. An error will occur if this is attempted; therefore, deleting SDK objects such as bodies or joints from within one of the above class methods is forbidden.

The following example shows how to create a user defined class:

```
class MyAllocator : public NxUserAllocator
{
     public:
     void * malloc(NxU32 size)
     {
        return ::malloc(size)
     }

     void * mallocDEBUG(NxU32 size,const char *fireName, int line)
     {
        return ::_malloc_dbg(size,_NORMAL_BLOCK, fileName, line);
     }

     void * realloc(void * memory, NxU32 size)
     {
        return ::realloc(memory,size);
     }

     void free(void * memory)
     {
        ::free(memory);
     }
} myAllocator;

gPhysicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION, &myAllocator, 0);
```

This creates a very simple memory allocator class which uses the standard C memory allocation routines. The main purpose of a custom allocator is to use a custom memory allocator scheme, as described in the Memory Management section.

API Reference

- NxUserAllocator
- NxUserOutputStream
- NxUserContactReport

- NxUserNotify
- NxUserRaycastReport
- NxUserTriggerReport
- NxUserEntityReport
- NxStream

Error Reporting

The Physics SDK provides the possibility to supply a user defined stream class for error logging. Basically, the user needs to subclass the abstract base class NxUserOutputStream:

```
class MyOutputStream : public NxUserOutputStream
{
    void reportError (NxErrorCode code, const char *message, const char* file, int line)
    {
        //this should be routed to the application
        //specific error handling. If this gets hit
        //then you are in most cases using the SDK
        //wrong and you need to debug your code!
        //however, code may  just be a warning or
        //information.

        if (code < NXE_DB_INFO)
        {
            MessageBox(NULL, message, "SDK Error", MB_OK),
            exit(1);
        }
    }

    NxAssertResponse reportAssertViolation (const char *message, const char *file,int line)
    {
        //this should not get hit by
        // a properly debugged SDK!
        assert(0);
        return NX_AR_CONTINUE;
    }

    void print (const char *message)
    {
        printf("SDK says: %s\n", message);
    }

} myOutputStream;
```

This object is then passed to NxCreatePhysicsSDK() :

```
gPhysicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION, 0, &myOutputStream);
```

Currently, messages are logged in standard syntax when an error occurs. For example:

```
c:\NVIDIA\SDKs\Physics\src\PhysicsSDK.cpp(123) : error 3: invalid parameter
```

If a stream object has been supplied, such an error message will be generated. In addition, the reportError() member of the error stream will be called with a numeric error code.

NOTE: No method from the SDK should be called from within the error reporting functions.

Threading

Errors are only reported in the context of the user thread, so it is not necessary to provide a thread safe error reporting callback.

API Reference

- NxUserOutputStream

Saving the Simulation State

Most of the simulation objects that are created using a descriptor have two methods: saveToDesc() and loadFromDesc(). These methods are useful for saving the simulation state, and then restoring it at a later time. In games, this can be used for saving games and demo playback. To save the state, call saveToDesc() on the object to save, and all the necessary state information will be written into the descriptor. Be sure to check the API documentation of saveToDesc() for the object at hand, because sometimes additional measures may be needed. The user is then responsible for serializing the data contained in the descriptor. This sometimes involves the conversion of pointers to an equivalent persistent identifier.

Later, when the object is to be restored, you must deserialize the descriptor data to its original state, and then either call loadFromDesc() on an existing object, or, better yet, create a new object by passing the descriptor to the create*() call.

Previous versions of the SDK provided explicit "core dump" functionality. This has been superseded by enhanced support for serialization. The same functionality is provided by the serialization class library.

SDK Parameters

The Physics SDK has a set of centrally managed, global parameters. They are defined by the enumeration NxParameter and can be accessed by the NxPhysicsSDK::setParameter() and getParameter() methods. Most simulations should not need to change these parameters.

Parameter Summary

Parameter	Default Value	Description
NX_PENALTY_FORCE		DEPRECATED
NX_SKIN_WIDTH	0.025	Default value for NxShapeDesc::skinWidt info. (range: [0, inf) Unit: distance. See Sk
NX_DEFAULT_SLEEP_LIN_VEL_SQUARED	(0.15*0.15)	The default linear velocity, squared, below sleep. (range: [0, inf))
NX_DEFAULT_SLEEP_ANG_VEL_SQUARED	(0.14*0.14)	The default angular velocity, squared, belo sleep. (range: [0, inf))
NX_BOUNCE_TRESHOLD	-2	A contact with a relative velocity below th 0])
NX_DYN_FRICT_SCALING	1	This lets the user scale the magnitude of th objects. (range: [0, inf))
NX_STA_FRICT_SCALING	1	This lets the user scale the magnitude of th objects. (range: [0, inf))
NX_MAX_ANGULAR_VELOCITY	7	See API Reference for NxBody::setMaxAn
NX_CONTINUOUS_CD	0	Enable/disable continuous collision detecti
NX_VISUALIZATION_SCALE	0	This overall visualization scale gets multip Setting to zero turns off debug visualizatio
NX_VISUALIZE_WORLD_AXES	0	Visualize the world axes.
NX_VISUALIZE_BODY_AXES	0	Visualize a body's axes.
NX_VISUALIZE_BODY_MASS_AXES	0	Visualize a body's mass axes.
NX_VISUALIZE_BODY_LIN_VELOCITY	0	Visualize the bodies' linear velocity.
NX_VISUALIZE_BODY_ANG_VELOCITY	0	Visualize the bodies' angular velocity.
NX_VISUALIZE_BODY_JOINT_GROUPS	0	Visualize joint groups.
NX_VISUALIZE_JOINT_LOCAL_AXES	0	Visualize local joint axes.
NX_VISUALIZE_JOINT_WORLD_AXES	0	Visualize joint world axes.
NX_VISUALIZE_JOINT_LIMITS	0	Visualize joint limits.
NX_VISUALIZE_CONTACT_POINT	0	Visualize contact points.
NX_VISUALIZE_CONTACT_NORMAL	0	Visualize contact normals.
NX_VISUALIZE_CONTACT_ERROR	0	Visualize contact errors.

NX_VISUALIZE_CONTACT_FORCE	0	Visualize contact forces.
NX_VISUALIZE_ACTOR_AXES	0	Visualize actor axes.
NX_VISUALIZE_COLLISION_AABBS	0	Visualize bounds (AABBs in world space).
NX_VISUALIZE_COLLISION_SHAPES	0	Visualize shape.
NX_VISUALIZE_COLLISION_AXES	0	Visualize shape axes.
NX_VISUALIZE_COLLISION_COMPOUNDS	0	Visualize compound (compound AABBs in world
NX_VISUALIZE_COLLISION_VNORMALS	0	Visualize mesh & convex vertex normals.
NX_VISUALIZE_COLLISION_FNORMALS	0	Visualize mesh & convex face normals.
NX_VISUALIZE_COLLISION_EDGES	0	Visualize active edges for meshes.
NX_VISUALIZE_COLLISION_SPHERES	0	Visualize bounding spheres.
NX_VISUALIZE_COLLISION_STATIC	0	Visualize static pruning structures.
NX_VISUALIZE_COLLISION_DYNAMIC	0	Visualize dynamic pruning structures.
NX_VISUALIZE_COLLISION_FREE	0	Visualize "free" pruning structures.
NX_VISUALIZE_COLLISION_CCD	0	Visualize CCD tests.
NX_VISUALIZE_COLLISION_SKELETONS	0	Visualize CCD skeletons.
NX_VISUALIZE_FLUID_EMITTERS	0	Visualize emitter.
NX_VISUALIZE_FLUID_POSITION	0	Visualize particle position.
NX_VISUALIZE_FLUID_VELOCITY	0	Visualize particle velocity.
NX_VISUALIZE_FLUID_KERNEL_RADIUS	0	Visualize particle kernel radius.
NX_VISUALIZE_FLUID_BOUNDS	0	Visualize fluid AABB.
NX_VISUALIZE_FLUID_PACKETS	0	Visualize fluid packets.
NX_VISUALIZE_FLUID_MOTION_LIMIT	0	Visualize fluid motion limits.
NX_VISUALIZE_FLUID_DYN_COLLISION	0	Visualize fluid dynamic mesh collision.
NX_VISUALIZE_FLUID_STC_COLLISION	0	Not implemented: Visualize fluid static collision.
NX_VISUALIZE_FLUID_MESH_PACKETS	0	Visualize available fluid mesh packets.
NX_VISUALIZE_FLUID_DRAINS	0	Visualize fluid drain shapes.
NX_VISUALIZE_FLUID_PACKET_DATA	0	Visualize fluid data packets.
NX_VISUALIZE_CLOTH_MESH	0	Visualize cloth meshes.
NX_VISUALIZE_CLOTH_COLLISIONS	0	Visualize cloth rigid body collision.
NX_VISUALIZE_CLOTH_SELFCOLLISIONS	0	Visualize cloth self collision.
NX_VISUALIZE_CLOTH_WORKPACKETS	0	Visualize cloth clustering for the PPU.
NX_VISUALIZE_CLOTH_SLEEP	0	Visualize cloth sleeping.
NX_VISUALIZE_CLOTH_SLEEP_VERTEX	0	Visualize cloth sleeping with full per-vertex inform
NX_VISUALIZE_CLOTH_TEARABLE_VERTICES	0	Visualize tearable cloth vertices.
NX_VISUALIZE_CLOTH_TEARING	0	Visualize cloth tearing.
NX_VISUALIZE_CLOTH_ATTACHMENT	0	Visualize cloth attachments.
NX_VISUALIZE_CLOTH_VALIDBOUNDS	0	Visualize cloth valid bounds.
NX_VISUALIZE_SOFTBODY_MESH	0	Visualize soft body meshes.
NX_VISUALIZE_SOFTBODY_COLLISIONS	0	Visualize soft body collisions with rigid bodies.
NX_VISUALIZE_SOFTBODY_WORKPACKETS	0	Visualize soft body clustering for simulation on the
NX_VISUALIZE_SOFTBODY_SLEEP	0	Visualize soft body sleeping.

NX_VISUALIZE_SOFTBODY_SLEEP_VERTEX	0	Visualize soft body sleeping with full per-v
NX_VISUALIZE_SOFTBODY_TEARABLE_VERTICES	0	Visualize tearable soft body vertices.
NX_VISUALIZE_SOFTBODY_TEARING	0	Visualize soft body tearing.
NX_VISUALIZE_SOFTBODY_ATTACHMENT	0	Visualize soft body attachments.
NX_VISUALIZE_SOFTBODY_VALIDBOUNDS	0	Visualize soft body valid bounds.
NX_ADAPTIVE_FORCE	1	Used to enable adaptive forces to accelerat
NX_COLL_VETO_JOINTED	1	Controls default filtering for jointed bodies
NX_TRIGGER_TRIGGER_CALLBACK	1	Controls whether two touching triggers ge
NX_SELECT_HW_ALGO	0	Internal parameter, used for debugging and
NX_VISUALIZE_ACTIVE_VERTICES	0	Internal parameter, used for debugging and
NX_CCD_EPSILON	0.01	Distance epsilon for CCD algorithm.
NX_SOLVER_CONVERGENCE_THRESHOLD	0	Used to accelerate the solver.
NX_BBOX_NOISE_LEVEL	0.001	Used to accelerate HW Broad Phase.
NX_IMPLICIT_SWEEP_CACHE_SIZE	5.0	Used to set the sweep cache size.
NX_DEFAULT_SLEEP_ENERGY	0.005	The default sleep energy threshold. Objects threshold are allowed to go to sleep. **Note:** Only used when the NX_BF_ENER
NX_CONSTANT_FLUID_MAX_PACKETS	925	Constant for the maximum number of pack the fluid packet buffer size in NxFluidPack
NX_CONSTANT_FLUID_MAX_PARTICLES_PER_STEP	4096	Constant for the maximum number of new
NX_VISUALIZE_FORCE_FIELDS	0	Force field visualization.
NX_ASYNCHRONOUS_MESH_CREATION	0	[Experimental] Disables scene locks when
NX_FORCE_FIELD_CUSTOM_KERNEL_EPSILON	0.001	Epsilon for custom force field kernels.
NX_IMPROVED_SPRING_SOLVER	1	Enable/disable improved spring solver for
NX_PARAMS_NUM_VALUES		This is not a parameter, it just records the c

API Reference

- NxPhysicsSDK
- NxScene

Parameter Ranges

In order to provide a robust simulation, it is important to make sure that all parameters passed to the SDK are within a reasonable range. The API reference gives theoretical ranges for each parameter/structure member as a guide.

In practice, it is important to ensure that all quantities are within a similar order of magnitude. See Solver Accuracy for notes on improving the stability of your simulations.

The API reference refers to a number of abstract quantities, which are discussed below, when suggesting ranges for parameters. These quantities are inter-dependent and should all be kept within a tight range. For example, when specifying a linear velocity you also need to consider the magnitude of the objects mass to ensure that the momentum of the object does not become larger than the range of numbers which can be accurately represented.

Examples

[or] - Represents an inclusive bound, i.e., the range of allowable values includes the end value.

(or) - Represents an exclusive bound, i.e., the range of allowable values does not include the end value.

[0,inf) - All values greater than or equal to zero are allowed.

[-inf,inf] - All values are allowed, including infinity (represented in practice with NX_MAX_REAL).

[0,param1] - All values are allowed between zero and param1, inclusive of zero and param1.

Quantities

- **rigid body transform** - A combination of a position vector and a rotation matrix.
- **position vector** - Positon vector, in local or world space. Local space position vectors should still be within a reasonable range even when transformed to world space.
- **rotation matrix** - A rotation matrix, i.e., the determinant of the matrix should be one (within a small epsilon) and the inverse should equal the transpose.
- **unit quaternion** - A unit length quaternion, used to represent a rotation.
- **direction/extents vector** - Offset/Direction vector, in local or world space. Should still be within a reasonable range when added to an associated position vector and/or transformed into world space.
- **force vector** - Linear force vector. Forces are added to the momentum of a body; care should be taken to keep the momentum from growing too large.
- **torque vector** - Torque vector. A torque is added to the angular momentum of a body; care should be taken to keep the angular momentum within a reasonable range.
- **velocity vector** - Linear velocity vector. Consider the magnitude of the velocity when multiplied by the mass, i.e., a large velocity when applied to a body with a large mass will result in a very large momentum.
- **angular velocity vector** - Angular velocity vector. Consider the magnitude of the angular velocity when multiplied by the inertia tensor, i.e., the angular momentum.
- **momentum vector** - Linear momentum vector.
- **angular momentum vector** - Angular momentum vector.

Utility Functions

The SDK provides a range of utility functions that are useful when working with physical simulations. Some of these functions are listed below:

- Intersection Testing, e.g., NxBoxBoxIntersect()
- FPU control, e.g., NxSetFPUPrecision24()
- Shape manipulation, e.g., NxComputeBoxAroundCapsule()
- Inertia Tensors, e.g., NxDiagonalizeInertiaTensor()
- Other math functions, e.g., NxFindRotationMatrix()

Refer to the API reference for a complete list.

In versions of the SDK prior to 2.3, these functions were exposed as static exports from NxPhysics.dll. But with the move to PhysXLoader.lib / PhysXCore.dll, exporting the functions in this way is no longer supported.

Instead these functions must be accessed through the NxUtilLib interface. To obtain a reference to an NxUtilLib object you can call NxGetUtilLib() after calling NxCreatePhysicsSDK(). The reference returned is a singleton, i.e., multiple calls to NxGetUtilLib() will return the same object.

To provide a measure of source compatibility with applications written for previous versions of the SDK, inline wrapper functions, which call these functions, are provided. However, unlike previous versions of the SDK, NxCreatePhysicsSDK() must be called before using the functions.

For new code it may be preferable to obtain a reference to NxUtilLib and repeatedly call its members directly. Below is an example:

```
NxUtilLib *gUtilLib;
...

gUtilLib=NxGetUtilLib();
...

gUtilLib->NxSetFPUPrecision24();
gUtilLib->NxFindRotationMatrix(vecA,vecB,M);
```

API Reference

- NxUtilLib

Dynamics

The key ideas underlying the SDK's simulation functionality are described in this section.

Scenes

Simulations take place in scenes (abstracted by the NxScene class). A scene is a container for a simulation's actors, joints, and effectors. Several scenes can be created and filled with objects, then simulated in parallel. The simulation of two scenes is completely disjointed, so objects in different scenes do not influence each other. However, you may add external forces which lead to a sort of communication between scenes.

Scenes do not have any particular spatial extents; they can be thought of as infinite. They also provide various practical functionalities such as a uniform gravity field that influences all of the contained objects, ray casting, enabling and disabling collision detection, and so on. Of course, all of these features only work with the objects that exist in the given scene.

Most simulations will only use a single scene. One practical application for multiple scenes is a client/server multi player game where a single process must perform the simulation for the server and a local client. This process would create two scenes. The client scene would simulate only the immediate perceivable vicinity of the client's avatar, while the server scene would simulate the entire world.

A scene can be created by leaving all of the scene descriptor's properties at their defaults. A common member to set to a special value would be gravity. The scene gravity may be changed at any time using NxScene::setGravity().

Example

```
NxSceneDesc sceneDesc;

sceneDesc.gravity.set(0,-9.8f,0);

gScene = gPhysicsSDK->createScene(sceneDesc);
if (!gScene)     Error("Can't create scene!");
```

API Reference

- NxScene
- NxSceneDesc
- NxPhysicsSDK

Simulation Timing

The main feature of a scene is its capability to actually perform the physics simulation. This results in various object properties evolving over time, e.g., bodies' positions and velocities.

Simulation is done one time step at a time, typically using a fixed step size ranging between 1/100th of a second and 1/50th of a second. For a real-time simulation, the application must perform several of these time steps to synchronize the physics behavior with the rendered frame rate.
Note that longer timesteps lead to poor stability in the simulation, and that values much above the aforementioned range are not recommended.

The main method used for advancing the simulation is the following:

```
void simulate(NxReal elapsedTime);
```

Calling this method instructs the SDK to advance the simulation by elapsedTime; however, the exact behavior of this method is dependent on the setTiming() method shown below:

```
void setTiming(NxReal maxTimestep=1.0f/60.0f, NxU32 maxIter=8, NxTimeStepMethod method=NX_TIM
```

Timing parameters can also be set in the scene descriptor prior to scene construction. The default settings are appropriate for most applications.

Using fixed time sub steps is very important to ensure a stable and reproducible simulation and is recommended for most applications, specified using NX_TIMESTEP_FIXED.

Internally, the SDK can divide elapsedTime into a number of fixed time sub steps. When an application calls simulate() with a value of elapsedTime greater than maxTimestep, the SDK executes as many whole sub steps of maxTimestep length as possible and accumulates the remainder to be added onto the next call to simulate(). maxIter provides a cap on the number of sub steps executed, per time step. If the number of possible sub steps exceeds maxIter, then maxIter sub steps are executed and the remaining time is added on to the accumulator to be executed on the next time step.

The alternative to use variable time steps (NX_TIMESTEP_VARIABLE) is still available. When NX_TIMESTEP_VARIABLE is specified, the SDK does not divide the time steps, giving the user direct control over the stepping.

Recommended Time Stepping Method

The recommended time stepping method for a typical application is fixed time steps where maxTimestep is an exact multiple of elapsedTime and elapsedTime is a constant. This way, the user knows the number of sub steps taken, which do not vary. (elapsedTime is the parameter passed to simulate())

The advantage is that the behavior, when applying forces and moving kinematic actors, is deterministic. If the user executes a variable number of sub steps on each call to simulate, then the time period over which forces are applied and kinematic actors move will vary (causing the velocity of kinematic actors to vary). When accounting for variable frame rates, the user can make multiple calls to simulate, with new values supplied to moveGlobalPose() and addForce(), etc.

Example

```
NxScene* gScene;
NxReal myTimestep = 1.0f/60.0f;
gScene->setTiming(myTimestep / 4.0f, 4, NX_TIMESTEP_FIXED);
```

```
...

void mySimulationStepFunction()
{

    gScene->simulate(myTimestep); //perform 4 substeps to advance simulation by 1/60th of a second.
    gScene->flushStream();

    //...perform useful work here using previous frame's state data

    gScene->fetchResults(NX_RIGID_BODY_FINISHED, true);
}
```

API Reference

- NxScene

Asynchronous Stepping

The NVIDIA PhysX SDK is multi-threaded; the physics simulation calculations run in their own thread, separate from the application thread. The state of the simulation is updated by calling a sequence of functions that (1) start the simulation, (2) ensure that all necessary data has been sent to the simulation thread, (3) check to see whether the simulation is finished, and if so, update the state data in the buffer, and (4) swap the state data buffers so that the next simulation step will be performed on the alternate buffer, leaving the current results accessible to the application. The function sequence is illustrated in the following snippet of pseudocode:

```
NxScene* gScene;
NxReal myTimestep = 1.0f/60.0f;
...

void mySimulationStepFunction()
{

    gScene->simulate(myTimestep);
    gScene->flushStream();

    //...perform useful work here using previous frame's state data

    gScene->fetchResults(NX_RIGID_BODY_FINISHED, true);
}
```

There are several variations to the fetchResults() function call, selectable by the parameters of the function, or by using the function checkResults() in its place. The version illustrated above is a blocking function: the function will not return until the simulation thread has completed its calculations on all rigid bodies (i.e., NX_RIGID_BODY_FISHED == true). Using non-blocking calls, the application can do more useful work on the previous frame's state data, or work unrelated to the physics data, while the simulation thread is processing the calculations:

```
void mySimulationStepFunction()
{
    gScene->simulate(myTimestep);
    gScene->flushStream();

    //...perform useful work here using previous frame's state data

    while(!gScene->checkResults(NX_RIGID_BODY_FINISHED, false)
    {
        // do something useful
    }

    gScene-> fetchResults(NX_RIGID_BODY_FINISHED, true);

}
```

or

```
void mySimulationStepFunction()
{
    gScene->simulate(myTimestep);
    gScene->flushStream();

    //...perform useful work here using previous frame's state data

    while(!gScene->fetchResults(NX_RIGID_BODY_FINISHED, false)
    {
```

```
            // do something useful
        }

    }
```

The parameter NX_RIGID_BODY_PHYSICS is a flag of type NxSimulationStatus.

For additional information, review the NxScene.h header file and related API reference documentation.

NOTE: Prior to v2.1.2 of the SDK, the simulation state was advanced through the NxScene::startRun() and NxScene::finishRun() function set. These functions have been changed as of v2.1.2, and should be replaced with the NxScene::simulate() and NxScene::fetchResults() function set.

NOTE: Because the SDK double buffers data to allow asynchronous stepping state modification is not visible to some functions(eg overlap and raycasting) until a simulate()/fetchResults() pair has been executed.

Asynchronous Physics Programming

Next-generation platforms are moving toward multi-processor architectures to provide more computing power for rendering, animation, AI, physics, sound, and other game elements. This is the basic motivation underlying the SDK's transition to multi-threaded physics simulation, as described above. In a multiprocessor environment, the thread running the simulate() / fetchResults() function block can be offloaded from the main processor, and in a properly designed game engine, expand the scale and quality of physics content while providing a better and more consistent frame rate.

A multi-threaded physics API is the main ingredient for achieving state-of-the-art game play physics as well as physical effects, but designers of game engines for next-generation platforms face challenges surrounding synchronization, load balancing, and system bottlenecks. NVIDIA is committed to providing leading-edge products for next-gen systems, along with the support necessary to maximize their potential.

Samples

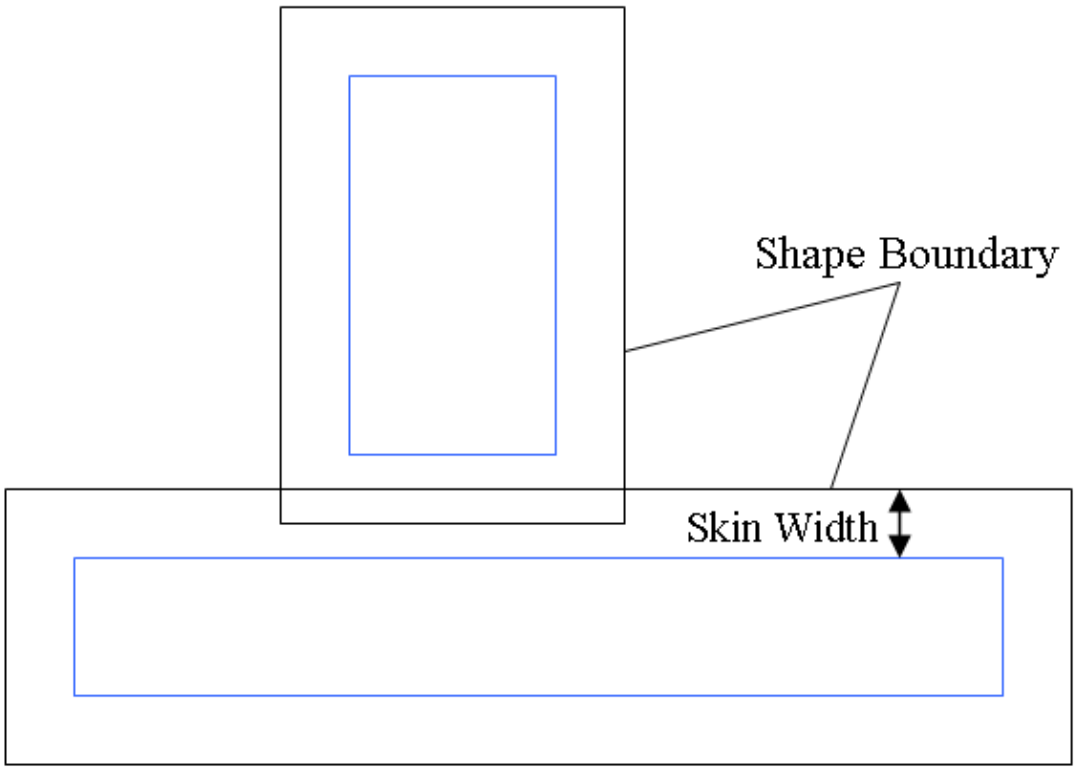
Sample Async Boxes

API Reference

- NxScene

Skin Width

The simulation deals with inaccuracy when stacking objects by letting them slightly inter penetrate each other. The amount of permitted inter penetration can be regulated at a scene level using the NX_SKIN_WIDTH parameter. Clearly, a lot of inter penetration is visually undesirable. On the other hand, forbidding inter penetration is even worse, because objects may repel each other to the point where they separate, and then fall back down on each other in a subsequent time frame. This leads to visible jittering. The amount of inter penetration that is best permitted depends on many things such as the size of the objects involved (so that the inter penetrating region is visually negligible) but also on the stability of the simulation, which is usually governed by the gravity setting as well as the time step size. (Lower gravity and smaller time steps typically result in more stable simulations.)



Skin width specifies how much objects can inter penetrate, as opposed to how much they are separated.

There are two options for setting the same property; either by using the following method:

```
void NxShape::setSkinWidth(NxReal skinWidth);
void NxShape::getSkinWidth();
```

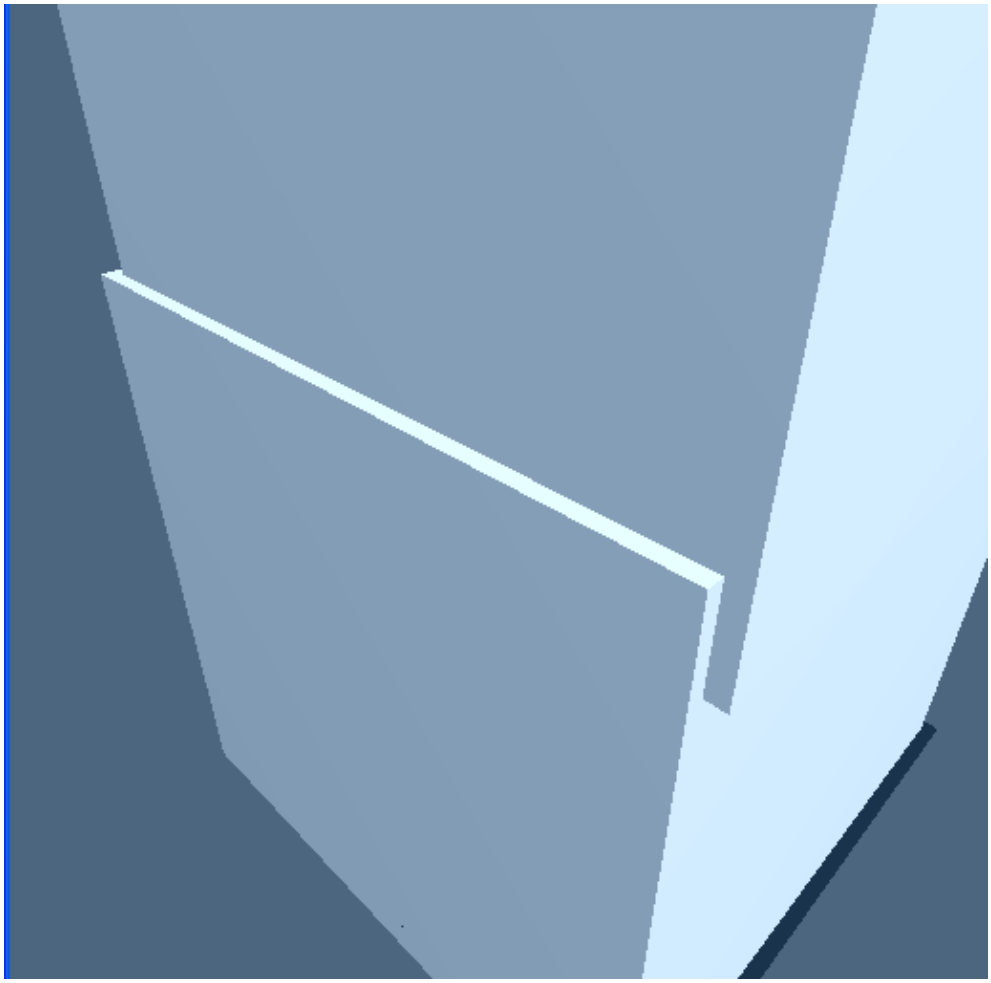
or by using the skinWidth member of NxShapeDesc.

To reduce the visual effect of interpenetrations due to the skin width, it is possible to inflate the size of physics objects with respect to their graphical representation. The SDK provides support for this when cooking convex meshes. See NxCookingParams::skinWidth. For other types of shapes, the user must manually scale shapes when providing them to the API.

Example

Object with a small skin width:

Object with a larger skin width:



API Reference

- NxScene
- NxShape

Solver Accuracy

When the motion of a rigid body is constrained either by contacts or joints, the constraint solver comes into play. The solver satisfies the constraints on the bodies by reiterating the constraints restricting the motion of the body a certain number of times. The more iterations, the more accurate the results become. The solver iteration count defaults to 4, but may be set individually for each body using the solverIterationCount field of NxBodyDesc. So far we have only found it necessary to set this significantly higher for objects with lots of joints and a small tolerance for joint error. A setting higher than 30 has not been exceeded.

The solver iteration count places an effective limit on the number of bodies that can be jointed together and simulated realistically; In the worst case, forces working on a body will only affect connected bodies so many joints distant.

Fast Rotation

Objects shaped like a pencil are difficult to simulate because they can store a lot of energy while rotating around a short axis. This energy is converted to a very high rotational velocity when the shape rotates around a longer axis. High rotational velocities can lead to problems because certain linear approximations of the rotational motion fail to hold. For this reason, the SDK automatically limits the rotational velocity of a body to a user definable maximum value. Because this may prevent intentional fast rotation in objects such as wheels, the user can override it on a per body basis with the following:

```
actor->setMaxAngularVelocity(maxAV);
```

Floating Point

For consistent and stable simulation results, it is necessary to take care when choosing the magnitude of quantities involved in the simulation. It is important to keep the distances, masses, and forces involved in the simulation within a tight range to avoid loss of precision. The exact range of values you use is dependent on the scale of the world, the complexity of the joints, the number of contacts, etc.

Before you begin designing your physically simulated levels, etc., it is a good idea to come up with guidelines for these values.

An example might be the following:

Quantity	**Range/Value**
Mass	1-10
Position	-4000 - 4000
Gravity	-9
Solver Iterations	5
...	...

Adaptive force

By default, PhysX uses an adaptive force scheme, in which the acceleration of bodies that are touching large numbers of other bodies is scaled down (on condition that some of the bodies are in contact with a static object). This causes a dramatic improvement in solver stability in many cases.
Adaptive force may in some instances lead to unwanted physical artifacts in the way objects move. Because of this, there is the option of disabling adaptive force throughout the SDK:

```
gPhysicsSDK->setParameter(NX_ADAPTIVE_FORCE, 0);
```

However, because of the abovementioned stability implications, this should not be done lightly.

Tips

- Use joint springs instead of spring and damper effectors. Joint springs are implicitly integrated within the solver and correctly account for transfer of momentum across the spring, which also improves behavior with sleeping bodies. However, spring and damper effectors are explicitly integrated and are not as numerically stable.
- When using springs, try not to make them too stiff (i.e., keep the spring constant and amount of damping relatively low).
- Make sure that you set the bodies center of mass to a reasonable value. As a rule of thumb, the center of mass should be inside the collision volume of the object if possible. This is especially important if the body has springs attached to it. A poorly tuned center of mass can cause the system to be poorly conditioned, thus the SDK solver may have trouble converging on a solution. The symptom of this is generally jittering bodies.
- The stability of the simulation when stacking objects can be improved by carefully tuning the skin width.

Shapes in Actors

Overview

An actor typically contains shapes - sometimes one, sometimes more, sometimes none. The shapes represent the physical space the actor occupies in the simulation, and are the core of collision detection.

You can create an actor with no shapes, which will then behave as a 'ghost' and collide with nothing (though it may still be jointed to other actors and affected by gravity and other applied forces or torques).
However, a shapeless actor must have a body, i.e. it may not be static.

Usage

Shapes are assigned to an actor before creating it, by inserting shape descriptors into the 'shapes' array of the actor descriptor:

```
NxSphereShapeDesc shapeDesc;
//Set properties of the shape
...
NxActorDesc actorDesc;
//Set properties of the actor
...
actorDesc.shapes.pushBack(&shapeDesc);
gScene->createActor(actorDesc);
```

You can also add shapes to an existing actor using the 'createShape' method of NxActor:

```
NxSphereShapeDesc shapeDesc;
//Set properties of the shape
...
actor->createShape(shapeDesc);
```

Finally, shapes can be removed from an actor using 'releaseShape'. You cannot release the last shape of a static actor this way. The following requirements exist when creating an actor, depending on the shapes making it up:

- A static actor must have at least one shape.
- An actor with no solid shapes (in other words, either no shapes at all or nothing but trigger shapes) must have a body (i.e. be dynamic), with a specified mass and inertia tensor.
- A dynamic actor with solid shapes must have either:
    - A non-zero mass, zero density and no inertia tensor (the tensor will be computed based on the mass)
    - Zero mass, non-zero density and no inertia tensor (mass and tensor will be computed based on density)
    - Zero density, but specified mass and inertia tensor

  otherwise the actor will not be created.

See inertia tensor to learn more about the inertia tensor.

Compound shapes

Whenever more than one shape are assigned to an actor, a compound shape will be created automatically. The compound acts as a container for several shapes belonging to one rigid body. A compound shape is conceptually similar to actors held together by fixed joints; however, unlike fixed joints compound shapes are

always kept perfectly rigid and incur no performance cost when simulating.

There are some performance implications of compound shapes that the user should be aware of:

- Adding a new shape to a non-compound incurs an overhead as a compound shape is created and the old shape is moved into it.
- Adding shapes to an existing compound is more time-consuming than creating the actor with all shapes at once.
- You should avoid static actors being compounds; there's a limit to the number of triangles allowed in one actor's mesh shapes and subshapes exceeding the limit will be ignored.

API Reference

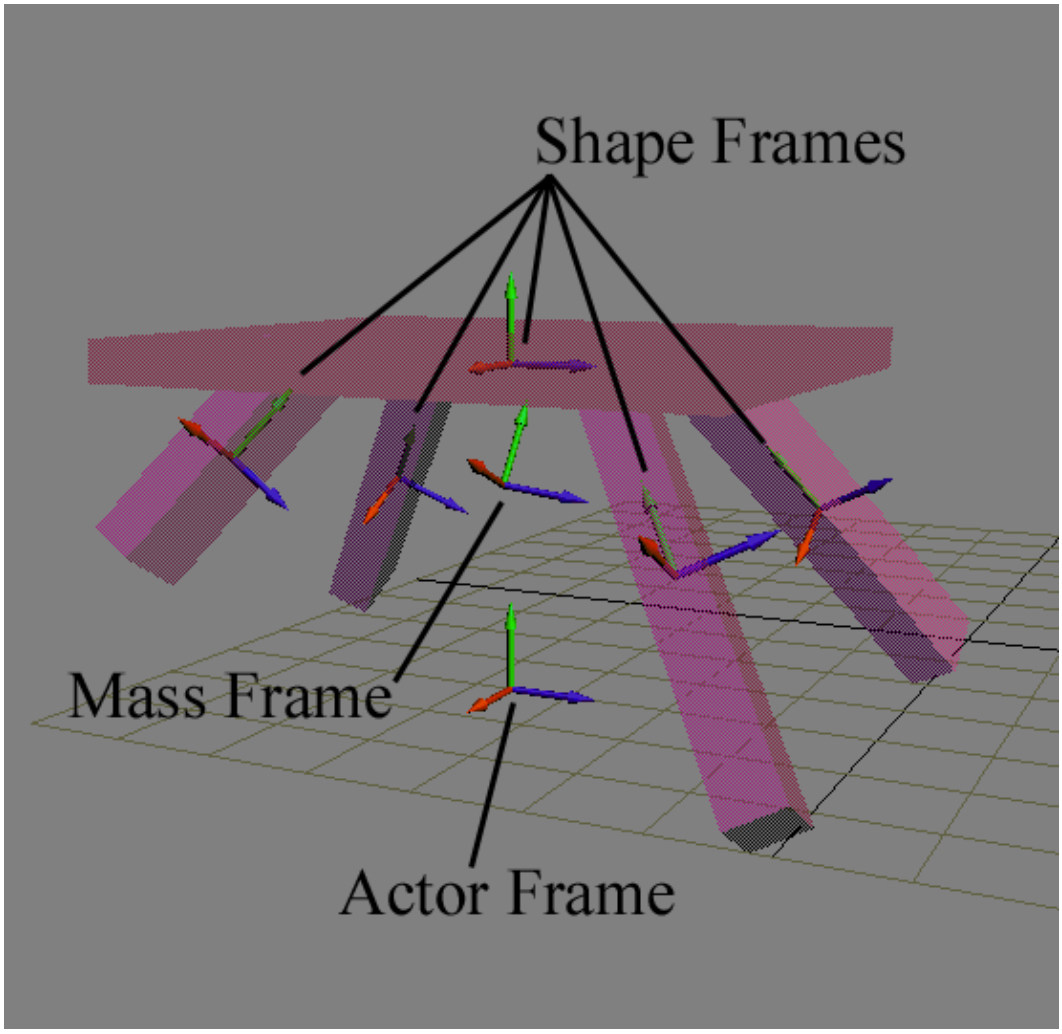
- NxActor
- NxActorDesc
- NxShape
- NxShapeDesc

Reference Frames

The most important property of an actor is its pose (position and orientation) in the world. Because the SDK provides a good deal of flexibility in this area, it is important to understand some spatial relationships involved.

For example, suppose we would like to simulate a table. The table is modeled in 3d software using a rectangular table top and four rectangular legs. The table will be an art asset for a computer game, and it will be instanced and placed at various spots using a level editor.

Both high detail geometry and bounding boxes for the table top and legs are exported from the 3d software (either manually or automatically). The physics SDK is not concerned with the graphical representation, but the bounding boxes will be instanced in the SDK as shape objects. The image below shows what the table may look like in the game editor, along with its significant reference frames.



NOTE: To make things more interesting, our table has a large chunk cut from one leg.

The space in which all the actors are positioned is called the world or global space. A space is also sometimes called a frame or reference frame.

The actor object corresponding to the entire table has its own frame, the actor frame. This is also known as the local space of the actor. The actor frame is defined so that when placed at zero height in the world, the legs of the table touch the floor, making it easy for people editing the level to place tables in a simple way, eventually using the editor's snap-to-grid functionality to make sure the table is at a correct height. In other words, actor

frames can be randomly placed and are usually chosen with other practical or artistic considerations in mind.

In regards to collision detection, the table is made up of five bounding box shapes. These have fixed poses relative to the actor frame. At its origin, defined as center, each box has its own local frame called the shape frame, oriented along the primary axis of the box. Other types of shapes have similar local frames defined. For example, the local frame of a triangle mesh is the space in which the triangle vertices are defined (which can otherwise be arbitrary).

Given the placement of shapes, the SDK computes the actor's center of mass, assuming it is dynamic. (You may manually specify a center of mass if preferred.) The center of mass is the point around which real physical objects rotate unless they are constrained in some way. The center of mass may have an orientation which is not always the same as that of the actor frame. In our example, the table is missing half of a leg. This makes the mass distribution a bit asymmetric, physically expressed by the center of mass frame's rotation. The center of mass frame is often abbreviated to body or mass frame. The center of mass pose is stored relative to the actor frame.

The NxActor methods,

```
void setGlobalPose(const NxMat34&);
void setGlobalPosition(const NxVec3&);
void setGlobalOrientation(const NxMat33&);
void setGlobalOrientationQuat(const NxQuat&);
```

serve to position the actor frame in world space, and the corresponding get*() methods retrieve it.

Note that the method,

```
const NxMat34& getGlobalPose()const
```

is normally the most convenient way to obtain the actor's transform for rendering. This can then be sent to OpenGL (or similarly to Direct3D) through the following:

```
float glmat[16];//OpenGL matrix.
actor->getGlobalPose()->getColumnMajor44(glmat);
glMultMatrixf(glmat);//Send to OpenGL.
```

The methods,

```
void setCMassOffsetLocalPose(const NxMat34&);
void setCMassOffsetLocalPosition(const NxVec3&);
void setCMassOffsetLocalOrientation(const NxMat33&);
```

set the mass frame's pose relative to the actor frame, or in other words, determine the actor's center of mass. This is computed and defaulted by the SDK to the correct values based on the mass distribution of the actor's shapes. You may instead use the follwing method to set the mass center's frame relative to the global frame:

```
void setCMassOffsetGlobalPose(const NxMat34&);
void setCMassOffsetGlobalPosition(const NxVec3&);
void setCMassOffsetGlobalOrientation(const NxMat33&);
```

To set the pose of the component boxes (or other shapes) relative to the actor frame, the following methods of NxShape may be used:

```
void setLocalPose(const NxMat34&);
void setLocalPosition(const NxVec3&);
void setLocalOrientation(const NxMat33&);
```

Setting the position of an object directly is sometimes unavoidable (most commonly at the start of a simulation, or if you want to 'teleport' an object to a different place), but may also be convenient.

The SDK does not prevent you from setting the poses of constrained objects. For example, if you had a linked chain of bodies, and added a certain value to the bodies' positions, the simulation would continue without problem. However, if you changed the position of only one body, the simulation would try to fix the constraint error you have inadvertently introduced by applying huge forces to the bodies during the next time step. This may lead to the simulation failing or the scene 'exploding'. To summarize, setting the pose of objects directly is not necessarily a bad thing, just watch out that you don't accidentally pull joints apart or put bodies into each other - your collision detection system won't like it.

NOTE: When specifying the orientation, there is no performance difference between the quaternion or the matrix representation, as the SDK uses both formats internally.

Samples

Sample Transforms

API Reference

- NxActor

Rigid Body Properties

Dynamic actors have a set of special properties that are used for rigid body simulation. One additional property of a body is that it has a center of mass position and orientation, as discussed in the Reference Frames section. At this center point, some of the mass is concentrated; its spatial distribution is expressed using the inertia tensor. Another is that it has velocity. Forces may act on the body to produce the motion, as long as this is not prevented by colliding shapes or joints.

NOTE: All rigid body properties have a linear and angular version. The linear version is used in computations involving rectilinear motion along a path, and the angular version is used when the body is rotating. Most movement is a combination of both.

Linear Quantity	**Angular Quantity**
mass (scalar)	inertia (3-vector) - The mass distribution of the body along its three dimensions.
position (3-vector) - The position of the body's center of mass relative to the actor's frame.	orientation (quaternion or 3x3-matrix) - The orientation of the principal moments relative to the actor's frame.
velocity (3-vector) - Linear velocity.	angular velocity (3-vector) - Represented as an axis of rotation with a length equal to the magnitude of the angular velocity.
force (3-vector) - The amount of force that is being exerted on this body.	torque (3-vector) - The amount of angular force that is being exerted on this body. Represented as an axis of rotation with a length equal to the magnitude of the angular velocity.

Initial settings for these properties may be assigned to NxBodyDesc * NxActorDesc::body, as well as queried and changed using appropriate methods of NxActor.

API Reference

- NxActor
- NxBodyDesc

---

2 Rigid Body Properties

The Inertia Tensor

For developers lacking dynamics experience, inertia tensor may be the most unusual aspect of a body. As mentioned in the Rigid Body Properties section, its purpose is to describe the bodies mass distribution, which may not be available otherwise, since the actual shape is not always stored. Even if shapes are available, sometimes it is preferred to use mass settings not strictly compatible with the shapes in order to achieve some particular effect.

Thus the inertia tensor expresses how hard it is to rotate the shape in various directions. For example, one can intuitively imagine that a long thin cylinder (like a printing drum) is easy to turn along its length axis, but if it were suspended on a chain, it would be much more difficult to swing around (especially if you push it near the middle where the chain is attached, as opposed to the ends which give a great deal more leverage).

Formulas in high school physics books tell you how to compute the moment of inertia for a particular shape. Unfortunately, these entry-level books only give you a scalar value, which is the moment of inertia along some specified axis. In the SDK we need to use a matrix quantity, which describes the inertia along all possible axes. The inertia can be expressed either in the local space of the body or the global space, as can all vector or matrix quantities.

Internally, this matrix is decomposed into a rotation and a vector, which is both faster and more intuitive to use than the dense matrix because the rotation gives the center of mass an orientation relative to the actor.

When you create a body, you will want to specify a moment of inertia. The default is the identity, which is most likely inappropriate for your shape. The easiest option is to assign collision detection shapes to the actor, along with either a density or a total mass. The inertia tensor will be automatically computed from this data. You may, of course, set the tensor yourself (having computed it earlier with the previous method), which is more efficient. The SDK contains a wide selection of utility functions to compute and manipulate inertia tensors. See the files NxExportedUtils.h and NxInertiaTensor.h in the SDK API documentation. Also, for meshes, the methods NxConvexMesh::getMassInformation and NxTriangleMesh::getMassInformation are useful.

The NxActor methods,

```
void setMassSpaceInertiaTensor(const NxVec3 &m);
NxVec3getMassSpaceInertiaTensor();
```

let you access the constant diagonal inertia tensor of the actor. The rotation that is found in the diagonalization of the tensor is accessed using

```
void setCMassLocalOrientation (const NxMat33 &);
NxMat33 getCMassLocalOrientation ();
```

The methods,

```
void getGlobalInertiaTensor (NxMat33 &dest);
NxMat33 getGlobalInertiaTensorInverse();
```

rotate this tensor using the current actor orientation into world space; as a result, the tensor changes every time the actor moves.

API Reference

- NxActor
- NxActorDesc

- NxExportedUtils.h
- NxInertiaTensor.h

Manual Computation of Inertia Tensors

At times, it can be useful to manually specify a center of mass and inertia tensor in actors with an uneven mass distribution (e.g., a toy with a spherical base and a weight at the bottom to make it stand up). Specifying an unusually low center of mass can enhance the effect.

By default, the SDK computes a center of mass and inertia tensor when the actor is created. However, the user can override this by setting the actor's massLocalPose, massSpaceInertia and mass for the body with the following SDK functions:

```
NxReal  NxComputeSphereMass (NxReal radius, NxReal density);
NxReal  NxComputeSphereDensity (NxReal radius, NxReal mass);
NxReal  NxComputeBoxMass (const NxVec3& extents, NxReal density);
NxReal  NxComputeEllipsoidMass (const NxVec3& extents, NxReal density);
...

void NxComputeBoxInertiaTensor (NxVec3& diagInertia, NxReal mass, NxReal xlength, NxReal ylen
void NxComputeSphereInertiaTensor(NxVec3& diagInertia, NxReal mass, NxReal radius, bool hollo
```

The massLocalPose is simply a transformation applied to the diagonal inertia to move it away from its default pose (centered on the origin with no rotation).

Another useful function is the following:

```
bool NxDiagonalizeInertiaTensor(const NxMat33 & denseInertia, NxVec3 & diagonalInertia, NxMat
```

This allows you to convert a dense inertia tensor that has already been transformed into a rotation/vector form.

API Reference

- NxActor
- NxExportedUtils.h
- NxInertiaTensor.h

Applying Forces and Torques

The most physics friendly way of interacting with a body is to apply external force. In classical mechanics, most interactions between bodies are typically solved by using force because of the following law:

```
f = m a (force = mass * acceleration)
```

Force directly controls a body's acceleration, and indirectly controls its velocity and position. For this reason, control by force may be inconvenient if you need immediate response. Its advantages, however, are regardless of what type you apply to the bodies in the scene, the simulation will be able to keep all the defined constraints (joints and contacts) satisfied. The spring and damper effector and gravity also work by applying force to bodies.

Unfortunately, applying large forces to articulated bodies at the resonance frequency of the system may lead to ever-increasing velocities, and eventually to the failure of the constraint solver to maintain the joint constraints. This is not unlike a real world system where the joints would simply break.

The forces acting on a body are accumulated before each simulation frame, applied to the simulation, and then reset to zero in preparation for the next frame. You can apply your own forces and torques to a body in a variety of ways. The relevant methods of NxActor are listed below. Please refer to the API reference to see further detail:

```
void addForceAtPos(const NxVec3 & force, const NxVec3 & pos, NxForceMode);
void addForceAtLocalPos(const NxVec3 & force, const NxrVec3 & pos, NxForceMode);
void addLocalForceAtPos(const NxVec3 & force, const NxVec3 & pos, NxForceMode);
void addLocalForceAtLocalPos(const NxrVec3& force, const NxVec3& pos, NxForceMode);

void addForce(const NxVec3 &, NxForceMode);
void addLocalForce(const NxVec3 &, NxForceMode);
void addTorque(const NxVec3 &, NxForceMode);
void addLocalTorque(const NxVec3 &, NxForceMode);
```

The NxForceMode member defaults to NX_FORCE to apply simple forces. Use NX_IMPULSE to apply an impulsive force. NX_VELOCITY_CHANGE also applies impulsive force, but ignores the mass of the body, effectively leading to an instantaneous velocity change. See the API documentation of NxForceMode for more options.

Gravity

Gravity is such a common force in simulations that we have made it particularly simple to apply. If you need a scene-wide gravity effect, or any other uniform force field, you can apply the NxScene class' gravity vector using setGravity().

The parameter is the acceleration due to gravity vector. If you work with meters and seconds, this equals about a 9.8 magnitude on earth. It should point downwards in your scene. The force that will be applied at each body's center of mass is this acceleration vector multiplied by the actor's mass.

However, certain special effects can require that some dynamic actors not be influenced by gravity. To do this you can call NxActor::raiseBodyFlag(NX_BF_DISABLE_GRAVITY).

NOTE: Be careful changing gravity (or enabling/disabling) during the simulation. The change will not wake up sleeping actors for performance reasons. It may be necessary to iterate through all actors and call NxActor::wakeUp() manually.

API Reference

- NxActor

Setting the Velocity

Setting a body's velocity will immediately get it moving in a certain direction. Setting its momentum is another option, which may be more convenient if you don't know the mass of the body but want its speed to depend on it.

If you set the velocity of a body and then run its scene, after the simulation is complete, the velocity of the body may no longer be what you specified. This is because the simulation overrides your settings if they are in conflict with a specified constraint. For example, if a ball is resting on a table and you give it a downward velocity, it will change back to 0 when the constraint is resolved in the solver. If you set the velocity of a chain link that is joined to other chain links, the velocity of the chain link you set will decrease, and the other chain links' velocity will increase, in order for the chain to stay together.

For linear velocity, note that it is the velocity of the actor's center of mass you set. If this does not coincide with the origin of the actor's frame, then the velocity of the actor frame may be different, if the actor is also rotating.

The methods of NxActor that help to set the velocity of a body are summarized below:

```
void setLinearVelocity(const NxVec3 &);
void setAngularVelocity(const NxVec3 &);

void setLinearMomentum(const NxVec3 &);
void setAngularMomentum(const NxVec3 &);
```

Note: Setting a velocity large enough to carry an actor outside of the range of a float is a user error and might produce artifacts; this situation should be avoided.

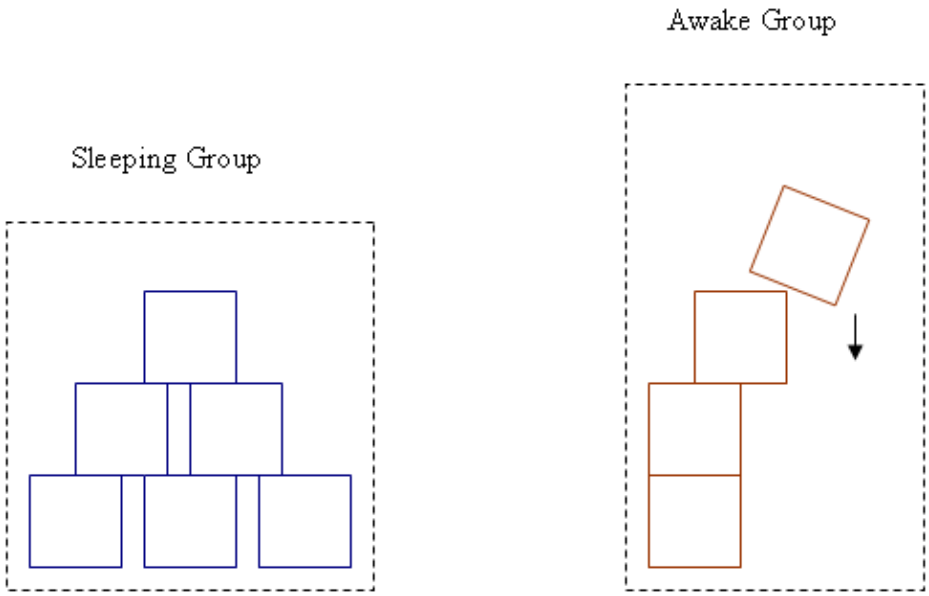
API Reference

- NxActor
- NxActorDesc

Sleeping

Overview

When a body does not move for a period of time, it is assumed that it will remain immobile until an external force throws it out of equilibrium. Until this force happens, it is no longer simulated in order to save time. This state is called sleeping.



In previous versions of PhysX, a body that was not moving for a certain time was considered sleeping and would not be integrated. When all bodies in an island had fallen asleep, they were pulled out of the simulation. In 2.5 and later, bodies will only be considered as being asleep when the whole island they're in wants to fall asleep. This means that bodies that have not moved for some time may still be awake, if they are in the same island as something that is moving. However, when the whole island falls asleep, it is taken out of the simulation totally.
Note that bodies that are put to sleep have their velocities set to zero, even if they were not exactly zero before.
The length of time over which the SDK judges an object to be sleeping is defined by the constant NX_SLEEP_INTERVAL. Each body has a sleep counter which is decremented each step based on if the body is determined to be awake(eg a force applied). When the counter reaches zero the body becomes a candidate to sleep.

The counter is specified by the body desc member:

```
NxBodyDesc::wakeUpCounter
```

Sleep Determination

There are a number of methods used to determine if a body can progress to a sleeping state, and hence a group, if all bodies within the group satisfy this condition. The methods are described below. The reason for a

number of sleep determination methods is that it is not always easy to determine if a body is genuinely in equilibrium. For example a long pendulum may have a low angular velocity but still display a lot of movement at the end furthest away from the pivot.

Simple Sleeping

Simple sleeping was the default in previous version of the SDK. It simply judges an object to be asleep if its linear and angular velocity are below a threshold for the interval specified by NX_SLEEP_INTERVAL.

This is controlled using:

```
NxBodyDesc::sleepLinearVelocity
NxBodyDesc::sleepAngularVelocity
```

and the methods:

```
NxActor::setSleepLinearVelocity(NxReal threshold)
NxActor::setSleepAngularVelocity(NxReal threshold)
```

Averaged Velocity

Keeping a running average of the velocity across time steps can lead to smoother sleep determination, which allows oscillating bodies to fall asleep more often. This mode is enabled using the body flag : NX_BF_FILTER_SLEEP_VEL.

Energy Based Sleeping

Using mass-normalized kinetic energy thresholds instead of velocity thresholds can also give better results. For two reasons: There is only one parameter to tweak and it automatically compensates for non-uniform inertia tensors. Long thin objects are very hard to tweak since the same angular velocity threshold is used around all three axes without compensating for the non-uniform tensor. The energy method also uses a scheme for checking the first derivative of the energy. It compares the current energy to the energy last frame, and the wake counter is reset whenever the energy increases. This means that bodies must strictly lose energy to be put to sleep. Close to sleep threshold, jitter tends to increase energy a little, so there's a small threshold for this mechanism to kick in (currently 5% of energy threshold).

To enable energy based sleeping a body must have the NX_BF_ENERGY_SLEEP_TEST raised (this is the default). The sleepEnergyThreshold parameter controls the point at which the kinetic energy is considered low enough to classify the body as sleeping.

```
NxBodyDesc::sleepEnergyThreshold
NxBodyDesc::flags (NX_BF_ENERGY_SLEEP_TEST)
```

Adaptive Damping

In addition to the above methods it is possible to specify a sleepDamping parameter for bodies. This allows the objects to go to sleep in a smooth manner, instead of coming to a complete stop. Sleep damping is applied over the interval NX_SLEEP_INTERVAL, as a body progresses towards a sleeping state the amount of damping is increased from zero up to sleepDamping.
NOTE: Sleep damping may cause unwanted artifacts if used in conjunction with articulated objects such as ragdolls. If you observe this phenomenon, consider applying damping only to the most important bodies.

```
NxBodyDesc::sleepDamping
```

Numerical considerations

Since bodies are awakened whenever contacts are lost, actors may have a problem falling asleep when they're in a position such that numerical imprecision in the collision detection causes contacts to appear and disappear without any movement. This is not commonly a result of normal simulation, but may occur if, for example, two boxes are created with faces that exactly meet. To avoid this effect, you can create the boxes with a slight overlap, up to the skin width, which will make the contacts persistent.

Sleep Control

Because an object automatically wakes up when a non sleeping object touches it or when a parameter is changed the mechanism should be mostly transparent to the user. However there are cases when it is desirable or required to exert direct control over object sleeping.

For example:

- When the user changes the contact groups in a way which causes the set of contacts to change (e.g., when calling NxActor::setGroup(), etc.).
- When changing global parameters such as gravity.
- When creating spring and damper effectors.
- When altering the collision filtering constants (e.g., NxScene::setFilterBool()).
- When the user wishes to throttle the performance of a scene by force actors to sleep.

You can query a body's sleep state with the following methods:

```
bool NxActor::isGroupSleeping();
bool NxActor::isSleeping();
```

But from version 2.5 these methods will return the same value, since bodies no longer fall asleep individually.

If you need to explicitly wake up a sleeping object, or force an object to sleep, use the following methods:

```
void NxActor::wakeUp();
void NxActor::putToSleep();
```

Island retrieval

If you need data on the sleeping island a certain actor is part of, you can get it using the getBoundForIslandSize() and getIslandArrayFromActor() methods of NxScene:

```
NxActor *actor = ...

NxU32 it = 0;
NxU32 nbIslandActors = gScene->getBoundForIslandSize(*actor);
NxActor** islandActors = (NxActor**)alloca(sizeof(NxActor*)*nbIslandActors);
nbIslandActors = gScene->getIslandArrayFromActor(*actor, islandActors, nbIslandActors, it);
```

You may also use the iterator parameter to extract the island one bit at a time, for example into a fixed-size buffer.

API Reference

- NxActor
- NxActorDesc
- NxBodyDesc

Sleep Events

Overview

The Sleep Events feature allows an application to request callbacks providing lists of NxActor pointers that point to NxActors in a given scene that have either just recently gone to sleep (deactivated) or woken up (activated).

Usage

In order to use this feature, you must first define a new class derived from the NxUserNotify class. It specifies the following pure-virtual functions for Sleep Events:

```
virtual void onSleep(NxActor** actors, NxU32 count);
virtual void onWake(NxActor** actors, NxU32 count);
```

Here is an example implementation:

```
class MyCallback : public NxUserNotify
{
public:

    virtual void onSleep(NxActor** actors, NxU32 count)
    {
        while(count--)
        {
            NxActor *thisActor = *actors;
            // do whatever you need to do
            // with actors that have gone to sleep
            actors++;
        }
    }

    virtual void onWake(NxActor** actors, NxU32 count)
    {
        while(count--)
        {
            NxActor *thisActor = *actors;
            // do whatever you need to do
            // with actors that have woken up
            actors++;
        }
    }
};
```

Next, you must create your simulation scene using this new class:

```
static NxPhysicsSDK* gPhysicsSDK = NULL;
static NxScene* gScene = NULL;
static MyCallback gSleepCallback;

void InitPhysX(void)
```

```
    {
        gPhysicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION, NULL, new ErrorStream());

        // create a scene
        NxSceneDesc sceneDesc;
        sceneDesc.userNotify = &gSleepCallback;
        gScene = gPhysicsSDK->createScene(sceneDesc);
    }
```

Now, whenever you call gScene->fetchResults(), you will receive callbacks via your MyCallback class whenever the scene reports NxActors that have either just gone to sleep, or have just woken up.

API Reference

- NxActor
- NxActorDesc
- NxUserNotify

Active Transform Notification

Overview

The Active Transform Notification feature allows an application to query the NxScene for a list of the NxActors that were active during the previous simulation step (and thus may have moved). This can be convenient for scenes with large numbers of dynamic objects, since most of the time the vast majority of dynamic objects are inactive. By providing a list containing only the active NxActors, the application does not have to iterate through all of the objects in the scene on the off chance that their transforms have changed.

Static actors that are moved do *not* generate active transform notifications.

Usage

First, you need to activate Active Transform Notification in your scene:

```
sceneDesc.flags |= NX_SF_ENABLE_ACTIVETRANSFORMS;
```

The active transforms are exposed to the application through the following API function in NxScene:

```
NxActiveTransform *NxScene::getActiveTransforms(NxU32 &nbActiveTransforms);


struct NxActiveTransform
{
    NxActor* actor;
    void* userData;
    NxMat34 actor2World;
};
```

Calling this function returns a pointer to the list of NxActiveTransforms generated during the last call to fetchResults(). The parameter nbActiveTransforms returns the number of elements in the list. You can call NxScene::getActiveTransforms() at any time. The list and number of elements you are returned will remain valid until the next call to fetchResults. It is preferable that the application not actually dereference the actor pointer (for memory coherency purposes), but refer to the cached userData and actor2World variables.

Example

```
// Get the Active Transforms from the scene
NxU32 nbTransforms = 0;
NxActiveTransform *activeTransforms = gScene->getActiveTransforms(nbTransforms);

// Start simulation (non blocking)
gScene->simulate(1.0f/60.0f);

// update the game objects for only the actors with active transforms
if(nbTransforms && activeTransforms)
{
    for(NxU32 i = 0; i < nbTransforms; ++i)
    {
        // the user data of the actor holds the game object pointer
        GameObject *gameObject = (GameObject *)activeTransforms[i].userData;
```

```
            // update the game object's transform to match the NxActor
            if(gameObject)
                gameObject->setTransform(activeTransforms[i].actor2World);
        }
    }

    // ...

    // Fetch simulation results
    gScene->flushStream();
    gScene->fetchResults(NX_RIGID_BODY_FINISHED, true);
```

API Refrence

- NxScene
- NxActiveTransform

Static Actors

Static actors do not have any of the dynamic properties discussed in the Rigid Body Properties section. Create a static actor by setting the body member to NULL in the actor descriptor.

Once the static actor is created, do not do anything with it. Even though operations such as changing the position, adding more shapes, or even deleting the static actor are not explicitly forbidden, they are not recommended for two reasons: 1) the SDK assumes that static actors are indeed static, and does all sorts of optimizations that rely on this property. Changing the static actor may force time consuming re-computation of such data structures, and 2) the code driving dynamic actors and joints is written with the assumption that static actors will not move or be deleted for the duration of the simulation, which permits a number of optimizations for this very common scenario. For example, moving a static actor out from under a stack of sleeping boxes will not wake these up, thus they will levitate in midair. Even if they are woken up, the collision response between moving static and dynamic actors is of low quality.

To achieve the behaviors of movable static actors, use 'kinematic' actors. Read about them in the Kinematic Actors section.

Kinematic Actors

A kinematic actor is a special kind of dynamic actor that is not moved directly by the user nor by the SDK. This means that it does not move in response to user forces, gravity, collision impulses, or if tugged by joints. Further, this type of actor lacks real velocity or momentum.

Instead, it becomes mobile when the user sets a slightly different position with the moveGlobal*() functions. NOTE: Do not use the setGlobal*() functions to make the actor mobile; it will disable many of the advantages that kinematic objects provide, such as having the appearance of infinite mass and the ability to push regular dynamic actors out of the way.

The move command will result in a velocity that, when successfully carried out inside simulate() (i.e., the motion is not blocked due to joints or collisions), moves the body into the desired pose. After the move is carried out during a single time step, the velocity is returned to zero. Thus, the move command must be called in every time step so kinematic actors can continue along a path. This is because the move functions simply store the move destination until simulate() is called, so consecutive calls will overwrite the stored target variable.

To create a kinematic actor, first create a dynamic actor, and then set its kinematic flag either in the body descriptor or with NxActor::raiseBodyFlag(NX_BF_KINEMATIC). When you want the kinematic actor to become a normal dynamic body again, turn off the kinematic flag with clearBodyFlag(NX_BF_KINEMATIC). While you do need to provide a mass for the kinematic body as for all dynamic bodies, this mass will not actually be used for anything while the actor is in kinematic mode.

Kinematic actors are great for moving platforms or characters where direct motion control is desired.

Caveats

- If a dynamic actor is squished between a kinematic and a static or between two kinematics, then it will have no choice but to get pressed into one of them. In future versions, it may be possible to have the kinematic motion blocked in this case.
- Kinematic actors will collide with true dynamic actors but not with static actors. This may also change in the future.
- When calling NxScene::simulate() with a value for elapsedTime which is not multiple of the sub step size, a different number of sub steps may be executed due to accumulated fractional sub steps. If there is a remainder, it will be accumulated until a whole sub step can be executed. The result is that kinematic actors may not move a constant distance each time step, since we have the requirement that the velocity is constant during a sub step.

API Reference

- NxActor
- NxActorDesc
- NxScene

Adaptive Force

The adaptive force feature provides a simple method to dramatically improve the performance of stacking. The SDK scales the amount of gravitational force applied to objects based on the number of constraints attached to the object. To reduce the artifacts from this approach this is now only applied when the objects are linked to a static object(in most cases the ground).

To control the amount of scaling applied, use the SDK parameter: NX_ADAPTIVE_FORCE.

API Reference

- NxScene

Joints

Without joints or shapes, actors would be able to do little else but float through space. Joints provide a consistent way to connect two actors. They require that the two actors always move relative to each other. The specific way in which they limit movement is determined by the type of joint.

Joints and contacts (see Collision Detection) are both called constraints because they constrain the motion of a body. The SDK only supports pairs of constraints. In other words, each joint or contact is between exactly two actors. One of these actors may be the implicit world actor, which represents the immovable global reference frame. To specify a joint between an actor and the world, pass NULL as one of the two actor pointers when you create the joint. The actor which is not NULL must be dynamic, as connecting two static actors, implicitly immobile, would be pointless.

Joints are created by calling the NxScene::createJoint() method. This method uses a certain type of joint descriptor as a parameter that contains all the information needed to create the joint. NOTE: When working with the PPU there is a limitation on 65536 joints, of which only 4096 may be active at any given time.

The simulation cost of a joint is determined by the degrees of freedom (DOFs) it removes from the system. A degree of freedom is a property of an object that can change, here referring to a component of the body's position or orientation. For example, a body has 3 linear degrees of freedom along the X, Y and Z axis, plus 3 angular degrees of freedom for its rotation around the X, Y and Z axis. (See Other Resources for more suggested reading on this topic.)

The sections within the 'Joint' folders describe the different joint types and give the number of degrees of freedom they remove.

Joint Flags

In addition to the joint specific flags, the following flags can be applied to all joints through the jointFlags member of NxJointDesc:

- NX_JF_COLLISION_ENABLED - Enable collision detection between the two bodies attached to the joint. By default the SDK does not generate contact points for pairs of bodies connected by a joint.
- NX_JF_VISUALIZATION - Enable the debug renderer for this joint.

API Reference

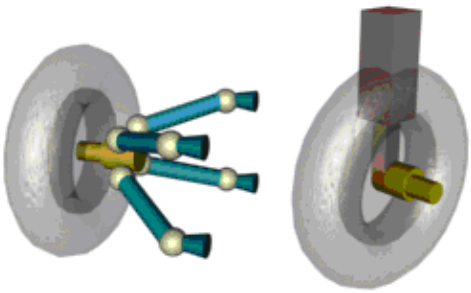
- NxJoint
- NxJointDesc
- NxSphericalJoint
- NxRevoluteJoint
- NxPrismaticJoint
- NxCylindricalJoint
- NxFixedJoint
- NxDistanceJoint
- NxPointInPlaneJoint
- NxPointOnLineJoint
- NxPulleyJoint
- NxD6Joint

Modeling with Joints

When considering an appropriate joint from which to model a real world machine, making an exact copy may be possible, but is almost never the optimal choice. This is because engineers who design real world objects have constraints you do not have within a simulation. For example, shock absorber elements in the real world cannot go through the wheel which they support - for this reason the engineer has to use additional joints to lead the suspension mechanism around the wheel. In a virtual simulation, this is not a consideration.

Also in the real world, an asymmetric construction may be impractical because it might lead to added friction or less stability. In the simulation, you don't have friction unless you purposely create it. Similarly, a real world car that has twice as many shock absorbers as another will not necessarily be slower, but your simulation of it will.

Consider the picture below of a real world car suspension assembly versus a virtual approximation. They are quite different. The real thing looks much more "realistic". For this reason, it may be useful to create the real assembly as a graphical prop, displayed and animated in place of the simplified dynamics scheme that you use for the actual simulation.

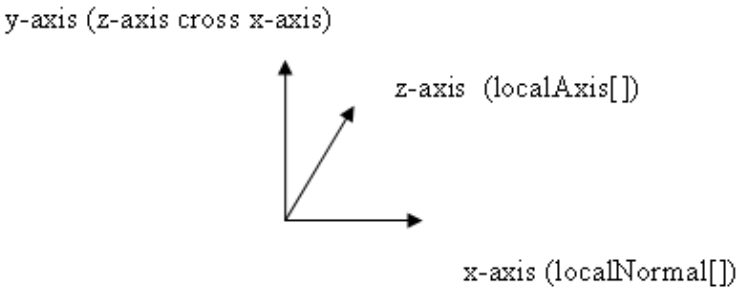


The figure shows a real mechanical component (single wheel suspension of a car) made up of a large number of joints, and an alternative simpler scheme with about the same degrees of freedom.

Joint Frames

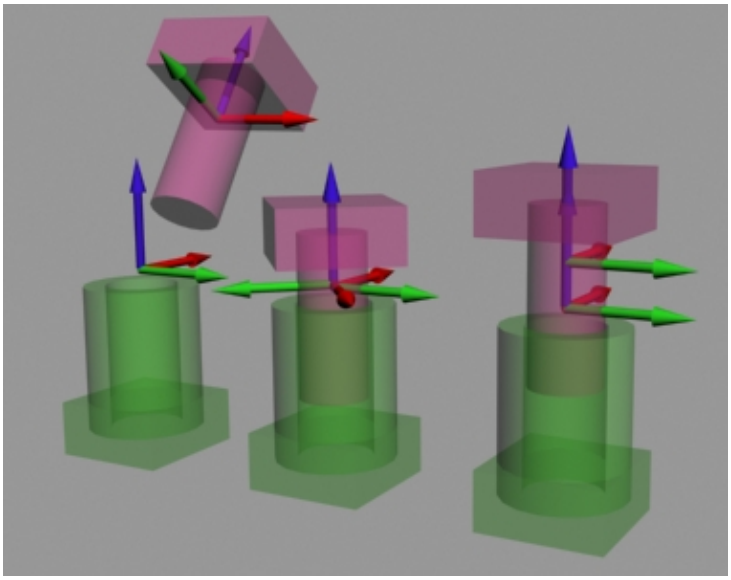
In the SDK, all joint types share some common properties which are collected in the NxJoint class and the corresponding descriptor NxJointDesc. First, they all have two actor pointers. Each of the actors, including the implicit world actor represented by a NULL actor pointer, have a joint frame that is specified relative to its actor frame. The two frames are stored in NxJointDesc as two basis vectors and a point, shown below:

```
NxVec3 localNormal[2];
NxVec3 localAxis[2];
NxVec3 localAnchor[2];
```



The first (X) basis vector of joint space is called the 'normal', the second (Z) is called the 'axis', and the third (Y) is called the 'binormal', computed as a cross product of the first two. The 'anchor' is the point where the frame is located. The word 'local' denotes that the frames are not in the world but rather in the actor space.

These two frames uniquely identify the geometric placement of the joint within the simulation. They never change because they are fixed relative to the respective actors; however, the frames can end up in different places. The joint moves when the connected actors move, therefore these frames, when transformed to the world space via the actor's transform, also move. The image below shows joint frames for a cylindrical joint. In the far left configuration, the blue z axis of the frames does not match up, meaning that this configuration suffers from significant joint error. The other two configurations are permissible.

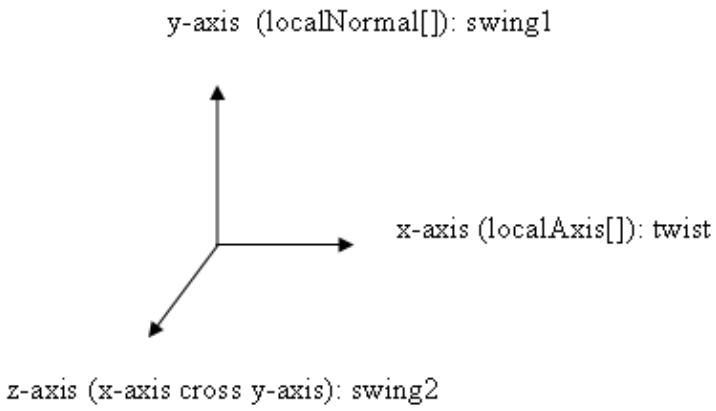


Use the NxJoint or NxJointDesc methods setGlobalAxis() / setGlobalAnchor() to specify the joint frames in world space. They will be transformed internally to the local spaces of the actors, which are needed because the simulation does not work without some error. For example, the ball joint may be pulled apart by a large force so that the two local frames no longer match up in world space. It is important to let the user have access to this error, like all internal states, so that it is possible to save and restore at a later time with the simulation proceeding exactly where it left off. (See Joint Projection for a method to correct severe joint errors.)

Not all joint types need an entire frame to specify their geometry. For example, a simple spherical joint only needs the attachment point to be specified relative to each actor. However, the additional features of the spherical joint, such as the limits and spring behavior, need the additional axes. Typically the axis vector is the primary axis of the joint, while the other two vectors are orthogonal to this. (See the API documentation on different joint types to see how the joints interpret the basis vectors.)

NOTE: The axis convention for joint frames follows a left-handed rule, differing from the world frame convention which uses a right-handed rule.

The 6 DOF joint is an exception, using a right-handed convention as shown below:



Joint Limits

In the real world, most joints can only move in a certain range, usually because the hinges' own construction gets in the way. For example, a door hinge can only move in one direction until it straightens out at the 180 degree position, and can perhaps reach the 20 degree mark in the other direction. A more complex anatomic joint, like a person's shoulder, which as previously mentioned is a spherical joint, has a much more complicated range of motion.

The SDK provides two different kinds of limit functionality. The limit point/limit planes functionality of the NxJoint class works generically with all the different joint types. The disadvantage of generic functionality is that it is less intuitive to use than the other joint specific approaches, which are provided only for Spherical, 6 DOF and Revolute joints. In the future, custom joint limits may be added to other joint types, but for now, the 6 DOF joint provides a superset of most other joint type functionality.

The custom method for creating joint limits consists of specifying a minimum and maximum rotational angle around a certain rotation axis. For revolute joints this is the joint axis. For spherical joints there are two different limits - a twist limit and a swing limit.

The generic method for creating joint limits consists of electing a point attached to one of the two bodies to act as the limit point, and specifying several planes attached to the other actor. The points and planes move together with the actor they are attached to. The simulation then makes certain that the pair of actors only move relative to each other so that the 'limit point' stays on the positive side of all limit planes. The positive side of a plane is the one in which its 'normal' points. To better understand how this model works, examine the pictures below:

Hinge joint with limits, and the resulting two extreme positions. The blue sphere is the 'limit point', and moves with the silver actor. The planes move with the gold actor.	Slider joint with four limit planes, and two possible extreme positions. The planes are specified to move with the silver actor.	The point on line joint is limited by two planes which move with the silver actor.

See the API reference documentation for the following methods to limit a joint:

```
void NxJoint::setLimitPoint(const NxVec3 & point, bool pointIsOnBody2 = true);
bool NxJoint::addLimitPlane(const NxVec3 & normal, const NxVec3 & pointInPlane);
```

Example

```
NxPrismaticJointDesc prismaticDesc;

prismaticDesc.actor[0] = actor0;
prismaticDesc.actor[1] = actor1;

prismaticDesc.setGlobalAnchor(globalAnchor);
prismaticDesc.setGlobalAxis(globalAxis);

NxJoint* joint = gScene->createJoint(prismaticDesc);

//Limit the prismatic joint so that it can only move from its initial position 1.5 units in e

joint->setLimitPoint(globalAnchor);
joint->addLimitPlane(-globalAxis, globalAnchor + 1.5 * globalAxis);
joint->addLimitPlane(globalAxis, globalAnchor - 1.5 * globalAxis);
```

Breakable Joints

Joints can be set to be breakable. This means that if a joint is tugged on with a large enough force, it will snap apart. To set a breakable joint, set the maxForce and maxTorque members of NxJointDesc to the desired values - the smaller the value, the more easily the joint will break. The maxForce defines the limit for linear forces that can be applied to a joint before it breaks, while the maxTorque defines the limit for angular forces. The exact behavior depends on the type of joint. It is possible to change these parameters for an existing joint using NxJoint::setBreakable().

When a joint breaks, the onJointBreak() method of the NxUserNotify class will be called, assuming the user has supplied the SDK with an instance of this object using the NxScene::setUserNotify() method. Further, the joint is put into a broken state, which basically deactivates it. At this point, it may be deleted by returning 'true' in onJointBreak(). The user may not delete the joint simply by calling releaseJoint() from onJointBreak() because the SDK forbids such reentrant calls that change the simulation state. No other action is permitted on broken joints.

NOTE: Even non-breakable joints are put in a broken state when the user releases one of the two bodies that the joint was referencing. However, in this case no onJointBreak() event is issued.

maxForce and maxTorque

The distinction between maxForce and maxTorque is dependent on how the joint is implemented internally, which may not be obvious. For example, what appears to be an angular degree of freedom may be constrained indirectly by a linear constraint.

Therefore, in most practical applications the user should set both maxTorque and maxForce to similar values.

In the current implementation, the following joints use angular constraints:

- 6 DOF Joint - Only in the case where angular DOFs are locked.
- Fixed Joint
- Prismatic Joint
- Revolute Joint

NOTE: The breakingImpulse parameters passed to the onJointBreak() callback is now clamped to the values of maxForce (or maxTorque, depending on what made the joint break) that was specified for the joint. This differs from the behavior in versions prior to 2.5, where the actual force/torque was provided.

Example

```
NxRevoluteJointDesc revDesc;

revDesc.actor[0] = actor0;
revDesc.actor[1] = actor1;

revDesc.setGlobalAnchor(globalAnchor);
revDesc.setGlobalAxis(globalAxis);

revDesc.jointFlags |= NX_JF_COLLISION_ENABLED;//enable collision between the bodies attached

gMyBreakableJoint = gScene->createJoint(revDesc);
```

```
NxReal maxForce = 2000;
NxReal maxTorque = 100;
joint->setBreakable(maxForce,maxTorque);

//When the joint breaks, the application is notified through NxUserNotify
//Remember to register the callback class with gScene->setUserNotify();

class MyUserNotify : public NxUserNotify
{
    public:

    virtual bool onJointBreak(NxReal breakingImpulse, NxJoint & brokenJoint)
    {
        if((&brokenJoint) == gMyBreakableJoint)
        {
            cout << "BANG, The joint broke" << endl;
            return true; //delete the joint
        }

        return false; //don't delete the joint
    }
}
```

Threading

The NxUserNotify class is only called in the context of the user thread. It is not necessary to make it thread safe.

Joint Motors, Springs, and Special Limits

Joint motors are a very robust and stable way to simulate motorized joints, such as the wheels of a car or the joints of a robot. You can either control the maximum force output of the motor to reach a certain speed, or set this force to a large number and control the speed that is to be maintained. The monster truck demo uses revolute joint motors to drive all four wheels.

The following are joints with motor support:

- 6 DOF Joint
- Revolute Joint
- Pulley Joint

Joint springs are similar to joint motors. You can set up a spring and a damper constant that resists the motion applied to the joint's degrees of freedom.

The following are joints with spring support:

- 6 DOF Joint
- Spherical Joint
- Revolute Joint
- Distance Joint

Finally, custom angular limits may be defined for the joints below, which are much easier to use than the generic limits described in previous sections:

- 6 DOF Joint
- Spherical Joint
- Revolute Joint

NOTE: The 6 DOF joint is recommended for situations which require sophisticated motor and joint limit support.

Joint Projection

Joint projection is a method for correcting large joint errors. Joint errors occur when a joint's constraint is violated - imagine the ball being pulled out of the socket in a spherical joint. Under normal circumstances, joint errors are small yet unavoidable due to the imprecise nature of floating point math and numerical integrators. The SDK applies minimal correcting forces to the simulation to try and reduce joint errors over time.

However, if a joint error becomes very large, more drastic measures are necessary, such as joint projection. The SDK can, in some situations, project (or change) the position of objects directly to fix the joint error. But when projecting the joint it is not desirable to completely remove the joint error because the correcting forces, which also affect velocity, would become zero.

To enable joint projection, set the projectionMode member of the appropriate joint descriptor to either of the following:

- NX_JPM_POINT_MINDIST: Performs both linear and angular projection.
- NX_JPM_LINEAR_MINDIST: Only projects linear distances, for improved performance.

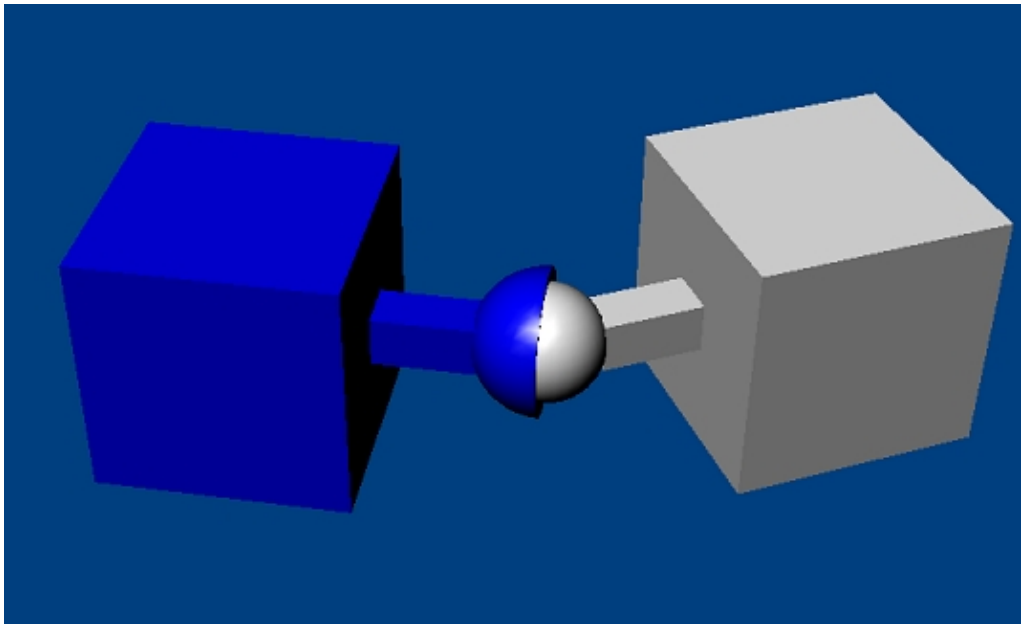
It is also necessary to set projectionDistance to a small value greater than zero. When the joint error is larger than projectionDistance the SDK will change it so that the joint error is equal to projectionDistance. Setting projectionDistance too small will introduce unwanted oscillations into the simulation.

The following joints support projection:

- 6 DOF Joints
- Revolute Joints
- Spherical Joints

Note: Joint projection is a somewhat "blunt instrument". It simply moves one actor, disregarding any collisions that might otherwise intervene. This may lead to penetration of collision geometries; this is particularily inconvenient when CCD is enabled, as this may prevent the geometries to de-penetrate afterward. Also, it is not possible to specify which actor is the one to move in this way.
Because of this, it is recommended that joint projection be used judiciously, and avoided unless there are real problems in tems of joint violation.

Spherical Joint



A spherical joint is the simplest kind of joint. It constrains two points on two different bodies from coinciding. This point, located in world space, is the only parameter that has to be specified (the other parameters are optional). Specifying the anchor point (point that is forced to coincide) in world space guarantees that the point in the local space of each body will coincide when the point is transformed back from local into world space.

By attaching a series of spherical joints and bodies, chains/ropes can be created. Another example for a common spherical joint is a person's shoulder, which is quite limited in its range of motion (see the Spherical Joint Limits section below).

DOFs removed: 3
DOFs remaining: 3

Joint Parameters

Parameter	Description
actor[0]	First actor
actor[1]	Second actor
localAnchor[0]	Attachment point on the first actor.
localAnchor[1]	Attachment point on the second actor (the attachment points should coincide).
Limits:	
localAxis[0]	The joint axes are used to enforce joint limits.
localAxis[1]	
swingAxis	Defines the center of the swing limit cone, attached to actor 0.
swingLimit	Defines the angle which the joint can rotate away from the swing axis.
twistLimit	Defines the high and low limits for twist around the joint axis.

Springs:	
twistSpring	Spring which attempts to return the twist to the target value.
swingSpring	Spring which attempts to return the joint axis to the swingAxis.
jointSpring	Specifies how much the joint can be pulled apart if joint spring is enabled.
flags:	
NX_SJF_TWIST_LIMIT_ENABLED	true if the twist limit is enabled
NX_SJF_SWING_LIMIT_ENABLED	true if the swing limit is enabled
NX_SJF_TWIST_SPRING_ENABLED	true if the twist spring is enabled
NX_SJF_SWING_SPRING_ENABLED	true if the swing spring is enabled
NX_SJF_JOINT_SPRING_ENABLED	true if the joint spring is enabled

NOTE: When setting localAnchor[] it is generally convenient to use setGlobalAnchor() to set the anchor with a world space point.

NOTE: When setting localAxis[] it is generally convenient to use setGlobalAxis() to set the axis with a world space axis.

NOTE: When specifying the joint axis it is also important to specify the localNormal[] which should be orthogonal to the localAxis[]. The localNormal[] is needed to specify joint limits.

Spherical Joint Limits

Spherical joints allow limits that can approximate the physical limits of motion on a human arm. It is possible to specify a cone which limits how far the arm can swing from a given axis. In addition, a twist limit can be specified which controls how much the arm is allowed to twist around its own axis.

NOTE: There are similar restrictions on the twist limits for spherical joints as there are for revolute joints.

Example:

```
NxSphericalJointDesc sphericalDesc;

sphericalDesc.actor[0] = actor0;
sphericalDesc.actor[1] = actor1;

//Point which both of the bodies are constrained to share (relative to each other).

sphericalDesc.setGlobalAnchor(globalAnchor);

NxSphericalJoint* sphereJoint=gScene->createJoint(sphericalDesc);
```

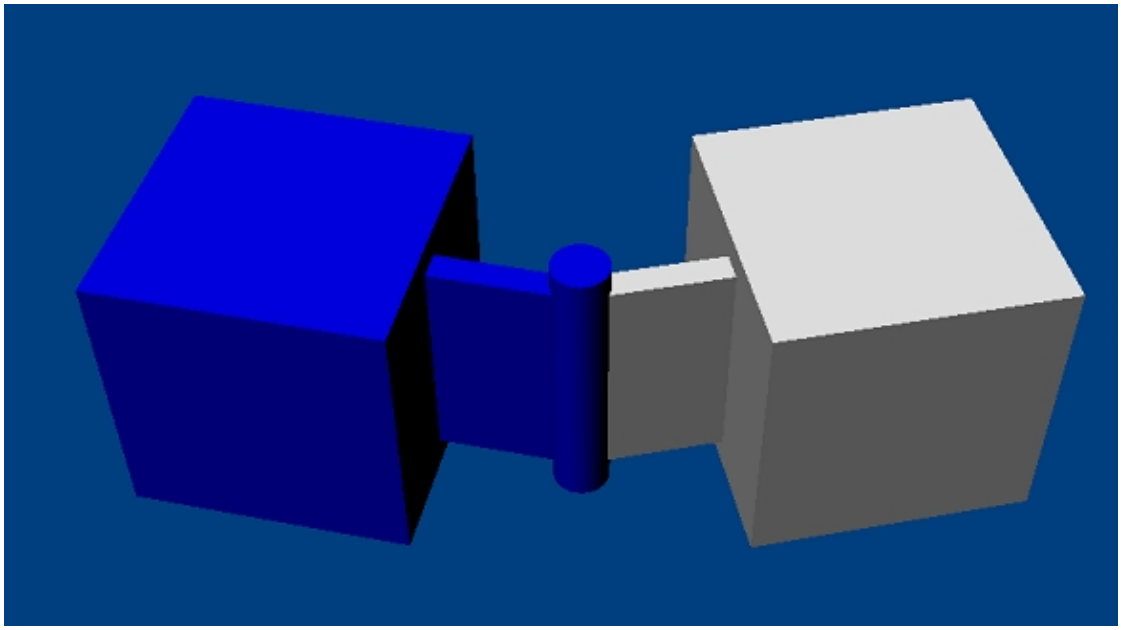
Samples

Sample Joints

API Reference

- NxJoint
- NxJointDesc
- NxSphericalJoint
- NxSphericalJointDesc

Revolute Joint



A revolute joint removes all but a single rotational degree of freedom from two objects. The axis along which the two bodies may rotate is specified with a point and a direction vector. In theory, the point along the direction vector does not matter, but in practice, it should be near the area where the bodies are closest to improve simulation stability.

An example for a revolute joint is a door hinge. Another example would be using a revolute joint to attach rotating fan blades to a ceiling. The revolute joint could be motorized, causing the fan to rotate.

DOFs removed: 5
DOFs remaining: 1

Joint Parameters

Parameter	Description
actor[0]	First actor
actor[1]	Second actor
localAnchor[0]	Point on the axis in the first actor's frame.
localAnchor[1]	Point on the axis in the second actor's frame (the points should coincide in the global frame).
localAxis[0]	Revolute axis in the first actor's frame.
localAxis[1]	Revolute axis in the second actor's frame (should match localAxis[0] in the global frame).
Limits:	
limit	Defines a lower and upper limit on the angle of rotation for the joint.
Springs:	
springs	Defines an angular spring which will try and rotate the joint to the springs target value (it is not applied if the joint has a motor).
Motor:	

motor	Defines an angular motor to apply to the joint.
flags:	
NX_RJF_LIMIT_ENABLED	true if the limits are enabled
NX_RJF_MOTOR_ENABLED	true if the motor is enabled
NX_RJF_SPRING_ENABLED	true if the spring is enabled (the spring will only take effect if the motor is disabled)

NOTE: When setting localAnchor[] it is generally convenient to use setGlobalAnchor() to set the anchor with a world space point.

NOTE: When setting localAxis[] it is generally convenient to use setGlobalAxis() to set the axis with a world space axis.

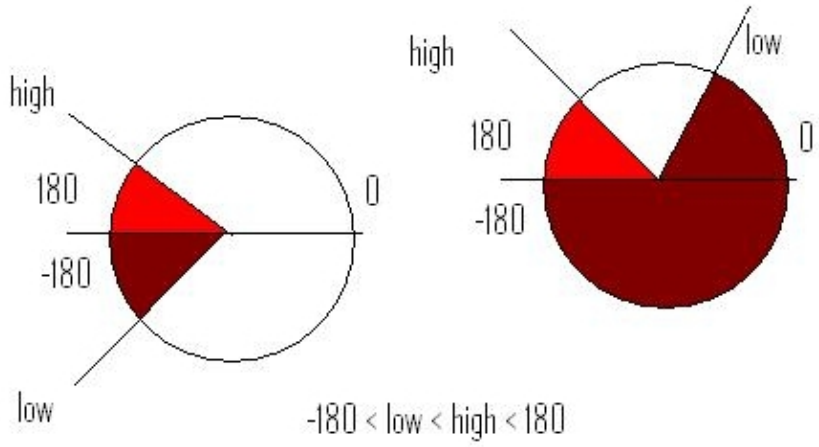
NOTE: When specifying the joint axis it is also important to specify the localNormal[] which should be orthogonal to the localAxis[]. The localNormal[] is needed to specify joint limits and to calculate the joint's angle of rotation.

Revolute Joint Limits

A revolute joint allows limits to be placed on how far it rotates around the joint axis. For example, a hinge on a door cannot rotate through 360 degrees; rather, it can rotate between 20 degrees and 180 degrees.

The angle of rotation is measured using the joints normal (axis orthogonal to the joints axis). This is the angle reported by NxRevoluteJoint::getAngle(). The limits are specified as a high and low limit, which must satisfy the condition -Pi < low < high <Pi degrees.

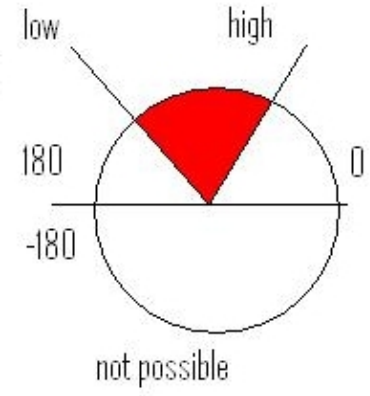
Below are valid revolute joint limits in which the joint is able to move between low and high:



NOTE: The white region represents the allowable rotation for the joint.

Limitations of Revolute Joint Limits

As shown below, it is not possible to specify certain limit configurations without rotating the joint axes, due to the restrictions on the values of low and high:



To achieve this configuration, it is necessary to rotate the joint counter-clockwise so that low is below the 180 degree line.

NOTE: If the angular region that is prohibited by the twist limit (as in the above figures) is very small, only a few degrees or so, then the joint may "push through" the limit and out on the other side if the relative angular velocity is large enough in relation to the time step. Care must be taken to make sure the limit is "thick" enough for the typical angular velocities it will be subjected to.

Example

```
NxRevoluteJointDesc revDesc;

revDesc.actor[0] = actor0;
revDesc.actor[1] = actor1;

revDesc.setGlobalAxis(globalAxis); //The direction of the axis the bodies revolve around.
revDesc.setGlobalAnchor(globalAnchor); //Reference point that the axis passes through.

NxRevoluteJoint *revJoint=(NxRevoluteJoint *)gScene->createJoint(revDesc);
```

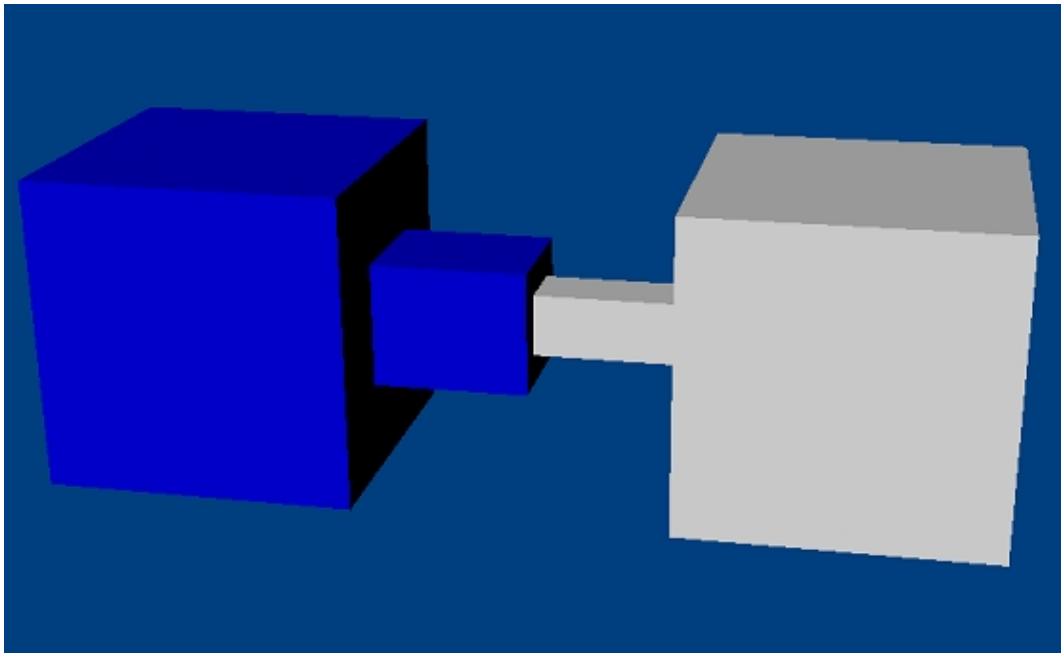
Samples

Sample Joints

API Reference

- NxJoint
- NxJointDesc
- NxRevoluteJoint
- NxRevoluteJointDesc

Prismatic Joint



A prismatic joint permits relative translational movement between two bodies along an axis, but no relative rotational movement at all. It is usually necessary to add joint limits to prevent the bodies from getting too far from each other along the joint axis. If the distance becomes too great, then the SDK can have difficulty maintaining the joint constraints (see Joint Limits). NOTE: A prismatic joint is similar to a cylindrical joint except that it prevents rotation around the joint axis.

An example for a prismatic joint is a pair of motorcycle shock absorbers.

DOFs removed: 5
DOFs remaining: 1

Joint Parameters

Parameter	Description
actor[0]	First actor
actor[1]	Second actor
localAnchor[0]	Point attached to slide axis on the first actor.
localAnchor[1]	Point attached to slide axis on the second actor (the attachment points should coincide).
localAxis[0]	The axis along which to slide in the first actor's frame.
localAxis[1]	The axis along which to slide in the second actor's frame (should match localAxis[0] in the global frame).

NOTE: When setting localAnchor[] it is generally convenient to use setGlobalAnchor() to set the anchor with a world space point.

NOTE: When setting localAxis[] it is generally convenient to use setGlobalAxis() to set the axis with a world space axis.

NOTE: When specifying the joint axis it is also important to specify the localNormal[] which should be

orthogonal to the localAxis[]. The localNormal[] is needed to specify joint limits.

Example

```
NxPrismaticJointDesc prismaticDesc;

prismaticDesc.actor[0] = actor0;
prismaticDesc.actor[1] = actor1;

prismaticDesc.setGlobalAnchor(globalAnchor); //The point constrained to the slide axis.
prismaticDesc.setGlobalAxis(globalAxis); //The axis along which to slide.

NxPrismaticJoint* prismJoint = (NxPrismaticJoint *)gScene->createJoint(prismaticDesc);

prismJoint->setLimitPoint(globalAnchor); //Constrain sliding a specific range.

prismJoint->addLimitPlane(-globalAxis, globalAnchor + 1.5*globalAxis);
prismJoint->addLimitPlane(globalAxis, globalAnchor - 1.5*globalAxis);
```

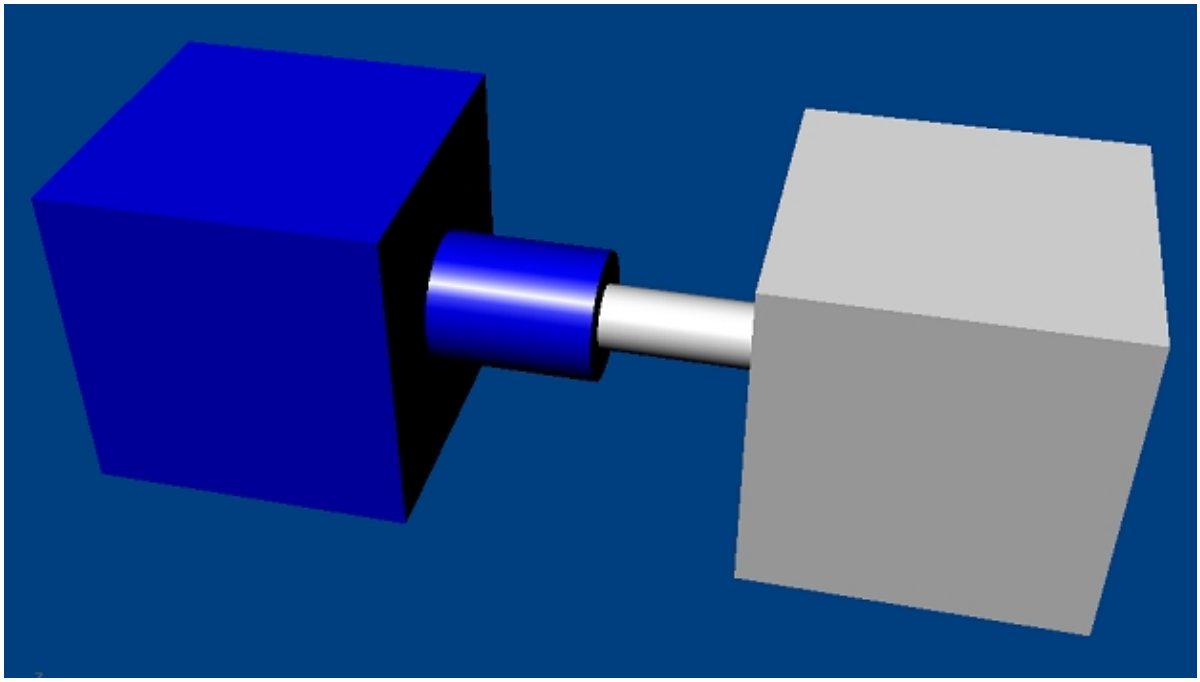
Samples

Sample Joints

API Reference

- NxJoint
- NxJointDesc
- NxPrismaticJoint
- NxPrismaticJointDesc

Cylindrical Joint



A cylindrical joint permits both relative translational and rotational movement between two bodies along a single axis (i.e., the bodies are allowed to both slide and twist along the axis of the joint). It is usually necessary to add joint limits to prevent the bodies from getting too far from each other along the joint axis. If the distance becomes too great, then the SDK can have difficulty maintaining the joint constraints (see Joint Limits).

An example for a cylindrical joint is a telescopic radio antenna. Another example is a piston on a machine such as a steam engine.

DOFs removed: 4
DOFs remaining: 2

Joint Parameters

Parameter	Description
actor[0]	First actor
actor[1]	Second actor
localAnchor[0]	Point on the slide axis in the first actor's frame.
localAnchor[1]	Point on the slide axis in the second actor's frame (the anchor points should coincide).
localAxis[0]	The slide axis in the first actor's frame.
localAxis[1]	The slide axis in the second actor's frame (should coincide with localAxis[0] in the global frame).

NOTE: When setting localAnchor[] it is generally convenient to use setGlobalAnchor() to set the anchor with a world space point.

NOTE: When setting localAxis[] it is generally convenient to use setGlobalAxis() to set the axis with a world space axis.

NOTE: When specifying the joint axis it is also important to specify the localNormal[] which should be orthogonal to the localAxis[]. The localNormal[] is needed to specify joint limits.

Example

```
NxCylindricalJointDesc cylDesc;

cylDesc.actor[0] = actor0;
cylDesc.actor[1] = actor1;
cylDesc.setGlobalAnchor(globalAnchor);
cylDesc.setGlobalAxis(globalAxis);

NxJoint* joint = gScene->createJoint(cylDesc);

joint->setLimitPoint(globalAnchor);

//Add limiting planes (to restrict how far the bodies can slide relative to each other).

joint->addLimitPlane(-globalAxis, globalAnchor + 1 * globalAxis);
joint->addLimitPlane(globalAxis, globalAnchor - 1 * globalAxis);
```

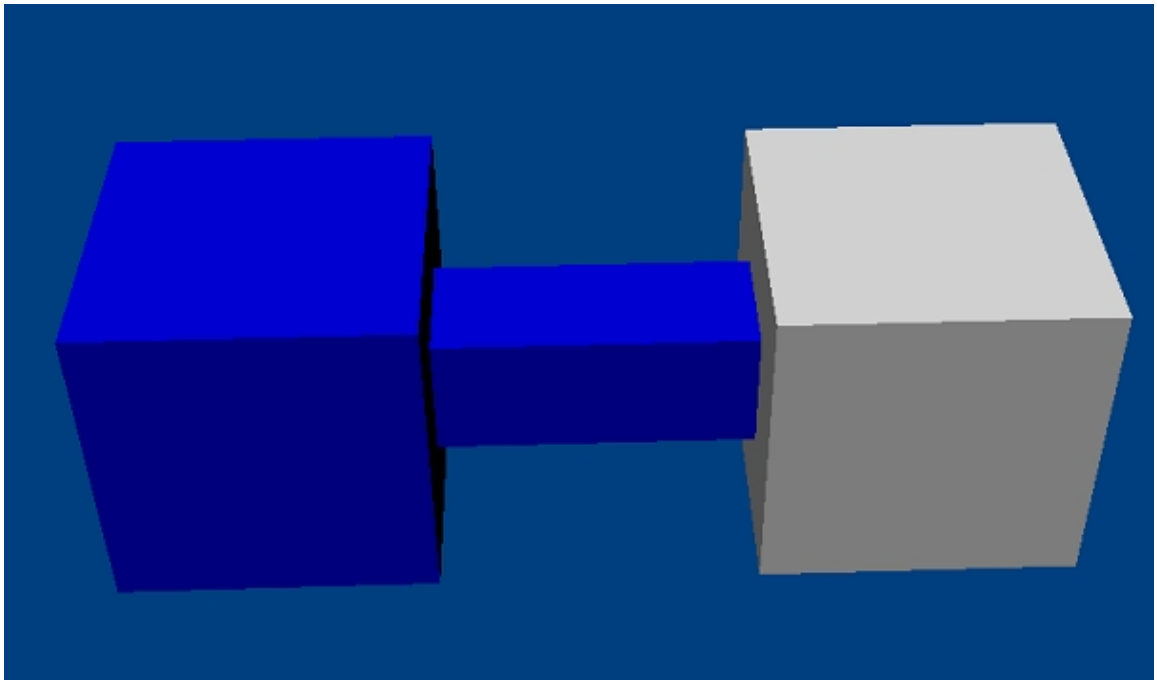
Samples

Sample Joints

API Reference

- NxJoint
- NxJointDesc
- NxCylindricalJoint
- NxCylindricalJointDesc

Fixed Joint



The fixed joint effectively glues two bodies together with no remaining degrees of freedom for relative motion. It is useful to set it to be breakable (see Breakable Joint) to simulate simple fracture effects.

An example for a fixed joint is a factory chimney divided into sections, each section held together with fixed joints. When the chimney is hit, it will break apart and topple over by sections rather than unrealistically falling over in one piece.

DOFs removed: 6
DOFs remaining: 0

Joint Parameters

Parameter	Description
actor[0]	First actor
actor[1]	Second actor

Example

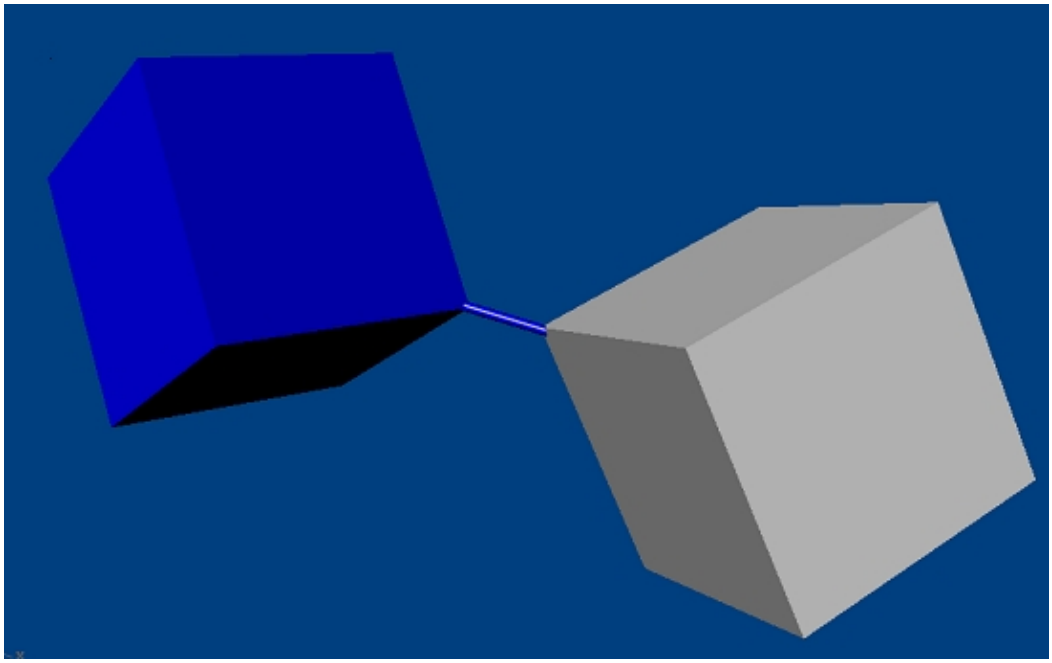
```
NxFixedJointDesc fixedDesc;

fixedDesc.actor[0] = actor0;
fixedDesc.actor[1] = actor1;

NxFixedJoint *fixedJoint=(NxFixedJoint*)gScene->createJoint(fixedDesc);
```

API Reference

- NxJoint
- NxJointDesc
- NxFixedJoint

- NxFixedJointDesc

Distance Joint



The distance joint tries to maintain a certain minimum and/or maximum distance between two points attached to a pair of actors. It can be set to springy in order to behave like a rubber band.

An example for a distance joint is a pendulum swinging on a string, or in the case of a springy distance joint, a rubber band between two objects.

DOFs removed: 1
DOFs remaining: 5

Joint Parameters

Parameter	Description
actor[0]	First actor
actor[1]	Second actor
localAnchor[0]	Point from which the distance is measured on the first actor.
localAnchor[1]	Point from which the distance is measured on the second actor.
maxDistance	The maximum allowed distance between the two points.
minDistance	The minimum allowed distance between the two points.
flags:	
NX_DJF_MAX_DISTANCE_ENABLED	true if the joint enforces the maximum separation distance
NX_DJF_MIN_DISTANCE_ENABLED	true if the joint enforces the minimum separation distance
NX_DJF_SPRING_ENABLED	true if the joint is to be springy

Example

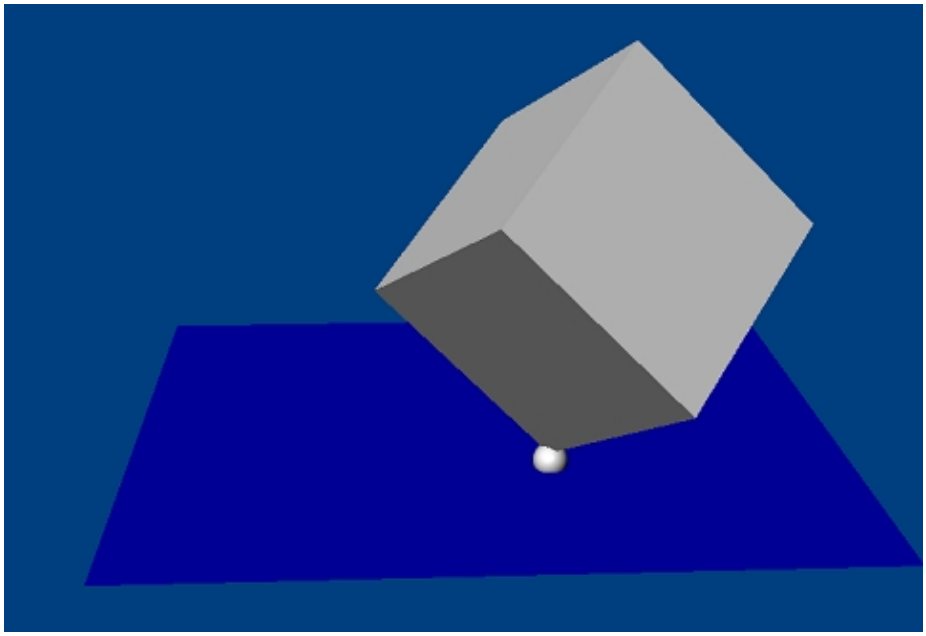
```
NxDistanceJointDesc distDesc;

distDesc.actor[0] = actor0;
```

```
distDesc.actor[1] = actor1;

distDesc.localAnchor[0]=NxVec3(0.0f,1.0f,0.0f);
distDesc.localAnchor[1]=NxVec3(0.0f,-1.0f,0.0f);

distDesc.maxDistance=10.0f;
distDesc.minDistance=10.0f;

distDesc.flags=NX_DJF_MIN_DISTANCE_ENABLED | NX_DJF_MAX_DISTANCE_ENABLED; //Enforce a fixed distance

NxDistanceJoint* distJoint=(NxDistanceJoint)gScene->createJoint(distDesc);
```

API Reference

- NxJoint
- NxJointDesc
- NxDistanceJoint
- NxDistanceJointDesc

Point In Plane Joint



A point in plane joint constrains a point on one actor to only move inside a plane attached to another actor. The point attached to the plane is defined by the anchor point. The joint's axis specifies the plane normal.

An example for a point in plane joint is a magnet on a refrigerator.

DOFs removed: 1
DOFs remaining: 5

Joint Parameters

Parameter	Description
actor[0]	First actor
actor[1]	Second actor
localAnchor[0]	Point attached to the plane on the first actor.
localAnchor[1]	Point attached to the plane on the second actor (the attachment points should coincide).
localAxis[0]	The plane normal in the first actor's frame.
localAxis[1]	The plane normal in the second actor's frame (should match localAxis[0] in the global frame).

NOTE: When setting localAnchor[] it is generally convenient to use setGlobalAnchor() to set the anchor with a world space point.

NOTE: When setting localAxis[] it is inconvenient to use setGlobalAxis() to set the axis with a world space axis.

NOTE: When specifying the joint axis it is also important to specify the localNormal[] which should be orthogonal to the localAxis[]. The localNormal[] is needed to specify joint limits.

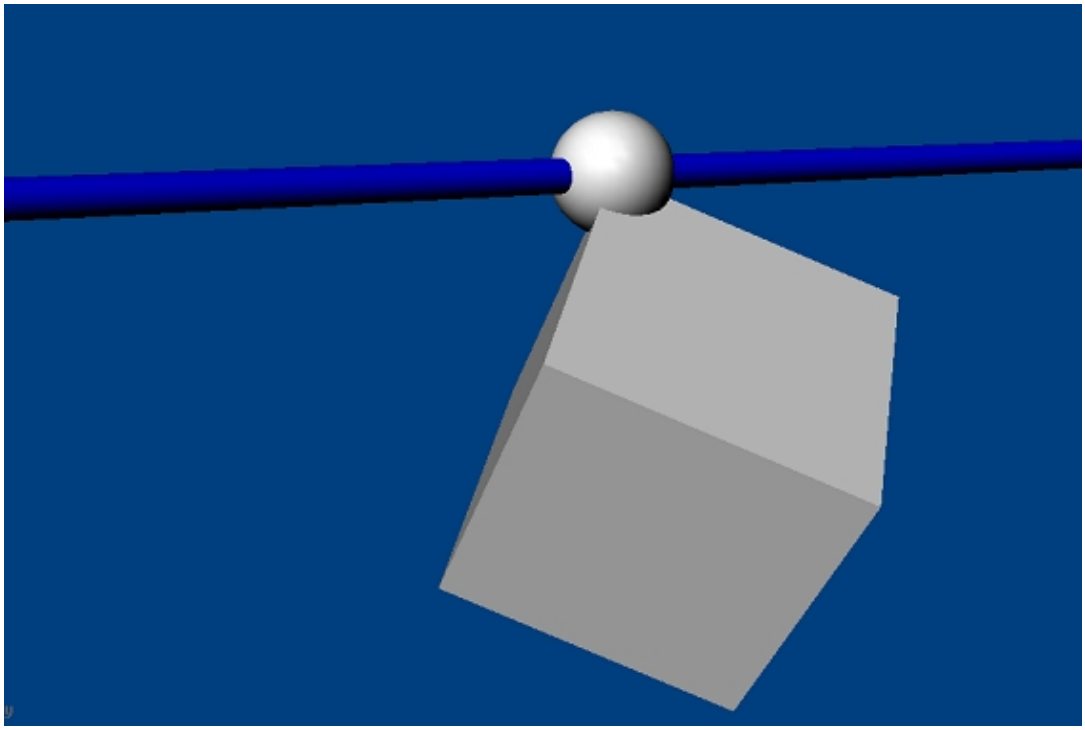
Example

```
NxPointInPlaneJointDesc pipDesc;

pipDesc.actor[0] = actor0;
pipDesc.actor[1] = actor1;

pipDesc.setGlobalAnchor(globalAnchor); //Point attached to plane.
pipDesc.setGlobalAxis(globalAxis); //Point plane normal.

pipDesc.jointFlags |= NX_JF_COLLISION_ENABLED; //Enable collision detection between each actor.

NxPointInPlaneJoint *pipJoint=(NxPointInPlaneJoint*)gScene->createJoint(pipDesc);
```

API Reference

- NxJoint
- NxJointDesc
- NxPointInPlaneJoint
- NxPointInPlaneJointDesc

Point On Line Joint



A point on line joint constrains a point on one actor to only move along a line attached to another actor. The point attached to the line is the anchor point for the joint. The line through this point is specified by its direction (axis) vector.

An example for a point on line joint is a curtain hanger widget.

DOFs removed: 2
DOFs remaining: 4

Joint Parameters

Parameter	Description
actor[0]	First actor
actor[1]	Second actor
localAnchor[0]	Point attached to the line on the first actor.
localAnchor[1]	Point attached to the line on the second actor (the attachment points should coincide in the global frame).
localAxis[0]	The axis direction of the line in the first actor's frame.
localAxis[1]	The axis direction of the line in the second actor's frame (should match localAxis[0] in the global frame).

NOTE: When setting localAnchor[] it is generally convenient to use setGlobalAnchor() to set the anchor with a world space point.

NOTE: When setting localAxis[] it is generally convenient to use setGlobalAxis() to set the axis with a world space axis.

NOTE: When specifying the joint axis it is also important to specify the localNormal[] which should be orthogonal to the localAxis[]. The localNormal[] is needed to specify joint limits.

Example

```
NxPointOnLineJointDesc polDesc;

polDesc.actor[0] = actor0;
polDesc.actor[1] = actor1;

polDesc.setGlobalAnchor(globalAnchor); //Point attached to line.
polDesc.setGlobalAxis(globalAxis); //Direction of line.

polDesc.jointFlags |= NX_JF_COLLISION_ENABLED; //Enable collision detection between each actor.

NxPointOnLineJoint *polJoint=(NxPointOnLineJoint *)gScene->createJoint(pipDesc);
```

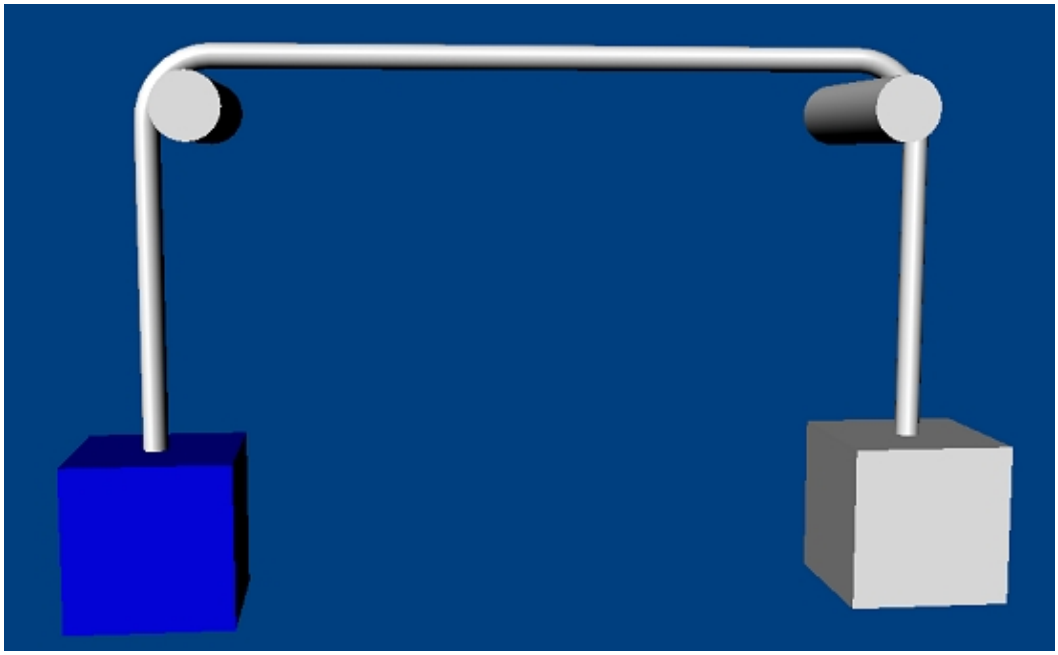
Samples

Sample Joints

API Reference

- NxJoint
- NxJointDesc
- NxPointOnLineJoint
- NxPointOnLineJointDesc

Pulley Joint



The pulley joint simulates a rope that can be thrown across a pair of pulleys. In this way, it is similar to the distance joint (the length of the rope is the distance) but the rope doesn't connect the two bodies along the shortest path, rather it leads from the connection point on one actor to the pulley point (fixed in world space), then to the second pulley point, and finally to the other actor.

The pulley joint can also be used to simulate a rope around a single point by making the pulley points coincide.

Note that a setup where either object attachment point coincides with its corresponding pulley suspension point in world space is invalid. In this case, the simulation would be unable to determine the appropriate direction in which to pull the object and a random direction would result. The simulation will be unstable. Note that it is also invalid to allow the simulation to end up in such a state.

Joint Parameters

Parameter	Description
actor[0]	First actor
actor[1]	Second actor
localAnchor[0]	Attachment point of the rope in the first actor's frame.
localAnchor[1]	Attachment point of the rope in the second actor's frame.
pulley[0]	First suspension point in the global frame.
pulley[1]	Second suspension point in the global frame.
distance	The remaining length of the rope connecting the two objects.
stiffness	Stiffness of the constraint, between 0 and 1 (1 being the stiffest).
ratio	Transmission ratio (e.g., if set to 2, would cause the joint to apply twice the force on actor1 as actor0).
flags:	
NX_PJF_IS_RIGID	true if the joint maintains a minimum and maximum distance

NX_PJF_MOTOR_ENABLED	true if the motor is enabled

Example

```
NxPulleyJointDesc pulleyDesc;

pulleyDesc.actor[0] = actor0;
pulleyDesc.actor[1] = actor1;

pulleyDesc.localAnchor[0] = NxVec3(0,2,0); //Point on actor0 where the rope is attached.
pulleyDesc.localAnchor[1] = NxVec3(0,2,0); //Point on actor1 where the rope is attached.

pulleyDesc.pulley[0] = pulley0; //Suspension points of two bodies in world space.
pulleyDesc.pulley[1] = pulley1; //Suspension points of two bodies in world space.

pulleyDesc.distance = distance; //The remaining length of rope connecting the two objects.
pulleyDesc.stiffness = 1.0f; //Stiffness of the constraint, between 0 and 1 (stiffest).
pulleyDesc.ratio = 1.0f; //Transmission ratio.

pulleyDesc.flags = NX_PJF_IS_RIGID; //NX_PJF_IS_RIGID instructs the joint to maintain the minimum ar

NxPulleyJoint* pulleyJoint = (NxPulleyJoint *)gScene->createJoint(pulleyDesc);
```

Samples

Pulley Joint

API Reference

- NxJoint
- NxJointDesc
- NxPulleyJoint
- NxPulleyJointDesc

6-Degree-of-Freedom Configurable Joint

This joint type can be configured to model nearly any joint imaginable. Each degree of freedom - both linear and angular - can be selectively locked or freed, and separate limits can be applied to each. The 6 DOF joint provides motor drive on all axes independently, and also allows soft limits.

The joint axis (localAxis[]) defines the joint's local x-axis. The joint normal (localNormal[]) defines the joint's local y-axis. The local z-axis is computed as a cross product of the first two.

When constraining the angular motion of the joint, rotation around the x-axis is referred to as *twist*, rotation around the y-axis as *swing1*, and rotation around the z-axis as *swing2*.

DOFs removed: 0-6
DOFs remaining: 0-6

Joint Parameters

Parameter	Description
actor[0]	First actor
actor[1]	Second actor
localAnchor[0]	Anchor/reference point in the first actor's frame.
localAnchor[1]	Anchor/reference point in the second actor's frame.
localAxis[0]	Joint local x-axis, attached to actor[0].
localAxis[1]	
localNormal[0]	Joint local y-axis, attached to actor[0].
localNormal[1]	
twistMotion	If locked, do not allow rotation around the local x-axis.
swing1Motion	If locked, do not allow rotation around the local y-axis.
swing2Motion	If locked, do not allow rotation around the local z-axis.
xMotion	If locked, do not allow translation motion along the local x-axis.
yMotion	If locked, do not allow translation motion along the local y-axis.
zMotion	If locked, do not allow translation motion along the local z-axis.
Limits:	
linearLimit	Used to set a translational limit.
swing1Limit	Used to set a rotational limit around the local y-axis.
swing2Limit	Used to set a rotational limit around the local z-axis.
twistLimit	Used to set a rotational limit around the local x-axis.
Motors:	
xDrive	Apply a linear drive along the local x-axis (to a specific velocity or position).
yDrive	Apply a linear drive along the local y-axis.
zDrive	Apply a linear drive along the local z-axis.
swingDrive	Drive to apply to swing1 and swing2 (y and z axis).
twistDrive	Drive the joint rotation around the local x-axis.
slerpDrive	Drive to apply when in SLERP mode (shortest arc to target).

drivePosition	Position to drive joint towards.
driveOrientation	Orientation to drive joint towards.
driveLinearVelocity	Linear velocity to drive the joint towards.
driveAngularVelocity	Angular velocity to drive the joint towards.
flags:	
NX_D6JOINT_SLERP_DRIVE	Drive along the shortest spherical arc.
NX_D6JOINT_GEAR_ENABLED	when the flag NX_D6JOINT_GEAR_ENABLED is set, the angular velocity of the second actor is driven towards the angular velocity of the first actor times gearRatio (both w.r.t. their primary axis).

NOTE: When setting localAnchor[] it is generally convenient to use setGlobalAnchor() to set the anchor with a world space point.

NOTE: When setting localAxis[] it is generally convenient to use setGlobalAxis() to set the axis from a world space axis.

NOTE: When specifying the joint axis it is also important to specify the localNormal[] which should be orthogonal to the localAxis[]. The localNormal[] is needed to specify joint limits and rotational degrees of freedom.

Examples

Fixed Joint Configuration:

```
NxD6JointDesc d6Desc;

d6Desc.actor[0] = actor0;
d6Desc.actor[1] = actor1;

d6Desc.setGlobalAnchor(globalAnchor);//Important when DOFs are unlocked.
d6Desc.setGlobalAxis(globalAxis);//Joint configuration.

d6Desc.twistMotion = NX_D6JOINT_MOTION_LOCKED;//Create a fixed joint.
d6Desc.swing1Motion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.swing2Motion = NX_D6JOINT_MOTION_LOCKED;

d6Desc.xMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.yMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.zMotion = NX_D6JOINT_MOTION_LOCKED;

d6Desc.projectionMode = NX_JPM_NONE;

NxD6Joint d6Joint=(NxD6Joint*)gScene->createJoint(d6Desc);
```

Revolute Joint Configuration:

```
...
d6Desc.twistMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.swing1Motion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.swing2Motion = NX_D6JOINT_MOTION_LOCKED;

d6Desc.xMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.yMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.zMotion = NX_D6JOINT_MOTION_LOCKED;
...
```

Spherical Joint Configuration:

```
...
d6Desc.twistMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.swing1Motion = NX_D6JOINT_MOTION_FREE;
d6Desc.swing2Motion = NX_D6JOINT_MOTION_FREE;

d6Desc.xMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.yMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.zMotion = NX_D6JOINT_MOTION_LOCKED;
...
```

Samples

Sample D6 Joint

API Reference

- NxJoint
- NxJointDesc
- NxD6Joint
- NxD6JointDesc

Frames

Setting up the joint constraint frames is the first step in successfully configuring a joint. A frame marks a position and orientation in space. It is viewed as a set of three right angle axes meeting at a point that is considered the origin of the frame.

The d6joint uses a pair of frames, one fixed in the world and one fixed in the constrained body. The frame locations are specified by their fixed offset relative to their respective actor frame (or the world frame).

A common way to specify 6 DOF frame offsets in computer graphics is as 4x4 homogeneous rigid body transformation matrices that contain both the position offset of the origin and the rotational offset of the frame as a 3x3 submatrix. In the NVIDIA PhysX SDK, joint frames are specified using an anchor point and a pair of unit vectors which represent the joint frames orientation relative to the body frames. The third axis can be derived from the other two axes since they are orthogonal, where the z-axis is computed as a cross product of x and y (see the diagram in the 6-Degree-of-Freedom Configurable Joint section).

The x-axis is the principal axis in a d6joint frame for the purposes of a twist and swing decomposition of the joint's rotation; an arbitrary orientation is achieved by (1) a twist rotation about the x-axis followed by (2) a swing rotation about an axis in the yz plane.

Parent and Child Constraint Frames

Using the parent/child naming conventions, the first frame is the parent, corresponding to actor[0] and the second frame is the child, corresponding to actor[1]. Usually the parent frame is fixed in the world or located in the limb that is closer to the central root in a hierarchy, but this is not enforced.

The principal axis of the child frame will usually be aligned with the symmetry axis of its actor (e.g., the long axis of a limb or the rotation axis of a wheel). The principal axis of the parent frame is used to specify the center of a swing limit so should be set in the middle range of motion of the child limb.

The y and z axes provide a reference for the twist rotation of the principal axis between parent and child. The y and z axes of the parent frame are also used for aligning an optional swing limit.

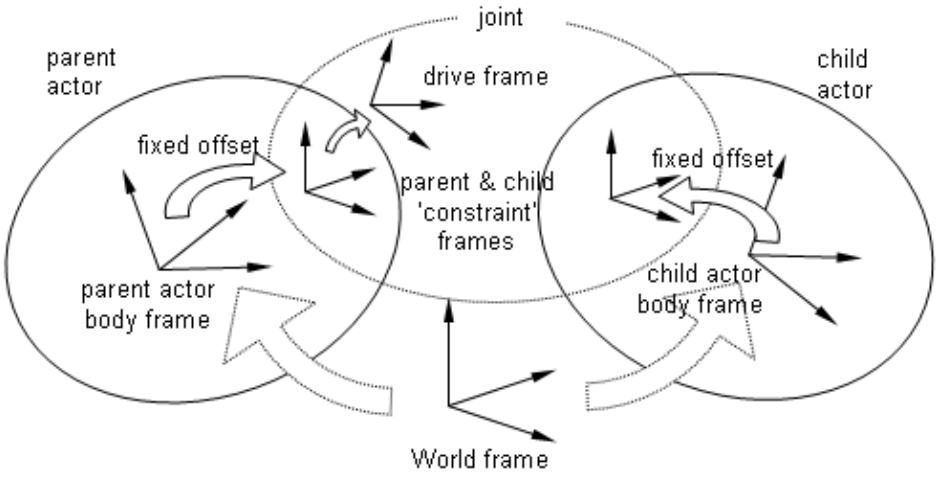
Drive Frame

When driving the joint it can be useful to imagine a third drive frame specified by the drive position input &"" lang="EN-US">Frame Relationship Summary

By now there are six frames involved, including the world frame and the actor frames but ignoring other frames such as the center of mass frames which are not directly involved with the joint configuration. There is plenty of scope for confusion if it is not clear how frame offsets are being specified or what conventions are being used.

Note that once the joint constraint frames have been set up, as above, at some (fixed) offset from their respective actor body frames then the actor body frames can be forgotten; their position and alignment has no effect on the action of the joint, only the constraint frames are involved in the operation of the joint.

The following sketch shows the relative frame offsets as they are stored and treated by the d6joint.



The joint frames are sketched separate from each other for clarity, but the joint constraints will act to bring the two constraint frames together and drive constraints will act to bring the drive frame and the child frame together.

The NVIDIA PhysX SDK allows for frame setup in global coordinates (i.e., relative to the world frame or absolute). A single global frame position and orientation can be used to initialize both constraint frames. This can be convenient in simple cases when the actors are in suitable starting positions; however, for more complex cases, like setting up rag doll joint limits, parent and child frames are best set up separately.

The NxJointDesc::setGlobalAnchor() and NxJointDesc::setGlobalAxis() functions are used to set the frame position and orientation in the global frame (remember when using setGlobalAxis() that the x-axis is the principle axis for a d6joint, unlike other joints).

D6Joint Coordinates

For specifying position constraints, it is useful to think of the joint's 6DOF as being parameterized by six coordinates, three linear and three angular, between the constraint frame fixed in the parent actor and the constraint frame fixed in the child actor.

For the angular DOFs, the concept of ‘twist and swing’ is used to structure the API for specifying angular joint types and limits and to actually implement the default angular drive behavior so that twist and swing freedoms are decoupled.

Linear Coordinates: Cartesian X, Y, Z

The three linear DOFs are uncomplicated; they are naturally specified by the three Cartesian origin coordinates of the child frame in the parent frame. Similarly, the drive frame position is specified by Cartesian coordinates relative to the parent frame, and the velocity components are simply derivatives of the position coordinates.

Default Angular Coordinates: Twist & Swing

Twist and swing coordinates are the default angular coordinates for the d6joint. They decouple a 3D rotation into a 1D twist rotation about the principal frame axis and a swing rotation about an axis in the orthogonal plane of the two remaining frame axes.

Twist and swing coordinates have a singularity; when the child limb swings a full 180 degrees so that its principal axis points in exactly the reverse direction of the parent’s principal axis, the limb’s twist is not defined. Avoid this singularity; limits should be used when a zero-twist joint is configured or when driving a ball-and-socket joint with the default twist and swing coordinates.

This intuitive idea of twist and swing is all that is needed to configure joints and limits. More details are given in the Angular Drives section.

API Reference

- NxJoint
- NxJointDesc
- NxD6Joint
- NxD6JointDesc

---

span style="font-family: "Times New Roman"; font-style: normal; font-variant: normal; font-weight: normal;

normal2font-size: 7pt; line-height: normal; font-size-adjust: none; font-stretch: normal;" D6Joint Coordinates

normal2font-size: 7pt; line-height: normal; font-size-adjust: none; font-stretch: normal;" D6Joint Coordinates

Linear Joint Types

Four different types of linear joints can be configured by locking zero, one, two, or all three linear DOFs.

The table below lists all eight linear joint possibilities with their suggested names:

X	Y	Z	Joint Name
free	free	free	free-linear
locked	free	free	point-in-plane-X
free	locked	free	point-in-plane-Y
free	free	locked	point-in-plane-Z
free	locked	locked	point-in-line-x
locked	free	locked	point-in-line-y
locked	locked	free	point-in-line-z
locked	locked	locked	fixed-linear

Examples

free-linear joint:

There are no linear constraints. The child actor is free to translate relative to the parent actor.

Any of the linear DOFs may be limited and/or driven.

```
d6Desc.xMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.yMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.zMotion = NX_D6JOINT_MOTION_FREE;
```

point-in-plane joint:

These are also known as planar joints. A single linear DOF is constrained.

The child actor is constrained so that a point, the origin of its constraint frame, is restricted to a plane fixed in the parent actor selected as the X, Y or Z plane of its constraint frame (X=yz, Y=zx, Z=xy).

```
d6Desc.xMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.yMotion = NX_D6JOINT_MOTION_LOCKED; //Constraint to the y==0 plane
d6Desc.zMotion = NX_D6JOINT_MOTION_FREE;
```

point-in-line joint:

These are also known as cylindrical joints or, when all the angular DOFs are also constrained, prismatic joints. One linear DOF is left free and the other two are constrained.

The child actor is constrained so that a point, the origin of its constraint frame, is restricted to a line fixed in the parent actor selected as the x, y or z axis of its constraint frame.

```
d6Desc.xMotion = NX_D6JOINT_MOTION_FREE; //Constraint to the x-axis
d6Desc.yMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.zMotion = NX_D6JOINT_MOTION_LOCKED;
```

fixed-linear joint:

All three linear DOFs are constrained. This is a common case, where the angular constraints define the interesting characteristics of the joint. For example, a ball and revolute joint will have all its linear DOFs locked.

```
d6Desc.xMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.yMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.zMotion = NX_D6JOINT_MOTION_LOCKED;
```

API Reference

- NxJoint
- NxJointDesc
- NxD6Joint
- NxD6JointDesc

Angular Joint Types

There are five types of angular joints that can be configured by locking or freeing different DOFs based on the twist and swing decomposition of the angular freedoms.

The table below lists all eight angular constraint possibilities with their suggested names:

Twist	Swing1	Swing2	constraint name	freedom name	Joint Name
free	free	free		free-angular	spherical
locked	free	free	zero-twist	free-swing	iso-universal-x
free	locked	free	zero-swing1	twist&swing2	universal-xz
free	free	locked	zero-swing2	twist&swing1	universal-xy
free	locked	locked	zero-swing	free-twist	twist-revolute
locked	free	locked		free-swing1	swing1-revolute
locked	locked	free		free-swing2	swing2-revolute
locked	locked	locked	fixed-angular		fixed-joint

Examples

free-angular joint:

There are no angular constraints. Locking the linear DOFs creates a ball-and-socket/spherical joint.

Any limits or drives can be used. This is the only joint configuration to which the SLERP drive option applies.

```
d6Desc.twistMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.swing1Motion = NX_D6JOINT_MOTION_FREE;
d6Desc.swing2Motion = NX_D6JOINT_MOTION_FREE;
```

zero-twist joint (aka, iso-universal):

The zero-twist joint allows the primary axis of the child to swing freely away from the primary axis of the parent but only with zero twist in the sense that the shortest rotation is taken to align the two axes.

This creates a kind of isotropic universal joint which avoids the problems of the usual 'engineering style' universal joint (see below) that is sometimes used as a kind of twist constraint.

There is a nasty singularity at 180 degree swing so the remaining swing freedoms should be limited with a swing cone limit and/or driven to avoid the singularity.

```
d6Desc.twistMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.swing1Motion = NX_D6JOINT_MOTION_FREE;
d6Desc.swing2Motion = NX_D6JOINT_MOTION_FREE;
```

zero-swing1 or zero-swing2 joint (aka, Universal):

The zero-swing1 joint constrains the x-axis of the child to remain orthogonal to the z-axis of the parent. The zero-swing2 joint constrains the x-axis of the child to remain orthogonal to the y-axis of the parent.

The selected swing DOF is constrained so the lower limb is free to twist about its twist axis and to swing about the remaining free swing axis.

In character applications, this joint can be used to model an elbow swing joint incorporating the twist freedom of the lower arm or a knee swing joint incorporating the twist freedom of the lower leg.

In vehicle applications, these joints can be used as ‘steered wheel’ joints in which the child actor is the wheel, free to rotate about its twist axis, while the free swing axis in the parent acts as the steering axis.

In fact, the constraint involved in these joints is exactly that of an ‘engineering style’ universal joint but whose primary axis is one of the swing axes. Universal joints are not recommended for use in game applications because of their anisotropic behavior and singularities at 90 degrees . The zero-twist joint is a better behaved alternative for most use cases. The d6joint does not support the universal joint as a first class type (i.e., whose primary axis is the twist axis). These joints provide a way to implement universal joints but you are on your own – beware the dreaded gimbal lock.

```
        d6Desc.twistMotion = NX_D6JOINT_MOTION_FREE;
        d6Desc.swing1Motion = NX_D6JOINT_MOTION_LOCKED;
        d6Desc.swing2Motion = NX_D6JOINT_MOTION_FREE;

or

        d6Desc.twistMotion = NX_D6JOINT_MOTION_FREE;
        d6Desc.swing1Motion = NX_D6JOINT_MOTION_FREE;
        d6Desc.swing2Motion = NX_D6JOINT_MOTION_LOCKED;
```

free-twist, free-swing1 and free-swing2 joint (aka, Revolute Joint):

Locking any two angular DOFs leaves a single angular DOF. The selected axis in both parent and child frame is held in alignment and so acts as the axis of rotation between the two frame orientations.

This is clearly the angular constraint part of a revolute joint. With linear DOFs locked, the free-twist constraint completes a hinge joint, free to rotate about the x-axis (free-swing1 = y-axis, free-swing2 = z-axis).

While the actual constraint in all of these joints is the same, it is recommended that revolute joints are implemented as twist revolute joints because joint limits and drives are geared towards this case.

```
        d6Desc.twistMotion = NX_D6JOINT_MOTION_FREE;
 6dDesc.swing1Motion = NX_D6JOINT_MOTION_LOCKED;
        d6Desc.swing2Motion = NX_D6JOINT_MOTION_LOCKED;
```

fixed-joint:

The fixed joint constrains all three angular DOFs. If just one of the linear DOFs is allowed free motion, then the d6joint becomes a prismatic joint.

```
        d6Desc.twistMotion = NX_D6JOINT_MOTION_LOCKED;
        d6Desc.swing1Motion = NX_D6JOINT_MOTION_LOCKED;
        d6Desc.swing2Motion = NX_D6JOINT_MOTION_LOCKED;
```

API Reference

- NxJoint
- NxJointDesc
- NxD6Joint
- NxD6JointDesc

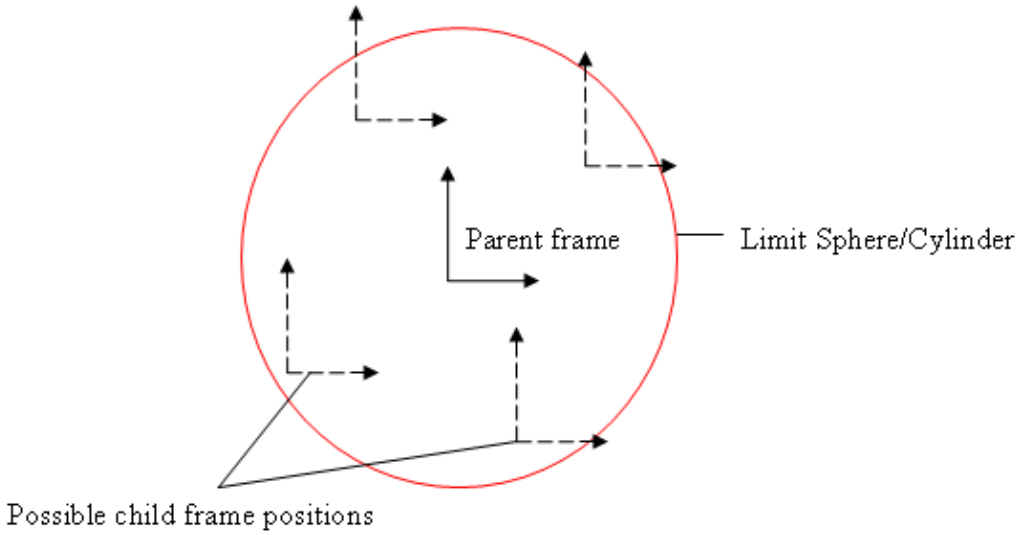
Linear Joint Limits

Linear joint limits are currently specified by a single radial distance parameter applying to all three DOFs. It is a radial parameter because with two or three limited DOFs the limits are combined into a circle or a sphere, rather than a square or a cubical box, in order to avoid having multiple limits active at corners.

Examples

Three linear DOFs limited: Spherical limit

Limiting all three DOFs gives a spherical limit surface which can be visualized as a bubble centered on the parent frame origin, such that the origin of the child frame is constrained to remain inside the sphere.



The red circle becomes a sphere when extended to 3 dimension.

```
//Spherical linear limit (hard limit)
d6Desc.xMotion = NX_D6JOINT_MOTION_LIMITED;
d6Desc.yMotion = NX_D6JOINT_MOTION_LIMITED;
d6Desc.zMotion = NX_D6JOINT_MOTION_LIMITED;

d6Desc.linearLimit.value=limitRadius;
d6Desc.linearLimit.damping=0.0f;
d6Desc.linearLimit.restitution=0.0f;
```

Two linear DOFs limited: Cylindrical limit

Limiting two linear DOFs and leaving one free creates a cylindrical limit surface centered on the free axis. The child frame origin is constrained to remain inside the cylinder.

See the above illustration, but instead of the red circle becoming a sphere, when extended to 3 dimension, it becomes a cylinder along the axis into the page.

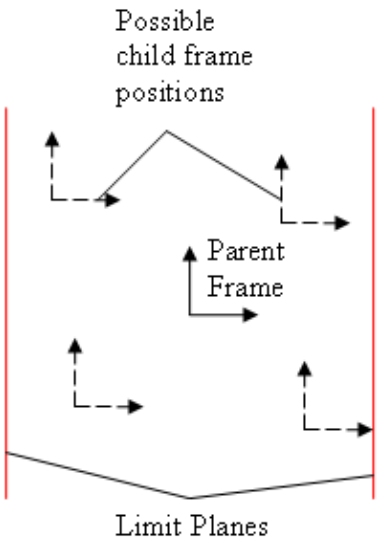
If the third DOF is locked rather than left free, the joint collapses to a planar joint with a circular limit.

```
//Cylindrical linear limit. The axis of the cylinder is the x axis.
d6Desc.xMotion = NX_D6JOINT_MOTION_FREE;
```

```
d6Desc.yMotion = NX_D6JOINT_MOTION_LIMITED;
d6Desc.zMotion = NX_D6JOINT_MOTION_LIMITED;

d6Desc.linearLimit.value=limitRadius;
d6Desc.linearLimit.damping=0.0f;
d6Desc.linearLimit.restitution=0.0f;
```

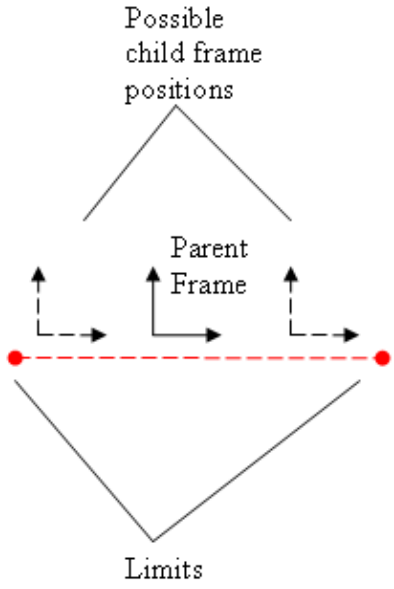
One linear DOF limit: Pair-of-planes limit

Limiting just one DOF and leaving the others free creates a pair of parallel limit planes fixed in the parent and separated along the selected limited axis by the radial distance either side of the parent frame origin. The child frame origin is constrained to lie between the two planes.



If one of the free DOFs is locked then the joint collapses to a planar joint and the origin of the child frame is constrained to lie between a pair of parallel lines centered in the selected plane of the parent frame.

If both of the free DOFs are locked then the joint collapses to a point-in-line joint with the child frame origin constrained to lie between ± the specified radial distance along the selected parent frame axis.




```
//Single linear limit
d6Desc.xMotion = NX_D6JOINT_MOTION_LIMITED;
d6Desc.yMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.zMotion = NX_D6JOINT_MOTION_FREE;

d6Desc.linearLimit.value=limitRadius;
d6Desc.linearLimit.damping=0.0f;
d6Desc.linearLimit.restitution=0.0f;
```

API Reference

- NxJoint
- NxJointDesc
- NxD6Joint
- NxD6JointDesc

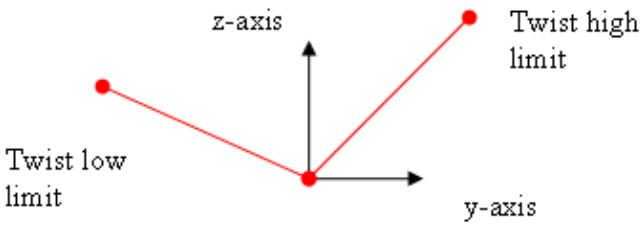
Angular Joint Limits

Limits on the 1DOF twist and the 2DOF swing motions are configured separately; therefore, unlike the linear case, two limit constraints can be active on angular DOFs.

Also unlike linear, each DOF has its own separate maximum limit parameter - angles in this case. As with the linear radius parameter, they are specified as maximum angles measured symmetrically on both sides of the zero angle so that the full angular separation between the limits is twice the specified angle.

Twist Limit

The twist limit is a little different than the swing limits, specified using a pair of angles rather than just a single radius. This allows the limit orientations to be nonsymmetrical around the twist axis, unlike the swing axis.



x-axis(twist) coming out of the page.

To visualize the affect of the limits, it is best to consider them with respect to a particular axis, such as the z-axis. In the diagram above, the child frame and parent frame are aligned. When the joint twists, the z-axis can rotate around the twist axis, but only as far as the limits (red lines).

```
//Set a hard twist limit, which limits the twist angle to twistLowLimit < twistAngle < twistH

d6Desc.twistMotion = NX_D6JOINT_MOTION_LIMITED;

d6Desc.twistLimit.low.value=NxMath::degToRad(twistLowLimit); //d6Joint angles are specified i
d6Desc.twistLimit.low.damping=0.0f;
d6Desc.twistLimit.low.restitution=0.0f;
d6Desc.twistLimit.low.spring=0.0f;

d6Desc.twistLimit.high.value=NxMath::degToRad(twistHighLimit);
d6Desc.twistLimit.high.damping=0.0f;
d6Desc.twistLimit.high.restitution=0.0f;
d6Desc.twistLimit.high.spring=0.0f;
```

Swing Limit Types

The table below lists all five swing limit possibilities with suggested their names:

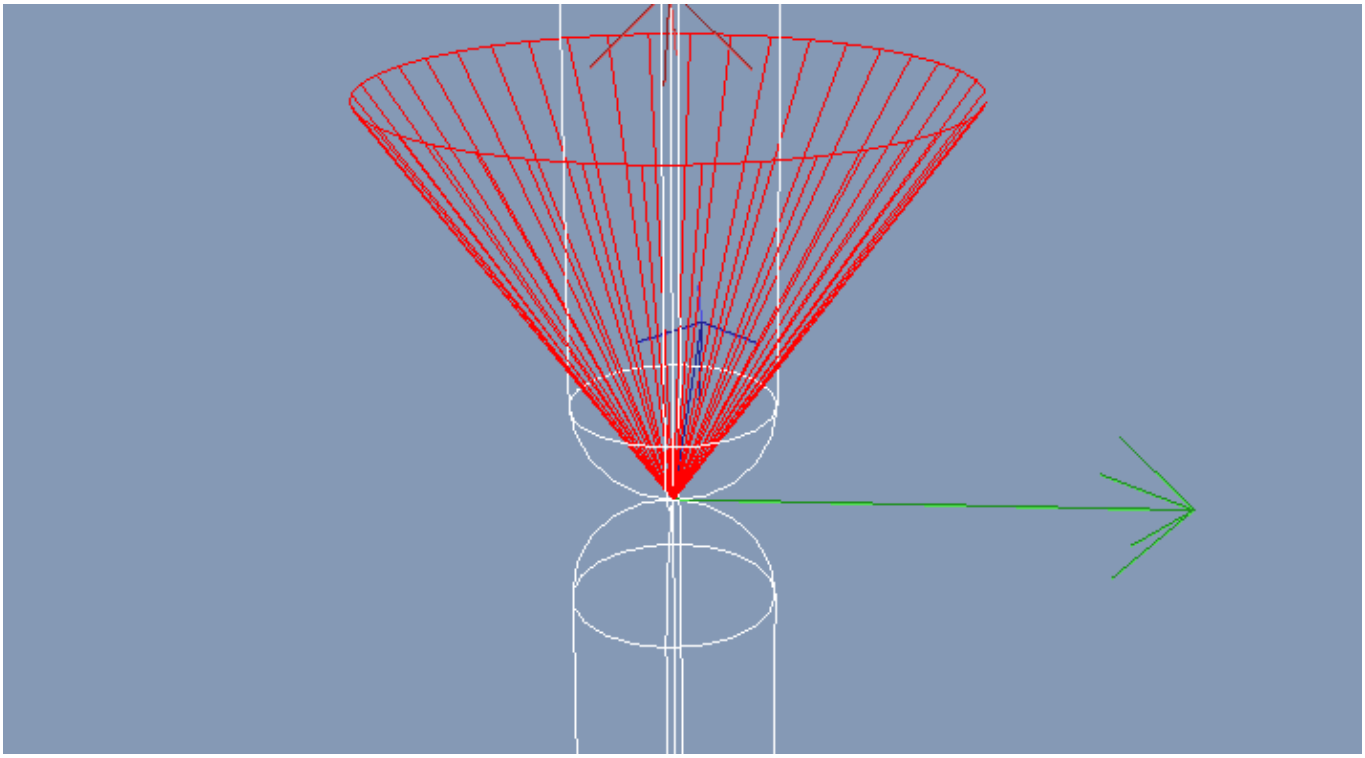
Swing1	Swing2	Limit Name
limited	limited	(elliptic) swing-cone limit
limited	free	swing1 limit

free	limited	swing2 limit
limited	locked	zero-swing2 joint with swing1 limit
locked	limited	zero-swing1 joint with swing2 limit

Examples

swing-cone limit:

If both swing DOFs are limited, then the limit surface geometry is a cone centered on the principal axis of the parent. The principal axis of the child is constrained to stay inside the parent-fixed cone.



In the above screen shot, the cone is represented in red. The principal child axis (x-axis) is a red arrow near the center of the cone; it is constrained to lie within the cone.

If both parameters specifying the maximum swing angles are equal, then the cone is circular. Generally, however, the cone has an elongated elliptical shape.

The maximum swing angles may be set between 0 and 180°.

Avoid cases where one swing limit angle is much smaller than the other as this results in an extremely elongated (highly eccentric elliptic) swing cone limit which may cause problems in simulation (the small swing DOF could be locked instead).

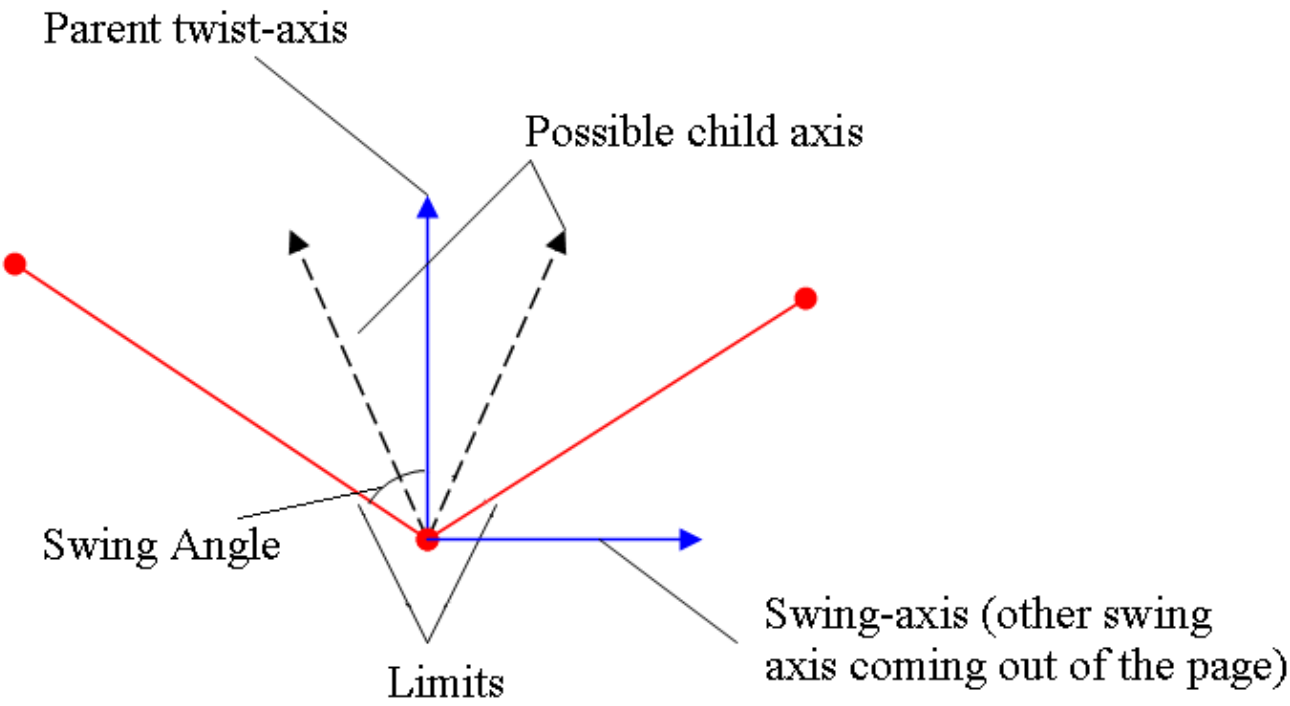
```
//Limit motion on the swing axis to lie within the elliptical cone specified by the
//angles swing1Limit and swing2Limit, relative to the parent frame.

d6Desc.swing1Motion = NX_D6JOINT_MOTION_LIMITED;

d6Desc.swing1Limit.value=NxMath::degToRad(swing1Limit);
d6Desc.swing1Limit.damping=0.0f;
d6Desc.swing1Limit.restitution=0.0f;
```

```
d6Desc.swing2Motion = NX_D6JOINT_MOTION_LIMITED;

d6Desc.swing2Limit.value=NxMath::degToRad(swing2Limit);
d6Desc.swing2Limit.damping=0.0f;
d6Desc.swing2Limit.restitution=0.0f;
```

single-swing limit:

If only one swing DOF is limited and the other is left free, this is interpreted as a limit surface in the shape of a double cone aligned with the free swing axis of the parent frame. The principal axis of the child is constrained to be excluded from the double cone.

If the other swing DOF is locked rather than left free, then swing sphere reduces to a circle and the single swing limit is just a segment of the circle located on the specified swing limit angle either side of zero.




```
d6Desc.swing1Motion = NX_D6JOINT_MOTION_LIMITED;

d6Desc.swing1Limit.value=NxMath::degToRad(parent->swing1Limit);
d6Desc.swing1Limit.damping=0.0f;
d6Desc.swing1Limit.restitution=0.0f;
```

API Reference

- NxJoint
- NxJointDesc
- NxD6Joint
- NxD6JointDesc

Soft Limits and Restitution

Soft Limits

The d6joint allows a limit to be 'soft' (i.e., allows a certain amount of limit violation when the joint is pushing against a limit). This creates a smoother and more realistic response when an object hits a limit, in addition to improving the joint's stability.

A common scenario for using soft limits is when the joint is being driven by animation data. In some cases, the data may not be perfect and try to force the joint outside the range defined by its limits. In this case, it looks more realistic to give a little than to stop the motion dead.

To enable soft limits for a joint, set the spring and damping parameters of the appropriate joint limit to a positive value. A value of zero is the default, used to signify a hard limit.

```
//Single linear limit
d6Desc.xMotion = NX_D6JOINT_MOTION_LIMITED;
d6Desc.yMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.zMotion = NX_D6JOINT_MOTION_FREE;

d6Desc.linearLimit.value = limitRadius;
d6Desc.linearLimit.spring = 0.5f;
d6Desc.linearLimit.damping = 0.1f;
```

Restitution

When a joint hits a limit, it is possible to give the limit a bounce so that it reflects back with a proportion of the velocity that it hits the limit with (along the axis of the limit). This is similar to specifying a restitution value when dealing with contacts. A restitution value of zero is the default which causes the joint to stop dead - a value of 1 will cause the joint to bounce back from the limit with the same velocity that it hit.

In situations where the joint has many locked DOFs (e.g., 5) the restitution may not be applied correctly. This is due to a limitation in the solver which causes the restitution velocity to become zero as the solver enforces constraints on the other DOFs.

This limitation applies to both angular and linear limits; however, it is generally most apparent with limited angular DOFs.

Disabling joint projection and increasing the solver iteration count may improve this behavior to some extent.

Also, combining soft joint limits with joint motors trying to drive past those limits may affect stability. This can be alleviated through either tweaking the limit's and drive's parameters (such as making the target lie within the limits), or avoiding the situation entirely, either by eschewing the combination or by switching off the motor.

```
//Single linear limit
d6Desc.xMotion = NX_D6JOINT_MOTION_LIMITED;
d6Desc.yMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.zMotion = NX_D6JOINT_MOTION_FREE;

d6Desc.linearLimit.value = limitRadius;
d6Desc.linearLimit.restitution = 0.5f;
```

API Reference

- NxJoint
- NxJointDesc
- NxD6Joint
- NxD6JointDesc

Linear Drives

A joint with a single linear DOF, i.e. a slider or prismatic joint, has an obvious coordinate to drive; its linear extension is represented by a single Cartesian coordinate - x, y or z. Each linear DOF that is not locked can be driven by its Cartesian coordinate so multi-DOF linear joints are easily understood as multiple 1DOF joints. Drive velocities are the rate of change of the Cartesian coordinates. NOTE: Angular Drive is more involved, so be sure to understand the linear drive model before progressing to angular drive.

Linear Drive Model

The linear drive model for the d6joint consists of the following parameters:

- Drive targets are specified in relation to the parent actor's (actor[0]) frame using the following members of NxD6JointDesc:
  - ♦ drivePosition **-** a vector representing a target position.
  - ♦ driveLinearVelocity **-** a vector representing the target velocity.
- Separate drive descriptors for x (xDrive), y (yDrive) and z (zDrive) can be specified using these NxJointDriveDesc members:
  - ♦ driveType **-** used to apply a type of drive with the following flags:
    - ◊ NX_D6JOINT_DRIVE_POSITION - Enable drive to a positional goal.
    - ◊ NX_D6JOINT_DRIVE_VELOCITY - Enable drive to a velocity goal.
  - ♦ spring **-** amount of force needed to move the joint to its target position proportional to the distance from the target (not used for a velocity drive).
  - ♦ damping **-** applied to the drive spring (used to smooth out oscillations about the drive target).
  - ♦ forceLimit **-** maximum force applied when driving towards a velocity target (not used for a positional drive).

Position drive attempts to follow the desired position input with the configured stiffness and damping properties. A physical lag due to the inertia of the driven body acting through the drive spring will occur; therefore, sudden step changes will result over a number of time steps.

Physical lag can be reduced by stiffening the spring. Another way to control the lag is to apply a velocity target in addition to a positional target.

With a fixed position input, a position drive will spring about that drive position with the specified springing/damping characteristics. Adding a velocity drive in this case will deflect the drive from the desired position, just as an external force stretches the spring.

In addition to specifying drive targets through the joint descriptor, the d6joint provides methods to update the targets only, as shown below:

```
void setDrivePosition(const NxVec3 &position);
void setDriveOrientation(const NxQuat &orientation);

void setDriveLinearVelocity(const NxVec3 &linVel);
void setDriveAngularVelocity(const NxVec3 &angVel);
```

Example

```
d6Desc.xMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.yMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.zMotion = NX_D6JOINT_MOTION_FREE;

d6Desc.zDrive.driveType = NX_D6JOINT_DRIVE_POSITION;
```

```
d6Desc.zDrive.forceLimit = FLT_MAX;
d6Desc.zDrive.spring = 100.0f;
d6Desc.zDrive.damping = 0;

d6Desc.drivePosition.set(0.0f, 0.0f, 0.0f); //Drive the joint to the local(actor[0]) origin.
```

Why Not Just Drive with Forces?

One way to drive a joint is to apply a force. While this can be effective for modeling soft, springy effects, it quickly becomes unstable for stiff springs and forces, due to a limitation of the numerical integration used to advance time from one time step to the next. The use of forces for direct drive should be avoided unless you are fully aware of the technical meaning of stiff, and therefore understand the potential pitfalls of simulation using a first order method.

The d6joint uses a drive constraint in which the springs are implicitly integrated within the solver, as this is the best way to model stiff behavior for stable simulation. Use of drive constraints are also convenient for modeling soft behavior so there is little reason to use direct force.

If you do use forces, be careful to apply equal and opposite forces to the parent and child actors or the result will be physically incorrect.

API Reference

- NxJoint
- NxJointDesc
- NxD6Joint
- NxD6JointDesc

Angular Drives

Angular drive differs from linear drive in a fundamental way: it does not have a simple and intuitive representation free from singularities. For this reason, the d6joint provides two angular drive models - twist and swing and SLERP (Spherical Linear Interpolation).

Twist and swing is intuitive in many situations; however, there is a singularity when driven to 180 degrees swing. In addition, the drive will not follow the shortest arc between two orientations.

On the other hand, SLERP drive will follow the shortest arc between a pair of angular configurations, but may cause un-intuitive changes in the joint's twist and swing.

The d6joint angular drive takes a quaternion as a target orientation. In a SLERP drive, the quaternion is used directly. In a twist and swing drive, it is decomposed into separate twist and swing components and each component is interpolated separately.

Angular Drive Model

The angular drive model for the d6joint consists of the following parameters:

- Velocity targets are specified in relation to the parent actor's (actor[0]) frame using the following members of NxD6JointDesc:
    - ♦ driveOrientation **-** a quaternion representing a relative orientation.
    - ♦ driveAngularVelocity **-** a vector representing the target velocity as a scaled axis.
- Separate drive descriptors for SLERP (slerpDrive), swing (swingDrive) and twist (twistDrive) can be specified using these NxJointDriveDesc members:
    - ♦ driveType - used to apply a type of drive with the following flags:
        - ◊ NX_D6JOINT_DRIVE_POSITION - Enable drive to a goal orientation.
        - ◊ NX_D6JOINT_DRIVE_VELOCITY - Enable drive to a velocity goal.
    - ♦ spring **-** amount of torque needed to move the joint to its target orientation proportional to the angle from the target (not used for a velocity drive).
    - ♦ damping **-** applied to the drive spring (used to smooth out oscillations about the drive target).
    - ♦ forceLimit **-** maximum torque applied when driving towards a velocity target (not used for an orientation drive).

Angular orientation drive will attempt to follow the desired orientation input with the configured stiffness and damping properties, while the joint constraints are enforced by the solver. Ideally, the inputs should be consistent with the joint constraints, requiring more effort compared to the linear case.

The default drive behavior for the angular DOFs interpolates between orientations with decoupled twist and swing freedoms, using separate drive properties for twist and swing (both swing DOFs use the same drive properties). In fact, twist and swing is the only drive method for 1DOF or 2DOF angular joints when one or two angular DOFs are locked.

Instead of using the default twist and swing drive, 3DOF angular joints (i.e., no angular DOFs locked) with orientation drive can utilize the SLERP option that interpolates between orientations via a SLERP/shortest arc rotation.

1DOF Angular Drive

A joint with a single angular DOF, i.e. a hinge or rotary joint, is similar to the equivalent linear 1DOF joint. The obvious coordinate in this case is the joint angle but here it must be input in quaternion form. The velocity input is the single angular velocity component for the selected DOF.

It is recommended that revolute joints are implemented as twist hinges because the d6joint is optimized for this case and provides slightly more flexible limit functionality. In other words, the user can specify a low and high angle separately, similar to the dedicated revolute joint.

```
d6Desc.twistMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.swing1Motion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.swing2Motion = NX_D6JOINT_MOTION_LOCKED;

d6Desc.twistDrive.driveType=NX_D6JOINT_DRIVE_POSITION;
d6Desc.twistDrive.forceLimit=0.0f;
d6Desc.twistDrive.spring=250.0f;
d6Desc.twistDrive.damping=0.01f;

...

joint->setDriveOrientation(NxQuat(angle,NxVec3(1.0f,0.0f,0.0f)); //Angle in degrees.
```

2DOF Angular Drives

There are two types of joints with two angular DOFs: the zero-twist joint and the zero-swing joint.

zero-twist joint drive (aka Iso-Universal)

The two swing DOFs remain to be actuated.

The swing orientation is extracted from the input quaternion by decomposing it into twist and swing as shown at the end of this document, or it can be computed and input directly as shown here:

```
Qswing = Qyz = (Cyz, 0, Sy, Sz)
```

Velocity inputs, if required, can be derived using the cross product to find the axis of rotation from the current twist-axis swing direction to its desired swing direction, as demonstrated below:

```
d6Desc.twistMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.swing1Motion = NX_D6JOINT_MOTION_FREE;
d6Desc.swing2Motion = NX_D6JOINT_MOTION_FREE;

d6Desc.swingDrive.driveType=NX_D6JOINT_DRIVE_POSITION;
d6Desc.swingDrive.forceLimit=0.0f;
d6Desc.swingDrive.spring=250.0f;
d6Desc.swingDrive.damping=0.01f;

...

joint->setDriveOrientation(targetOrient);
```

zero-swing joint drive (aka Universal)

The 2DOFs are a pair of sequential rotations about orthogonal axes:

- The zero-swing1 joint DOFs are rotations about the child x-axis and the parent z-axis.
- The zero-swing2 joint DOFs are rotations about the child x-axis and the parent y-axis.

The joint constraint keeps the two rotation axes orthogonal.

The input quaternion is computed by composing the two rotations (e.g., for the zero-swing2 joint):

```
QyQx = (Cy, 0, Sy, 0)(Cx, Sx, 0, 0) = (CyCx, CySx, SyCx, -SySx)
```

The angular velocity input vector is found by adding two rotation-axis-aligned angular velocity vectors as shown below:

```
d6Desc.twistMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.swing1Motion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.swing2Motion = NX_D6JOINT_MOTION_FREE;

d6Desc.twistDrive.driveType=NX_D6JOINT_DRIVE_POSITION;
d6Desc.twistDrive.forceLimit=0.0f;
d6Desc.twistDrive.spring=250.0f;
d6Desc.twistDrive.damping=0.01f;

d6Desc.swingDrive.driveType=NX_D6JOINT_DRIVE_POSITION;
d6Desc.swingDrive.forceLimit=0.0f;
d6Desc.swingDrive.spring=250.0f;
d6Desc.swingDrive.damping=0.01f;

...

joint->setDriveOrientation(targetOrient);
```

3DOF Angular Drives

The non-integrable ('nonholonomic') behavior manifests itself in joints with 3DOF (i.e., none of the angular DOFs are locked).

Pure angular velocity drive is useless for actuating the orientation of a body in the full 3DOFs as the orientation will quickly drift away from what might be expected.

For an orientation drive, there is a choice between the default twist and swing drive or the optional SLERP drive.

twist and swing drive

Internally, the angular drive defaults to using twist and swing coordinates. This is the preferred choice for all joints within a skeletal character because it gives the expected interpolation behavior, decoupling twist and swing freedoms. However, many animation systems deal with quaternions and perform SLERPs or LERPs between the quaternions representing individual key frames. In these systems, the SLERP drive may be the preferred system as it most closely mimics the interpolation used in the key framing system.

Another case where the twist and swing drive may prove useful is with AI controlled objects, such as gun turrents, where the natural parameterization of the joint is a twist and a swing.

The twist and swing drive acts on the absolute coordinates between parent frame and child frame, attempting to bring these coordinates to the desired values between parent frame and drive frame.

Because it acts on absolute coordinates, the child frame should not be driven near the twist and swing singularity at a full 180 degree swing.

SLERP drive

The d6joint angular drive has a SLERP option to replace the default drive. The SLERP option decouples twist and swing coordinates with an isotropic drive of all three angular DOFs. This uses rotation vector coordinates in which a rotation vector lies in the single shortest rotation axis so the 3D rotation isn't being decomposed into a sequence of rotations as with twist and swing.

The SLERP drive acts locally between the drive frame and the child frame. The parent frame is not directly involved in the drive behavior; it only acts as a root from which to specify the drive frame offset.

Because it acts locally, the SLERP option is useful when the child object can rotate to any arbitrary orientation relative to the parent without limit.

There is a singularity that could cause the child to get stuck if its frame is 180 degrees away from the drive frame, but this is unlikely to occur (and will correct itself with any small disturbance).

```
d6Desc.flags|=NX_D6JOINT_SLERP_DRIVE;

d6Desc.slerpDrive.driveType=NX_D6JOINT_DRIVE_POSITION;
d6Desc.slerpDrive.forceLimit=0.0f;
d6Desc.slerpDrive.spring=250.0f;
d6Desc.slerpDrive.damping=0.01f;

...

joint->setDriveOrientation(animTarget);
```

API Reference

- NxJoint
- NxJointDesc
- NxD6Joint
- NxD6JointDesc

Projection

The d6joint provides the ability to correct large joint errors by projecting the joint back to valid configuration. This is similar to the projection functionality provided by other joints such as the spherical joint and revolute joint.

Projection is provided for joint errors on locked DOFs and limit violations on limited DOFs with a couple of exceptions.

The d6joint provides projection for the following cases:

- Free linear motion
- Limited linear motion
- Free and Limited twist motion
- Both swing axes locked
- Swing 1 locked and Swing 2 limited
- Swing 2 locked and Swing 1 limited
- Swing 1 and Swing 2 locked

The d6joint does NOT provide projection for the instances below:

- Swing 1 limited and Swing 2 free
- Swing 2 limited and Swing 1 free

By default, joint projection is disabled. To enable joint projection, it is necessary to set the members of NxD6JointDesc appropriately as demonstrated below:

```
d6Desc.projectionMode = NX_JPM_POINT_MINDIST;
d6Desc.projectionDistance = 0.1f;
d6Desc.projectionAngle = 0.0872f;
```

NOTE: In some cases, projection may negatively influence the behavior of a system, depending on how the system is tuned. This is particularly true for small projection distances when the solver fights with the way the joint is projected. Obtaining the best configuration is generally a matter of experimentation.

API Reference

- NxJoint
- NxJointDesc
- NxD6Joint
- NxD6JointDesc

Twist and Swing

Twist and swing coordinates for 3D orientation are based on its decomposition into a two orthogonal rotation sequence. They sit between Euler angle decompositions into a three orthogonal rotation sequence and direct representations of the single shortest rotation axis/angle.

In the twist and swing decomposition, an orientation is achieved by a sequence of two rotations:

(1) a twist rotation about a selected axis and (2) a swing rotation about an axis in its perpendicular plane.

Hence, twist describes the DOF spin about the selected twist axis, while swing describes the direction in space of the twist axis. With two bodies involved, the twist and swing decomposition is based on a selected twist axis in both the parent and child; the twist axis in the parent acts as the central reference direction relative to which the swing direction of the child twist axis is measured.

Globe Visualization

Imagine the twist axis as a radius tracing out points on the globe as it swings and the central reference swing direction positioned towards the North Pole. A swing rotation plots a point from the North Pole to the target point by following the shortest path on the sphere - a line of longitude in this case.

The swing rotation axis lies in the plane of the equator 90 degrees east of the swing longitude. NOTE: The direction is computed from the cross product of the parent and child twist axes.

Now set a zero reference for twist as a given heading at the North Pole. This twist reference is transferred to other target points on the globe by maintaining the heading along the shortest path to the target point. This is an isotropic definition of twist in that it is independent of the swing direction to the target point.

The Singularity at Full-Swing

The South Pole is a problem case because there is no unique single shortest path from the North Pole; all lines of longitude meet at both poles, therefore a 180 degree swing rotation around any arbitrary equatorial axis takes a point from the North to the South Pole. Twist cannot be defined and swing is redundant.

Twist Parameter

Twist can be specified and stored as an angle; however, performing the rotation requires trig functions to be evaluated. Therefore, it is more efficient to store a pre-computed trig function, which will be demonstrated later in this section.

Swing Parameters as Rotation Vector Components

Swing is parameterized by the two Cartesian coordinates of its rotation vector in the equator plane. The swing direction is inherent in the direction of the vector, i.e., the relative values of its two coordinates. The swing angle is encoded in the magnitude of the vector, i.e., its length as given by the norm of its two coordinates. The options of suitable trig functions for mapping angle to magnitude is covered below.

Note that swing is NOT parameterized by swing-angle and swing-direction, or latitude and longitude angles as a picture of the globe might seem to suggest; these are the natural coordinates for a pair of sequential Euler angle rotations rather than two parameters of a single rotation. They are singular at the origin; namely, the North Pole, where swing-angle is zero and swing-direction is redundant.

Swing Parameters as Projected Components

The two swing parameters can also be seen as components of the twist axis projected flat onto the equatorial plane. The equivalent rotation vector lays 90 degrees east of the projected axis. The projection is more general than just a shadow; it can be any suitable function for mapping angle to magnitude.

Choice of Projection for Mapping Magnitude

The angle itself is an obvious choice for the magnitude function; however, to execute the rotation requires costly sine and cosine evaluations. Below are four trig functions based on spherical projections in order of increasing suitability for numerical work.

sin-angle

The cross product of the parent and child twist axes is a vector in the direction of the swing rotation axis. Its magnitude is the sine of the swing angle. This corresponds to parallel projection form, a light source far above the North Pole. However, note that the shadow of the twist axis on the equatorial plane does not distinguish a North hemisphere point from its reflection through the equator in the South hemisphere.

The sine function is not a suitable magnitude-mapping function because it only works for angles up to 90 degrees.

tan-half-angle

This choice is of more theoretical than practical use.

A light source located at the South Pole maps all North hemisphere points to inside the equator and South hemisphere points to outside the equator; however, as the South Pole is approached, the image goes to infinity. This is the stereographic projection. Its magnitude is the tangent of half the swing angle. This projection is the unique conformal projection between the sphere and the plane in that it preserves all angles. Circles on the sphere generally map to circles on the plane except for great circles which map to straight lines.

Instead of the familiar 2-sphere, i.e., a 2D spherical surface embedded in 3D space, imagine the 3-sphere, the space inhabited by unit quaternions, as a 3D spherical surface embedded in 4D space. In fact, only half the 3-sphere is needed because diametrically opposite points represent the same orientation. The tan-half-angle projection is the central projection from the hemi-3-sphere onto the tangent (hyper) plane at its pole! In the central projection, the light source is placed at the center of the sphere.

sin-half-angle

This choice corresponds to the magnitude of the vector part of the quaternion which makes it convenient for computations. It would seem that accuracy could be lost as the angle approaches 180 degrees, because the gradient approaches zero, but this can be overcome by generating the scalar part of the quaternion, i.e., cos-half-angle, a computation that requires a square root.

This is the parallel projection from the hemi-3-sphere of quaternions into 3D space.

tan-quarter-angle

This corresponds to the stereographic projection from the quaternion sphere into 3D space. It is the unique conformal projection from the space of rotations into Euclidean space.

It is easy to compute the quaternion (cos-half-angle and sin-half-angle) from tan-quarter-angle.

Twist and Swing is not Axis and Angle

Do not confuse swing and twist with axis-and-angle. Both describe the direction of an axis and an angle of rotation about that axis. The difference is in the definition of the axis.

The axis-and-angle axis is the axis of the single shortest rotation. It is not fixed in the world or the body; its arbitrary direction in space is decided by the difference between reference and required orientations. The angle is the magnitude of the single shortest rotation between the orientations.

The twist and swing axis is a physical axis fixed in the body. Its direction is specified by a swing rotation about an axis in the perpendicular plane. The twist angle specifies a second, orthogonal, decoupled rotation about the axis.

Twist and Swing Decomposition in Quaternion Components

This section shows how the twist and swing decomposition is done with quaternion mathematics.

The quaternion representing the general 3D orientation is written below,

Qxyz = (Qw, Qx, Qy, Qz)

with 'w' indexing the scalar and 'x','y','z' indexing the components of the vector.

If the x-axis is the special axis in defining twist and swing, then the decomposition is computed in quaternion multiplication, Qxyz=Qswing Qtwist, with the following:

$$\boldsymbol{q}_{\text{twist}} = \left(\frac{q_w}{\sqrt{q_w^2+q_x^2}}, \frac{q_x}{\sqrt{q_w^2+q_x^2}} \quad 0 \quad 0 \right) \quad \boldsymbol{q}_{\text{swing}} = \left(\sqrt{q_w^2+q_x^2}, 0 \quad \frac{q_w q_y - q_x q_z}{\sqrt{q_w^2+q_x^2}} \quad \frac{q_w q_z +$$

The proof begins by writing the quaternion representing the twist rotation around the x-axis as,

Qtwist = Qx = (Cx, Sx, 0, 0)

and the quaternion representing the swing rotation around an axis in the yz-plane with the following:

Qswing = Qyz = (Cyz, 0, Sy, Sz)

Below, 'c' represents the scalar and 's' represents the vector components in relation to the cosine and sine of the half angles:

$$c_x = \text{cos-half-twist-angle} \quad s_x = \text{sin-half-twist-angle}$$

$$c_{yz} = \text{cos-half-swing-angle} \quad \sqrt{s_y^2 + s_z^2} = \text{sin-half-swing-angle}$$

Remember that generally $q_w^2 + q_x^2 + q_y^2 + q_z^2 = 1$ so we

have $c_x^2 + s_x^2 = 1$ and $c_{yz}^2 + s_y^2 + s_z^2 = 1$

Using quaternion multiplication, the rotations compose right-to-left just like rotation matrices, shown below:

$$\boldsymbol{q}_{xyz} = \boldsymbol{q}_{yz}\boldsymbol{q}_x = (c_{yz}, 0\ s_y\ s_z)(c_x, s_x\ 0\ 0)$$

The rotations are treated as active rotations around global fixed axes rather than the current local axes. Expanding the quaternion multiplication looks like this:

$$(q_w, q_x\ q_y\ q_z) = ((c_x c_{yz}), (s_x c_{yz})\ (c_x s_y + s_x s_z)\ (c_x s_z - s_x s_y))$$

Extracting the twist quaternion from this is straightforward once you see that Qw^2 + Qx^2 = Cyz^2

$$\boldsymbol{q}_{twist} = \boldsymbol{q}_x = (c_x, s_x\ 0\ 0) = (q_w, q_x\ 0\ 0) / \sqrt{q_w^2 + q_x^2}$$

Then the swing quaternion is extracted by inverting (conjugating) the twist quaternion and multiplying

$$\boldsymbol{q}_{swing} = \boldsymbol{q}_{xyz}\boldsymbol{q}_{twist}^* = (q_w, q_x\ q_y\ q_z)(q_w, -q_x\ 0\ 0) / \sqrt{q_w^2 + q_x^2}$$

Expanding the quaternion multiplication gives

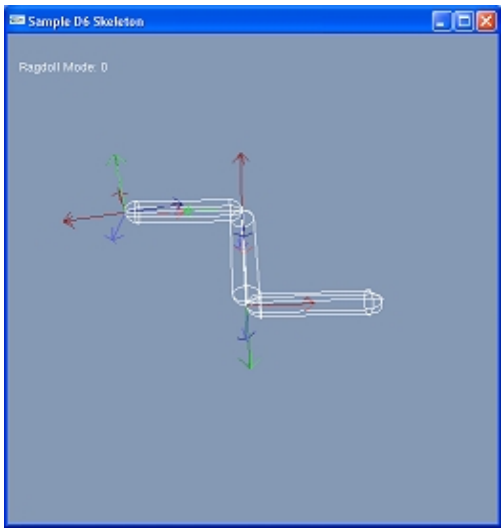
$$(c_{yz}, 0\ s_y\ s_z) = ((q_w^2 + q_x^2), 0\ (q_w q_y - q_x q_z)\ (q_w q_z + q_x q_y)) / \sqrt{q_w^2 + q_x^2}$$

which completes the proof.

API Reference

- NxJoint
- NxJointDesc
- NxD6Joint
- NxD6JointDesc

Skeletal Animation and Rag Dolls



One of the primary aims when creating the d6joint was to integrate the flexibility and features necessary for sophisticated character animation and rag doll effects. To this end, the d6joint provides the following features which make it particularly suitable for character animation:

- Sophisticated angular and linear drive model
- Capability to be driven using combined angular position and velocity targets - important for achieving smooth animation
- Choice between twist and swing or SLERP angular drive. A SLERP drive moves the joint along the shortest spherical arc to a target orientation which reflects the behavior of many existing skeletal animation systems.
- Flexible joint limits. Being able to specify realistic joint limits for skeletal joints is important for realistic rag doll effects. The range of motion, for example, of a shoulder, is quite complicated; specifying this range of motion using simple point/plane limits is not generally possible.

A Simple Skeletal Animation System

A skeletal animation system generally provides a base pose for a skeleton along with a set of animation keys. In an average system, the keys can take the form of rotational keys or positional keys. The following example considers only rotational keys since they are the most common and the most difficult.

For each frame of animation, a blended rotation key is computed and applied as a relative rotation to the base pose.

Creating Bone Shapes

The first step is to create actors to represent each bone; a convenient primitive for this is the capsule shape. When specifying bone transforms, the convention that the bone lies along the x-axis in its local space is used, which matches the d6joint axis convention. To orient the capsule shapes correctly, compute and set a shape

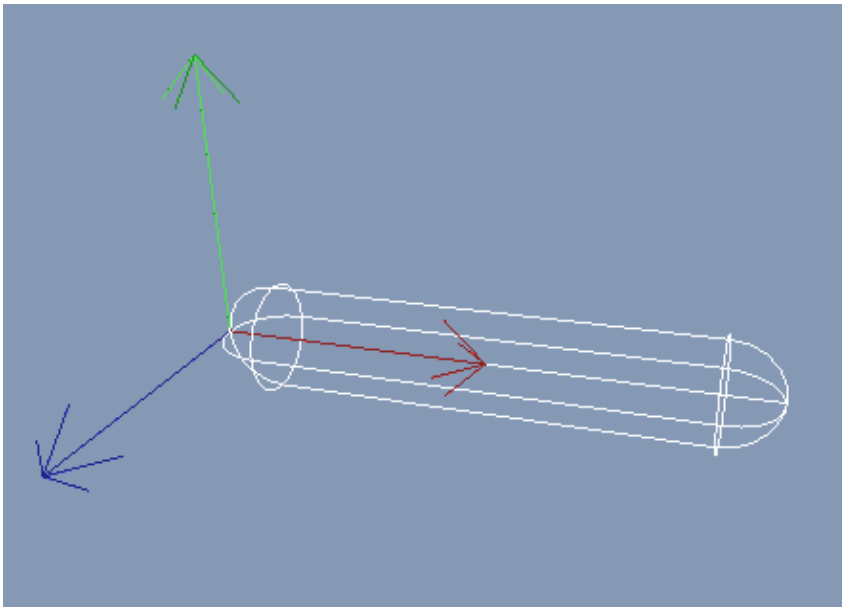
rotation which orients the capsule to point along the x-axis (the SDK convention is that a capsule points along the y-axis). Translate the capsule so that its end point touches the actors origin.

```
NxVec3 parentPt=parent->basePose*NxVec3(0.0f,0.0f,0.0f);
NxVec3 ourPt=basePose*NxVec3(0.0f,0.0f,0.0f);

NxCapsuleShapeDesc capsuleDesc;

capsuleDesc.radius=0.1f;
capsuleDesc.height=boneLength-capsuleDesc.radius*2.0f;

NxFindRotationMatrix(NxVec3(0.0f,1.0f,0.0f),NxVec3(1.0,0.0f,0.0f),capsuleDesc.localPose.M);

capsuleDesc.localPose.t=NxVec3(boneLength/2.0f,0.0f,0.0f);

actorDesc.shapes.pushBack(&capsuleDesc);
```

NOTE: Be aware that changing a shapes transform in this way will alter the center of mass and inertia tensor of the actor.

Create the Joints

It is necessary to create the d6joints for each bone orientated appropriately in the base pose of the skeleton. For this exercise, use the convention that the twist axis (x-axis) points along the child bone and that the joint frames of the parent and child body are aligned in the base pose.



The red axis in the above screen shot is the twist axis, and the bone is represented as a capsule. The green and blue axes are the swing axes.

In the example code below, the joint is attached to the parent bone. Then the joint's axis is oriented to point along the x-axis in the child frame. This requires transforming the localAxis[0] member into the frame of the parent actor since localAxis (and localNormal) are relative to their respective actors.

localPose is the relative transform between child and parent, while basePose is the transform to the global frame.

```
if(parent!=NULL)
    d6Desc.actor[0]=parent->actor;
else
    d6Desc.actor[0]=NULL;

d6Desc.actor[1]=actor;

d6Desc.localAxis[0]=localPose.M*NxVec3(1.0f,0.0f,0.0f);
d6Desc.localAxis[1]=NxVec3(1.0f,0.0f,0.0f);

d6Desc.localNormal[0]=localPose.M*NxVec3(0.0f,1.0f,0.0f);
d6Desc.localNormal[1]=NxVec3(0.0f,1.0f,0.0f);

d6Desc.setGlobalAnchor(basePose*NxVec3(0.0f,0.0f,0.0f));
```

Since only applying angular animations to the bones, setup the d6Joint as a spherical joint by locking all the linear DOFs and leaving the angular ones free or limited.

```
d6Desc.xMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.yMotion = NX_D6JOINT_MOTION_LOCKED;
d6Desc.zMotion = NX_D6JOINT_MOTION_LOCKED;

d6Desc.twistMotion = NX_D6JOINT_MOTION_FREE;
d6Desc.swing1Motion = NX_D6JOINT_MOTION_FREE;
d6Desc.swing2Motion = NX_D6JOINT_MOTION_FREE;
```

Disabling Contact Generation between Bones

When animating a bone, we do not want contacts to be generated between shapes which are attached to the same parent. If this happens, the contact points will interfere with the motion of the bones. To ensure this doesn't happen, call NxScene::setActorPairFlags() specifying each pair of child bones in turn.

```
foreach(np in boneChildren)
    foreach(nq in boneChildren)
    {
    if(np==nq)
        continue;

    gScene->setActorPairFlags(np->actor,nq->actor,NX_IGNORE_PAIR);
    }
```

Angular Drive

Once the above setup work is complete, it is time to drive the skeleton so that it follows the key framed animation. As a first pass, drive the angular orientation/position to match just the position of the key frames. In the next section, we will drive the velocity for smoother animation.

To set the angular drive when creating the joint, specify the appropriate angular drive parameters. Use a SLERP drive (instead of a twist and swing) to mimic the behavior of the animation system.

The spring parameter specifies the amount of force needed to keep the physical representation in sync with the animation data. Setting a higher value will cause the physical bodies to match the animation more closely. The damping parameter is used to smooth out oscillations of the actors when correcting their position to match the key framed animation.

```
d6Desc.flags|=NX_D6JOINT_SLERP_DRIVE;

d6Desc.slerpDrive.driveType=NX_D6JOINT_DRIVE_POSITION;

d6Desc.slerpDrive.forceLimit=0.0f; //Not used with the current drive model.
d6Desc.slerpDrive.spring=250.0f;
d6Desc.slerpDrive.damping=0.01f;
```

Next, update the target position for each bone as they are animated. This is simply a matter of calling NxD6Joint::setDriveOrientation() for each animation frame with a new relative orientation to the joint's parent frame.

```
joint->setDriveOrientation(parent->animTarget);
```

NOTE: For this example, assume that joint animation data is the same for each child bone; therefore, store the data in the parent.

Angular Velocity Drive

In some cases, applying an additional velocity drive to the joint can improve the joint's ability to follow the animation track, i.e., the velocity drive provides an additional cue to the SDK about where the joint will be in the next time step, in addition to providing the position of the joint in the current time step.

Enabling the velocity drive is simply a matter of specifying the appropriate flag on joint creation:

```
d6Desc.flags|=NX_D6JOINT_SLERP_DRIVE;

d6Desc.slerpDrive.driveType=NX_D6JOINT_DRIVE_POSITION | NX_D6JOINT_DRIVE_VELOCITY;
d6Desc.slerpDrive.forceLimit=0.0f;
d6Desc.slerpDrive.spring=250.0f;
d6Desc.slerpDrive.damping=0.01f;
```

It is necessary to update the drive's angular velocity target and the positional target at the same time:

```
if((parent!=NULL)&&(joint!=NULL))
    {
    joint->setDriveOrientation(parent->animTarget);
    joint->setDriveAngularVelocity(parent->animVelTarget);
    }
```

The velocity target can be approximated from the animation track in a number of ways. Below is a simple approximation using quaternion math:

```
NxVec3 SkeletonNode::ComputeAnimDeriv(float t)
    {
    const float h=0.2f;
    const float h2=2.0f*h;

    NxQuat rot0=ComputeAnimRot(t-h);
    NxQuat rot1=ComputeAnimRot(t+h);

    rot0.invert();

    NxQuat deltaRot=rot1*rot0;

    if(deltaRot.w<0.0)//Quaternions should be in the positive hemisphere.
        {
        deltaRot.x=-deltaRot.x;
        deltaRot.y=-deltaRot.y;
```

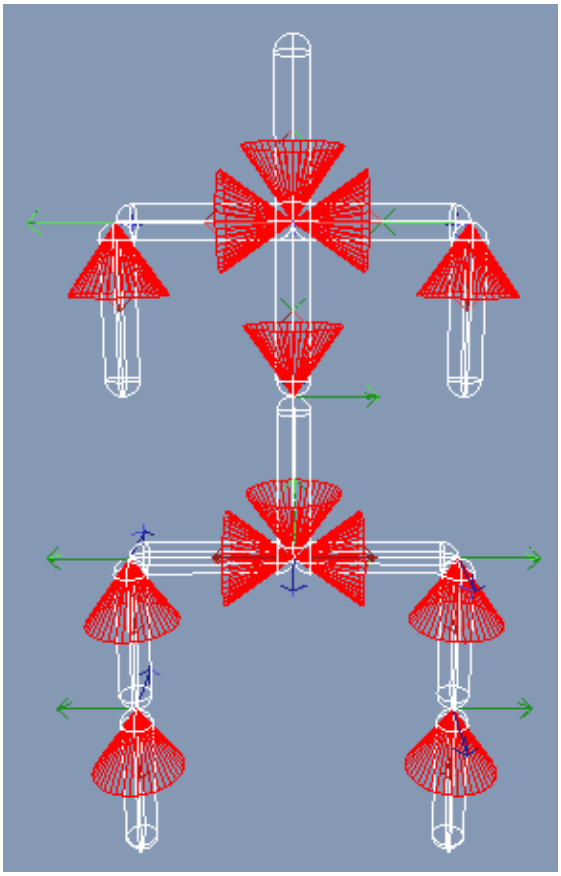
```
            deltaRot.z=-deltaRot.z;
            deltaRot.w=-deltaRot.w;
            }

        return NxVec3(deltaRot.x/h2,deltaRot.y/h2,deltaRot.z/h2);
        }
```

Turning the Skeleton into a Rag doll

At some point, for example when a character dies, it becomes necessary to turn off key frame animation of the skeleton and resort to pure physical simulation. This is simply a matter of disabling the SLERP drive for each of the d6joints.

```
if(isKeyframed)
{
    d6Desc.slerpDrive.driveType=NX_D6JOINT_DRIVE_POSITION;
    d6Desc.slerpDrive.forceLimit=0.0f;
    d6Desc.slerpDrive.spring=250.0f;
    d6Desc.slerpDrive.damping=0.01f;
}
else
{
    d6Desc.slerpDrive.driveType=0;
    d6Desc.slerpDrive.forceLimit=0.0f;
    d6Desc.slerpDrive.spring=0.0f;
    d6Desc.slerpDrive.damping=0.0f;
}

joint->loadFromDesc(d6Desc);
```

Joint Limits



As soon as the key framed animation is turned off, notice that the skeleton folds up in an unrealistic manner. A real skeletal joint has anatomically imposed limits on its motion. To make the rag doll behave in a more realistic manner, it is necessary to approximate these limits.

The first type of limit to apply is a twist limit, which constrains how much a joint can rotate about its twist axis (bone axis). To do so, specify a high and low angular value for the limit. In anatomical joints, the amount of twist tends to be quite small, although it varies between joints. For example, a shoulder joint can twist more than an elbow joint.

```
if((parent->twistLowLimit!=0.0f)||(parent->twistHighLimit!=0.0f))
{
    d6Desc.twistMotion = NX_D6JOINT_MOTION_LIMITED;

    d6Desc.twistLimit.low.value=NxMath::degToRad(parent->twistLowLimit);
    d6Desc.twistLimit.low.damping=0.0f;
    d6Desc.twistLimit.low.restitution=0.0f;
    d6Desc.twistLimit.low.spring=0.0f;


    d6Desc.twistLimit.high.value=NxMath::degToRad(parent->twistHighLimit);
    d6Desc.twistLimit.high.damping=0.0f;
    d6Desc.twistLimit.high.restitution=0.0f;
    d6Desc.twistLimit.high.spring=0.0f;
}
```

The swing limits constrain how much the joint can rotate around its other two axes. They are slightly less flexible than the twist joint, in that only a single angle can be specified; the limit is assumed to be symmetrical around the base pose. To achieve realistic limits, it is necessary to set up the base pose so that each joint is at the center of its arc (or alternatively manipulate the joint frames during setup). If both swing1 and swing2 limits are specified, the joint has a range of motion which can be represented by an elliptical cone.

```
if(parent->swing1Limit!=0.0f)
{
    d6Desc.swing1Motion = NX_D6JOINT_MOTION_LIMITED;
    d6Desc.swing1Limit.value=NxMath::degToRad(parent->swing1Limit);
    d6Desc.swing1Limit.damping=0.0f;
    d6Desc.swing1Limit.restitution=0.0f;
}

if(parent->swing2Limit!=0.0f)
{
    d6Desc.swing2Motion = NX_D6JOINT_MOTION_LIMITED;
    d6Desc.swing2Limit.value=NxMath::degToRad(parent->swing2Limit);
    d6Desc.swing2Limit.damping=0.0f;
    d6Desc.swing2Limit.restitution=0.0f;
}
```

API Reference

- NxJoint
- NxJointDesc
- NxD6Joint
- NxD6JointDesc

Spring and Damper Element

While technically not a joint, this is a good place to explain the general spring and damper element implemented by the NxSpringAndDamperEffector class because it exerts a force between two bodies, proportional to the relative positions and/or the relative bodies' velocities.

Create the element like any other dynamics object:

```
NxSpringAndDamperEffectorDesc sdd;

NxSpringAndDamperEffector * effector = scene->createSpringAndDamperEffector(sdd);
```

The descriptor does not yet have any interesting settings. Next, tell it between which two actors it is to act, and where exactly on the actors (but in global space) the two ends of the effector are attached:

```
effector->setBodies(actor1,positon1, actor2, position2);
```

When the above call is made, the effector reads the bodies' poses so that it can remember what the relative position is when the spring is relaxed. At this initial position, no spring force is applied. When the spring is stretched or compressed by moving the bodies relative to each other, a force will be exerted. The magnitude of this force is controlled by the NxSpringAndDamperEffector::setLinearSpring() method. See the API reference for explanation about its parameters.

Similarly, when the two bodies begin moving relative to each other, they will have a relative velocity. The damping force, which is always trying to stop the bodies, is a function of this velocity. The exact magnitude of the damping force is controlled by the NxSpringAndDamperEffector::setLinearDamper() method.

This effector element is not as numerically stable as the NxSpringDesc spring elements built into certain types of joints.

NOTE: The spring and damper element is deprecated, therefore, the distance joint should be used in its place for the following reasons:

- Spring and Damper elements are not as numerically stable as genuine joints.
- Joints have proper sleep behavior - bodies fall asleep and wake up at appropriate times with regard to forces being transferred across the joint's spring.

API Reference

- NxSpringAndDamperEffector
- NxSpringAndDamperEffectorDesc

Materials

When two actors collide, the collision behavior that results depends on the material properties of the actors' surfaces. For example, the surface properties determine if the actors will or will not bounce, or if they will slide or stick. Currently, the only special feature supported by materials is anisotropic friction, but other effects such as moving surfaces and more types of friction are slotted for future release.

Surface properties are determined by the members of the NxMaterial class. The SDK maintains a global list of materials, which are referenced using the 16 bit NxMaterialIndex indices. To create a material, its properties must be specified. Below is an example for setting the material properties:

```
NxMaterialDesc materialDesc;
materialDesc.restitution = 0.7f;
materialDesc.staticFriction = 0.5f;
materialDesc.dynamicFriction = 0.5f;
```

Once the material descriptor has been set up, a material object can be created:

```
NxMaterial *newMaterial=gScene->createMaterial(materialDesc);
```

When assigning a material to a shape, it is necessary to retrieve the material's index with the following code:

```
NxActorDesc actorDesc;
NxBoxShapeDesc boxDesc;

boxDesc.dimensions.set(4,4,5);
boxDesc.materialIndex= newMaterial->getMaterialIndex();
actorDesc.shapes.pushBack(&boxDesc);
...

NxActor * myActor = gScene->createActor(actorDesc);
```

It is also possible to get and set the material index of a shape during the simulation using the NxShape::setMaterial() and NxShape::getMaterial() methods.

Combine modes

Both friction and restitution are really properties of pairs of materials rather than of single materials. However, providing values for every possible combination of materials that may touch is probably not feasible; instead, an effective value for the property in question is generated whenever two objects of different materials interact.

The method used to generate the combined value can be set via the NxMaterialDesc::frictionCombineMode and NxMaterialDesc::restitutionCombineMode descriptor members. Please refer to the API description for more information.

Note: Restitution values near to 1 may cause increasing energy in bounces, depending on the restitution combine mode of the materials. It is therefore recommended that you take care to avoid combined restitution values near 1, at least in cases where the participants are supposed to be able to come to rest touching each other.

Default Material

You can change the properties of the default material by querying for material index 0. The default material is used when no other material is specified for a shape.

```
NxMaterial* defaultMaterial = gScene->getMaterialFromIndex(0);

        defaultMaterial->setRestitution(0.5);
        defaultMaterial->setStaticFriction(0.5);
        defaultMaterial->setDynamicFriction(0.5)
```

Special Contact Behaviors

Contact points are typically used to model the local interaction of two hard surfaces. Material properties can be used to define the local characteristics of the surfaces. The first, most basic possibility to customize the behavior beyond just hard surfaces is the coefficient of restitution, which permits a very simple simulation of soft elastic bodies. Sometimes it is practical to put contacts to even more creative use (e.g., Anisotripic Friction to simulate skis).

Samples

Sample Materials

API Reference

- NxMaterial
- NxMaterialDesc

Anisotropic Friction

Some materials are more slippery in one direction than in another, and sometimes more complex interactions must be modeled, such as the way wheels or skis interact with the ground. For this case it is possible to define a direction of anisotropy in the material class. This vector, relative to the shape's own coordinate frame, defines the primary direction of anisotropy. The component of the sliding direction projected along this vector ('U' direction) uses the regular set of friction coefficients, and sliding orthogonal to this direction ('V' direction) uses the dynamicFrictionV and staticFrictionV. If the coefficients for the two directions are set to different values, the object will have a tendency to slide along the direction with lower friction coefficients.

NOTE: Prior to version 2.1.2, the NxMaterial::dirOfAnisotropy was defined in actor space. It is now assumed to be in shape space.

```
    //Create an anisotropic material
    NxMaterialDesc material;

    //Anisotropic friction material

    material.restitution = 0.0f;
    material.staticFriction = 0.1f;
    material.dynamicFriction = 0.1f;
    material.dynamicFrictionV = 0.8f;
    material.staticFrictionV = 1.0f;
    material.dirOfAnisotropy.set(0,0,1);
    material.flags = NX_MF_ANISOTROPIC;

NxMaterial *anisoMaterial = gScene->createMaterial(material);
```

Samples

Sample Materials

API Reference

- NxMaterial
- NxMaterialDesc

Materials per Triangle

While in most cases it is sufficient to assign a single material to an entire shape, at times, a triangle mesh shape is used to model a large environment. When this occurs, it is necessary to assign different materials to certain regions of the triangle mesh by specifying a material index for each one before cooking it. (See the section on Cooking for more information).

Point the materialIndices member to an array of NxMaterialIndex when filling in the NxTriangleMeshDesc. The materialIndexStride should be set to the size of a NxMaterialIndex plus the number of bytes between entrys in the array. Typically, this is set to sizeof(NxMaterialIndex) for a densely packed array.

The number of material indices is assumed to be equal to the number of triangles in the mesh. Follow the example below:

```
unsigned int BUNNY_NBVERTICES=453;

unsigned int BUNNY_NBFACES=902;

float gBunnyVertices[BUNNY_NBVERTICES*3]={
    -0.334392f,0.133007f,0.062259f,
    -0.350189f,0.150354f,-0.147769f,
...
};

int gBunnyTriangles[BUNNY_NBFACES*3]={
    126,134,133,
    342,138,134,
...
};

/*
NEW:
Now we also have a material index array, with one entry for each triangle.
Each index must be a valid NxMaterialIndex, returned by, for example,
NxMaterial::getMaterialIndex().

For the bunny, the material indices may be arranged so that the bunny's
ears be more slippery (have lower friction) than the rest of the model.
*/

NxMaterialIndex gBunnyMaterials[BUNNY_NBFACES] = {
    1,
    3,
    ...
};

NxTriangleMeshDesc bunnyDesc;

bunnyDesc.numVertices = BUNNY_NBVERTICES;
bunnyDesc.numTriangles = BUNNY_NBFACES;
bunnyDesc.pointStrideBytes = sizeof(NxVec3);
bunnyDesc.triangleStrideBytes = 3*sizeof(NxU32);

bunnyDesc.points = gBunnyVertices;
bunnyDesc.triangles = gBunnyTriangles;
bunnyDesc.flags= 0;

//NEW: add the mesh material data:

bunnyDesc.materialIndexStride = sizeof(NxMaterialIndex);
bunnyDesc.materialIndices = gTerrainMaterials;
```

```
//Cook the triangle mesh...
```

Note: Per-triangle materials will cause the PPU to fall back to software, which will have an impact on performance.

Samples

Sample Mesh Materials

API Reference

- NxMaterial
- NxMaterialDesc
- NxTriangleMesh
- NxTriangleMeshDesc

Collision Detection

Many dynamic applications are not concerned with the forces that occur when bodies collide or touch (e.g., a rocket taking off, or a cannon ball flying in an arc due to gravity). On the other hand, most interesting applications are concerned with collision detection - for the cannon ball to bounce on the ground or an articulated body to fall down stairs, the SDK must be provided with the shapes of the bodies that can touch, and also the properties of the surfaces (e.g., how slippery they are). This collision detection functionality can range from very simple (the cannon ball is simply dependent on whether the ball is above or below ground level) to quite complex (in the case of the articulated body falling down the stairs). A body falling down the stairs is more complex because it involves many shapes intersecting. For example, the SDK may need to collide a convex against a mesh, and must resolve the system in a way to maintain the constraints within the body.

Shapes

The SDK lets you create shapes - these are objects which occupy a well-defined volume in space. The Dynamics section served as an introduction on how several shapes are attached to an actor's frame of reference.

The easiest way to create shapes is together with the actor that owns them. For example, the below code creates a mace shaped actor made up of a spherical head and a round handle:

```
NxSphereShapeDesc head;
NxCapsuleShapeDesc handle;
NxActorDesc desc;
NxBodyDesc bodyDesc;

/*Implicitly define the mace's origin to be near the lower
end of its handle, where a character would grip to hold it.*/

head.radius = 0.05f;//Mace head has radius of 5cm.
head. localPose.t.set(0,45,0); //Head's position relative to the mace's origin.

handle.radius = 0.01f; //Handle has a radius of 1 cm.
handle.height = 0.50f; //Handle is 50 cm long along its y axis.
handle.localPose.t.set(0,15,0); //Handle's position relative to the mace's origin.

/*Set the position of the mace in the world here. This would have to be the pose of a
character's hand in world space.*/

desc. globalPose = myPose.
desc. body = &bodyDesc; //Set dynamics properties here.

desc.density = 7600; //This is the density of iron in kg/m3.

//Assign the shapes to the actor:
desc.shapes.pushBack(&head);
desc.shapes.pushBack(&handle);

//Create the actor:

scene->createActor(desc);
```

The localPose set above can also be accessed and changed using the NxShape::setLocalPose(), etc., methods.

API Reference

- NxShape
- NxShapeDesc
- NxPlaneShape
- NxBoxShape
- NxSphereShape
- NxCapsuleShape
- NxConvexShape
- NxTriangleMeshShape

Collision Interactions

The PhysX SDK supports collision detection and contact generation between most shape types, however some are not beneficial to the user. Below is a table describing the collision interactions which are supported:

	NxSphereShape	NxBoxShape	NxCapsuleShape	NxPlaneShape	NxConvexShape	NxHeightfieldShape	NxWheelShape	NxTriangle
NxSphereShape	✓	✓	✓	✓	✓	✓	✓	✓
NxBoxShape	✓	✓	✓	✓	✓	✓	✓	✓
NxCapsuleShape	✓	✓	✓	✓	✓	✓	✓	✓
NxPlaneShape	✓	✓	✓	✗	✓	✗	✓	✓
NxConvexShape	✓	✓	✓	✓	✓	✓	✓	✓
NxHeightfieldShape	✓	✓	✓	✗	✓	✗	✓	✗
NxWheelShape	✓	✓	✓	✓	✓	✓	✗	✓
NxTriangleMeshShape	✓	✓	✓	✓	✓	✗	✓	✗
NxTriangleMeshShape (heightfield)	✓	✓	✓	✗	✓	✗	✓	✗
NxTriangleMeshShape (pmap)	✓	✓	✓	✗	✓	✗	✓	✗

NOTE: PMaps are not a recommended collision primitive and in future support may be dropped.

API Reference

- NxTriangleMesh
- NxConvexMesh
- NxConvexShape
- NxTriangleMeshShape
- NxSphereShape
- NxBoxShape
- NxCapsuleShape
- NxHeightfieldShape
- NxWheelShape

Broad Phase Collision Detection

The first step of collision detection is to find out which pairs of objects from all the possible pairs in a scene have a chance of touching. Because there are approximately n*n/2 potential pairs in a set of n shapes, checking all of them is too time consuming for large scenes. Instead, the SDK automatically partitions the space occupied by the shapes. This way a shape is only tested against nearby shapes.

NOTE: In previous versions, the SDK allowed selection of the broadpahse algorithm. Support for this has been dropped as the default is best.

Shape Pair Filtering

Once a pair of shapes is identified as possibly colliding, three consecutive checks are performed to see if the user is interested in the pair. Only after all three checks pass will time consuming contact determination be performed. Collision detection between shapes a and b occurs if the following is true:

```
(a->getActor()->isDynamic() || b->getActor()->isDynamic())
    && NxScene::getGroupCollisionFlag(a->getGroup(), b->getGroup())
    && (!(NxScene::getShapePairFlags(a,b) & NX_IGNORE_PAIR))
```

The first test simply means that static objects which cannot move anyway do not collide against each other. We look at the last three conditions in the following sections.

In addition to the above checks, an additional mechanism for contact filtering which uses bit masks and boolean operations can be applied. See Contact Filtering for more information.

Collisions between pairs of jointed bodies are disabled by default. To enable collisions between jointed bodies the NX_COLL_VETO_JOINTED parameter can be set to false. See SDK Parameters for more information.

Collision Groups

The first condition checks the shapes collision group membership. All shapes can be assigned to a collision group with the following call:

```
NxShape::setGroup(NxCollisionGroup)
```

The CollisionGroup is simply an integer between 0 and 31. For example, the below call assigns our shape to the 11th group:

```
myShape->setGroup(11);
```

All shapes start out in group 0, but this does not really come with any built in meaning. The SDK maintains a 32x32 symmetric Boolean matrix which identifies whether or not shapes of a particular group should be collided against shapes of another group. Initially, all group pairs are enabled. You can change the entries of the matrix with the following call:

```
NxScene::setGroupCollisionFlag(CollisionGroup g1, CollisionGroup g2, bool enable);
```

For example, the below call disables collisions between group 11 and group 0:

```
gScene->setGroupCollisionFlag(11, 0, false);
```

Collision groups are useful if you know ahead of time that there are certain kinds of actors which should not collide with certain other kinds.

Actors can also be assigned to groups. Actor groups are orthogonal to shape groups, even though they are used for similar things. They work the same way, however there are 0x7fff possible actor groups, instead of only 32 as with shapes. This allows for more flexibility. First, each actor can be assigned to a group as shown below:

```
myActor->setGroup(333);
```

By default each actor starts out in group 0. Next, set up relationships between pairs of actor groups:

```
gScene->setActorGroupPairFlags(333,334, NX_NOTIFY_ON_START_TOUCH | NX_NOTIFY_ON_END_TOUCH);
```

A combination of the following flags are permitted:

NX_NOTIFY_ON_START_TOUCH, NX_NOTIFY_ON_END_TOUCH, NX_NOTIFY_ON_TOUCH.

See the section on Triggers for an explanation of these flags.

Disabling Pairs

The second broad phase condition checks to see if a pair that is about to be collision tested has been disabled or not. Any pair of actors or shapes can be disabled with the following call:

```
NxScene::setActorPairFlags(NxActor&, NxActor&, NxU32 flags);
NxScene::setShapePairFlags(NxShape&, NxShape&, NxU32 flags);
```

The second method is useful if an actor is made up of multiple shapes, and you only want to disable collisions between some of them. If this is the case, the only flag of interest is NX_IGNORE_PAIR. For example, the below call disables collisions between the two actors passed:

```
gScene->setActorPairFlags(a1, a2, NX_IGNORE_PAIR);
```

NOTE: Other possible flags for the setActorPairFlags() call will be discussed later.

User actor filtering

It is possible to provide a custom method for actor pair filtering. This gives you total control over which actors can interact with which. To enable this feature, do the following:

- Implement the NxUserActorPairFiltering interface, including the onActorPairs callback. This callback will be called with arrays of actor pairs, for each of which you may specify whether to filter that pair out or not.
- Set the NX_AF_USER_ACTOR_PAIR_FILTERING flag for any actors you wish to involve in custom filtering.
- Whenever you change the policy used in your callback, or change the callback itself, you need to also call NxActor::resetUserActorPairFiltering() for each actor that would change its filtering state with regard to any of its current interactions (to be safe you may want to simply call it for every actor with the flag set). Otherwise the changed policy would have no effect until the pair separated and contacted once more.

Custom user actor pair filtering overrides any other actor pair filtering.

Samples

Sample Filtering

API Reference

- NxScene
- NxActor
- NxShape

Contact Filtering

In addition to the filtering mechanisms implemented using NxScene::setGroupCollisionFlag() and NxScene::setShapePairFlags() the SDK provides a more sophisticated filtering mechanism based on boolean operations and flags.

The user is able to specify a 128 bit group mask for each shape. This mask is combined with user specified constants and operators, which results in a boolean value indicating if contacts should be generated for a pair of shapes. The shape culling operates according to the following pseudocode:

```
bool shouldGenerateContacts(shape0Bits, shape1Bits)
{
    value = (shape0Bits op0 constant0) op2 (shape1Bits op1 constant1)

    return ((bool)value)==filterBool;
}
```

In other words, the bit mask associated with shape 0 is combined with constant 0 using operator 0. The bit mask associated with shape 1 is combined with constant 1 using operator 1. Finally, the two results are combined with operator 2 and compared to filterBool. If the result is true then contacts are generated for the shape pair.

Available operators (members of NxFilterOp):

- NX_FILTEROP_AND - result = A & B
- NX_FILTEROP_OR - result = A | B
- NX_FILTEROP_XOR - result = A ^ B
- NX_FILTEROP_NAND - result = ~ (A & B)
- NX_FILTEROP_NOR - result = ~ (A | B)
- NX_FILTEROP_NXOR - result = ~ (A ^ B)
- NX_FILTEROP_SWAP_AND -

```
results.bits0 = A.bits0 & B.bits2;
results.bits1 = A.bits1 & B.bits3;
results.bits2 = A.bits2 & B.bits0;
results.bits3 = A.bits3 & B.bits1;
```

To set the filter operators/constants and filterBool value, NxScene provides the following methods:

```
void NxScene::setFilterOps(NxFilterOp op0, NxFilterOp op1, NxFilterOp op2);
void NxScene::setFilterBool(bool flag);
void NxScene::setFilterConstant0(const NxGroupsMask& mask);
void NxScene::setFilterConstant1(const NxGroupsMask& mask);
```

NxGroupsMask is a class which encapsulates a 128 bit value as a set of 32 bit values:

```
class NxGroupsMask
    {
    public:

    NxU32 bits0, bits1, bits2, bits3;
    };
```

Finally, NxShape provides methods to set/get the group mask associated with the shape:

```
void NxShape::setGroupsMask(const NxGroupsMask& mask);
const NxGroupsMask NxShape::getGroupsMask() const;
```

In addition, the groupsMask can be specified in the shape descriptor on creation:

```
NxShapeDesc::groupsMask;
```

For this example, 4 sets of shapes will collide with each other as demonstrated below (* represents that contacts will be generated):

	Shapes A	Shapes B	Shapes C	Shapes D
Shapes A	*			*
Shapes B		*		*
Shapes C			*	
Shapes D	*	*		*

First assign each group a bit, called the base mask. Then read down the table ORing appropriate masks together so that only those groups which should collide share a 1 in the appropriate place. This yields the value for the group's mask.

Shapes	Base Mask	groupsMask
A	0001	A = 0001
B	0010	B = 0010
C	0100	C = 0100
D	1000	A | B | D = 1011

To set up the operators/constants, ensure that contacts between a pair of shapes are generated if their masks produce a non zero value when ANDed together:

```
gScene->setFilterOps(NX_FILTEROP_OR, NX_FILTEROP_OR, NX_FILTEROP_AND);

void NxScene::setFilterBool(true);

NxGroupsMask zeroMask;
zeroMask.bits0=zeroMask.bits1=zeroMask.bits2=zeroMask.bits3=0;

void NxScene::setFilterConstant0(zeroMask);
void NxScene::setFilterConstant1(zeroMask);
```

Setting the filter constants to zero and operators 0 and 1 to NX_FILTEROP_OR effectively makes them no ops. The final NX_FILTEROP_AND causes the groupsMasks to be ANDed together and the result compared to the filterBool value producing the intended result.

Note 1: Since it is undefined which of any two shapes checked for collision is assigned "G0" and which "G1" in the above formula, you should refrain from using filters that give different results depending on this assignment.

Samples

Sample Filtering

API Reference

- NxScene
- NxActor
- NxShape

Triggers

A trigger is a shape that permits other shapes to pass through it. Each shape passing through it can create an event for the user when it enters, leaves, or simply stays inside the shape. Triggers can be used to implement behaviors such as automatic doors opening when an object approaches.

Triggers can be created from any type of shape except wheel shapes. Triangle mesh triggers count as hollow surfaces for collision detection, not volumes. Note that convex vs. triangle mesh and mesh vs. mesh triggers are not implemented. Triggers are created and attached to actors just like any other shape. The only difference is that they need to be marked as triggers in the shape descriptor:

```
//This trigger is a cube.

NxBoxShapeDesc boxDesc;

boxDesc.dimensions = NxVec3(10.0f, 10.0f, 10.0f);
boxDesc.shapeFlags |= NX_TRIGGER_ENABLE;

NxActorDesc actorDesc;
actorDesc.shapes.pushBack(&boxDesc);

NxActor * triggerActor = gScene->createActor(actorDesc);
```

Above, NX_TRIGGER_ENABLE specifies that all possible trigger events should be sent to the user. Alternatively, it is possible to select a subset of the flags NX_TRIGGER_ON_ENTER, NX_TRIGGER_ON_LEAVE, and NX_TRIGGER_ON_STAY.

To receive trigger events, the SDK must be told where to send them. Trigger events are passed to a user defined object of type NxUserTriggerReport. Below is an example implementation, plus the code used to assign it to the trigger scene:

```
class TriggerCallback : public NxUserTriggerReport
{

    void onTrigger(NxShape& triggerShape, NxShape& otherShape, NxTriggerFlag status)
    {
        if(status & NX_TRIGGER_ON_ENTER)
        {
            //A body entered the trigger area for the first time
            gNbTouchedBodies++;
        }

        if(status & NX_TRIGGER_ON_LEAVE)
        {
            //A body left the trigger area
            gNbTouchedBodies--;
        }

        //Should not go negative
        NX_ASSERT(gNbTouchedBodies>=0);
    }

}myTriggerCallback;

gScene->setUserTriggerReport(&myTriggerCallback);
```

Caveats

- Trigger shapes will not contribute to a shape's mass when computing the mass from the shapes. For this reason, a mass and inertia must be explicitly supplied when a dynamic actor is created from trigger shapes alone.
- Triggers are excluded from raycasts and sweep tests.
- The SDK Parameter NX_TRIGGER_TRIGGER_CALLBACK controls if trigger events are generated when a trigger touches another trigger.
- Static triggers do not trigger when touched by a static shape.
- All other combinations, e.g., kinematic vs. static, kinematic vs. dynamic, dynamic trigger vs. kinematic, etc., do trigger.
- The state of the SDK should not be modified from within the onTrigger() routine. In particular, objects should not be created or destroyed. If it is necessary to modify a state, then the updates should be stored to a buffer and performed after the simulation step.
- Convex vs. triangle mesh and mesh vs. mesh triggers are not implemented.
- Triggers do not take part in Continuous Collision Detection (CCD)

Threading

The NxUserTriggerReport class is only called in the context of the user thread. It is not necessary to make it thread safe.

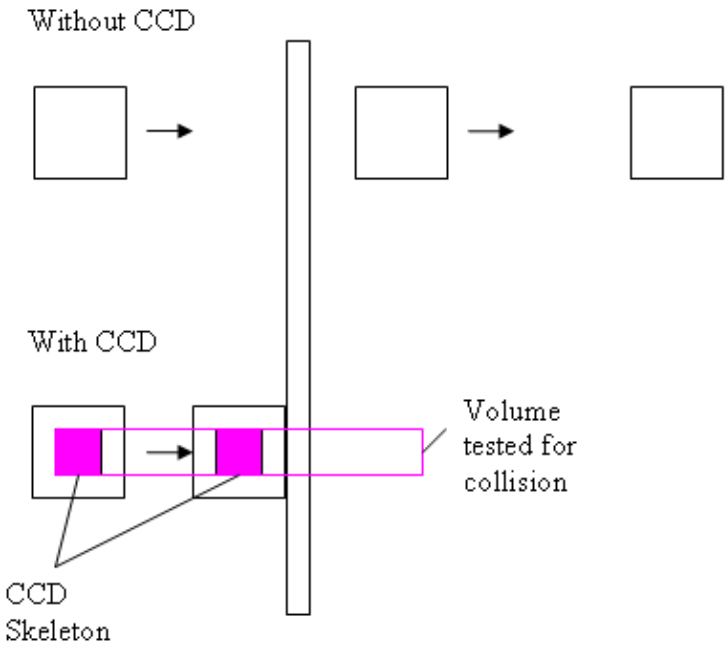
API Reference

- NxShape
- NxShapeDesc
- NxUserTriggerReport

Continuous Collision Detection

Continuous collision detection is useful for fast moving objects. With traditional collision detection, fast moving objects pass through other objects during a single time step. This effect is known as tunneling. Imagine a bullet traveling towards a thin metal plate, in the first time step it would be on one side and in the next it would be on the other. The SDK would never detect a collision and hence not constrain the motion of the bullet.

To deal with this problem, a technique known as continuous collision detection (CCD) is necessary. Instead of testing for collision at discrete points, it tests an extruded volume which represents the object's motion during the whole time step. If a collision is detected, the time of impact can be computed and the object's motion constrained appropriately. See the diagram below:



In the current version of the SDK, CCD is supported for dynamic vs. dynamic objects and dynamic vs. static objects.

CCD Skeletons

CCD is performed with a skeleton mesh embedded within the object, which is simpler than the geometry used for discrete collision detection. The skeleton stops the object flying through other objects. A higher frequency discrete collision object can provide more realistic behavior when the object is resting or rolling on another.

A CCD skeleton is a kind of mesh; however, there are less restrictions on a CCD skeleton than a standard mesh. Stray vertices are permitted (in other words, vertices not referenced by any triangles), but degenerate triangles (triangles that have a triangle index 2 or 3 times) are not. Stray vertices are supported so that you can use a single vertex to do a raycast style CCD test.

In addition, a single CCD skeleton can be associated with multiple shapes to reduce the memory overhead.

The SDK parameter NX_CCD_EPSILON is used to give triangles within the CCD skeleton a slight thickness to improve numerical robustness. When a point tries to move through a triangle, the CCD stops the point and positions it so that it is just touching the triangle. However, due to numerical precision issues, the point could

be classified as being on either side of the triangle in the next step. If it were classified as being on the wrong side of the triangle, it could tunnel through the triangle. Hence, NX_CCD_EPSILON is used to correctly classify the point. Because NX_CCD_EPSILON is a genuine numerical epsilon, it can be set to a very small value.

To create a CCD skeleton, use the NxPhysicsSDK::createCCDSkeleton() function. This function takes an NxSimpleTriangle mesh object, which describes the mesh. To associate a CCD skeleton with a shape, use the NxShape::setCCDSkeleton() method.

Guidelines for CCD skeletons:

- Performing CCD is more expensive than regular collision detection, so it's good to use as few triangles/vertices as possible.
- For actors with multiple shapes, the CCD skeletons of each subshape are merged into a single skeleton. The below restrictions on vertices and edges apply to this compound skeleton.
- Skeletons with more than 64 vertices are not supported.
- Skeletons are limited to a maximum of 256 edges.
- Static objects do not need CCD skeletons and do not use them, but rather go by the full mesh of the shape. (Note that only convexes and triangle meshes participate in CCD.)
- The CCD skeleton geometry should fit completely inside its associated shape's volume. In fact, it should keep a reasonable distance (see below) from the outside surface of this shape.

To explain the last bullet point further: The actual size of the CCD skeletons in relation to the shapes is something that needs to be tuned for the best performance in your specific game, but some points need to be kept in mind when doing this tuning:

- Do you want the CCD skeleton as accurate as possible, or as small and efficient as possible? A one-vertex only CCD skeleton is the most efficient choice.
- A small skeleton might let the actor escape through narrow passages, where it would normally not be able to move.
- A large CCD skeleton will make the CCD algorithm kick in more often.

When CCD becomes active (see below for further information about this) it stops ALL motion for the actor, including lateral sliding motion, which may not be desirable. In order to stop this from happening too often, you want to make the CCD skeleton smaller than the actual shape (minus the skin width), so that the CCD skeleton only comes in contact with other objects when it is in deeper than normal penetration. One situation where penetration depths normally increase past the skin width is in large stacks, where the iterative solver is working hard to de-penetrate objects. There are two main ways of finding a CCD skeleton that fits your game scenario, depending on what you are optimizing for:

1. **Accurate collision**
   A good way to find a CCD skeleton offset that fits your game scenario, is to start with a big CCD skeleton and make it smaller until you no longer see too many CCD blockages at low velocities.
2. **Fast collision**
   A good way to find a CCD skeleton for you scenario is to start off with a one-vertex skeleton and grow it until objects stop escaping through narrow passages, or stop missing in dynamic/dynamic collision scenarios.

To see if the CCD skeletons have been created properly, you can visualize them with NX_VISUALIZE_COLLISION_SKELETONS. Remember that the skeletons are typically inside the shape, so you may need to turn on transparency/wire frame mode to see them.

Example

```
NxSimpleTriangleMesh triMesh;
//Fill in triMesh...

NxCCDSkeleton *newSkeleton=gPhysicsSDK->createCCDSkeleton(triMesh);

NxShape *myShape=...

myShape->setCCDSkeleton(newSkeleton);
```

When is CCD applied?

To enable CCD computations for the scene, the NX_CONTINUOUS_CD SDK Parameter must be set.

Because CCD is slow in comparison to discrete collision detection methods, there are a couple of optimizations. First, CCD is only applied to fast moving objects. If an object is simply resting on a surface, then discrete collision detection is sufficient.

To control the threshold at which CCD is applied, use the NxActor::setCCDMotionThreshold() (and the corresponding NxBodyDesc member CCDMotionThreshold) to specify a minimum velocity for the use of CCD. In other words, below the motion threshold for the body, only regular discrete collision detection is used. This applies to absolute velocities and relative velocities (for dynamic vs. dynamic CCD).

For shapes which should take part in dynamic vs. dynamic CCD, the shape flag NX_SF_DYNAMIC_DYNAMIC_CCD must be raised.

The NX_CCD_EPSILON is used to give triangles within the CCD skeleton a slight thickness to improve numerical robustness. When a point tries to move through a triangle, the CCD stops the point and positions it so that it is just touching the triangle. However, due to numerical precision issues, the point could be classified as being on either side of the triangle in the next step. If it were classified as being on the wrong side of the triangle, it could tunnel through the triangle. Hence, NX_CCD_EPSILON is used to correctly classify the point. Because NX_CCD_EPSILON is a genuine numerical epsilon, it can be set to a very small value.

The following checks are used to determine if dynamic vs. static CCD is performed:

- Only if NX_CONTINUOUS_CD is raised
  OR
  we're in a compartment where NX_CF_INHERIT_SETTINGS is cleared and NX_CF_CONTINUOUS_CD is set.
- Only if the dynamic object has a CCD skeleton.
- Only if the point's velocity on the body is above CCDMotionThreshold.
- Only if the static object is an NxConvexMeshShape, NxTriangleMeshShape or compound of these.

The following checks are used to determine if dynamic vs. dynamic CCD is performed:

- Only if NX_CONTINUOUS_CD is raised
  OR
  we're in a compartment where NX_CF_INHERIT_SETTINGS is cleared and NX_CF_CONTINUOUS_CD is set.
- Only if both actors have bodies (are dynamic).
- Only if one of the shapes in a pair is tagged with NX_SF_DYNAMIC_DYNAMIC_CCD.
- Only if both shapes have CCD skeletons.
- Only if a point's velocity on either of the bodies is above CCDMotionThreshold.
- Only if the relative velocity of the bodies is above CCDMotionThreshold.

Serialization

CCD skeletons can be quickly serialized by saving them to a memory buffer using the following method:

```
NxU32 NxCCDSkeleton::save(void * destBuffer, NxU32 bufferSize);
NxU32 NxCCDSkeleton::getDataSize();
```

To load the skeleton, perform the following:

```
NxCCDSkeleton *NxPhysicsSDK::createCCDSkeleton(const void * memoryBuffer, NxU32 bufferSize);
```

Using the memory buffer approach may be faster than manually recreating the skeleton because the skeleton has the opportunity to store the skeleton in its internal format.

Limitations

- 64 vertex limit on CCD skeletons.
- Skeletons limited to 256 edges.
- Dynamic vs. static does not work when the static shape is an NxSphereShape, NxCapsuleShape, NxPlaneShape, etc. These shapes are not used for static geometry in many games and it would require significant work to support them since they are not polygon based shapes (the CCD algorithm works with meshes).
- If the geometry of the static mesh is very complex and high detail (not common in a typical game), the dynamic object may get stuck on its surface after the collision due to conflicting contact points. Minimize this effect by increasing the size of the CCD skeleton, but depending on the mesh, it may not be fully resolved.
- The dynamic vs. dynamic CCD uses an approximation which assumes one shape is rotating much faster than the other. If both shapes in a pair are rotating very quickly, the approximation may break down (causing unrealistic results).
- The CCD code assumes shapes will not rotate by more than 180 degrees in a time step.
- CCD works by limiting the motion of objects. When an interpenetration is detected, the objects are stopped dead, even if it is halfway through a time step. This may make objects appear to hang in mid air for one frame, etc.
- Triggers do not take part in CCD.
- Be very careful with assigning CCD skeletons only to some of a dynamic actor's shapes. There is the risk of a non-CCD shape penetrating another object before a CCD shape arrests the motion, leading to the actor getting stuck in penetration. Only create such actors if you can in some way insure that this will not occur.
- Kinematic objects are never halted by CCD and so using CCD for fast-moving kinematic bodies has no effect. (This only applies when it's the kinematic object that is fast-moving, fast-moving dynamic CCD objects are halted by kinematic ones as usual.)
- CCD does not work with restricted hardware scenes (see Hardware Scene Creation).

Example

```
gPhysicsSDK->setParameter(NX_CONTINUOUS_CD, 1);

...

int nbShapes=newAct->getNbShapes();
NxShape *const* shapeArray=newAct->getShapes();

for(int i=0;i <nbShapes;i++)
{
    NxShape *shape=shapeArray[i];
```

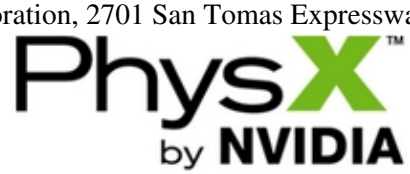
```
        NxSimpleTriangleMesh triMesh;
        CreateMeshFromShape(triMesh,shape);

        NxCCDSkeleton *newSkeleton=gPhysicsSDK->createCCDSkeleton(triMesh);

        FreeSimpleMesh(triMesh);

        shape->setCCDSkeleton(newSkeleton);
    }

    static void CreateMeshFromShape(NxSimpleTriangleMesh &triMesh,NxShape *shape)
    {

        NxBoxShape *boxShape=shape->isBox();

        if(boxShape!=NULL)
        {
            NxBox obb=NxBox(NxVec3(0.0f,0.0f,0.0f),boxShape->getDimensions(),NxMat33(NX_IDENTITY_

            triMesh.points=new NxVec3[8];
            triMesh.numVertices=8;
            triMesh.pointStrideBytes=sizeof(NxVec3);
            triMesh.numTriangles=2*6;
            triMesh.triangles=new NxU32[2*6*3];
            triMesh.triangleStrideBytes=sizeof(NxU32)*3;
            triMesh.flags=0;

            NxComputeBoxPoints(obb,(NxVec3 *)triMesh.points);
            memcpy((NxU32 *)triMesh.triangles,NxGetBoxTriangles(),sizeof(NxU32)*2*6*3);
        }
        else
            NX_ASSERT(!"Invalid shape type");

        NX_ASSERT(triMesh.isValid());

    }
```

Samples

Sample CCD Explosion
Sample CCD Dynamic

API Reference

- NxSimpleTriangleMesh
- NxShape
- NxPhysicsSDK
- NxCCDSkeleton



Dominance Groups

With dominance groups one can have all constraints (contacts and joints) created between actors act in one direction only. This is useful if you want to make sure that the movement of the rider of a vehicle or the pony tail of a character doesn't influence the object it is attached to, while keeping the motion of both inherently physical.

An NxDominanceGroup is a 5 bit group identifier (legal range from 0 to 31). By default every actor is created in group 0, and statics can never leave this group. You change this either in the descriptor (the dominanceGroup member) or by calling NxActor::setDominanceGroup().

Meaning

Whenever a constraint (i.e. joint or contact) between two actors (a_0, a_1) needs to be solved, the groups (g_0, g_1) of both actors are retrieved. Then the NxConstraintDominance setting for this group pair is retrieved with getDominanceGroupPair(g_0, g_1).

In the constraint, NxConstraintDominance::$dominance_0$ becomes the dominance setting for a_0, and NxConstraintDominance::$dominance_1$ becomes the dominance setting for a_1. A $dominance_N$ setting of 1.0f, the default, will permit a_N to be pushed or pulled by $a_{(1-N)}$ through the constraint. A $dominance_N$ setting of 0.0f, will however prevent a_N to be pushed or pulled by $a_{(1-N)}$ through the constraint. Thus, a NxConstraintDominance of (1.0f, 0.0f) makes the interaction one-way.

Default behavior

The matrix sampled by getDominanceGroupPair(g_1, g_2) is initialized by default such that:

- if $g_1 == g_2$, then (1.0f, 1.0f) is returned
- if $g_1 < g_2$, then (0.0f, 1.0f) is returned
- if $g_1 > g_2$, then (1.0f, 0.0f) is returned

In other words, we permit actors in higher groups to be pushed around by actors in lower groups by default.

These settings should cover most applications, and in fact not overriding these settings may likely result in higher performance. To change the settings, use NxScene::setDominanceGroupPair().

It is not possible to make the matrix asymmetric, or to change the diagonal. In other words:

- it is not possible to change (g_1, g_2) if (g_1==g_2)
- if you set (g_1, g_2) to X, then (g_2, g_1) will implicitly and automatically be set to ~X, where:
  - ~(1.0f, 1.0f) is (1.0f, 1.0f)
  - ~(0.0f, 1.0f) is (1.0f, 0.0f)
  - ~(1.0f, 0.0f) is (0.0f, 1.0f)

  These two restrictions are to make sure that constraints between two actors will always evaluate to the same dominance setting, regardless of which order the actors are passed to the constraint.

Dominance settings are currently specified as floats 0.0f or 1.0f because in the future we may permit arbitrary fractional settings to express 'partly-one-way' interactions.

API Reference

- NxScene

- NxActor

Raycasting

Raycasting is basically collision detection with rays. However, instead of attaching the line segment to an actor like other shapes, it gets cast into the scene at the user's request, stabbing one or more shapes in the process. It has many uses, from picking up objects with the mouse to checking if there is a line of sight between two characters.

There are several functions of raycasting; each one slightly different. They are all members of an NxScene called raycast*(), and are listed below:

- raycastAnyBounds, raycastAnyShape – these methods are the most primitive, but also the fastest. They simply return true if the ray that is specified hits any shape's axis aligned bounding box, or the shape itself.
- raycastClosestBounds, raycastClosestShape – similar to the first two methods, here the closest shape that is stabbed by the ray is returned, along with the distance along the ray and the ray-shape intersection point.
- raycastAllBounds, raycastAllShapes – these are the most complex versions, which return all of the shapes stabbed by the ray, including distances and intersection points. To use these, you must provide an NxUserRaycastReport interface, which will be called with information for each of the stabbed shapes.

Below is an example for implementing NxUserRaycastReport with the raycastAllShapes() function:

```
class myRaycastReport : public NxUserRaycastReport
{

    virtual bool onHit(const NxRaycastHit& hits)
    {

        //Record information here.
        return true; //Or false to stop the raycast.
    }

}gMyReport;

NxRay worldRay;
worldRay.orig= cstart;
worldRay.dir= cend - cstart;
worldRay.dir.normalize(); //Important!!

NxU32 nbShapes = gScene->raycastAllShapes(worldRay, gMyReport, NxUserRaycastReport::ALL_SHAPE
```

Caveats

- Wheel shapes are not detected by raycasts.
- The shapes are not guaranteed to be passed to onHit() in the order of their geometric layout along the ray.
- The SDK returns two face indices for raycast hits. faceID is the face index for the mesh before it is cooked and internalFaceID is the post-cooking face index (cooking can re-map face indices). The internalFaceID should be used when referencing the cooked version of the mesh using readback functions and saveToDesc(). (See Reading Back Mesh Data for more information.)
- The state of the SDK should not be modified from within the onHit() routine. In particular, objects should not be created or destroyed. If it is necessary to modify state then the updates should be stored to a buffer and performed after the raycast.
- Because the SDK double buffers data to allow asynchronous stepping state modification is not visible

to some functions(eg overlap and raycasting) until a simulate()/fetchResults() pair has been executed.

Threading

The NxUserRaycastReport class is only called in the context of the user thread. It is not necessary to make it thread safe.

Samples

Sample Raycast

API Reference

- NxScene
- NxUserRaycastReport
- NxRaycastHit
- NxTriangleMesh

Overlap Testing

In addition to providing Raycast and Trigger support, the NVIDIA PhysX SDK provides a set of general methods for testing collision primitive directly against shapes. These are useful when the other collision methods do not fit the user's needs.

The methods provided by NxScene allow efficient testing against the entire collision database, while the methods of NxShape test against specific objects.

NxTriangleMesh also provides methods for testing an AABB against the triangle mesh and retrieving the set of triangles which overlap the AABB.

NOTE: Because the SDK double buffers data to allow asynchronous stepping state modification is not visible to some functions(eg overlap and raycasting) until a simulate()/fetchResults() pair has been executed.

NxScene::overlapSphereShapes()

```
NxU32 overlapSphereShapes(const NxSphere& worldSphere, NxShapesType shapeType, NxU32 nbShapes
NxUserEntityReport<NxShape*>* callback, NxU32 activeGroups=0xffffffff, const NxGroupsMask* gr
bool accurateCollision=false);
```

This method tests a sphere (in the global frame) against shapes in the scene. The user can test against static, dynamic, or all shapes by setting the shapeType parameter to NX_STATIC_SHAPES, NX_DYNAMIC_SHAPES or NX_ALL_SHAPES respectively.

The user has two options for retrieving the overlapping shapes, detailed below:

- Provide a buffer large enough to hold all the overlapping shapes using the shapes and nbShapes parameters. If this buffer is insufficient in size, then only the first nbShapes are returned.
- Provide an NxUserEntityReport callback which directs the SDK to call the callback with sets of shapes as they are detected. A small buffer is allocated on the stack to return shapes, unless a buffer is provided using the shapes parameter, in which case the SDK uses this temporarily.

The activeGroups and groupsMask parameters provide an optional mechanism for filtering shapes based on their group membership. See Broad Phase Collision Detection for more information.

The accurateCollision parameter provides the option to either collide against the actual shapes (true) or the AABBs (false) shapes.

Example

```
NxSphere worldSphere(NxVec3(0,0,0), 5);

NxU32 nbShapes = gScene->getNbDynamicShapes();

NxShape** shapes = new NxShape* [nbShapes];

for (NxU32 i = 0; i < nbShapes; i++)
    shapes[i] = NULL;

gScene->overlapSphereShapes(worldSphere, NX_DYNAMIC_SHAPES, nbShapes, shapes, NULL);
```

NxScene::overlapAABBShapes()

```
NxU32 overlapAABBShapes(const NxBounds3& worldBounds, NxShapesType shapeType, NxU32 nbShapes, NxShap
NxUserEntityReport<NxShape*>* callback, NxU32 activeGroups=0xffffffff, const NxGroupsMask* groupsMas
bool accurateCollision=false);
```

This method is identical to overlapSphereShapes(), except it tests an axis aligned bound box against the shape database instead of a sphere.

Example

```
NxBounds3 worldBounds;
worldBounds.set(NxVec3(-5,-5,-5), NxVec3(5,5,5));

NxU32 nbShapes = gScene->getNbDynamicShapes();

NxShape** shapes = new NxShape* [nbShapes];

for (NxU32 i = 0; i < nbShapes; i++)
    shapes[i] = NULL;

gScene->overlapAABBShapes(worldBounds, NX_DYNAMIC_SHAPES, nbShapes, shapes, NULL);
```

NxScene::overlapOBBShapes()

```
NxU32 overlapOBBShapes(const NxBox& worldBox, NxShapesType shapeType, NxU32 nbShapes, NxShape** shap
NxU\serEntityReport<NxShape*>* callback, NxU32 activeGroups=0xffffffff, const NxGroupsMask* groupsMa
bool accurateCollision=false) = 0;
```

This method is identical to overlapSphereShapes(), except it tests an oriented bounding box against the shape database instead of a sphere.

NxScene::overlapCapsuleShapes()

```
NxU32 overlapCapsuleShapes(const NxCapsule& worldCapsule, NxShapesType shapeType, NxU32 nbShapes, Nx
NxUserEntityReport<NxShape*>* callback, NxU32 activeGroups=0xffffffff, const NxGroupsMask* groupsMas
bool accurateCollision=false) = 0;
```

This method is identical to overlapSphereShapes(), except it tests a world space capsule against the shape database instead of a sphere.

NxScene::cullShapes()

```
NxU32 cullShapes(NxU32 nbPlanes, const NxPlane* worldPlanes, NxShapesType shapeType, NxU32 nbShapes,
NxUserEntityReport<NxShape*>* callback, NxU32 activeGroups=0xffffffff, const NxGroupsMask* groupsMas
```

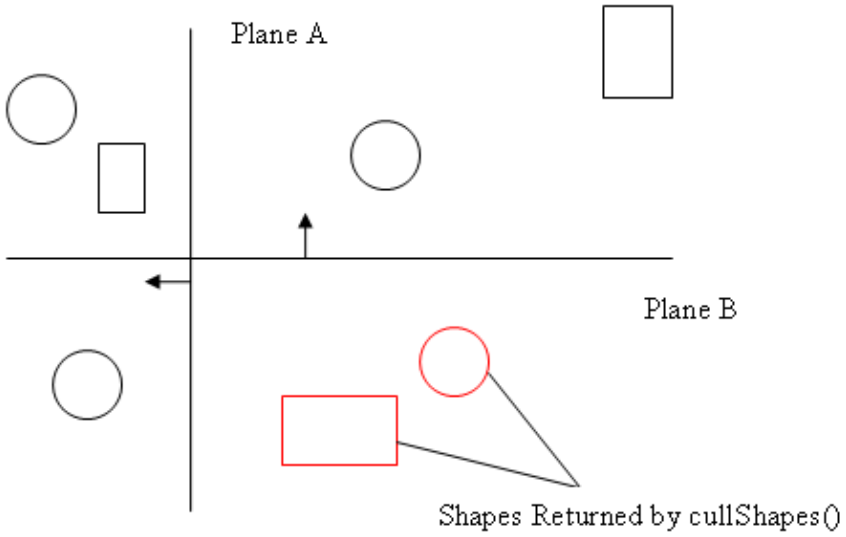
This method is a little different; it tests all shapes in the collision database against a set of planes. It tests objects against the planes and returns them if they are in the plane's negative half space (i.e., on the side which the normal points away from).

Note that the set of shapes returned by cullShapes() is not conservative; additional shapes may be returned which do not intersect the volume formed by the plane's positive half space. This allows a more efficient

implementation.

A limitation of this function is that the number of planes should not exceed 32.

The primary intent of this function is for view frustum culling. The user can pass through the 6 planes which define the view frustum to receive all the objects which intersect it for further processing.



Example

```
NxPlane worldPlanes[2];

worldPlanes[0]=NxPlane(-1.0f,0.0f,0.0f,0.0f);
worldPlanes[1]=NxPlane(0.0f,1.0f,0.0f,0.0f);

NxU32 nbShapes = gScene->getNbDynamicShapes();

NxShape** shapes = new NxShape* [nbShapes];

for (NxU32 i = 0; i < nbShapes; i++)
    shapes[i] = NULL;

gScene->cullShapes(2, worldPlanes, NX_DYNAMIC_SHAPES, nbShapes, shapes, NULL);
```

NxScene::checkOverlapSphere()

```
bool checkOverlapSphere(const NxSphere& worldSphere, NxShapesType shapeType=NX_ALL_SHAPES, Nx
    const NxGroupsMask* groupsMask=NULL, bool accurateCollision=false);
```

This method tests the sphere against shapes in the collision database. The "accurateCollision" parameter controls if this method should return true only if the sphere overlaps the actual shape or the AABB of the shape (accurateCollision=false). The default is to only test against the AABB of the shape.

Example

```
NxSphere worldSphere(NxVec3(0,0,0), 5);

bool intersection = gScene->checkOverlapSphere(worldSphere, NX_DYNAMIC_SHAPES);
```

NxScene::checkOverlapAABB()

```
bool checkOverlapAABB(const NxBounds3& worldBounds, NxShapesType shapeType=NX_ALL_SHAPES, NxU32 acti
const NxGroupsMask* groupsMask=NULL, bool accurateCollision=false);
```

This method tests the specified AABB against shapes in the collision database. The "accurateCollision" parameter controls if this method should return true only if the specified AABB overlaps the actual shape or the AABB of the shape (accurateCollision=false). The default is to only test against the AABB of the shape.

Example

```
NxBounds3 worldBounds;
worldBounds.set(NxVec3(-5,-5,-5), NxVec3(5,5,5));

bool intersection = gScene->checkOverlapAABB(worldBounds, NX_DYNAMIC_SHAPES);
```

NxScene::checkOverlapCapsule()

```
bool checkOverlapCapsule(const NxCapsule& worldCapsule, NxShapesType shapeType=NX_ALL_SHAPES, NxU32
    const NxGroupsMask* groupsMask=NULL, bool accurateCollision=false);
```

This method tests the capsule against shapes in the collision database. The "accurateCollision" parameter controls if this method should return true only if the capsule overlaps the actual shape or the AABB of the shape (accurateCollision=false). The default is to only test against the AABB of the shape.

Example

```
NxCapsule worldCapsule(NxSegment(NxVec3(0.0f,0.0f,0.0f),NxVec3(0.0f,1.0f,0.0f)),1.0f);

bool intersection = gScene->checkOverlapCapsule(worldCapsule, NX_DYNAMIC_SHAPES);
```

NxScene::checkOverlapOBB()

```
bool checkOverlapOBB(const NxBox& worldBox, NxShapesType shapeType=NX_ALL_SHAPES, NxU32 activeGroups
    const NxGroupsMask* groupsMask=NULL, bool accurateCollision=false);
```

This method tests the OBB against shapes in the collision database. The "accurateCollision" parameter controls if this method should return true only if the OBB overlaps the actual shape or the AABB of the shape (accurateCollision=false). The default is to only test against the AABB of the shape.

Example

```
NxBox worldBox(NxVec3(0.0f,0.0f,0.0f),NxVec3(1.0f,1.0f,1.0f), NxMat33(NX_IDENTITY_MATRIX));

bool intersection = gScene->checkOverlapOBB(worldBox, NX_DYNAMIC_SHAPES);
```

NxShape::checkOverlapSphere()

```
bool checkOverlapSphere(const NxSphere& worldSphere);
```

This method is similar to NxScene::checkOverlapSphere(), except that only a single shape is tested.

NxShape::checkOverlapAABB()

```
bool checkOverlapAABB(const NxBounds3& worldBounds);
```

This method is similar to NxScene::checkOverlapAABB(), except that only a single shape is tested.

NxShape::checkOverlapCapsule()

```
bool checkOverlapCapsule(const NxCapsule& worldCapsule);
```

This method is similar to NxScene::checkOverlapCapsule(), except that only a single shape is tested.

NxShape::checkOverlapOBB()

```
bool checkOverlapOBB(const NxBox& worldBox);
```

This method is similar to NxScene::checkOverlapOBB(), except that only a single shape is tested.

NxTriangleMeshShape::overlapAABBTriangles() and NxHeightFieldShape::overlapAABBTriangles()

```
bool overlapAABBTriangles(const NxBounds3& bounds, NxU32 flags, NxUserEntityReport<NxU32>* ca
```

Use this method to test an AABB against a triangle mesh shape and retrieve a list of triangles which intersect the AABB. The user entity report(callback) is called with batches of triangle indices.

Triangle indices returned by overlapAABBTriangles() can be used with getTriangle() to obtain the triangle's position, edge normals and flags.

The flag's member provides the following options:

- NX_QUERY_WORLD_SPACE - If set, the AABB is specified in the global frame. Otherwise, it is assumed to be specified in the shape's frame.
- NX_QUERY_FIRST_CONTACT - If set, the overlapAABBTriangles() only returns the first intersecting triangle that it detects.

NxTriangleMeshShape::getTriangle()

```
NxU32 getTriangle(NxTriangle& worldTri, NxTriangle* edgeTri, NxU32* flags,
            NxTriangleID triangleIndex, bool worldSpaceTranslation=true, bool worldSpaceRotat
```

This method allows the user to retrieve information about a triangle specified by its index (triangleIndex). The edgeTri and flags parameters are optional and can be set to NULL.

- worldTri - The world space vertex positions of the triangle.
- edgeTri - The world space edge normals of the triangle.

- flags - Specify if an edge of the triangle is convex or not. NXTF_ACTIVE_EDGE01, NXTF_ACTIVE_EDGE12, NXTF_ACTIVE_EDGE20 are set if edge 0, 1 or 2 are convex.
- worldSpaceRotation, worldSpaceTranslation - allows you to get the result in the world frame or in the local frame, with any combination of world or local rotation/translation.

Caveat

When shapes' poses are updated by the application directly, such as through calling NxActor::setGlobalPose, their broad phase bounds are not automatically updated until the next simulate call. This may cause incorrect results for API calls like overlap tests or cloth attachment.
A newly created entity has as yet no bounds and is not affected by this shortcoming.
If you are experiencing problems with for example overlap testing because of this, you can work around it by deferring such relocations until after tests are performed.

API Reference

- NxScene
- NxShape
- NxTriangleMeshShape

Contact Reports

The NxUserContactReport class has functions for passing information to the user about collisions that occur. This information can be used for playing 3D sounds or other effects in response to simulated collisions. An example is demonstrated within this section.

NOTE: Do not rely on the exact positioning or geometry of contacts. The SDK is free to re-organize or merge contact points to improve efficiency as long as the simulation is not affected.

First, subclass NxUserContactReport:

```
class MyContactReport : public NxUserContactReport
{
    void onContactNotify(NxContactPair& pair, NxU32 events)
    {
    //You can read the contact information out of the
    //contact pair data here.
    }
} myReport;
```

Then register this object with the SDK as before:

```
scene->setUserContactReport(&myReport);
```

The MyContactReport::onContactNotify() function receives the contact information for each pair of actors that request notifications. The following options to set contact notifications are available:

- Request notification for a specific actor pair using NxScene::setActorPairFlags().
- Request notification for actor pairs of two specific actor groups using NxScene::setActorGroupPairFlags().
- Request notification for the pairs of a specific actor using NxActor::setContactReportFlags().
- Suppress notification for actor pairs which only involve a specific shape pair using NxScene::setShapePairFlags() and the NX_IGNORE_PAIR flag.
- Suppress notification for actor pairs where the contact normal force is below a certain threshold using the options above, NxActor::setContactReportThreshold() and the flags NX_NOTIFY_ON_START_TOUCH_FORCE_THRESHOLD, NX_NOTIFY_ON_END_TOUCH_FORCE_THRESHOLD or NX_NOTIFY_ON_TOUCH_FORCE_THRESHOLD.

The possible contact notify flags are NX_NOTIFY_ON_START_TOUCH, NX_NOTIFY_ON_END_TOUCH, NX_NOTIFY_ON_TOUCH, NX_NOTIFY_ON_START_TOUCH_FORCE_THRESHOLD, NX_NOTIFY_ON_END_TOUCH_FORCE_THRESHOLD, NX_NOTIFY_ON_TOUCH_FORCE_THRESHOLD, and NX_NOTIFY_FORCES, besides the NX_IGNORE_PAIR flag previously mentioned on the Broad Phase Collision Detection page.

In our example we request contact notification for a specific pair of actors.

```
scene->setActorPairFlags(actor0, actor1, NX_NOTIFY_ON_START_TOUCH | NX_NOTIFY_ON_END_TOUCH |
```

Passing a combination of the notify flags, as shown above, creates a behavior similar to triggers. However, when using a trigger, an event is received when a shape passes into or leaves the volume of another shape. Whereas in this situation, shapes start or stop touching each other's surface. More contact status flags will be added in the future to better support sounds for simulation.

Instead of requesting contact notification for a specific pair of actors, we could focus on all the pairs of an actor. Additionally, notifications for contacts with a normal force below a certain threshold shall be suppressed. This would look as follows:

```
actor->setContactReportFlags(NX_NOTIFY_ON_START_TOUCH_FORCE_THRESHOLD | NX_NOTIFY_ON_END_TOUCH_FORCE
actor->setContactReportThreshold(100);
```

The contact information provided to onContactNotify() is in the form of the NxContactPair structure, and the event's argument is the type of event that has happened. This is a combination of one or more of the flags that have been set with setActorPairFlags(). The most interesting member of NxContactPair is the contact stream (NxConstContactStream). This is a compressed data structure that contains detailed information about the contacts between a pair, including the contact normal and friction forces acting between the pair. You can use the NxContactStreamIterator class to read it. Please refer to the API documentation and code for the sample SampleContactStreamIterator for details on these classes.

Caveats

- Kinematics that move into contact with statics or with other kinematics will not generate contact reports.
- When enabling contact reporting using setActorPairFlags() the shapes for the actor must be created prior to the call since actor pair flags are in fact applied to the shapes belonging to the actor. If setActorPairFlags() is called for an actor with no shapes, the call will be ignored.
- The state of the SDK should not be modified from within the onContactNotify() routine. In particular, objects should not be created or destroyed. If it is necessary to modify state, then the updates should be stored to a buffer and performed after the simulation step.

Threading

The user contact report is only called in the context of the user thread, so it is not necessary to ensure that it is thread safe.

Samples

Sample Contact Stream Iterator

API Reference

- NxActor
- NxScene
- NxUserContactReport

Batched Scene Queries

The SDK exposes the following *scene-level queries*, in *NxScene*:

- Raycast functions
- Sweep tests
- Overlap tests
- Culling tests

In version 2.7.0 a new *NxSceneQuery* object was introduced to batch such queries together and execute all of them at the same time, instead of one after the other. Batched queries are usually faster than the individual calls.

Creating / Releasing Scene Query Objects

The first step is to create an *NxSceneQuery* object, using this function:

```
virtual NxSceneQuery* NxScene::createSceneQuery(const NxSceneQueryDesc&)
```

This object will store the batched queries, manage them, etc. It will eventually replace all the corresponding *NxScene* functions. When you are done with the object, release it with the following function:

```
virtual bool NxScene::releaseSceneQuery(NxSceneQuery&)
```

The object is created via a descriptor containing the following elements:

```
NxSceneQueryReport* NxSceneQueryDesc::report;
NxSceneQueryExecuteMode NxSceneQueryDesc::executeMode;
```

Those elements are explained in details in the following sections.

Queries

The scene queries that are available through the new batched interface are basically the same as the ones available directly through NxScene, with a few differences:

- The raycast functions against bounds are not supported anymore.
- Only the accurate mode is supported in overlap tests

Please refer to the original functions for details.

NxSceneQueryReport

This is a user-defined report object, called with query results. There are 4 different callbacks for 4 different kinds or results:

```
virtual NxQueryReportResult   onBooleanQuery(void* userData, bool result)

virtual NxQueryReportResult   onRaycastQuery(void* userData, NxU32 nbHits, const NxRaycastHit* hi

virtual NxQueryReportResult   onShapeQuery(void* userData, NxU32 nbHits, NxShape** hits)
```

```
virtual NxQueryReportResult   onSweepQuery(void* userData, NxU32 nbHits, NxSweepQueryHit* hits)
```

Whenever a result is returned through those callbacks, one can define what happens next, with an *NxQueryReportResult* return value. It is possible to continue enumerating the results for current query (NX_SQR_CONTINUE), stop enumerating the results for current query (NX_SQR_ABORT_QUERY), or stop enumerating all results for all batched queries (NX_SQR_ABORT_ALL_QUERIES).

NxSceneQueryExecuteMode

It is possible to execute the scene queries in two different modes: synchronous and asynchronous.

In *synchronous* mode, queries are not batched, but instead performed immediately. This is the same as calling the old *NxScene* functions. Results are available immediately, either through returned parameters, or through the callback object.

In *asynchronous* mode, queries are batched inside the query object. They are all performed at the same time in a later stage, when the user calls the *NxSceneQuery::execute* method. Results are then sent to the *NxSceneQueryReport* object exclusively. When the queries are performed in a separate thread, you can use the *NxSceneQuery::finish()* method to check whether the work has been completed or not.

Query Object Lifetime

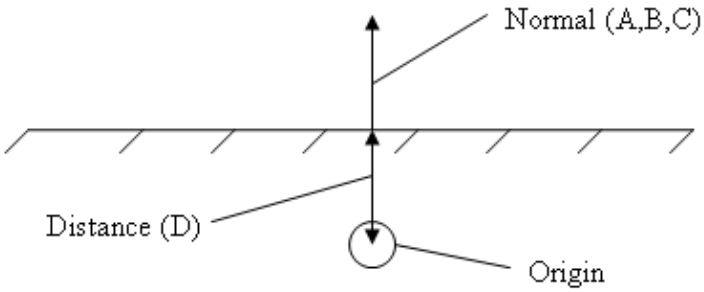
An *NxSceneQuery* should be re-used for multiple queries over multiple frames. It is not efficient to create a different object each time you perform a raycast, for example.

For batched queries (asynchronous mode), keep in mind that all queries and their parameters are saved inside the *NxSceneQuery* object. This uses some memory. The memory is recycled for subsequent batched queries, and only released when the *NxSceneQuery* object is deleted.

API Reference

- NxScene

Plane Shape



Planes (class NxPlaneShape) are useful for simple demos in which a ground plane is needed but little else. A plane is characterized in the world space plane equation with the following:

```
ax + by + cz = d
```

Planes are static and cannot be assigned to mobile actors; the plane equation is always in world space and the shape's relative transform is ignored. However, the plane can be moved manually by changing the equation.

The plane represents a half space rather than an infinitely thin shape. Thus, all points behind a plane are treated as being in contact with it. This way, when an object is dropped onto the plane, it will not pop through it and continue falling.

Because planes are assumed to be static objects, they never report contacts with other planes.

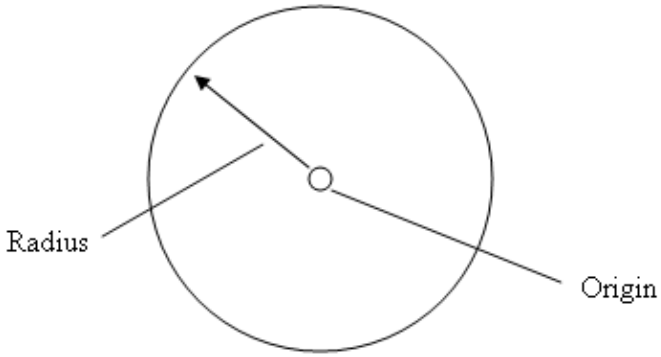
Example

```
NxPlaneShapeDesc planeDesc;
NxActorDesc actorDesc;

//create a plane at y=1
planeDesc.normal = NxVec3(0.0f,1.0f,0.0f);
planeDesc.d = 1.0f;

actorDesc.shapes.pushBack(&planeDesc);

NxActor *planeActor = gScene->createActor(actorDesc);
```

API Reference

- NxShape
- NxShapeDesc
- NxPlaneShape
- NxPlaneShapeDesc

Sphere Shape



Spheres (class NxSphereShape) are the simplest types of shapes. Their only property is their radius.

Example

```
NxBodyDesc bodyDesc;
NxActorDesc actorDesc;
NxSphereShapeDesc sphereDesc;

sphereDesc.radius = 1.0f;

actorDesc.shapes.pushBack(&sphereDesc);
actorDesc.body = &bodyDesc;
actorDesc.density = 10.0f;
actorDesc.globalPose.t = NxVec3(0.0f,0.0f,0.0f); //Position at the origin.

NxActor *actor=gScene->createActor(ActorDesc);
```

Sphere-Mesh Collision Detection

When performing collision detection between spheres and meshes, choose between two algorithms: normal and smooth. The normal algorithm assumes that the mesh is composed of flat triangles. When a ball rolls along the mesh surface, it will experience small, sudden changes in velocity as it rolls from one triangle to the next. The smooth algorithm, on the other hand, assumes that the triangles are just an approximation of a surface that is smooth. It uses the Gouraud Algorithm (recommended for simulating car wheels on a terrain) to smooth the triangles' vertex normals. This way the rolling sphere's velocity will change smoothly over time, instead of suddenly, making it more realistic. Choose which method to use for each mesh instance using the NxTriangleMeshShapeDesc::flags member's NX_MESH_SMOOTH_SPHERE_COLLISIONS bit. If the flag is raised, the smooth method is used.

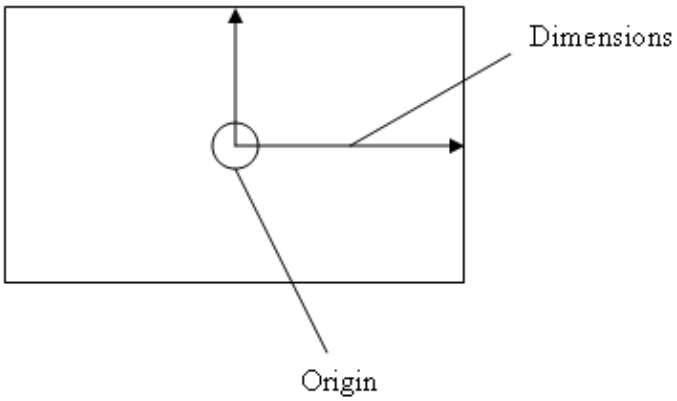
NOTE: This flag defaulted to being raised in SDK versions prior to 2.1.2. Now it defaults to being lowered.

API Reference

- NxShape
- NxShapeDesc
- NxSphereShape

- NxSphereShapeDesc

Box Shape



Boxes (class NxBoxShape) have a dimensions property. The elements of this vector are the volume's half width, half height and half depth. Specifying a large half width will yield a wide box.

Example

```
//Create a dynamic actor with a box shape.
NxActorDesc actorDesc;
NxBodyDesc bodyDesc;
NxBoxShapeDesc boxDesc;

boxDesc.dimensions.set(0.5f,0.5f,0.5f); //The actor has one shape, a 1m cube.

actorDesc.shapes.pushBack(&boxDesc);
actorDesc.body = &bodyDesc;
actorDesc.density = 10;
actorDesc.globalPose.t = NxVec3(0.0f,10.0f,0.0f);//Set initial position.

NxActor *dynamicActor=gScene->createActor(actorDesc);
```

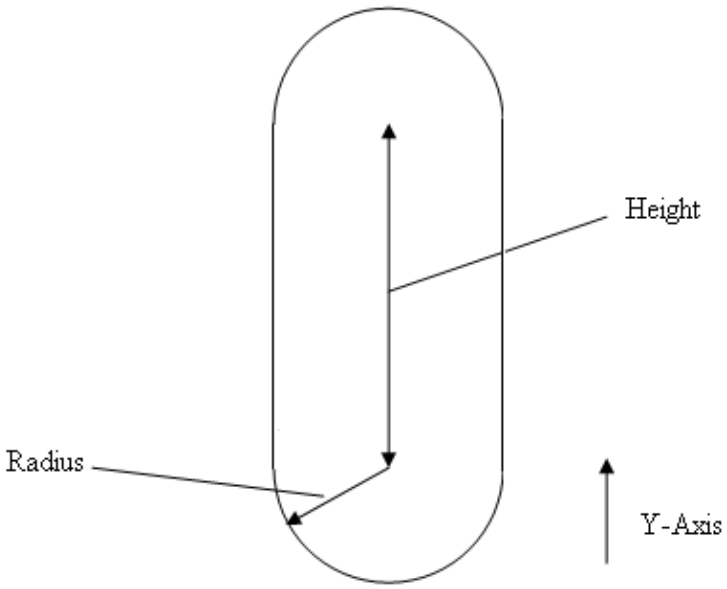
Samples

Sample Boxes

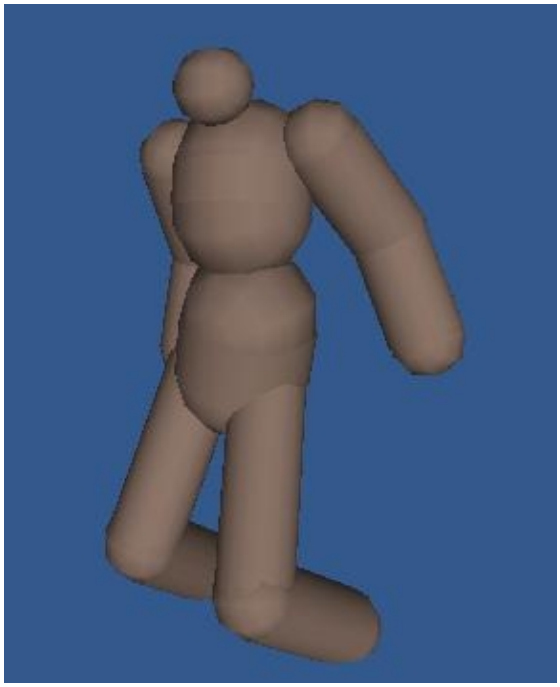
API Reference

- NxShape
- NxShapeDesc
- NxBoxShape
- NxBoxShapeDesc

Capsule Shape



Capsules (class NxCapsuleShape), also called line swept spheres, are like cylinders that have rounded ends. Capsules are great to approximate round shapes like body parts because their collision detection is very fast and robust, even more so than boxes. The image below shows a character whose bounding boxes are made using capsules.



Example

```
NxActorDesc actorDesc;
NxBodyDesc bodyDesc;

float height = 2.0f;
```

```
float radius = 0.5f;

//The actor has one shape, a capsule.
NxCapsuleShapeDesc capsuleDesc;

capsuleDesc.height = height;|
capsuleDesc.radius = radius;

actorDesc.shapes.pushBack(&capsuleDesc);
actorDesc.body = &bodyDesc;
actorDesc.density = density;
actorDesc.globalPose.t = NxVec3(0,radius+(NxReal)0.5*height,0);//Rest the capsule on the y=0 plane.

NxActor *capsuleActor=gScene->createActor(actorDesc);
```

Samples

Sample Pulley Joint
Sample D6 Joint

API Reference

- NxShape
- NxShapeDesc
- NxCapsuleShape
- NxCapsuleShapeDesc

Heightfield Shape

In previous versions of the old NovodeX SDK/PhysX SDK, support for heightfields/terrain was provided as a sub feature of general mesh support. This led to significant memory overhead due to persistent storage of mesh vertices, indices and connectivity information.

However, the SDK now supports a specialized heightfield shape which stores the heightfield data as elevation samples. This provides memory savings in addition to potential performance improvements due to better cache utilization and simpler adjacency/contact generation logic.

Heightfields are similar to meshes in that a heightfield object contains the bulk of the data and then an instance of that in the form of a shape. The heightfield object is called NxHeightField and instances are created using NxHeightfieldShapes. NxHeightfieldShapes belongs to actors in the same way as other shapes such as NxTriangleMeshShape, NxSphereShape, etc.

Heightfield objects contain the following:

- Elevation samples in a grid
- The vertical extent or thickness of the heightfield
- Tessellation flags
- Material offset indices (which are added to the material indices supplied to the shape)

Heightfield shapes contain the following:

- Vertical and horizontal scale of the heightfield instance
- Material indices
- The hole/not present material

Creation - NxHeightField

Currently, the SDK supports heightfields that use just one sample format defining elevation, materials, etc. per sample:

- height - 16 bits, a signed 16 bit integer which specifies the elevation at the sample
- materialIndex0 - 7 bits, material index
- tessFlag - 1 bit, tessellation flag
- materialIndex1 - 7 bits, material index
- unused/reserved - 1 bit

height	materialIndex0	tessFlag	materialIndex1	unused
16 bits (signed)	7 bits	1 bit	7 bits	1 bit

Heightfield samples are laid out in a row major format, with a stride specifying the number of bytes from the start of one sample to the next. The stride allows the user to interleave their own data with the heightfield data. The descriptor and heightfield data provided to the SDK can be deleted after the call to createHeightfield() as the SDK makes an internal copy of this information.

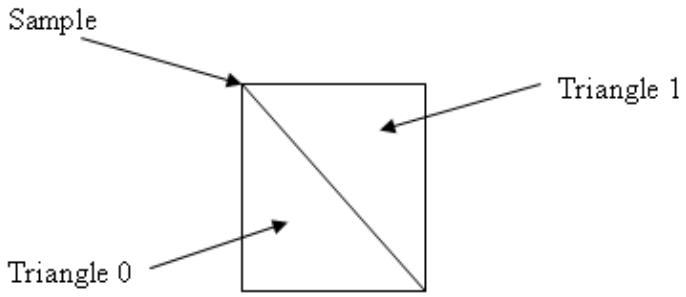
When defining the heightfield, the user must provide either a vertical extent or a thickness for the heightfield. Using a vertical extent is similar to modelling the heightfield in a box of sand, each sample will extend down to a certain depth. If you are using a thickness, then each sample has that thickness, which could be visualized as two copies of the heightfield surface, where one is vertically offset with the thickness. Please note that the thickness is per sample, not per face, thus on steep slopes the actual thickness is lower.

Vertical extent
When using a vertical extent, objects are considered colliding with the heightfield up until the vertical extent. The vertical extent is not assumed to be infinite because the user may wish to place objects on the other side of the heightfield (e.g., caves). The vertical extent must encompass the heightfield data, otherwise collision detection with the heightfield may behave in an unpredictable way. The vertical extent is specified with respect to the scaled heightfield values for each instance so it must encompass the largest scale.

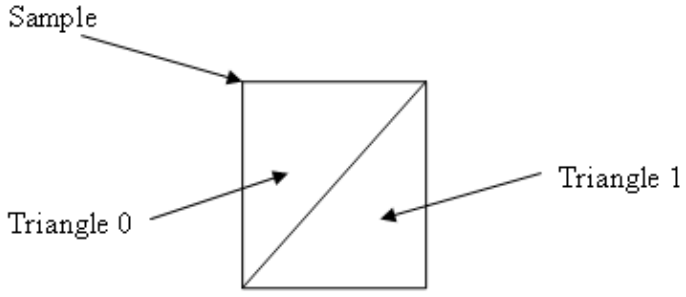
Thickness
If you want to model caves or have other areas under the heightfield, it might be better to specify that the heightfield should use a thickness instead of the vertical extent. Objects are considered colliding with the heightfield from the samples and to a depth equal to the thickness from each individual sample. As with the vertical extent, the thickness is not scaled.

The additional information in a heightfield sample defines the properties for the two triangles below and to the right of the sample point (in the sample array).

The tessellation flag defines which way the quad, formed by the four sample points to the right and bottom of the sample point, will be split. If the tessellation flag is set, then it will be split with an edge starting at the tessellation point and ending at the diagonally opposite vertex as shown in the following diagram:



However, if the flag is not set, then the quad will be split along the other diagonal as shown below:



In addition to specifying the tessellation flag, the sample also defines the materials for each triangle in the quad. The triangles in the quad are numbered so that triangle 0 is always the triangle which shares an edge with the triangle to the right of the quad (see the diagram above). In this way, triangle numbering is independent of the tessellation flag.

Once you have created the heightfield shape, you are free to delete the buffers used for initialization. They are copied by the SDK.

Example

```
NxHeightFieldDesc heightFieldDesc;

heightFieldDesc.nbColumns           = nbColumns;
heightFieldDesc.nbRows              = nbRows;
heightFieldDesc.verticalExtent      = -1000;
heightFieldDesc.convexEdgeThreshold = 0;

//Allocate storage for samples.
heightFieldDesc.samples             = new NxU32[nbColumns*nbRows];
heightFieldDesc.sampleStride        = sizeof(NxU32);

NxU8* currentByte = (NxU8*)heightFieldDesc.samples;

for (NxU32 row = 0; row < nbRows; row++)
    {
    for (NxU32 column = 0; column < nbColumns; column++)
        {
        NxHeightFieldSample* currentSample = (NxHeightFieldSample*)currentByte;

        currentSample->height         = computeHeight(row,column); //Desired height value. Si
        currentSample->materialIndex0 = gMaterial0;
        currentSample->materialIndex1 = gMaterial1;

        currentSample->tessFlag = 0;

        currentByte += heightFieldDesc.sampleStride;
        }
    }

NxHeightField* heightField = gScene->getPhysicsSDK().createHeightField(heightFieldDesc);

//Data has been copied therefore free the buffer.
delete[] heightFieldDesc.samples;
```

Creation - NxHeightFieldShape

When creating a heightfield shape, specify horizontal and vertical scale, material mapping, a hole material, etc.

- heightField - The heightfield object that will be used by the shape.
- heightScale - A scale value applied to the elevation values of the heightfield.
- rowScale - A scale value applied to the heightfield rows.
- columnScale - A scale applied to the heightfiel columns.
- materialIndexHighBits - Specifies the material for a heightfield triangle along with the low order bits of the material index stored in each heightfield sample.

The heightfield is mapped to shape space with the vertical/elevation mapped to the y axis. The rows of the heightfield grid are mapped to the x axis and the columns to the z axis. Rows and columns increase from the origin in the positive direction.

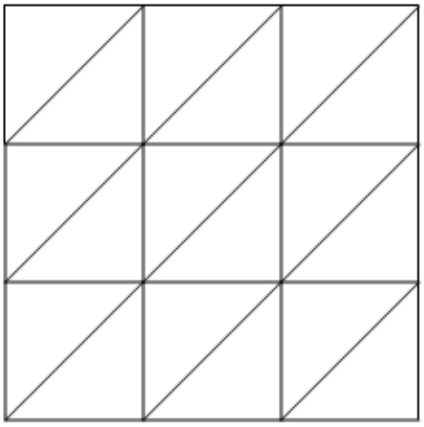
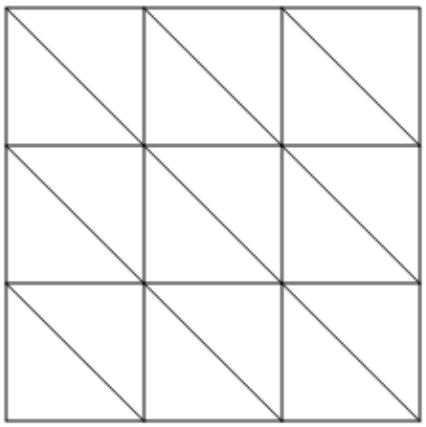
In other words, the scale factors are applied to the heightfield as if the sample points were mapped into the cube (0,0,0)=>(1,1,1). The scale factors must be non-zero, although negative scale factors are allowed. Negative scale factors mirror the heightfield across the appropriate axis, but do not change the direction of the normals (so the heightfield will still face upwards when it is mirrored across an axis).

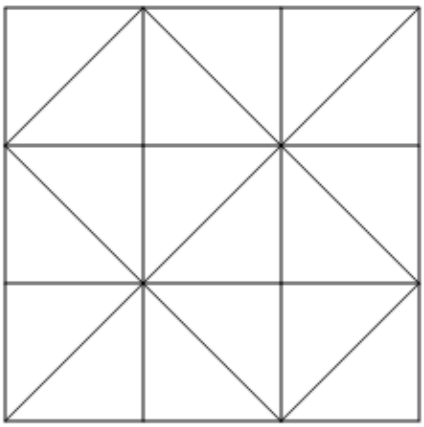
Example - Creating the Heightfield Shape

```
NxHeightFieldShapeDesc heightFieldShapeDesc;

heightFieldShapeDesc.heightField     = heightField;
heightFieldShapeDesc.heightScale     = gVerticalScale;
heightFieldShapeDesc.rowScale        = gHorizontalScale;
heightFieldShapeDesc.columnScale     = gHorizontalScale;
heightFieldShapeDesc.materialIndexHighBits = 0;
heightFieldShapeDesc.holeMaterial = 2;

NxActorDesc actorDesc;

actorDesc.shapes.pushBack(&heightFieldShapeDesc);
actorDesc.globalPose.t = pos;

NxActor* newActor = gScene->createActor(actorDesc);
```

Tessellation Flags

As discussed above, the tesselation flags allow the user to change the way cells in the heightfield are divided into triangles. This allows the user to choose a tessellation which most accurately represents their terrain/renderable data. Some examples are given below:

Tessellation flags	Result
0,0,0, 0,0,0, 0,0,0,	
1,1,1, 1,1,1, 1,1,1,	

0,1,0, 1,0,1, 0,1,0,	

Materials and Holes

As described above, each sample specifies materials for a triangle. A full material index, as returned from NxMaterial::getMaterialIndex(), is not fully stored in a sample (due to size restrictions). Instead, the material index for each heightfield sample, 7 bits, is combined with the materialIndexHighBits member of NxHeightFieldShapeDesc to form the full 16 bit index.

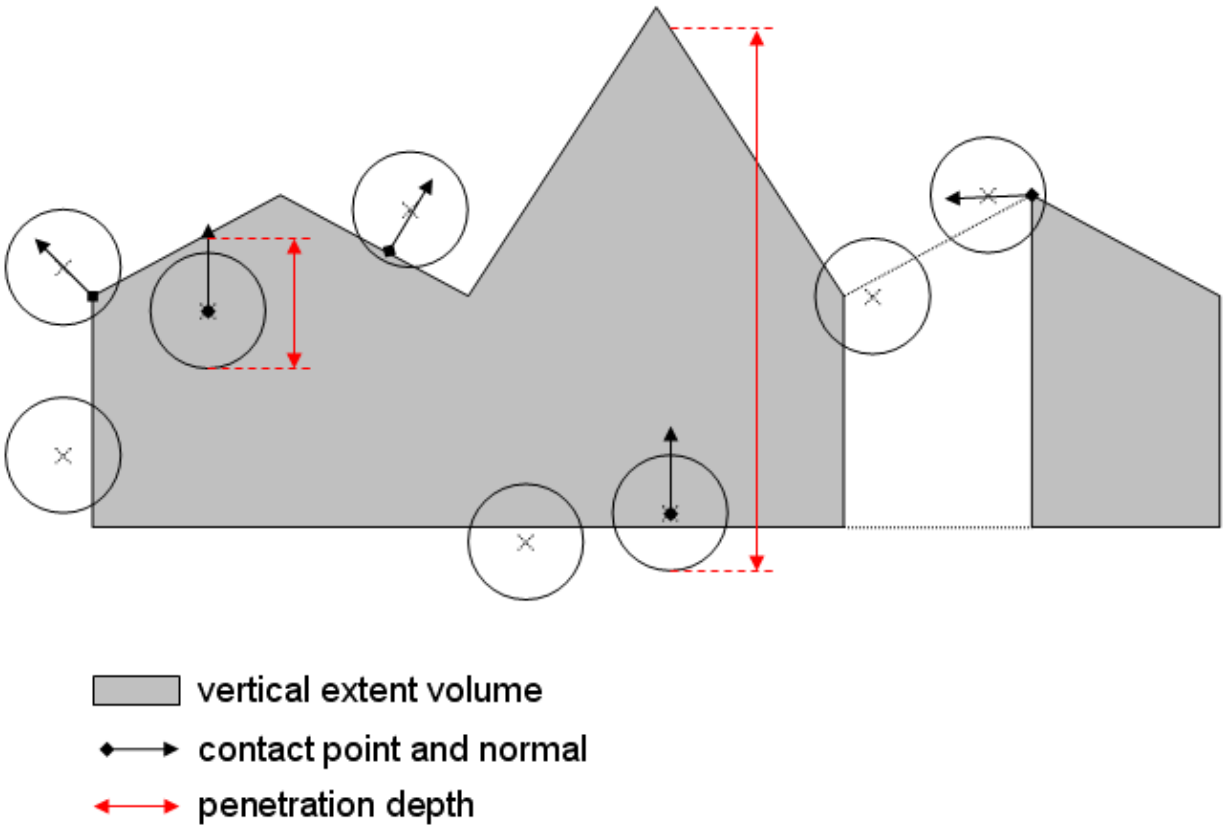
The SDK allows the user to specify a hole material when creating a heightfield shape. This material is treated in a special way; it effectively instructs the SDK to ignore certain triangles in the grid, creating holes in the terrain. A mesh or another heightfield could join with these holes to form caves and more complicated geometry.

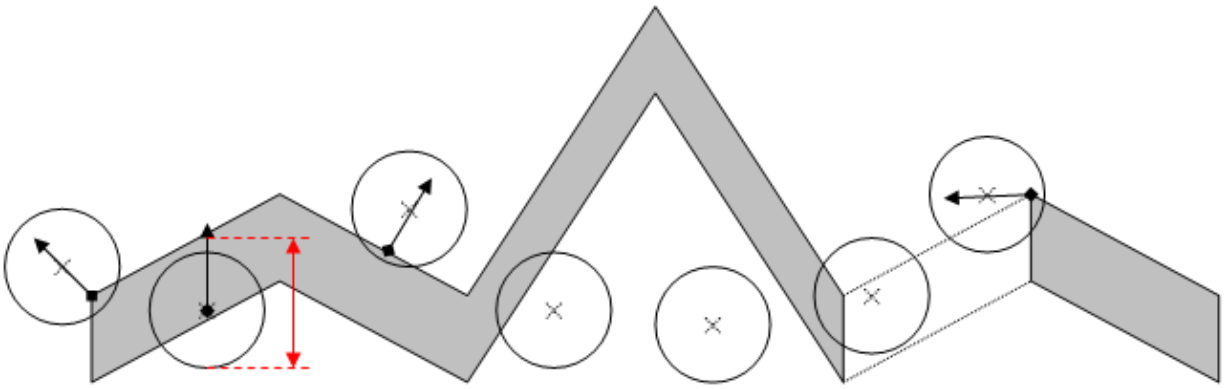
Contact Generation

Heightfield shapes support collision detection and triggers with the following shapes:

- NxSphereShape
- NxCapsuleShape
- NxBoxShape
- NxConvexShape
- NxWheelShape

The heightfield shape generates contacts when the appropriate shape is within its vertical extent. Contact normals are generated only with respect to the surfaces of the heightfield. For example, if an object intersects the side of the heightfield but not the surface, then only a contact pointing towards the surface will be generated. Consider this example of a simple heightfield with a hole (the white area):





NOTE: An optimization in the sphere and capsule contact generation causes the heightfield not to generate contacts until the center of the sphere/capsule is in contact with the vertical extent of the heightfield. The same optimization also leads to spheres and capsules only generating contact points against edges if their centers are above the heightfield surface (see the examples above).

Contact generation with triangle edges at the terrain's borders can be disabled using the NX_HF_NO_BOUNDARY_EDGES flag, allowing more efficient contact generation when there are multiple heightfield shapes arranged so that their edges touch.

When the NX_MESH_SMOOTH_SPHERE_COLLISIONS mesh flag is raised, the contact is the smooth heightfield normal. NOTE: The SDK modifies the position of the contacts for spheres when the NX_MESH_SMOOTH_SPHERE_COLLISIONS flag is specified. This is necessary because keeping the exact position may induce an unrealistic torque on the sphere.

Raycasting and Overlap Testing

The heightfield shape supports the following overlap/raycasting functions (see Overlap Testing for more information):

- NxShape::raycast()
- NxShape::checkOverlapSphere()
- NxShape::checkOverlapAABB()
- NxShape::getTriangle()

The character controller is supported for heightfields, since it relies on the following:

- NxShape::overlapAABBTriangles()
- NxShape::getTriangle()

API Reference

- NxHeightfield
- NxHeightfieldShape
- NxHeightfieldDesc
- NxHeightfieldShapeDesc
- NxScene
- NxShape

Background

There are two basic ways to create vehicles with the PhysX SDK:

1. Spherical wheel bodies using joints to attach them to a rigid body representing the chassis, eventually adding more rigid bodies that represent the pieces of the steering and suspension geometry. The advantage of this approach are the complex steering and suspension geometries. The disadvantages are a lack of a sophisticated tire friction model, and that simulating several rigid bodies and joints can be costly and inaccurate (joint error).

2. Raycast cars composed of a single rigid body with special ground contacts that simulate the behavior of a wheel. The wheels themselves are not modeled as separate rigid bodies, which makes for a much more lightweight simulation, and no joint error. The disadvantage is that the mass of the wheels and their inertia do not influence the motion of the car automatically, so all the effects that result from this need to be added as a post process.

Previous Raycast Wheel Support

Previously, raycast cars were supported by extending the material class with a spring contact mode. This made it possible to have the contact act like a car suspension (a spring + damper element). In theory this feature could be used for other applications besides vehicle suspensions, but so far nothing has been attempted.

The wheels' positions were determined by adding special capsule shapes to the vehicle body, with their y axis pointing downward toward the ground. The shape was flagged with NX_SWEPT_SHAPE to change its behavior from a capsule that would generate contacts with anything it touched, to a ray that disregarded the thickness of the capsule and shot a ray along the capsule's y axis. A single contact point was created between the ray and any shape it hit.

The other aspects of wheel simulation - steering, axle torque, and tire friction - were not supported by any special features.

Tire friction was roughly approximated with a skates model. Here at each wheel, contact anisotropic friction was used to make the contact slide easily in the direction of rolling, and slide less easily in the orthogonal (lateral) direction, in which the wheel does not roll. In general, the frequent and stiff change between static and dynamic friction made for problems.

The vehicle was steered by rotating the capsule shape relative to the car body around the vertical y axis, so that the x and z axes that defined the longitudal and lateral directions for the purpose of tire friction shifted. This made the direction of easy sliding change, and turned the vehicle.

The vehicle was powered by applying a force along the direction of motion, similar to how a horse-drawn sled is pulled.

User side read back of wheel state (such as whether ground contact exists, and what the ground normal force is) was not directly supported; rather, the user had to parse to the contact stream between the car body and other actors, and extract this information manually.

There were also some bugs in the general anisotropic friction implementation that only became apparent when the axes of anisotropy were changed by the user, such as during steering. In response to steering, the car sometimes rotated even when stationary.

Another problem is that the suspension axis and the contact normal are one and the same. This is appropriate on flat terrain where these two directions are aligned in reality. However, when a wheel is on sloping terrain, yet the car body is horizontal, the suspension axis should still be vertical, while the contact normal should be normal to the terrain. The simulation always uses the terrain normal for both. This leads to the suspension spring pushing the car somewhat horizontally, an undesirable effect.

Finally, because the shapes hit by the raycast all produced contacts, and all these contacts had normal forces and friction computed for them, the simulation was not very realistic when the vehicle rolled over several thin shapes in close proximity.

For legacy support, the old capsule wheel shape and springy material setup can now be simulated using a new flag - NX_WF_EMULATE_LEGACY_WHEEL, read more about mapping wheel shape settings.

Improvements / Bug Fixes to the Old Raycast Wheel Material Approach

The problem with anisotropic friction mentioned above has been fixed. Now a stationary car should never rotate in response to steering alone.

API Reference

- NxWheelShape
- NxWheelShapeDesc
- NxWheelContactData
- NxTireFunctionDesc

Features

While some improvements have fixed the most immediate issues with the old raycast wheel model, the problems with simulation fidelity and lacking features remain. One important customer driven requirement to easily query wheel information made it necessary to introduce an API object which had a 1:1 correspondence to each simulated wheel. Previously, the capsule shape object was in 1:1 correspondence with the wheel, making it possible to share between wheels and other shapes.

To improve the general usability of the wheel API, the NxWheelShape was introduced, in addition to NxWheelShapeDesc.

NxWheelShape has a simple but extensible interface that permits simple saving and loading of the entire wheel state, as well as the modification of any properties during the simulation. It is also possible to query the contact status directly from the wheel shape. This provides a complete set of contact properties, including the local surface material of the other shape and the currently acting tire forces.

Similarly, it is possible to disable ALL friction using the NX_MF_DISABLE_FRICTION flag. This, however, is more in support of the new raycast wheel.

The main objective is to introduce a more advanced tire slip based friction model than this standard textbook approach. Many references are available to explain the textbook version (for example "Race Car Vehicle Dynamics" by Milliken). Tire slip based friction separates the overall friction force into a longitudinal component (responsible for the direction of rolling, braking and acceleration) and a lateral component (orthogonal to rolling, responsible for keeping the car properly oriented). In both directions it is first determined how much the tire is slipping (i.e., what the speed difference is between the rubber and the road). Then this slip value is mapped to a tire force by means of a lookup table filled with empirical tire data. The predominant property of real tires is that for a certain low slip, they can exert high tire forces as the rubber compensates for the slip by stretching, and maintains a static friction of sorts. Later, when the slip gets really high, the forces are reduced as the tire really starts to slide/spin, and dynamic friction starts to act.

Rather than using an explicit lookup table associated with a certain storage and performance overhead, NxWheelShape introduces NxTireFunctionDesc, which contains the coefficient parameters of two cubic Hermite splines, with which the tire force function is approximated. The coefficients are very intuitive as they are at the extrema of the function.

To compute the tire slip in the longitudinal (rolling) direction, it is necessary to know the tire's angular velocity. Initially the customer wanted the user to keep track of this velocity and pass the current value to the wheel shape at every simulation step; however, this turned out to be unsatisfactory, as it is very important to be able to simulate slipless rolling. In this case, the speed of the wheel must correspond almost exactly to the ground speed. As the ground speed changes every sub step due to the progressing simulation, if the user only makes the wheel speed respond every step, then for several sub steps the wheel may look like it is slipping when in fact the update is simply not happening in sync.

To remedy this, the wheel shape contains the update of the wheel speed by integrating the sum torque acting on the wheel, which is made up of the sum motor, braking, and ground induced torques. The ground induced torque is equivalent to the longitudinal tire force that is computed internally. The motor and brake torques are user inputs, and are treated separately, because a motor torque opposing the wheel’s rotation will make it reverse rotation, while a very large sustained brake torque will simply stop the wheel and lock it in place.

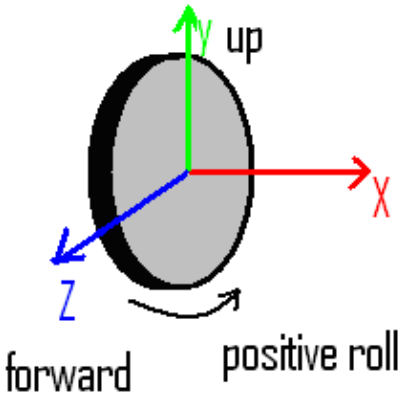
By providing a motor and brake axle torque input, it is no longer necessary to apply an explicit force simulating these effects to the car chassis, as was the case for the previous solution.

Further parameters necessitated by the simulation of the wheel rotation are the wheel radius (to map the axle angular velocity to a radial velocity at the contact patch), and the wheel mass (to map the axle torques to

accelerations).

For improved convenience, it is no longer needed to rotate the wheel shape to steer the vehicle. Instead, a steering angle parameter can be set. Also, the ray direction is defined to cast along the negative y axis, which means that simulations using y as the up axis no longer need to turn the shape on its head to properly orient the wheel.

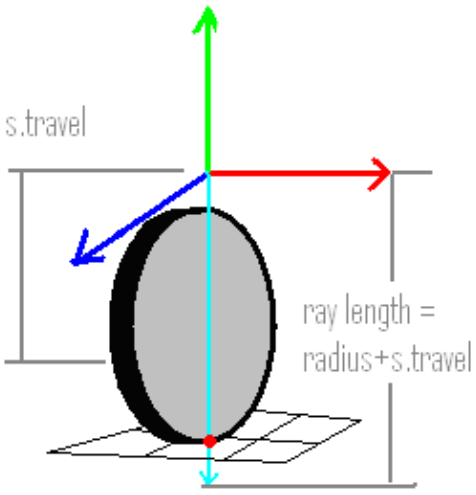
The below diagram illustrates the wheel shape axis conventions:



To simplify debugging, the wheel shape provides visualization of the wheel as part of the shape visualization mode.

The wheel suspension parameters are identical to the ones used by the old material based system, but are now stored directly in the wheel shape for convenience.

A new suspension travel parameter determines the length of the raycast, as seen in the figure below:



The meaning of the suspension NxSpringDesc::targetValue has also changed. It is now decoupled from the suspension travel, and is a value between 0 and 1 that determines at what % of whole travel the rest length of the spring should be, 0 mapping to the default of full extension along the suspension travel.

Another new feature is that when the suspension is completely compressed, a hard contact is generated in addition to the spring contact to make sure that a large force doesn't push the car with soft springs through the ground. This is a realistic simulation of a suspension that has bumped up against its hard stops.

Finally, the need to use addForce() to apply tire forces has been removed, as the explicit integration of such sustained forces, which are part of simulation loops, can lead to instability. The high tire forces acting at low velocities quickly become unstable because the forces not only stop the body, but depending on the size of the simulation step, can reverse its direction and lead to oscillation.

Instead, there is now a way to map the tire forces to a constraint that is passed into a solver. The results are unconditionally stable, even for very high tire forces. The mapping of the tire force to a constraint gave rise to another parameter, the stiffnessFactor of the tire descriptor. This parameter has no good physical interpretation and has simply to do with mapping the tire forces to a spring constant that the solver can work with.

Because the wheel implementation sorts through contacts along the ray and only takes the contact closest to the ray origin, multiple simultaneous contacts are not an issue.

API Reference

- NxWheelShape
- NxWheelShapeDesc
- NxWheelContactData
- NxTireFunctionDesc

Simulation Diagram

Keep the following points in mind regarding the simulation:

- The wheels are always integrated when the car's simulation group is not asleep, even if they don't have ground contact.
- Constraints for a wheel are only generated when they have ground contact.
- Tire forces are mapped to tire constraints by spring constraints to achieve slipless rolling in the longitudinal direction and no sliding in the lateral direction. The spring constant for these constraints is measured using stiffnessFactor times the tire force. A lower tire force results in a lower, thus softened, spring constraint.

API Reference

- NxWheelShape
- NxWheelShapeDesc
- NxWheelContactData
- NxTireFunctionDesc

API Usage

An example application of the new NxWheelShape is provided with the new RaycastCarController2 class in the Viewer (available to customers with a source release).

Create an NxWheelShape for each wheel. All the initial values for the wheel should be added in the wheel descriptor. In particular, pay attention to the wheel flags.

Wheels can now share a single material; specify the NX_MF_DISABLE_FRICTION flag so that the regular friction model is disabled. If this flag is forgotten, both regular and tire friction constraints are generated - a rather bizarre setup and not recommended. NOTE: It does not make sense to re-enable friction. To stop the car from moving when it is parked, manipulate the wheel's longitudinal and lateral friction as explained next.

It is currently mandatory to specify the NX_WF_INPUT_LNG_SLIPVELOCITY flag or no longitudinal forces are computed at all. The code path for using slip ratios, which would be the alternative, is not yet implemented. Specifying the NX_WF_INPUT_LAT_SLIPVELOCITY flag is a simpler way to compute the lateral slip than the alternative slip angle, but both are available. In general, it is better to use NX_WF_INPUT_LAT_SLIPVELOCITY if the results are adequate.

Once the wheels are created, take control of the vehicle by using setMotorTorque(), setBrakeTorque() and setSteerAngle().

RaycastCarController2 gives an example of how to plot the friction ellipse of each tire based on contact information retrieved using getContact().

API Reference

- NxWheelShape
- NxWheelShapeDesc
- NxWheelContactData
- NxTireFunctionDesc

Caveats

For the best simulation, keep the following criteria in mind:

- When using a digital input for steering (e.g., the keyboard) it is important to drastically scale down the maximum steering angle as the car's speed increases. At high speed, a steering angle of 1 degree will suffice.

- Avoid sudden changes in steering angle. The input should be smoothed, it is best to not change the steering angle by more than 1 degree per time step. NOTE: The input angle is specified in radians.

- As this wheel shape only provides wheel simulation, the user must simulate the engine and transmission by other means. Most importantly, the engine torque should decrease with increasing engine speed, otherwise the car will be able to accelerate indefinitely. A model for increasing air resistance at increasing speed would be beneficial.

- Do not use negative engine torque for braking – only use it for driving in reverse. If you do not have a separate brake input for a simple car game, detect if the user input is opposing the direction of travel, and apply a constant brake torque until the car comes to a stop. Then apply a negative engine torque to start moving in reverse.

- If problems occur with the suspension direction, try the NX_WF_WHEEL_AXIS_CONTACT_NORMAL on and then off to see the difference between the two settings.

- All shapes that are stabbed by the ray are added to the contact stream, even though all but one of these are ignored for the purpose of tire friction. Because the tire model has its own way of reporting contact and friction forces, these are not channeled into the contact stream. All the wheel contacts there will report 0 for all forces. Keep that in mind when looking at the contact stream.

- When a vehicle accelerates, the motor torque applied to the wheels results in an equal and opposite torque. This effect is responsible for pitching the chassis backwards when accelerating. However, with the raycast car model this force is not explicitly simulated and must be added by the user as a post process.
- Collision detection with wheel shapes are not hardware-accelerated.

API Reference

- NxWheelShape
- NxWheelShapeDesc
- NxWheelContactData
- NxTireFunctionDesc

Future Extensions

In the future, it would be useful to model the aligning torque of a wheel in response to tire friction. The user would supply a steering torque rather than a steering angle, and the simulation would be responsible for computing and applying a torque that would act to center the steering when the vehicle is moving.

The slip ratio mode is not implemented when the NX_WF_INPUT_LNG_SLIPVELOCITY flag is not specified. The problem is that the formula for this quantity breaks down at low velocities because there is a division by the speed. A decision will be made at some point that either 1) the slipVelocity mode is adequate, so the above flag will be removed from the API, or 2) computing the slip ratio to work for very low velocities must be created.

Currently, the suspension axis and contact axis, though different in real life, are simulated in the same direction. This problem needs a fix in the near future.

API Reference

- NxWheelShape
- NxWheelShapeDesc
- NxWheelContactData
- NxTireFunctionDesc

Sweep API

The PhysX Sweep API allows you to take a shape and project it along a line in space, detecting the place or places where it intersects other shapes in the scene. The principle is similar to raycasting; the difference is that a raycast projects only a single point through space, producing a line, whereas the Sweep API allows you to use more complex shapes. To visualize this imagine a sphere swept along a straight line, it becomes a capsule.

When performing a sweep the user specifies a motion vector, this specifies the direction and magnitude of the path the objects is swept over. The API will in turn return a "time of impact" for the sweep, this is the fraction of the motion vector that the object can sweep until it touches another object. Along with this a contact point and normal are provided, the contact point is the point(or one of the points) where the objects touch at the time of impact.

A sweep can be performed using the following entities:

- Box
- Capsule
- Actor

Note: A sphere shape sweep can be performed using a capsule sweep with a height of 0.

Results from the sweep test are available through the NxSweepQueryHit struct:

```
struct NxSweepQueryHit
{
    NxF32           t;              //!< Distance to hit
    NxShape*        hitShape;       //!< Hit shape
    NxShape*        sweepShape;     //!< Only nonzero when using NxActor::linearSweep.
                                    //!< Shape from NxActor that hits the hitShape.
    void*           userData;       //!< User-defined data
    NxU32           internalFaceID; //!< ID of touched triangle (internal)
    NxU32           faceID;         //!< ID of touched triangle (external)
    NxVec3          point;          //!< World-space impact point
    NxVec3          normal;         //!< World-space impact normal
};
```

NOTE: The internal and external faceIDs refer to the triangle index of the un-cooked and cooked mesh respectively.

Sweep Cache

A sweep cache can be used to speed up many similar queries. For example performing a raycast along an NPCs line of sight each many times. The sweep cache works by storing a list of objects which were close to the origin of the sweep. Thus on the next sweep with the cache the closest objects can be tested first culling many other objects very quickly. In addition the sweep cache may store versions of the geometry in a format which is optimal for sweep testing.

```
static NxSweepCache* cache = gScene->createSweepCache();

NxU32 nb = BoxActor->linearSweep(dir*dist, flags, NULL, 100, results, NULL, cache);
```

As an additional optimization the sweep cache can be restricted to a particular bounding volume. (So that only

shapes occupying this volume are stored in the sweep cache)

```
void NxSweepCache::setVolume(const NxBox& box);
```

NOTE: When performing sweep tests from multiple threads the sweep cache cannot be shared as access to the cache is not serialized. In other words there should be an independent sweep cache for each thread which needs a cache.

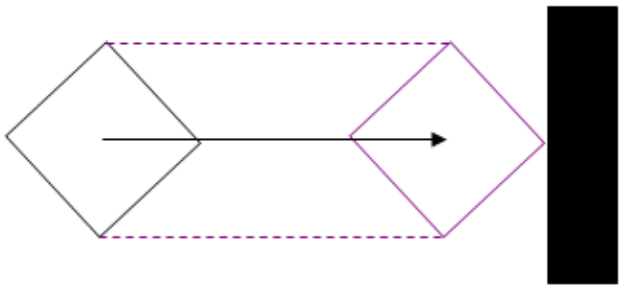
Caveats:

- Sweep testing does not work against NxWheelShape.
- Trigger shapes are excluded from sweep tests.
- All sweep tests are currently performed synchronously.
- It is undefined what happens when the swept volume already overlaps with something at the start of the query.

API Reference

- NxScene
- NxActor

Box sweeps



You can perform a sweep with a box shape by using the NxScene::linearOBBSweep() method. The results are returned via an NxSweepQueryHit structure.

```
NxU32 linearOBBSweep(const NxBox& worldBox, const NxVec3& motion, NxU32 flags, void* userData
                     NxU32 nbShapes, NxSweepQueryHit* shapes, NxUserEntityReport<NxSweepQuery
                     NxU32 activeGroups=0xffffffff, const NxGroupsMask* groupsMask=NULL);
```

- worldBox - A structure describing the box to sweep, in world space.
- motion - The motion vector for the sweep.
- flags
  - NX_SF_STATICS: Sweep against static shapes
  - NX_SF_DYNAMICS: Sweep against dynamic shapes
  - NX_SF_ALL_HITS: Report all hits rather than just closest hit
- userData - A user pointer which is provided to the user through the callback.
- nbShapes - Number of shapes which the shapes buffer can hold.
- shapes - A user supplied buffer to store touched shapes.
- callback - A callback which allows the user to receive results from the query.
- activeGroups - Mask used to filter out shapes based on the shapes group(NxShape::setGroup())
- groupsMask - Group mask used for filtering with the bitwise filtering system.

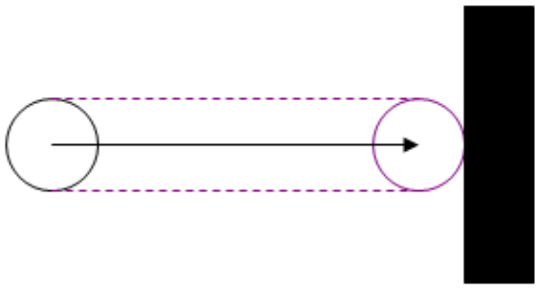
Example

```
NxBox testBox;

testBox.center      = NxVec3(0.0f, 100.0f, 0.0f);
testBox.extents     = NxVec3(10.0f, 10.0f, 10.0f);

float dist = 100.0f;
NxVec3 dir(0.0f, -1.0f, 0.0f);

NxU32 flags = NX_SF_STATICS|NX_SF_DYNAMICS;

NxSweepQueryHit result;

gScene->linearOBBSweep(testBox, dir*dist, flags, NULL, 1, &result, NULL);

NxShape *hitShape = result.hitShape;
NxVec3 impactPoint = result.point;
```

API Reference

- NxBox
- NxScene
- NxShape

Capsule sweeps



You can perform a sweep with a capsule shape by using the NxScene::linearCapsuleSweep() method. The results are returned via an NxSweepQueryHit structure.

```
NxU32 linearCapsuleSweep(const NxCapsule& worldCapsule, const NxVec3& motion, NxU32 flags, vo
                         NxU32 nbShapes, NxSweepQueryHit* shapes, NxUserEntityReport<NxSweepQ
                         NxU32 activeGroups=0xffffffff, const NxGroupsMask* groupsMask=NULL);
```

- worldCapsule - A structure describing the capsule to sweep, in world space.
- motion - The motion vector for the sweep.
- flags
  - NX_SF_STATICS: Sweep against static shapes
  - NX_SF_DYNAMICS: Sweep against dynamic shapes
  - NX_SF_ALL_HITS: Report all hits rather than just closest hit
- userData - A user pointer which is provided to the user through the callback.
- nbShapes - Number of shapes which the shapes buffer can hold.
- shapes - A user supplied buffer to store touched shapes.
- callback - A callback which allows the user to receive results from the query.
- activeGroups - Mask used to filter out shapes based on the shapes group(NxShape::setGroup())
- groupsMask - Group mask used for filtering with the bitwise filtering system.

Example

```
NxCapsule testCapsule;
testCapsule.radius  = 5.0f;
testCapsule.p0              = NxVec3(0.0f, 100.0f, 0.0f);
testCapsule.p1              = NxVec3(10.0f, 70.0f, 0.0f);

float dist = 100.0f;
NxVec3 dir(0.0f, -1.0f, 0.0f);

NxU32 flags = NX_SF_STATICS|NX_SF_DYNAMICS;

NxSweepQueryHit result;

gScene->linearCapsuleSweep(testCapsule, dir*dist, flags, NULL, 1, &result, NULL);

NxShape *hitShape = result.hitShape;
NxVec3 impactPoint = result.point;
```

API Reference

- NxCapsule
- NxScene
- NxShape

Actor sweeps

You can perform a sweep with an entire actor, including all its constituent shapes, by using the NxActor::linearSweep() method. The results are returned via an NxSweepQueryHit structure.

In the current implementation actor sweep tests are only conducted using the actor shapes which are:

- Capsules
- Spheres
- Boxes

If a sweep is performed with an actor which contains, for example a convex shape, then the convex shape will be ignored and no hits for this shape reported. Also note that trigger shapes in the actor will be excluded from the test.

```
NxU32 linearSweep(const NxVec3& motion, NxU32 flags, void* userData,
                  NxU32 nbShapes, NxSweepQueryHit* shapes, NxUserEntityReport<NxSweepQueryHit
                  const NxSweepCache* sweepCache=NULL);
```

- motion - The motion vector for the sweep.
- flags
  - NX_SF_STATICS: Sweep against static shapes
  - NX_SF_DYNAMICS: Sweep against dynamic shapes
  - NX_SF_ALL_HITS: Report all hits rather than just closest hit
- userData - A user pointer which is provided to the user through the callback.
- nbShapes - Number of shapes which the shapes buffer can hold.
- shapes - A user supplied buffer to store touched shapes.
- callback - A callback which allows the user to receive results from the query.
- sweepCache - A cache object used to accelerate sweep queries. See Sweep API.

Example

```
NxActor *testactor;

...

float dist = 100.0f;
NxVec3 dir(0.0f, -1.0f, 0.0f);
NxU32 flags = NX_SF_STATICS|NX_SF_DYNAMICS;
NxSweepQueryHit result;
testActor->linearSweep(dir*dist, flags, NULL, 1, &result, NULL);
NxShape *hitShape = result.hitShape; // Shape in scene
NxVec3 impactPoint = result.point;
NxShape *sweepShape = result.sweepShape; // Shape in actor
```

API Reference

- NxActor
- NxScene

Triangle Mesh

A triangle mesh is helpful in creating complex shapes that are too complex to represent with a bunch of boxes and spheres, or a better option than spending time to tediously fit these bounding boxes.

There are four basic ways that triangle meshes are provided to the SDK:

- convex mesh
- heightfield mesh
- arbitrary open mesh
- arbitrary closed mesh with a pmap (*NOTE: PMaps have been deprecated.*)

Each of these mesh types have the same API interface, but the collision detection codes they use when in contact with different shape types may be different. For a list of collision interactions applicable to mesh shapes see Collision Interactions.

Collision detection is less robust against triangle meshes than between primitives such as spheres or boxes. In particular, a primitive that, due to excessive velocity or imprudent insertion, ends up with its center inside the triangles of a triangle mesh will *not* register a collision. Continuous collision detection (CCD) can help with this problem.

The common practice of storing a single triangle mesh and then creating several differently posed instances of it is mirrored in the SDK by creating an NxTriangleMesh object and then a number of NxTriangleMeshShapes which reference it. The NxTriangleMesh shapes can be attached to actors and transformed individually.

To create a triangle mesh object it is necessary to first cook the data. See the following section on Cooking for more information.

API Reference

- NxTriangleMesh
- NxConvexMesh
- NxConvexShape
- NxTriangleMeshShape

Cooking

To create a triangle mesh object, it is necessary to first cook the mesh data into a form which allows the SDK to perform efficient collision detection. This process of cooking can be quite slow, so it is best to perform off-line or as a one time only operation. Once the data has been cooked, the application can save the generated stream to file for fast loading later.

When working with the PPU there are additional considerations. See Mesh Paging for more information.

The application uses the NxCooking library to create a cooked mesh. First, initialize the library by calling the following commands:

```
static NxCookingInterface *gCooking = NxGetCookingLib(NX_PHYSICS_SDK_VERSION);

gCooking->NxInitCooking();
```

Next, fill in an NxTriangleMeshDesc to describe the mesh and pass it to NxCookTriangleMesh(), which will process the mesh description and write the results to a user supplied stream. The stream will then write the cooked data to a file or a temporary memory buffer as shown below:

```
    //Build physical model
    NxTriangleMeshDesc bunnyDesc;

    bunnyDesc.numVertices = BUNNY_NBVERTICES;
    bunnyDesc.numTriangles = BUNNY_NBFACES;
    bunnyDesc.pointStrideBytes = sizeof(NxVec3);
    bunnyDesc.triangleStrideBytes = 3*sizeof(NxU32);
    bunnyDesc.points = gBunnyVertices;
    bunnyDesc.triangles = gBunnyTriangles;
    bunnyDesc.flags = 0;

    gCooking->NxCookTriangleMesh(bunnyDesc, UserStream("c:\\cooked.bin", false));
```

Later, when the application is ready to create the triangle mesh, it can pass the cooked stream to the SDK to load the following:

```
        bunnyTriangleMesh = gPhysicsSDK->createTriangleMesh(UserStream("c:\\tmp.bin", true));
```

If supplying a convex mesh to be cooked, use NxCookConvexMesh() instead.

NOTE: The cooking process is sensitive to numerical precision issues, and the result of the cooking can vary depending on the FPU mode. You should make sure that you always use the same FPU precision and rounding modes when cooking meshes, e.g. if you want the result to be consistent over multiple platforms.

NOTE: NxStream has append semantics, so when writing the new cooked data, it will be added to the end of the stream. The file pointer or memory stream must be manually reset before reusing the stream object (e.g., call MemoryWriteBuffer::clear() in the sample code).

In the above sample, the flag's member is zero, but it can be used to indicate special formatting, shown below:

- NX_MF_FLIPNORMALS – Triangles have a winding which is opposite to the default.
- NX_MF_16_BIT_INDICES – The triangle's vertex indices are 16 instead of 32 bits.
- NX_MF_CONVEX – The mesh is convex which permits certain optimizations.

The triangle's normals are not passed explicitly, and are assumed to be:

(v1 – v0) x (v2 – v0)

where v0, v1, and v2 are the vertices of the triangle, indexed in that order. This is the same as a counter-clockwise winding in a right handed coordinate system or
alternatively a clockwise winding order in a left handed coordinate system.

If the triangles are stored in another way, set the NX_MF_FLIPNORMALS flag as shown below:

```
meshDesc.flags |= NX_MF_FLIPNORMALS;
```

Notes on meshes:

- Be sure that you define face normals as facing in the direction you intend. Collision detection will only work correctly between shapes approaching the mesh from the outside, i.e., from the direction in which the face normals point.
- Do not duplicate identical vertices. If two triangles are sharing a vertex, this vertex should only occur once in the vertex list, and both triangles should index it in the index list. If you create two copies of the vertex, the collision detection code won't know that it is actually the same vertex, which leads to decreased performance and unreliable results.
- Avoid t-joints and non-manifold edges for the same reason. NOTE: A t-joint is a triangle vertex that is placed right on top of another triangle's edge, but this second triangle is not split into two triangles at the vertex. A non-manifold edge is an edge (a pair of vertices) that is referenced by more than two triangles.
- When a mesh is being cooked, its mass and inertia tensor is also computed. The inertia tensor computation uses triangles winding to tell which side of a triangle is solid. For this reason, improper winding may lead to a negative mass. An open mesh will probably give an errant mass, center of mass, and inertia tensor result.
- It is possible to assign a different material to each triangle in a mesh. See the Materials section for details.

Shape Creation

The above tasks complete, it is time to create instances of the bunny mesh. These are the actual shape objects that can be associated with an actor. The descriptor of the triangle mesh shape needs to have a reference to the triangle mesh object that is to be instanced:

```
NxTriangleMeshShapeDesc bunnyShapeDesc;
bunnyShapeDesc.meshData= bunnyTriangleMesh;

NxBodyDesc bodyDesc;
NxActorDesc actorDesc;

actorDesc.shapes.pushBack(&bunnyShapeDesc);
actorDesc.body= &bodyDesc;
actorDesc.density= 1.0f;

NxActor* newActor = gScene->createActor(actorDesc);
```

Samples

Sample Convex

API Reference

- NxTriangleMesh
- NxConvexMesh
- NxTriangleMeshDesc
- NxConvexMeshDesc
- NxTriangleMeshShape
- NxConvexShape

Heightfield Meshes

A heightfield is like a bumpy plane. It is also infinite in two dimensions, and defines an elevation along the third. All values below the elevation (or a user definable depth) are treated as being inside the heightfield volume. It is possible for the heightfield triangulation to be irregular. The only difference between a heightfield mesh and a general triangle mesh is that a heightfield is not a closed shape, but rather the boundary of a volume that extends downward for a certain distance. This way, fast moving objects will not fall through the terrain if traveling across its surface in a single time step as they would in a general triangle mesh.

Provide the triangles for the heightfield as in an arbitrary triangle mesh:

```
NxTriangleMeshDesc terrainDesc;

terrainDesc.numVertices= TERRAIN_NB_VERTS;
terrainDesc.numTriangles= TERRAIN_NB_FACES;
terrainDesc.pointStrideBytes= sizeof(NxVec3);
terrainDesc.triangleStrideBytes= 3*sizeof(NxU32);
terrainDesc.points= gTerrainVerts;
terrainDesc.triangles= gTerrainFaces;
```

Heightfield geometry must be flat in that the projections of all triangles onto the heightfield plane must be disjointed. (If the heightfield vertical axis is y, the heightfield plane is spanned by x and z). Set which local (vertex-space) axis should be the height axis, and how far along this axis the implicit heightfield volume should extend:

```
terrainDesc.heightFieldVerticalAxis= NX_Y;
terrainDesc.heightFieldVerticalExtent= -1000.0f;
```

Objects that are under the surface of the heightfield but above this cutoff are treated as colliding with the heightfield. The heightFieldVerticalExtent has to be outside of the mesh's vertex coordinate range along the heightFieldVerticalAxis. This may be set to a positive value, in which case the extent will be cast along the opposite side of the heightfield. A smaller finite value for the extent can be set if space beneath the heightfield is desired, such as a cave.

In the code sample above, NX_Y axis was used, other options being NX_X and NX_Z. Using a negative value for the extent (also shown) provides the necessary down direction, as opposed to making the extent positive which would simulate up. This complete, the remaining process follows the exact steps of a triangle mesh (i.e., the mesh data needs to be cooked first (see the Cooking section), then a triangle mesh is created from the cooked data).

NOTE:

- In most circumstances, the Heightfield Shape is preferable to a mesh-based heightfield when developing for the PhysX SDK 2.4 and above. The heightfield shape will use less memory and perform better than a heightfield mesh. The tessellation of triangles for a heightfield shape is assumed to be regular, unlike a heightfield mesh.
- Collisions with Heightfields are not hardware-accelerated.

Samples

Sample Terrain

API Reference

- NxTriangleMesh
- NxTriangleMeshDesc
- NxHeightfieldShape

Convex Meshes

The convex mesh feature is meant to satisfy the need of using shapes that are more complex than a box, yet simpler and more efficient than the general mesh-mesh collision models. The only requirement is that they should be convex. If this is the case, set the appropriate flags in the mesh descriptor when cooking and make sure to use the NxCookConvexMesh() instead of the NxCookTriangleMesh() function.

NOTE: The maximum number of polygons for a convex mesh is limited to 256.

Convex Hull Generation

The SDK provides automatic convex hull generation when cooking, i.e., the user can specify a point cloud from which the SDK computes a convex hull. To enable this functionality, specify the NX_CF_COMPUTE_CONVEX flag in the mesh descriptor.

The convex hull can be inflated using the skinWidth parameter of NxCookingParams. When a convex hull is inflated, its boundary is expanded beyond the point cloud by skinWidth.

NOTE: The hull generation may fail for input point sets which contain certain degenerate configurations. If you feel this to be an issue, please try not to provide input that contains two or more points close to each other, or in other ways would generate very thin triangles. NOTE: As of 2.5 the legacy cooker has been deprecated since its output is not compatible with the hardware collision code.

Example

```
//Create descriptor for convex mesh
NxConvexMeshDesc convexDesc;
convexDesc.numVertices          = nbVerts;
convexDesc.pointStrideBytes     = sizeof(NxVec3);
convexDesc.points               = verts;
convexDesc.flags                = NX_CF_COMPUTE_CONVEX ;

gCooking->NxInitCooking();

//Cooking from memory
MemoryWriteBuffer buf;

if(gCooking->NxCookConvexMesh(convexDesc, buf))
{
    NxConvexShapeDesc convexShapeDesc;

    convexShapeDesc.meshData = gPhysicsSDK->createConvexMesh(MemoryReadBuffer(buf.data));

    if(convexShapeDesc.meshData)
    {

        NxActorDesc ActorDesc;

        ActorDesc.shapes.pushBack(&convexShapeDesc);
        ActorDesc.body            = &BodyDesc;
        ActorDesc.density        = 10.0f;
        ActorDesc.globalPose.t  = pos;

        gLastActor = gScene->createActor(ActorDesc);
    }
}
```

Providing a Convex Mesh Directly

As an alternative to using the SDK's convex hull generation, the user can directly provide mesh triangles and vertices to create a convex mesh.

NOTE: The skinWidth cooking parameter is ignored in this case and the mesh is not inflated.

Example

```
    //Cook a convex mesh

    NxConvexMeshDesc coneMeshDesc;

    coneMeshDesc.numVertices = CONE_NBVERTICES; //Number of vertices in mesh.
    coneMeshDesc.numTriangles = CONE_NBFACES; //Number of triangles(3 indices per triangle).

    coneMeshDesc.pointStrideBytes = sizeof(NxVec3);//Number of bytes from one vertex to the next.
    coneMeshDesc.triangleStrideBytes = 3*sizeof(NxU32);//Number of bytes from one triangle to the next.

    coneMeshDesc.points = gConeVertices;
    coneMeshDesc.triangles = gConeTriangles;
    coneMeshDesc.flags = 0;

    gCooking->NxCookConvexMesh(coneMeshDesc, UserStream("c:\\cooked.bin", false));
```

Scaling convexes

Sometimes you may wish to create instances of the same convex mesh with different sizes. You can accomplish this by re-cooking a convex mesh using NxScaleCookedConvexMesh:

```
NxConvexMeshDesc convexDesc;
...
MemoryWriteBuffer buf;
if(gCooking->NxCookConvexMesh(convexDesc, buf))
{
        NxConvexShapeDesc convexShapeDesc1;
        convexShapeDesc1.meshData = gPhysicsSDK->createConvexMesh(MemoryReadBuffer(buf.data));

        MemoryWriteBuffer buf2;
        if(NxScaleCookedConvexMesh(MemoryReadBuffer(buf.data), 0.5, buf2)) //Resize the mesh by a factor
        {
                NxConvexShapeDesc convexShapeDesc2;
                convexShapeDesc2.meshData = gPhysicsSDK->createConvexMesh(MemoryReadBuffer(buf2.data));
        }
}
```

Samples

Sample Convex

API Reference

- NxConvexMesh
- NxConvexMeshDesc
- NxPhysicsSDK

Reading Back Mesh Data

In some situations, it is necessary to retrieve points, indices, normals, etc. from an NxTriangleMesh or NxConvexMesh, which can be useful for the following:

- Debugging
- Quick and dirty rendering
- AI
- Creating CCD skeletons

However, the read back process may be expensive because the data is probably not in the desired format.

Do *NOT* attempt to modify the mesh data you receive in this way; the results of such an operation are undefined.

The methods used for reading back mesh data are shown below:

```
NxU32 getSubmeshCount();
NxU32 getCount(NxSubmeshIndex submeshIndex, NxInternalArray intArray);
const void* getBase(NxSubmeshIndex submeshIndex, NxInternalArray intArray);
NxU32 getStride(NxSubmeshIndex submeshIndex, NxInternalArray intArray);
NxInternalFormat getFormat(NxSubmeshIndex submeshIndex, NxInternalArray intArray);
```

To retrieve the number of vertices, etc., use getCount(), and to specify which set of elements to retrieve, use the following members of NxInternalArray:

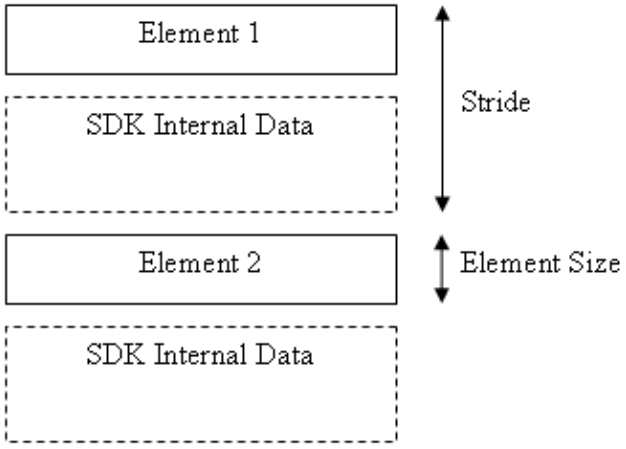
- NX_ARRAY_TRIANGLES
- NX_ARRAY_VERTICES
- NX_ARRAY_NORMALS
- NX_ARRAY_HULL_VERTICES
- NX_ARRAY_HULL_POLYGONS

To get the number of points in a mesh:

```
triangleMesh->getCount(0,NX_ARRAY_VERTICES);
```

NOTE: submeshIndex is not used and should be ignored. The SDK does not currently split the mesh into multiple submeshes.

Next it is necessary to retrieve the stride for the particular element - the number of bytes from the start of one element to the start of the next. This may not be equal to the element's size since the SDK may store additional data between each element.



...

```
NxU32 stride = triangleMesh->getStride(0,NX_ARRAY_VERTICES);
```

Next the user should retrieve the format of the elements. Indexes may be stored as 8, 16 or 32 bits. To do this use the call below:

```
NxInternalFormat format=triangleMesh->getFormat(0,NX_ARRAY_VERTICES);
```

The currently defined values for NxInternalFormat are as follows:

- NX_FORMAT_NODATA - Not available
- NX_FORMAT_FLOAT - 32 bit floating point
- NX_FORMAT_BYTE - 8 bits
- NX_FORMAT_SHORT - 16 bits
- NX_FORMAT_INT - 32 bits

When retrieving triangle indexes, they are stored as three consecutive integers of 8, 16 or 32 bits. Vertices are stored as three consecutive floating point values, as are normals.

Finally, to retrieve a pointer to the data, use getBase().

NOTE: Cooking may change the order of vertices/triangles or otherwise manipulate the mesh. For this reason, do not assume that the layout matches the original un-cooked mesh.

Example

This example shows how to construct a NxSimpleTriangleMesh from a convex mesh. It could be used to create a CCD skeleton. It is fairly limited in that it does not deal with meshes which have more than 1 sub mesh. It also does not handle indices which are 8 bits in size. Notice the use of NxFlexiCopy() to account for the potential difference between the element size and the stride.

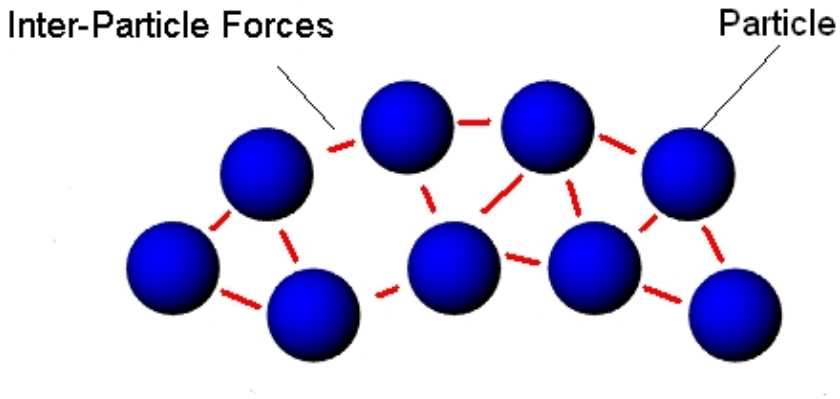
```
NxSimpleTriangleMesh &triMesh;
NxConvexShape *meshShape;

NxConvexMesh &convexMesh=meshShape->getConvexMesh();

int pointCount=convexMesh.getCount(0,NX_ARRAY_VERTICES);
int triCount=convexMesh.getCount(0,NX_ARRAY_TRIANGLES);
```

```
triMesh.points=new NxVec3[pointCount];
triMesh.numVertices=pointCount;
triMesh.pointStrideBytes=sizeof(NxVec3);
triMesh.flags=0;

//Extract triangle points
const void *ptr=convexMesh.getBase(0,NX_ARRAY_VERTICES);
NxU32 stride=convexMesh.getStride(0,NX_ARRAY_VERTICES);

NxFlexiCopy(ptr,(NxVec3 *)triMesh.points,pointCount,sizeof(NxVec3),stride);

//Extract triangle indices
if(triCount>0)
{
    NxInternalFormat format=convexMesh.getFormat(0,NX_ARRAY_TRIANGLES);

    if(format==NX_FORMAT_SHORT)
    {
        triMesh.flags|=NX_MF_16_BIT_INDICES;
        elemSize=sizeof(NxU16)*3;
    }
    else if(format==NX_FORMAT_INT)
        elemSize=sizeof(NxU32)*3;
    else
        NX_ASSERT(!"Unable to parse mesh format");

    triMesh.numTriangles=triCount;
    triMesh.triangles=new NxU8[triCount*elemSize];
    triMesh.triangleStrideBytes=elemSize;

    ptr=convexMesh.getBase(0,NX_ARRAY_TRIANGLES);
    stride=convexMesh.getStride(0,NX_ARRAY_TRIANGLES);

    NxFlexiCopy(ptr,(NxU8 *)triMesh.triangles,triCount,elemSize,stride);
}
```

API Reference

- NxTriangleMesh
- NxConvexMesh

Fluids



Fluids allow the simulation of liquids and gases using a particle system and emitters. A tap or fresh water spring introduce fluid particles into a scene and allow the user to control the manner in which they appear (e.g., the emission rate, the initial velocity of the fluid particles, the deviation (randomness) of the emission angle, etc.).

There are two ways to create fluid particles - either from a previously created layout recreated from a snapshot, or from implicit functions.

The fluid's API also allows for drains, used to remove fluids from a scene. For example, an emitter could be placed at the top of a waterfall with a drain at the bottom.

Another option for removing fluids is to set the lifetime counter for each individual particle. Designed with this setting upon creation, it counts down to zero, at which point it is removed from the simulation. This is especially useful for simulated gaseous effects where the lifetime, in addition to fading the graphical version of the particle, gives the impression that the gas is dissipating.

See Hardware Scenes for details on PhysX hardware support for fluids.

Fluid State

Each fluid has a set of properties which affect its behavior, such as the stiffness of the fluid and its viscosity.

The most important properties are listed below:

- *maxParticles* - maximum number of particles used to simulate the fluid.
- *restParticlesPerMeter* - particle per linear meter resolution, measured when the fluid is in its rest state (relaxed).
- *restDensity* - target density for the fluid (water is about 1000).
- *kernelRadiusMultiplier* - sphere radius of influence for particle interaction.
- *packetSizeMultiplier* - parallelization of the fluid.
- *stiffness* - stiffness of the particle interaction related to the pressure.
- *viscosity* - defines the fluid's viscous behavior.
- *damping* - velocity damping constant, globally applied to each particle.

- *externalAcceleration* - acceleration, applied to all particles at all timesteps.
- *simulationMethod* - defines whether or not particle interactions are considered in the simulation.

For more properties related to interaction with rigid bodies, go to Fluid Interaction with Rigid Bodies. For more information on particle interaction, see Fluid Particle Interaction.

Particle State

Each particle is associated with an individual state described below:

- *Position* - position of the particle.
- *Velocity* - linear velocity of the particle.
- *Density* - density of the fluid in the region surrounding the particle, though the mass of all particles in a fluid is fixed.
- *Life* - time the particles have left before dying (using the life counter, measured in seconds).

See Fluid Creation for how to set particle properties directly when they are created. For more information on how to retrieve particle state when rendering, see Rendering Particles.

API Reference

- NxFluid
- NxFluidEmitter

Fluid Particle Interaction

Fluid simulation can occur in two modes, one called SPH (Smoothed Particle Hydrodynamics) where interparticle forces are considered (e.g. viscous drag) and another simple mode where these forces are not considered. SPH is important when simulating dense collections of particles (e.g., liquids) but is not the best for performance; therefore, simple mode should be used when interparticle forces are not a consideration (e.g., a very thin gas).

Particle interaction is simulated in SPH using a number of force fields associated with each particle. The pressure force field has the effect of pushing particles apart and the viscosity forces have the effect of reducing the relative velocity between particles.

Surface tension is modeled through an attractive force between particles. Inside the fluid, the surface tension forces balance each other out. However, at the surface of the fluid they act in the opposite direction of the surface normal which tends to reduce the curvature of the fluid's surface.

When simulating a fluid, the mass of each particle remains constant; however, the density varies over time due to the fluid's compressibility. The change in density is therefore achieved through the movement of particles. An area of the fluid with high density is caused by a closely packed clump of particles.

Note: Different fluids cannot, at present, interact directly with each other.

Simulation Method

The simulationMethod member of NxFluidDesc controls how the fluid particles will be simulated, either with full particle interaction or as a set of independent particles.

- NX_F_SPH - simulates the fluid, taking into account particle interactions.
- NX_F_NO_PARTICLE_INTERACTION - simulates the fluid without interparticle interactions.
- NX_F_MIXED_MODE - alternates between SPH and simple mode (providing more performance than SPH alone, while maintaining some dense characteristics).

The simulation mode may be adjusted even during simulation using NxFluid::setSimulationMode(). Note that depending on the spatial arrangement of the particles, switching from NX_F_NO_PARTICLE_INTERACTION or NX_F_MIXED_MODE to NX_F_SPH might lead to an unstable simulation state.

Example

```
NxFluidDesc fluidDesc;

fluidDesc.simulationMethod = NX_F_SPH;
```

Kernel Radius Multiplier

The kernel radius multiplier, along with restParticlesPerMeter, controls the radius of influence for each particle.

$$radius = \frac{kernelRadiusMultiplier}{restParticlesPerMeter}$$

Units: [m] = [unit less] / [m^-1]

Example

```
NxFluidDesc fluidDesc;

fluidDesc.kernelRadiusMultiplier = 2.3f;
```

Rest Particles Per Meter

The cubed value of restParticlesPerMeter describes the number of particles in a cubic meter when the fluid is in its rest state. Although the parameter is called restParticlesPerMeter, this does not have to be the case. The user is free to choose whichever units are appropriate for the simulation as long as they are consistent.

Rest particles per meter has an indirect effect on the mass and radius of particles, along with the rest density and kernel radius multiplier.

$$radius = \frac{kernelRadiusMultiplier}{restParticlesPerMeter}$$

$$Mass = \frac{restDensity}{restParticlesPerMeter^3}$$

For radius: [m] = [unit less] / [m^-1]
For mass: [kg] = ([kg m^-3]) / [m^-3] = ([kg m^-3]) * [m^3]

Example

```
NxFluidDesc fluidDesc;

fluidDesc.restParticlesPerMeter= 10.0f;
```

Rest Density

The rest density of a fluid defines the mass of a particle along with the restParticles per meter.

$$Mass = \frac{restDensity}{restParticlesPerMeter^3}$$

Units: [Kg] = [Kg m^-3] / [m^-3]

Example

```
NxFluidDesc fluidDesc;

fluidDesc.restDensity = 1000.0f;
```

Viscosity

Viscosity controls a fluid's thickness. For example, a fluid with a high viscosity will behave like treacle while a fluid with a low viscosity will be more runny like water. The viscosity member scales the viscosity force field which applies force to reduce the relative velocity of particles within the fluid. If a pair of particles are close together and have a high viscosity, then a strong force will be applied to reduce there relative velocity. If they are far apart and have a low viscosity value, then only a small force will be applied to reduce there relative motion.

Reasonable values: 5-300

Example

```
NxFluidDesc fluidDesc;

fluidDesc.viscosity = 22.0f;
```

Stiffness

The stiffness (or gas constant) influences the calculation of the pressure force field. Pressure is calculated from (density-initialDenisty)*stiffness. Low values of stiffness make the fluid more compressible (i.e., springy), while high values make it less compressible.

The stiffness value has a weighty impact on the numerical stability of the simulation; setting very high values will result in instability.

Reasonable values: 1-200

Example

```
NxFluidDesc fluidDesc;

fluidDesc.stiffness = 200.0f;
```

Damping

The damping parameter is applied similarly to the damping parameter for rigid bodies. It is used to reduce the velocity of the particles over time.

Reasonable values: 0-1

Example

```
NxFluidDesc fluidDesc;

fluidDesc.damping = 0.0f;
```

External Acceleration

External acceleration is applied directly to particles. It can be used to counteract the effect of gravity or simulate wind, etc.

Units: [m s^-2]

Example

```
NxFluidDesc fluidDesc;

fluidDesc.externalAcceleration = NxVec3(0.0f,10.0f,0.0f);
```

Motion Limit Multiplier

The motion limit multiplier defines the maximum motion distance relative to the rest spacing of the fluid:

$$maxMotionDistance = \frac{motionLimitMultiplier}{restParticlesPerMeter}$$

This parameter is set to 3.0 * kernelRadiusMultiplier by default.

restParticlesPerMeter is the inverted distance (how many particles per meter) of the fluid in its rest state. Divide the motionLimitMultiplier by restParticlesPerMeter to get the maximum distance a particle can move within a timestep.

This can be shown using the units for the above equation: [m] = [unit less] / [m^-1]

If the motionLimitMultiplier is set higher, the fluid moves faster. If set lower, the static mesh cooked for fluids gets smaller and the collision can be computed more efficiently.

NOTE: The motionLimitMultiplier, kernelRadiusMultiplier, packetSizeMultiplier and restParticlesPerMeter have to be the same for all fluids within a scene to be compatible with one shared mesh for fluid collision. In other words, the parameter needs to be specified during fluid mesh cooking.

Packet Size Multiplier

The packet size multiplier controls the size of packets (groups of fluid particles) which are sent to the PPU. Increasing the packet size multiplier results in larger packets, which results in less overhead associated with managing them. However, this increase will also reduce the opportunities for the fluid to run in parallel. The multiplier can therefore be used to achieve faster fluid simulation by tuning it appropriately. Also, a single fluid packet cannot interact simultaneously with more than 8192 mesh triangles; thus you may need to adjust packet size to correspond to triangle density.

The packet size multiplier must be a power of two.

Flags

Gravity can be disabled using the fluid flags and the NX_FF_DISABLE_GRAVITY flag.

Example

```
NxFluidDesc fluidDesc;
```

```
fluidDesc.flags |= NX_FF_DISABLE_GRAVITY;
```

Samples

Sample Particle Fluid

API Reference

- NxFluid
- NxFluidDesc

---

6 Fluid Particle Interaction

Fluid Creation

To create a fluid, call NxScene::createFluid() with an appropriate descriptor. See Particle Interaction for a detailed description of the parameters affecting fluid behavior.
Directly specify the particles initial position, velocity, etc., or create new particles using an emitter. Pass the initial particle state when creating a fluid using the NxParticleData structure and the NxFluidDesc::initialParticleData member.

If you do not wish to create an initial set of particles, simply set numParticlesPtr in the initialParticleData member to NULL.
You will probably still want to specify output buffers (particleWriteData) for use in e.g. rendering.
If you do specify initial particle data, it is permissible to use the same NxParticleData structure for particleWriteData as well.
Only the following members of initialParticleData are read by the SDK when creating the fluid:

- *bufferPos* - position of each individual particle.
- *bufferVel* - velocity of each particle.
- *bufferLife* - lifetime of each particle.

If these members are NULL, then the SDK uses default values. The remainder of NxParticleData is only written to by the SDK during simulation; see particle rendering for more information.
You may add particles manually to the fluid later, using the NxFluid::addParticles function. There is a limit on how many particles that can be added this way, depending on user settings and the current number of particles in the fluid. The default max limit is 4096 particles per frame, but the exact limit is: Min(NxParameter::NX_CONSTANT_FLUID_MAX_PARTICLES_PER_STEP, NxFluidDesc::maxParticles - *NxParticleData::numParticlesPtr)

NOTE: New in version 2.7.0 is that the addParticles() method no longer buffers particles that could not be created (because the limit had been reached) for creating during the next frame.
NOTE: A new feature in 2.7.0 is that it is now possible to aquire the IDs of particles which have been added immediately, using: NxFluidDesc::particleCreationIdWriteData.
NOTE: Specify the stride for each of the state buffers (e.g., bufferPosByteStride) to allow for data that is not tightly packed or interleaved. Typically sizeof (element) is sufficient for a tightly packed array.

A fluid can be created to run in software rather than, as is the default, on hardware. To do this, simply lower the NX_FF_HARDWARE flag in the fluid descriptor.
This allows you to run fluids on a Windows PC without a PPU. The option is not currently available for other platforms.
Also note that the performance will be considerably worse than on hardware, especially in scenes with large numbers of dynamic objects.

Examples

Specifying Initial Particle State

```
//Set structure to pass particles, and receive them after every simulation step
NxParticleData particles;

particles.numParticlesPtr = &gParticleBufferNum;
particles.bufferPos = &gParticleBuffer[0].x;
particles.bufferPosByteStride = sizeof(NxVec3);
```

```
    //Create a fluid descriptor
    NxFluidDesc fluidDesc;

    fluidDesc.kernelRadiusMultiplier = 2.3f;
    fluidDesc.restParticlesPerMeter = 10.0f;
    fluidDesc.stiffness = 200.0f;
    fluidDesc.viscosity = 22.0f;
    fluidDesc.restDensity = 1000.0f;
    fluidDesc.damping = 0.0f;
    fluidDesc.simulationMethod = NX_F_SPH;
    fluidDesc.initialParticleData = particles;
    fluidDesc.particlesWriteData = particles;

    gFluid = gScene->createFluid(fluidDesc);
```

Particles Created Later with an Emitter

```
//Create fluid
    NxFluidDesc fluidDesc;
    fluidDesc.setToDefault();

    fluidDesc.simulationMethod = NX_F_SPH;

    fluidDesc.maxParticles = MAX_PARTICLES;
    fluidDesc.restParticlesPerMeter = 50;
    fluidDesc.stiffness = 1;
    fluidDesc.viscosity = 6;

    gParticles = new NxVec3[fluidDesc.maxParticles];

    fluidDesc.particlesWriteData.bufferPos = &gParticles[0].x;
    fluidDesc.particlesWriteData.bufferPosByteStride = sizeof(NxVec3);
    fluidDesc.particlesWriteData.numParticlesPtr = &gNumParticles;

    gScene->createFluid(fluidDesc);
```

Fluid Memory Sharing

A new feature in 2.7.0 is that if you create more than four fluids with the exact same NxFluidDesc::maxParticles setting, the PPU starts to share the memory between those fluids. This makes it possible to create more fluids, since the memory sharing frees up memory on the PPU.
Even though it is now possible to have more fluids on the PPU at the same time, it is a good idea to only simulate the fluids that are in view of the player, using the NX_FF_ENABLE flag.

A good use case would be:
A game level with 40 fluids, e.g. 20 with a maxParticles of M and one with a maxParticles of N, where not all of them are visible at the same time.
Note that only fluids within the same scene can share memory.

Samples

Sample Particle Fluid

API Reference

- NxFluid
- NxFluidDesc
- NxScene

- NxParticleData

Fluid Emitters

Fluid emitters provide an alternative to directly creating fluid particles. They allow things such as taps, hose nozzles and even bleeding wounds to be simulated.

There are two main modes of operation for an emitter:

- *Constant pressure*. In this case the state of the surrounding fluid is taken into account. The emitter tries to match the rest spacing of the particles. Nice rays of water can be generated this way. The rest spacing for fluid particles is specified using the NxFluidDesc::restParticlesPerMeter.
- *Constant flow rate.* In this case the emitter keeps emitting the same number of particles each frame. The rate can be adjusted dynamically. See the NxFluidEmitterDesc::rate member.

To turn an emitter on and off, use the NX_FEF_ENABLED flag with the NxFluidEmitterDesc::flags member.

Creation

To create an emitter, an NxFluidEmitterDesc needs to be specified and a call to NxFluid::createEmitter() must occur. Alternatively, it may be easier to create emitters when the fluid is created by adding elements to the NxFluidDesc::emitters array.

Important parameters of NxFluidEmitterDesc:

- *relPose* - pose of the emitter relative to the shape frame. If frameShape is NULL, the global frame is used.
- *type* - type of emitter, either constant pressure (NX_FE_CONSTANT_PRESSURE) or constant flow rate (NX_FE_CONSTANT_FLOW_RATE).
- *shape* - shape of the emitter. Presently there are two options: Rectangular (NX_FE_RECTANGULAR) or Elliptical (NX_FE_ELLIPSE).
- *particleLifetime* - time in seconds an emitted particle lives. If set to zero, the particle lives until it hits a drain.
- *maxParticles* - maximum number of particles which are emitted from this emitter.The emitter will stop creating particles when this limit is reached and start again once the number of fluid particles falls below the limit. If set to 0, the number of emitted particles is unrestricted (up to the maximum for the fluid).
- *fluidVelocityMagnitude* - velocity magnitude of the emitted fluid particles. Note that the maximal particle velocity is bound by NxFluid::motionLimitMultiplier - see NxFluid.
- *rate* - how many particles are emitted per second.The rate is only considered in the simulation if the type is set to NX_FE_CONSTANT_FLOW_RATE.
- *dimensionX, dimensionY* - sizes of the emitter in the directions of the first and the second axis of its orientation frame (relPose).
- *randomAngle* - range of random angular deviation from the emission direction.The emission direction is specified by the third orientation axis of relPose.
- *randomPos* - range of the randomized positions of emitted particles. The z value specifies the range of random positions along the direction of emission.
- *frameShape* - see Fluid Interaction with Rigid Bodies.

Example

```
//Create Emitter.

NxFluidEmitterDesc emitterDesc;

emitterDesc.setToDefault();
```

```
    emitterDesc.dimensionX = 0.03;
    emitterDesc.dimensionY = 0.03;

    emitterDesc.relPose.id();
    NxReal mat[] = {1,0,0,0, 0,0,1,0, 0,-1,0,0, 0,2,0,1};
    emitterDesc.relPose.setColumnMajor44(mat);

    emitterDesc.rate = 5.0;
    emitterDesc.randomAngle = 0.1f;
    emitterDesc.fluidVelocityMagnitude = 6.5f;
    emitterDesc.maxParticles = MAX_PARTICLES;
    emitterDesc.particleLifetime = 4.0f;
    emitterDesc.type = NX_FE_CONSTANT_FLOW_RATE;
    emitterDesc.shape = NX_FE_ELLIPSE;

    gFluidEmitter = gFluid->createEmitter(emitterDesc);
```

Usage

The following functions are useful when working with emitters:

```
//Sets the emission count to 0 and resets the reservoir
void NxFluidEmitter::resetEmission(NxU32 maxParticles)

//Returns the number of particles that have been emitted thus far by the emitter
NxU32 NxFluidEmitter::getNbParticlesEmitted()
```

API Reference

- NxFluidEmitter
- NxFluidEmitterDesc
- NxFluid

Fluid Drains

Fluid drains are not objects in there own right; rather, they are shapes with the NX_SF_FLUID_DRAIN flag set applied to the NxShapeDesc::shapeFlags member, causing particles colliding with them to be removed from the simulation. Below is a list of currently supported static and dynamic drain shapes:

- Box - NxBoxShape
- Sphere - NxSphereShape
- Capsule - NxCapsuleShape
- Convex - NxConvexShape (Only dynamic shapes)
- Plane - NxPlaneShape

Drains are an important method in keeping the particle count and spread under control. Place drains around the area in which a fluid is used to stop the fluid particles from spreading too far and negatively impacting performance.

Example

```
//Box to act as a drain.

NxBoxShapeDesc DrainDesc;
NxActorDesc DrainActorDesc;

DrainDesc.setToDefault();
DrainDesc.shapeFlags|=NX_SF_FLUID_DRAIN;
DrainDesc.dimensions.set(1.0f,0.2f,1.0f);
DrainActorDesc.shapes.pushBack(&DrainDesc);

NxActor *drainActor=gScene->createActor(DrainActorDesc);
```

API Reference

- NxShape
- NxBoxShape
- NxSphereShape
- NxCapsuleShape
- NxConvexShape
- NxPlaneShape

Fluid Usage

This section introduces a few Fluid features that do not fit into the other categories.

Particle Updates

You can affect each particle with a force, or tell the SDK to remove it by setting a deletion flag (NX_FP_DELETE). The updates are done through the NxFluid::updateParticles(const NxParticleUpdateData&) method.

The information needed by the call is passed via a user-allocated buffer, which needs to have (at least) enough space for NxFluidDesc::maxParticles of particles. The update buffer is defined as:

```
NxParticleUpdateData
{
        NxForceMode::forceMode;
        NxF32* bufferForce;
        NxU32* bufferFlag;
        NxU32 bufferForceByteStride;
        NxU32 bufferFlagByteStride;
        NxU32 numUpdates;
        NxU32* bufferId;
        NxU32 bufferIdByteStride;
}
```

There are two ways to provide particle updates: ID-based and indexed-based.

- In the ID-based approach, a subset of the particles can be updated. The particle IDs of this subset are provided in bufferId, the number off particles in the subset is provided in numUpdates, and the appropriate bufferIdByteStride is set. Only the particles with the provided IDs are updated.
- In the index-based approach, all particles are updated. To use this mode, set bufferId to null. The order of the particles in the buffers is the same as the ordering of the particles of the last simulation step, and numUpdates and bufferIdByteStride are ignored.

Notes and Limitations:

- For the index-based mode, the update order is equivalent to the ordering of the particles of the last simulation step. This means the update logic cannot be parallelized with the SDK simulation.
- Use NxFluidPacketData in order to optimize more complex particle updates.

Also see the SampleForce part of the SampleParticleFluid code for a use case.

Spatial Information

It is now possible to query the SDK for some spatial data connected to the Fluid simulation. This is done by setting a buffer which the SDK writes information to in each frame. The buffer needs to hold at least NxParameter::NX_CONSTANT_FLUID_MAX_PACKETS number of elements.
The API for this functionality is:

```
NxFluidPacketData NxFluidDesc::fluidPacketData
NxFluidPacketData NxFluid::getFluidPacketData()
void NxFluid::setFluidPacketData(const NxFluidPacketData& pData)
```

This can be used in conjunction with the NX_VISUALIZE_FLUID_PACKET_DATA parameter in order to have the SDK visualize the fluid packets.

Notes and Limitations:
The packet data can be used for spatial culling for general particle processing:

- Force/Deletion updates.
- View frustrum culling.

The packetID is currently not really useful, it might be used later for partial particle updates.

Also see the SamplePacketData part of the SampleParticleFluid code for a use case.

Fluid Events

In version 2.7.0 we added two events for the user to listen to, for making it easier to use the Fluid functionality. The events are raised when an emitter is empty and when a fluid is empty. The event callback class is called NxFluidUserNotify and controlled through the following interface:

```
void NxScene::setFluidUserNotify(NxFluidUserNotify* callback)
NxFluidUserNotify* NxScene::getFluidUserNotify() const
NxFluidUserNotify* NxSceneDesc::fluidUserNotify
```

The callback interface is described in the API documentation, but currently consists of the following two functions:

```
//The user needs to return whether he wants to delete the emitter, in the callback.
bool NxFluidUserNotify::onEmitterEvent(NxFluidEmitter& emitter, NxFluidEmitterEventType eventType)

//The user needs to return whether he wants to delete the fluid, in the callback.
bool NxFluidUserNotify::onEvent(NxFluid& fluid, NxFluidEventType eventType)
```

Also see the SampleEvents part of the SampleParticleFluid code for a use case.

Particle Priority Mode

A new feature is that the user can specify a certain amount of particles that should always be possible to add to the simulation during one frame. This is done by "reserving" an amount of particles when creating the Fluid using the NxFluidDesc::numReserveParticles field and rasing the NX_FF_PRIORITY_MODE flag. The SDK will make sure to delete all particles exceeding NxFluidDesc::maxParticles - NxFluidDesc::numReserveParticles. This way the SDK can guarantee that the user can always add at least numReserveParticles per simulation step. The SDK will delete the particles based on their lifetime, oldest first.
Related API:

```
NxU32 NxFluid::getNumReserveParticles()
void NxFluid::setNumReserveParticles(NxU32)
```

Notes and Limitations:
The priority mode will not work properly when setting lifetimes to infinity (0).

Also see the SampleUserData part of the SampleParticleFluid code for a use case.

Rendering Particles

Reading back particle data

To readback the positions of particles and render them, it is necessary to specify a buffer to write the appropriate particle information into when creating the fluid. This is done using the NxFluidDesc::particlesWriteData. After each simulation step, the SDK writes updated positions, etc., to these buffers. If any of the buffers in NxFluidDesc::particlesWriteData are NULL, then writing to these buffers is skipped. The buffers should be large enough to hold NxFluidDesc::maxParticles.

The SDK stores the number of particles written in the current timestep into the integer pointed to by NxFluidDesc::particlesWriteData::numParticlesPtr (this may be less than maxParticles if only a subset of the maximum number of particles are being simulated).

When creating a fluid, you can specify the following buffers in particlesWriteData:

- *bufferPos* - position of each individual particle.
- *bufferVel* - velocity of each particle.
- *bufferLife* - lifetime of each particle.
- *bufferDensity* - density for each particle.
- *bufferId* - SDK-generated unique particle ID.
- *bufferFlag* - NxParticleFlag values for each particle.

For example, the lifetime is useful for fading out particles as they reach the end of their life. The velocity is useful for motion blur/elongated particles (e.g., when rendering sparks, etc.).

Deletion and creation

Besides the buffers in particlesWriteData, the members particleDeletionIdWriteData and particleCreationIdWriteData can be used to respond to the deletion or creation of particles by the SDK, graphically or gameplay-wise. Together with IDs, these provide the means to easily tracking individual particles over their lifetime.

Examples

```
gParticles = new NxVec3[fluidDesc.maxParticles];

fluidDesc.particlesWriteData.bufferPos = &gParticles[0].x;
fluidDesc.particlesWriteData.bufferPosByteStride = sizeof(NxVec3);    fluidDesc.particlesWrit
```

Rendering Particles with OpenGL

```
//Run simulation step
//...

for (unsigned p=0; p<gNumParticles; p++)
{
    NxVec3 & particle = gParticles[p];

    glPushMatrix();
    glTranslatef(particle.x,particle.y,particle.z);
    glutSolidSphere(0.1,3,3);
    glPopMatrix();
}
```

Samples

Sample Particle Fluid

API Reference

- NxFluid
- NxFluidDesc

Fluid Interaction with Rigid Bodies

Rigid bodies acting on particles

When a particle collides with a rigid body, a collision impulse is applied to the particle. By default, collisions with static and dynamic shapes are both considered. To enable them separately, the user can specify the NX_F_STATIC and the NX_F_DYNAMIC flags in the NxFluidDesc::collisionMethod member; this can also be adjusted during simulation using NxFluid::setCollisionMethod.

The fluid's API allows a coefficient of restitution to be defined for collisions with static and dynamic shapes separately. These are specified using NxFluidDesc::restitutionForDynamicShapes and NxFluidDesc::restitutionForStaticShapes. Intuitively, these parameters define how much the particles bounce when they collide with shapes. The restitution coefficients must be between 0 and 1. A value of 0 keeps particles from bouncing and removes all their momentum in the direction of the contact normal. A value of 1 causes the particles to be reflected without any loss of momentum.

In addition, a friction (adhesion) factor can be defined which controls how easily particles slide along a surface. A value of 1 causes a particle to lose all its momentum tangential to the surface so that it sticks to the surface, while a value of 0 allows the particle to keep its momentum and thus slide. To visualize, imagine a viscous blob of liquid sliding smoothly across polished steel (a low friction factor) and more slowly down a slab of sandstone (a higher friction factor). The friction factors can be set using NxFluidDesc::dynamicFrictionForDynamicShapes and NxFluidDesc::dynamicFrictionForStaticShapes. Static friction thresholds (also called stiction) (NxFluidDesc::staticFrictionForDynamicShapes and (NxFluidDesc::staticFrictionForStaticShapes) are defined in the API but are not implemented yet. The same holds for normal direction attraction to the shape surface: (NxFluidDesc::attractionForDynamicShapes, NxFluidDesc::attractionForStaticShapes).

The distance that fluid particles maintain from the surface of rigid bodies can be adjusted using NxFluidDesc::collisionDistanceMultiplier. This is a multiplier, and the actual rest distance is the product of this number and the inverse of restParticlesPerMeter.

Fluids contacting meshes (i.e. triangle meshes, heightfields or convexes) require special processing of the triangles for fluid collision. Instead of relying on the SDK performing these calculations automatically, you can pre-cook areas you know will contact fluids using the NxScene::cookFluidMeshHotspot method. This way you can avoid performance spikes. Please see compartments for a list of limitations regarding the collision between rigid bodies and fluid particles.

Two-Way Interaction

It is possible to allow fluids to influence rigid bodies as they are introduced. See compartments for more information.
Note that different fluids can *not*, at present, interact directly with each other.

Attaching an Emitter to a Shape

Attaching an emitter to a shape is simply a case of setting the NxEmitterDesc::frameShape to the appropriate rigid body shape. The relPose transform is then used to position the emitter relative to the shape's frame so that it will move and rotate along with the actor.

The user may want the particles created by the emitter to have a velocity that is the sum of the emission velocity and the emitter velocity (which is attached to a dynamic actor). This can be achieved by setting the NX_FEF_ADD_BODY_VELOCITY flag in NxEmitterDesc::flags.

API Reference

- NxFluid
- NxFluidEmitter
- NxActor

Cloth and SoftBody interaction with Fluids

Introduction

Interaction between cloth/soft body and fluids is a new feature in 2.7.0 and is still in an experimental state. There are some things to think about when using this new feature:

- There are some types of scenarios that are difficult or impossible for the current algorithm to handle, e.g:
  - A piece of cloth or a soft body floating on a fluid.
  - A cloth bag filled with a fluid.
- The current implementation is not very efficient, and it is best to try and keep the number of cloths/soft bodies with the feature turned on at a minimum.
- Currently, each cloth / soft body with the fluid collision flag switched on collides against **all** fluids in the scene.
- Collision detection is performed for fluid particles against cloth / soft body particles, not against cloth triangles or soft body tetrahedra.
- The collision detection and response calculation is done in SW, even for HW cloth/soft bodies.

Usage

Even though the interaction feature affects both cloth/soft body and fluids, the feature is controlled through the API of cloth/soft body. The following section shows how to enable interaction for cloths, but the same scheme applies to softbodies. In the cases where there are differences, the soft body difference is noted in parenthesis.

To turn the interaction on, set the flag

```
NX_CLF_FLUID_COLLISION    (NX_SBF_FLUID_COLLISION)
```

in NxClothDesc.flags when creating the cloth. You can change the flag in runtime via the method

```
NxCloth.setFlags(NxU32 flags)
```

It is possible to control the intensity of the interaction on the cloth and on the fluid by specifying two response coefficients. It is possible to control the interaction so that for example the response of a collision is lower on the cloth than on the fluid.

```
//The intensity of the interaction on the cloth is controlled by
NxClothDesc.fromFluidResponseCoefficient

//The intensity of the interaction on the fluid is controlled by
NxClothDesc.toFluidResponseCoefficient
```

It is also possible to change these values in runtime using the following functions:

```
NxReal NxCloth.getFromFluidResponseCoefficient()
void NxCloth.setFromFluidResponseCoefficient(NxReal coefficient)

NxReal NxCloth.getToFluidResponseCoefficient()
void NxCloth.setToFluidResponseCoefficient(NxReal coefficient)
```

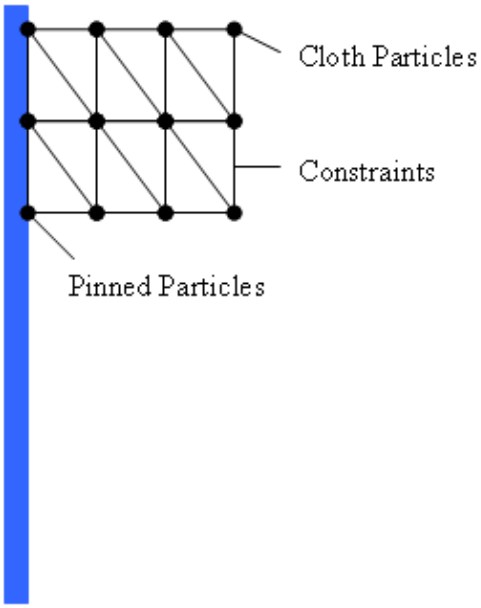
The interaction is always two-way, but you can simulate a one-way interaction by setting one of the coeffients to zero.

API Reference

- NxFluid
- NxCloth
- NxSoftBody

Cloth

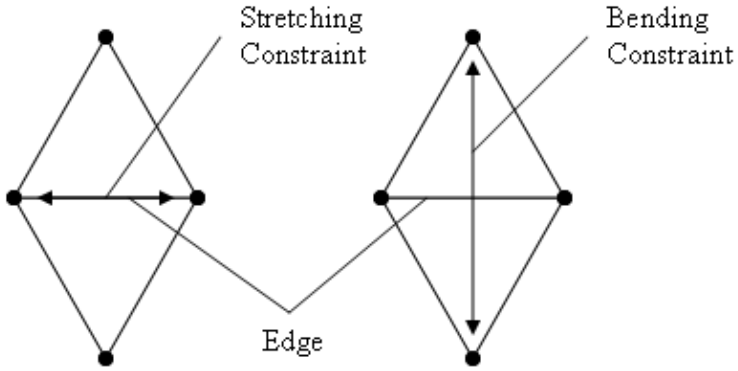
The cloth feature of the NVIDIA PhysX SDK allows for simulation of items made from cloth, such as flags, clothing, etc. This is accomplished by providing a mesh that is used to define a set of particles (vertices). The topology of the mesh allows the SDK to construct constraints between the particles that mimic how cloth can bend and stretch. In addition, cloth particles can be pinned to shapes and global positions, such as attaching a flag to a pole.

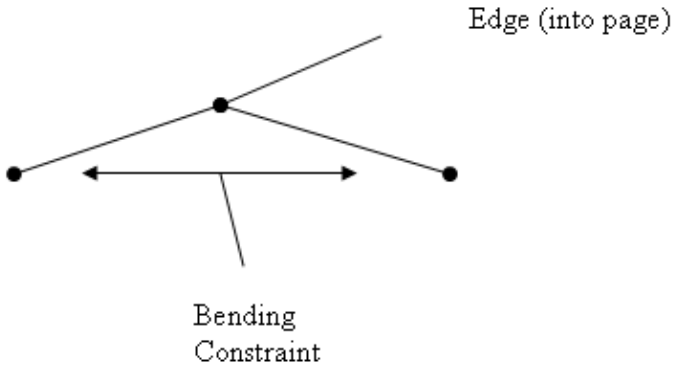


There are two main types of cloth constraints:

- Stretching - applied to maintain distance between each particle that is connected by an edge in a cloth mesh. The strength of this constraint is assigned using the SDK; specifying a small stretching constraint factor allows the particles to move apart more easily and gives the impression of more stretchy cloth, such as lycra. Specifying a larger constraint factor makes the cloth stiffer, like denim.

- Bending - applied to maintain the angle along an edge in a cloth mesh, either by constraining the angle directly or by constraining the distance between the pair of particles on either side of the edge (see the diagram below). An example of a material that would use a low bending constraint for simulation is cotton, while a substance such as paper or cardboard would use a high bending constraint.





Tips

- To create stiff cloth:
  - Use fewer vertices (fewer vertices makes stiffer cloth).
  - Use higher iteration counts (associated performance impact).
  - Use NX_CLF_BENDING_ORTHO mode (results in stiffer, more realistic cloth bending).

- To improve cloth performance:
  - Use fewer vertices.
  - Use simple bending constraints (NX_CLF_BENDING and not NX_CLF_BENDING_ORTHO).
  - Disable cloths that are not relevant using NX_CLF_STATIC.
  - Disable tearing for cloths which do not need tearing.
  - Avoid cloth self collision.

Caveats

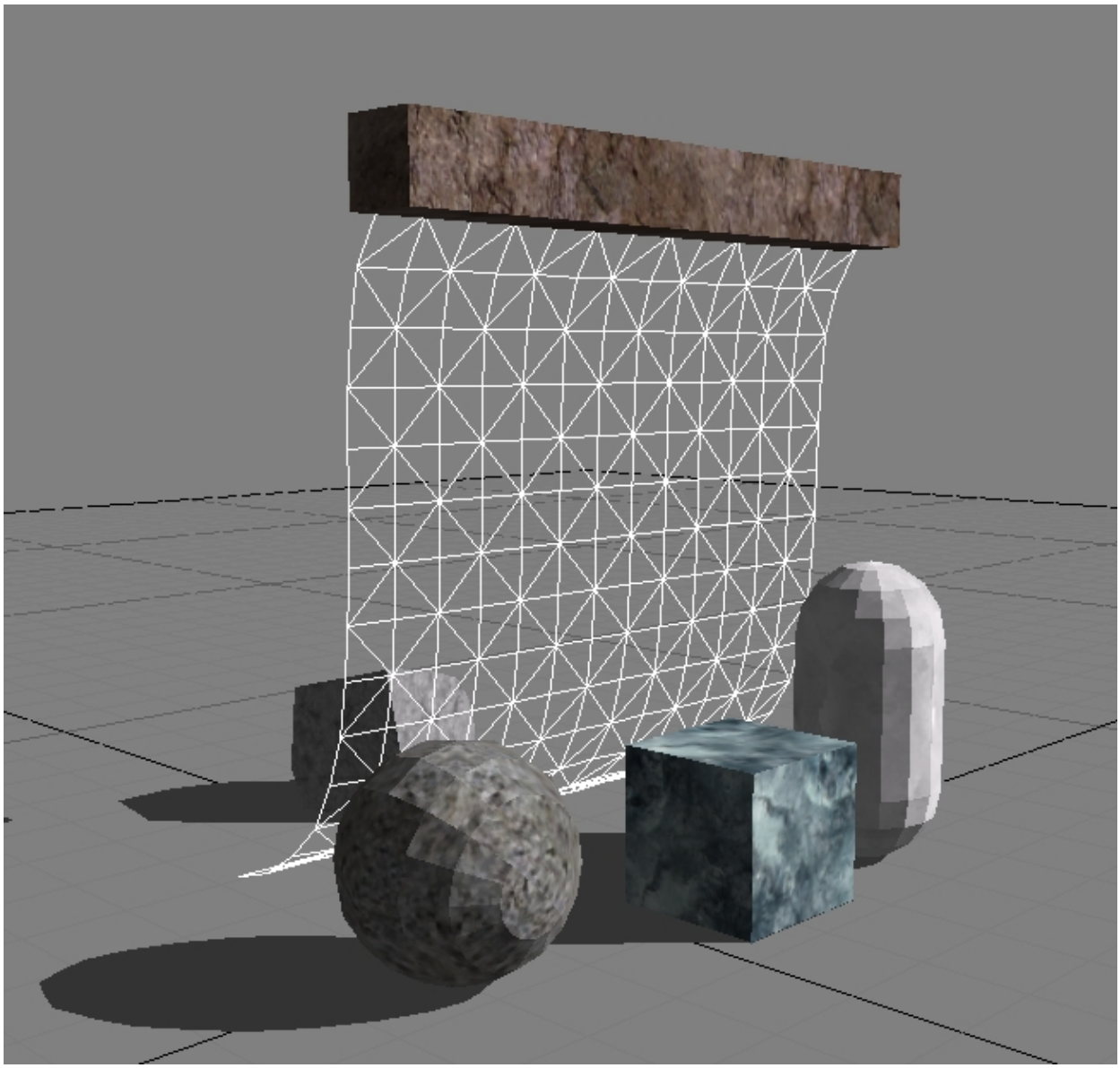
- Cloth collisions only take account of one constraint per particle per timestep. This means that in some cases a cloth can slip through a concave corner during mesh or terrain collisions. A workaround is to increase the cloth's thickness or bending stiffness to prevent cloth constraints from bending around the concave edge.
- When using two way interaction, it is important to set the appropriate density of the attached objects. If an object with a very low or high density is attached to a cloth then the simulation may behave poorly.
- Collision detection is only performed with cloth vertices, so if a cloth is stretched far enough objects will be able to slip between them.

API Reference

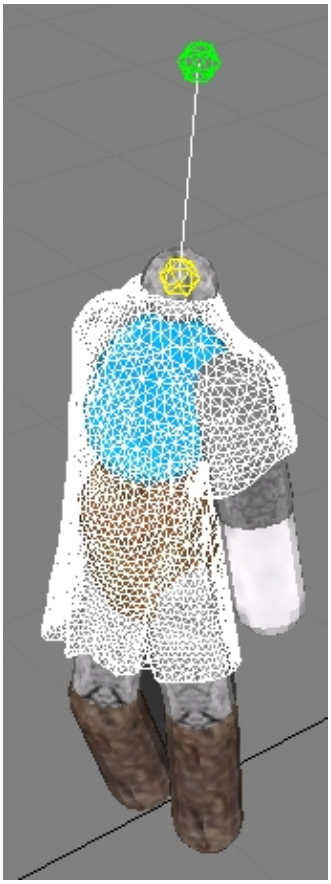
- NxCloth

Cloth Creation

A cloth is defined using a mesh which specifies cloth particle positions (vertices) and the constraints between them (edges). Below is a simple uniform triangle mesh used to represent a curtain.



More complicated cloths, such as the mesh below used to define a T-Shirt shape, can also be created.



Creating a Cloth Mesh

Similarly to a triangle mesh, it is necessary to cook a cloth mesh into a suitable format for simulation. To do so, fill in an NxClothMeshDesc structure and pass it to NxCookingInterface::NxCookClothMesh(). This function returns a stream of bytes that can either be saved and reused later or fed to NxScene::createClothMesh().

Once the NxClothMesh has been created, it can be instanced in the form of NxCloth objects to go with the mesh. For example, one T-Shirt cloth mesh may have many instances, for things like differentiating characters, or to illustrate certain conditions such as stiffness.

Examples

A Uniform Patch

```
/*
** Generate a uniform cloth patch, w and h are the width and height, d is the distance between verti
*/

NxClothMeshDesc desc;

int numX = (int)(w / d) + 1;
int numY = (int)(h / d) + 1;

desc.numVertices                = (numX+1) * (numY+1);
desc.numTriangles               = numX*numY*2;
desc.pointStrideBytes           = sizeof(NxVec3);
desc.triangleStrideBytes        = 3*sizeof(NxU32);
```

```
desc.points                     = (NxVec3*)malloc(sizeof(NxVec3)*desc.numVertices);
desc.triangles                  = (NxU32*)malloc(sizeof(NxU32)*desc.numTriangles*3);
desc.flags                      = 0;

int i,j;
NxVec3 *p = (NxVec3*)desc.points;

for (i = 0; i <= numY; i++) {
    for (j = 0; j <= numX; j++) {
        p->set(d*j, 0.0f, d*i);
        p++;
    }
}

NxU32 *id = (NxU32*)desc.triangles;

for (i = 0; i < numY; i++) {
    for (j = 0; j < numX; j++) {
        NxU32 i0 = i * (numX+1) + j;
        NxU32 i1 = i0 + 1;
        NxU32 i2 = i0 + (numX+1);
        NxU32 i3 = i2 + 1;

        if ((j+i)%2) {
            *id++ = i0; *id++ = i2; *id++ = i1;
            *id++ = i1; *id++ = i2; *id++ = i3;
        }
        else {
            *id++ = i0; *id++ = i2; *id++ = i3;
            *id++ = i0; *id++ = i3; *id++ = i1;
        }
    }
}
```

Cooking

```
gCooking->NxInitCooking();

NxClothMeshDesc desc;

//...



//Cook the mesh on the fly through a memory stream.
//A file stream could also be used to pre-cook the mesh.

MemoryWriteBuffer wb;

if (!gCooking->NxCookClothMesh(desc, wb))
    return false;

MemoryReadBuffer rb(wb.data);
mClothMesh = mScene->getPhysicsSDK().createClothMesh(rb);
```

Creating a Cloth

Fill in an NxClothDesc structure and call NxScene::createCloth() to create a cloth. Parameters for the cloth can be set either in the descriptor or later after the cloth has been created. See Cloth Parameters for details.

```
NxClothDesc desc;

NxMeshData receiveBuffers;
//Fill in receive buffers...

desc.clothMesh = mClothMesh;
desc.meshData = receiveBuffers;

mCloth = scene->createCloth(desc);
```

API Reference

- NxCloth
- NxClothMesh
- NxScene

Cloth Parameters

Cloth parameters may be specified during cloth creation, using members of the NxClothDesc, or after creation using members of NxCloth.

Flags

```
NxClothDesc::flags
```

or

```
NxCloth::setFlags()
```

The following flags may be used:

- NX_CLF_PRESSURE - Enables internal pressure for closed cloth meshes. See cloth pressure.
- NX_CLF_STATIC - Disables integration/simulation of the cloth. For example, a piece of cloth can be created in an authoring tool and simulated for awhile to achieve the desired effect, then made static to avoid the simulation overhead of continually simulating the cloth. This could be useful in making a tablecloth seem real. Simulation for the cloth could be enabled again if the player comes into contact with the table.
- NX_CLF_DISABLE_COLLISION - Disables collision detection between the cloth and other objects.
- NX_CLF_VISUALIZATION - Enables visualizations, displays cloth points, attachment deviation from attachment points, etc. At present, visualization is only intended for internal use. Available visualizations may change.
- NX_CLF_GRAVITY - Enables application of the scene gravity. To set the gravity, change the appropriate SDK parameter.
- NX_CLF_BENDING - Enables bending constraints. Must be set to enable ortho bending (in addition to NX_CLF_BENDING_ORTHO).
- NX_CLF_BENDING_ORTHO - Applies a more robust/realistic bending constraint. When ortho bending is not enabled, the SDK applies bending constraints based on the distance between the two points either side of an edge. When ortho bending is enabled, the constraint is computed based on the actual angle between the two triangles either side of the edge.
- NX_CLF_DAMPING - Enables damping on individual cloth particles.
- NX_CLF_COMDAMPING - Only effective if NX_CLF_DAMPING is set. Excludes the center of mass of the cloth from damping, making the cloth damping local instead of global.
- NX_CLF_COLLISION_TWOWAY - Applies the cloth's force on rigid bodies (e.g., a cloth could pull a rigid body).
- NX_CLF_TEARABLE - Enables Cloth Tearing.
- NX_CLF_HARDWARE - Directs the cloth to be simulated in hardware (flag has no effect after creation).
- NX_CLF_SELFCOLLISION - Enables self collision handling within a single piece of cloth.

- NX_CLF_VALIDBOUNDS - Enables valid bounds for this cloth.

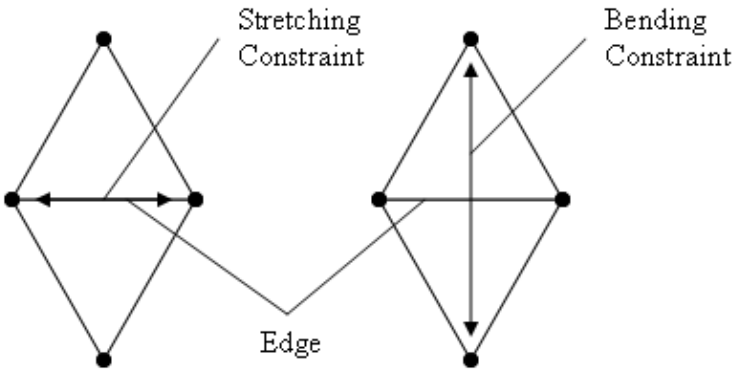
Bending Stiffness

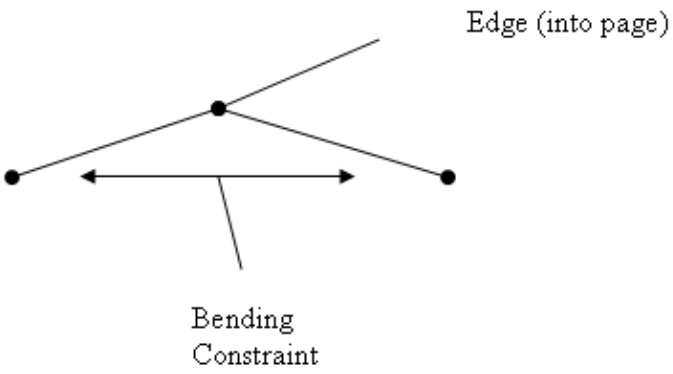
`NxClothDesc::bendingStiffness`

or

`NxCloth::setBendingStiffness()`

Bending stiffness defines the strength of the constraint that stops the triangles on either side of an edge from rotating. A value of one will produce a very stiff material, such as cardboard, while a value near zero will produce one which can bend very easily, like paper.





Stretching Stiffness

`NxClothDesc::stretchingStiffness`

or

`NxCloth::setStretchingStiffness()`

Stretching stiffness defines the strength of the constraint along a triangle edge that maintains the distance between the edge end points. A value of one will produce a cloth that is very difficult to stretch, such as denim pants, while a value near zero will produce one that can stretch easily, like nylon stockings (a value too near zero is not recommended for a stable simulation; zero is not allowed).

Density

`NxClothDesc::density`

NOTE: Density can only be specified when creating a cloth.

Density indirectly specifies the mass of the cloth particles. The mass of a particle is calculated as the sum of the following equation (for triangles sharing a point):

1/3 * triangleArea * density

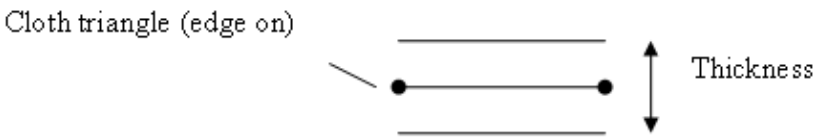
Thickness

`NxClothDesc::thickness`

or

`NxCloth::setThickness()`

Thickness defines the triangle's level of thickness for collision detection, improving its robustness. Setting the thickness very low is not recommended as this will lead to poor collision handling.



Damping

`NxClothDesc::damping`

or

`NxCloth::setDampingCoefficient()`

Controls how much damping is applied to the motion of cloth particles. Damping is always performed for the velocity of each individual particle.
If the NX_CLF_COMDAMPING flag is specified, the global rigid body modes (translation and rotation) are exempted from damping. If not, the global translation and rotation of the cloth are damped, same as the internal movement of the cloth.

Solver Iterations

`NxClothDesc::solverIterations`

or

`NxCloth::setSolverIterations()`

The solver iteration count controls how accurately the cloth is simulated. When simulating cloth, the SDK uses an iterative approach to enforce the cloth constraints - the higher the iteration, the better the result. However, more iterations are more expensive. Plus, there is a top-off point where applying more does not yield a noticeable gain in accuracy. A typical count for a cloth is 5 iterations.

Attachment Response Coefficient

`NxClothDesc::attachmentResponseCoefficient`

or

`NxCloth::setAttachmentResponseCoefficient()`

This coefficient controls the strength of the momentum transfer between a cloth and the shape(s) it is attached to. A large attachment response coefficient will cause the cloth to pull/push the body more easily and a lower value will mean the rigid body is more difficult to move.

See Cloth Attachments for details on how to attach a cloth to a rigid body.

Collision Response Coefficient

`NxClothDesc::collisionResponseCoefficint`

or

`NxCloth::setCollisionResponseCoefficient()`

The collision response coefficient is similar to the attachment response coefficient, except instead of applying forces due to the attachment, it applies forces due to collision.

The collision response coefficient only has an affect if both collisions and two way interactions are enabled (i.e., the CVF_COLLISION flag and the NX_CLF_COLLISION_TWOWAY are both set).

Friction

`NxClothDesc::friction`

or

`NxCloth::setFriction()`

The friction parameter controls the frictional response of cloth and rigid body contacts. A high friction parameter will cause more friction between a cloth and its rigid body contact, while a low friction parameter will result in a cloth that easily slides along a surface.

External Acceleration

`NxClothDesc::externalAcceleration;`

`or`

`NxCloth::setExternalAcceleration`

An external acceleration can be applied to a cloth to simulate different effects. The force is applied to each vertex that is not attached to an object. Interesting effects can be achieved at relatively little cost by varying the externalAcceleration over time.

Wind Acceleration

```
NxClothDesc::windAcceleration;

or

NxCloth::setWindAcceleration
```

As with external acceleration this applies an acceleration on each vertex. However this acceleration is only applied in the vertex normal direction, i.e. only the component of the external force that lies in the direction of the vertex normal is applied to the vertex. This makes it easy to simulate effects such as wind.

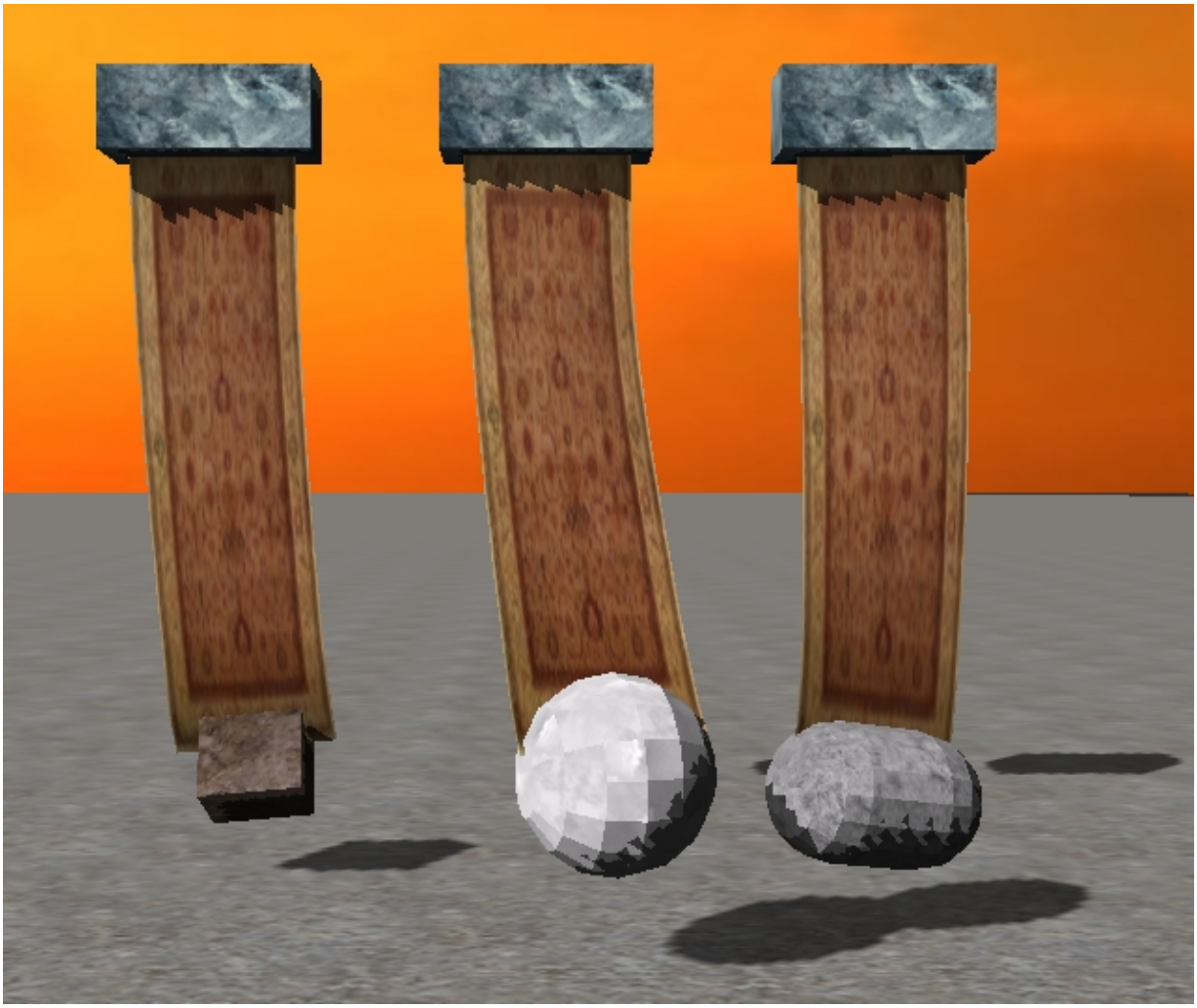
Valid bounds

By using valid bounds, activated by the NX_CLF_VALIDBOUNDS flag, you can specify an axis-aligned bounding volume to which the cloth is restricted. Any part of the cloth that leaves this volume (such as pieces torn off an otherwise attached flag) will be removed from the simulation.

API Reference

- NxCloth
- NxClothDesc

Cloth Attachments



Attaching Cloth to Fixed Points

```
void attachVertexToGlobalPosition(const NxU32 vertexId, const NxVec3 &pos);
```

Cloth particles can be attached to fixed points specified in the global frame. The vertexId specifies the cloth particle to attach to the global position, corresponding to the triangle vertex index of the mesh which is passed to the API when cooking the cloth. The first vertex of the second triangle would be three.

A useful method for obtaining a vertex ID, for example when dragging with a mouse, is to cast a ray against the cloth. See Other Cloth Features for more information.

Attaching Cloth to Shapes

```
void attachVertexToShape(NxU32 vertexId, const NxShape *shape, const NxVec3 &localPos, NxU32
```

Cloth can be attached to rigid body shapes in much the same way as attaching the cloth to a global position. But instead of specifying the global position, the position is relative to the shape (i.e., a position in the shape's coordinate frame). As the shape moves, this position will change in the global frame, pulling the cloth with it.

The attachment shape can be either static or dynamic (i.e., attached to a dynamic actor).

The attachmentFlags parameter specifies how the cloth interacts with the shape. The two possible values are shown below:

- NX_CLOTH_ATTACHMENT_TWOWAY - The object can pull the cloth and the cloth can move the object (only for dynamic shapes).

  The default is one way interaction. ie the object can pull the cloth, but the cloth cannot move the object. If the object moves then the cloth will follow it. However, if the cloth is pulled the object will behave as if it is fixed and not move.

- NX_CLOTH_ATTACHMENT_TEARABLE - When this flag is set the attachment can be torn. ie if enough force is exerted on the attachment then it will snap.

Attaching Cloth to Shapes from Collisions

```
void attachToCollidingShapes(NxU32 attachmentFlags);

void attachToShape(const NxShape *shape, NxU32 attachmentFlags);
```

Cloth can be attached to rigid bodies automatically by detecting colliding cloth particles and gluing these particles to the shape. For example, imagine a rigid body flag pole and a cloth to serve as the flag. The flag is modeled so that one of its edges is inside the geometry of the flag pole. The cloth can then be attached to the pole by attaching all the intersecting points of the flag to the pole.

attachToCollidingShapes() attaches to any rigid body shape in the scene. attachToShape() is less indiscriminate and only attaches to a specified shape. attachmentFlags behaves in a similar way to the attachmentFlags parameter of attachVertexToShape().

It should be noted that this method of attachment will not work with general triangle meshes because the inside/outside of a general triangle mesh is not defined.

Detaching Cloth

```
void detachFromShape(const NxShape *shape);

void freeVertex(const NxU32 vertexId);
```

Cloth particles can be detached as well as attached. detachFromShape() finds all the cloth particles that are attached to a particular shape and detaches them. freeVertex() detaches a specific vertex. The same vertex index that was used to attach a position or shape can be used again with attachVertexToGlobalPosition() or attachVertexToShape().

NOTE: Cloth and rigid bodies have separate solvers. This can cause stability problems in some cases, especially when the density of an attached object is set too low - the impulses generated by the attachment constraint overshoot the object and cause the system to get unstable. The workaround is to increase the density of the object or to add linear and angular damping to it.

Tearable Attachments

When attaching a cloth to an actor the attachment can be flagged as tearable using NX_CLOTH_ATTACHMENT_TEARABLE, this allows the cloth to break the attachment if a sufficient force is applied to the attachment.

In fact the attachment is broken when the stretching constraint attached to the object exceeds a certain factor of its rest length. This factor is specified using:

```
NxClothDesc::attachmentTearFactor

void NxCloth::setAttachmentTearFactor(NxReal factor) ;
NxReal NxCloth::getAttachmentTearFactor();
```

The default is 1.5 so breakable attachments break when the constraint is stretch 50% more than its rest length.

There are many uses for tearable attachments, for example towels which can be torn from there rails, leaves on plants which can be blown away by a strong gust of wind or a characters cape which can be torn off when it gets trapped in a door.

API Reference

- NxCloth
- NxShape

Cloth Rendering

Resulting cloth vertex positions are returned in a similar way as fluids (see rendering particles): 1) the user specifies a number of arrays to receive new cloth positions, 2) the SDK updates these positions during simulation, 3) they are valid after fetchResults(). The SDK also updates counter variables to tell the user how many indices/vertices have been provided (this can change when a cloth is torn).

To provide the buffers to the SDK, the user should fill in an NxMeshData structure. This structure contains pointers to the appropriate arrays, along with the stride and counts for the positions, etc.. The mesh data can be specified when creating the cloth through the NxClothDesc::meshData member or updated later with the NxCloth::setMeshData (NxMeshData &meshData) function.

If the user does not require one or more of the arrays provided by the SDK, the appropriate pointer can be set to NULL. For example, if normals are not required, then the verticesNormalBegin field can be set to NULL and the SDK will not provide normal information.

Unlike previous versions of the SDK, the mesh is always provided in indexed form (indices and vertices):

- verticesPosBegin - Pointer to the beginning of the array used to hold vertex positions. This array should be of at least sizeof(float) * maxVertices * 3. The vertex is stored as a 3 component float vector (xyz).
- verticesNormalBegin - Pointer to the beginning of the array used to store normals. This array should be of at least sizeof(float) * maxVertices * 3. The normal is stored as a 3 component float vector (xyz).
- verticesPosByteStride - The stride (number of bytes) from the start of one position to the next. This member allows application data to be interleaved with the positions (possibly interleaving the normal and position array, etc.).
- verticesNormalByteStride - The stride (number of bytes) from the start of one normal to the next.
- maxVertices - Maximum number of vertices/normals which can be held by the arrays pointed to by verticesPosBegin and verticesNormalBegin.
- numVerticesPtr - Pointer to an integer value which is used by the SDK to communicate the actual number of vertices used (which may be less than the maximum available).
- indicesBegin - Pointer to an array of triangle indices. Triangles are specified as groups of 3 indices.
- indicesByteStride - The stride from the start of one index to the next.
- maxIndices - Max number of indices the index buffer can contain. This is three times the max number of triangles.
- numIndicesPtr - Pointer to an integer which receives the number of triangles which have been generated.
- parentIndicesBegin - The pointer to the user specified buffer for vertex parent indices.
- parentIndicesByteStride - The stride from the start of one parent index to the next.
- maxParentIndices - Max number of parent indices the parent index buffer can contain.

- numParentIndicesPtr - Pointer to an integer which receives the number of parent indices that have been generated.

- dirtyBufferFlagsPtr - Pointer to a user allocated buffer of dirty flags. If the SDK changes the content of a given buffer, it also sets the corresponding flag of type NxMeshDataDirtyBufferFlags

- flags
    - NX_MDF_16_BIT_INDICES - Specifies that indices are stored as 16 bit integers instead of 32 bit integers.

After the user retrieves the data, it can be rendered as an ordinary triangle mesh or processed in some other way (e.g., constructing some form of higher order surface).

Parent indices are needed when vertices are duplicated by the SDK (e.g., cloth tearing). The parent index of an original vertex is its position in the verticesPos buffer. The parent index of a vertex generated by duplication is the index of the vertex it was copied from.

API Reference

- NxCloth
- NxClothMesh
- NxScene

Cloth Tearing

Cloth can be made to have a limited resistance to stretching and thus to get torn in pieces under a large enough force. In order to enable tearing, set the NX_CLF_TEARABLE flag in the descriptor when creating the cloth:

```
desc.flags |= NX_CLF_TEARABLE;
```

The tolerance of the cloth is set in terms of the elongation factor, using the descriptor member tearFactor; for example:

```
desc.tearFactor = 2.0f;
```

will cause the cloth to tear when the distance between two particles in the cloth is stretched past twice the nominal distance. Alternatively, you can set the tear factor using NxCloth::setTearFactor(). The default value for the tear factor is 1.5.
This means that the actual force required to tear the cloth is dependent upon both the tear factor and the stretching stiffness: higher stiffness means a greater force is required in order to exceed the elongation required for tearing.

Tearing works by splitting vertices, not by cutting the edges between vertices. Thus the total number of vertices increases when tearing occurs. This means that the buffers you provide when creating the cloth must be correspondingly larger in order to accommodate new vertices; when the buffers are filled, no more tearing can occur. The margin necessary will depend on the application and how torn up you expect the cloth to get. A typical equilateral triangle mesh may increase by up to a factor of 6 if torn up completely, or 2 if shredded into thin strips, while a single tear across a piece of cloth will only generate as many new vertices as lie along the tear.

Explicit tearing

In some situations you may wish the cloth to tear not from internal stresses, but for some other game-related reason, such as gunshots, slashes with a sword or simply a scripted event. To do this, call NxCloth::tearVertex() for the vertices you wish to tear the cloth at. You must also specify a plane through the vertex that you wish to use as "cutting plane", allowing you to choose along which triangle edges the cloth is to tear.

```
bool NxCloth::tearVertex(const NxU32 vertexId, const NxVec3 &normal);
```

Preset vertex tearing (tear lines)

The SDK provides a method to more precisely control the tearing of a cloth. It is possible to specify if vertices are tearable

An example of this is provided in the image above, the vertices forming the triangular top sections of cloth have had tearing disabled. This functionality could be used for a variety of effects such as a players cloak which can be torn and made to appear ragged, but not completely destroyed. Another use would be to restrict how far a cloth can be torn to prevent accumulating too many pieces, which could spread far apart and cause performance problems(due to the cloth having a very large bounding box).

To specify that a vertex can be torn(split) use the NX_CLOTH_VERTEX_TEARABLE flag. The flags array should be of type NxU32 and should have the same number of flags as the mesh has vertices.

In all other ways preset vertex tearing is identical to normal tearing. For example you should make sure there is enough additional space for the vertices created during tearing and the tear factor + NX_CLF_TEARABLE should be set.

Example

```
NxClothMeshDesc meshDesc;

static const NxU32 vFlags[] = {0, 0, NX_CLOTH_VERTEX_TEARABLE, ...}; //two normal vertices followed

meshDesc.vertexFlagStrideBytes =  sizeof(NxU32);
meshDesc.vertexFlags = vFlags;
...
```

Tips

- Make sure meshData.maxVertices and the corresponding buffers in meshData are large enough for the new vertices generated by tearing. When the buffer cannot hold the new vertices anymore, tearing stops.
- For tearing in hardware, make sure you cook the mesh with the flag NX_CLOTH_MESH_TEARABLE set in the NxClothMeshDesc.flags.
- You can use parent indices (see Cloth Rendering) to handle data associated with vertices duplicated through tearing, such as texture coordinates. The parent index of a vertex indicates the vertex in the

original untorn cloth from which the vertex was torn and duplicated.

API Reference

- NxCloth
- NxClothMesh
- NxScene

Cloth Pressure

Closed cloth meshes may be made to simulate a constant enclosed volume of air, allowing the simulation of balloons or other soft deformable objects. To enable this for a mesh, set the NX_CLF_PRESSURE flag:

```
desc.flags |= NX_CLF_PRESSURE;
```

Note: The mesh must be closed for the above to be meaningful.

The pressure of the enclosed volume of air is set in either of the following ways:

```
desc.pressure = 2.0f;
```

```
cloth.setPressure(2.0);
```

This will fill the closed cloth with an equivalent of 2 atmospheres of pressure, at its nominal volume.

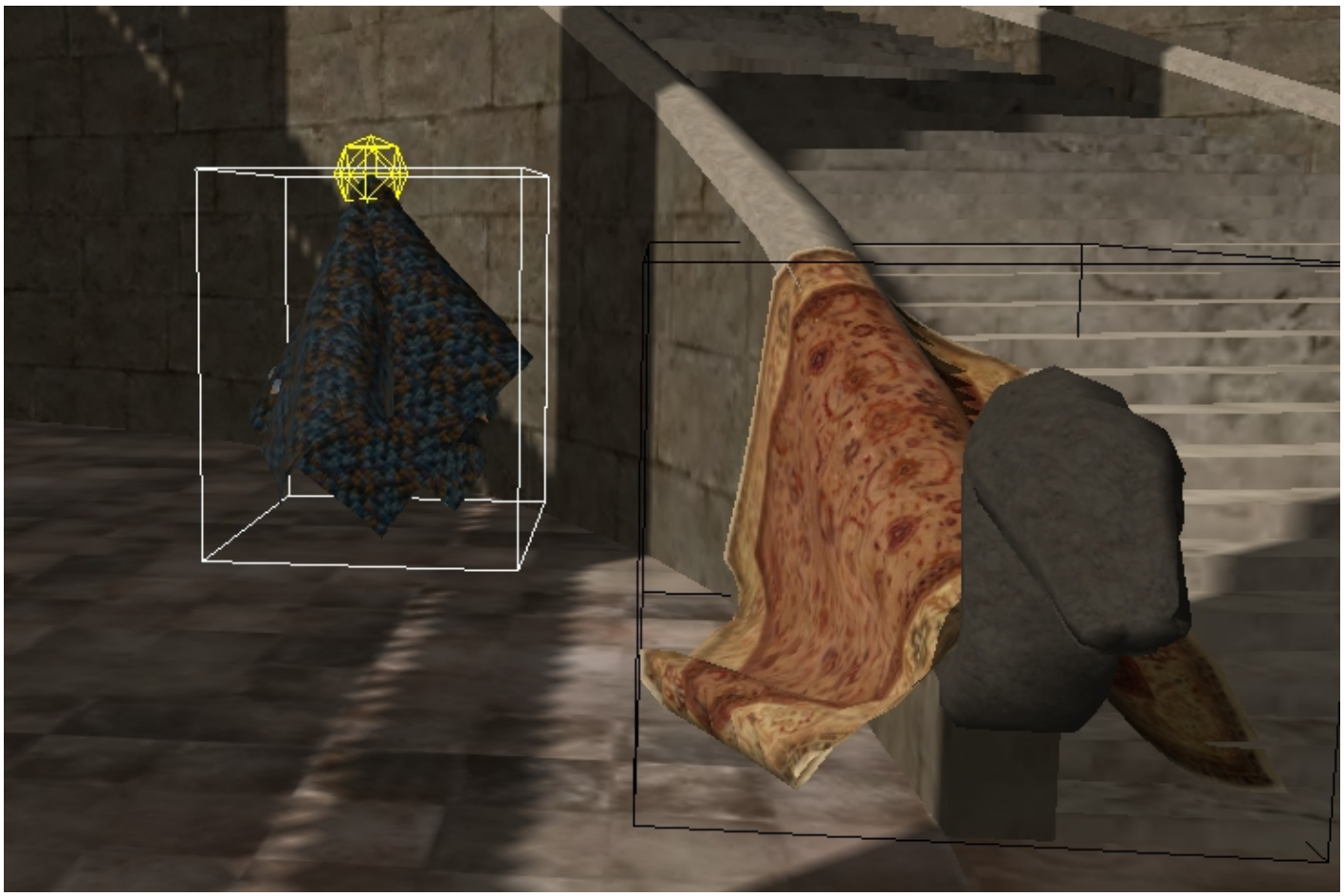
Tips

- If the pressure is set to a value less than 1.0 (the default), the pressure will be insufficient to preserve the initial shape of the cloth and it will collapse until it has been compressed enough to have an internal pressure of 1.0.
- If the pressure is set to a value greater than 1.0, the cloth will expand beyond its initial size; how much will depend on the stretching stiffness of the cloth.

API Reference

- NxCloth
- NxClothMesh
- NxScene

Cloth Sleeping



In a similar way to rigid bodies (see rigid body sleeping), cloths can go to sleep after a period of inactivity. The objective of sleeping is to improve performance by not simulating cloths which are not moving.

Inactivity is defined as all of the cloth vertices being below a linear velocity threshold. When all vertices in a cloth are determined to be inactive a counter is decremented until it reaches zero at which point the cloth transitions into a sleeping state and all particle velocities are zeroed.

The sleep state for cloth can be visualized using the NX_VISUALIZE_CLOTH_SLEEP SDK parameter. A white bounding box is drawn around cloths which are awake and a black one for sleeping cloths.

The most important parameter for the user to be concerned with is the velocity threshold used for sleep determination. This is set and retrieved using the NxCloth members:

```
NxReal NxCloth::getSleepLinearVelocity() const;
void NxCloth::setSleepLinearVelocity(NxReal threshold);
```

The threshold should be set high enough so that the cloth actually goes to sleep when it is not being interacted with, but low enough that the cloth does not remain awake due to oscillations and insignificant interactions with the cloth.

The user can query if a cloth is asleep using:

```
bool NxCloth::isSleeping() const;
```

Sometimes it will be necessary to put cloths to sleep, for example during level loading a user may wish to force all the cloths in the level to sleep, so as to prevent extremely high cpu/ppu load during the start of the level while the cloth settles and drops to sleep. When putting a cloth to sleep the SDK zeros the velocity of the cloth particles, so that when the cloth wakes up it is not instantly moving again.

```
void NxCloth::putToSleep();
```

Inversely there are occasions when the user wishes to wake up a cloth. For example when a user enters the vicinity of a cloth which has been forced to sleep.

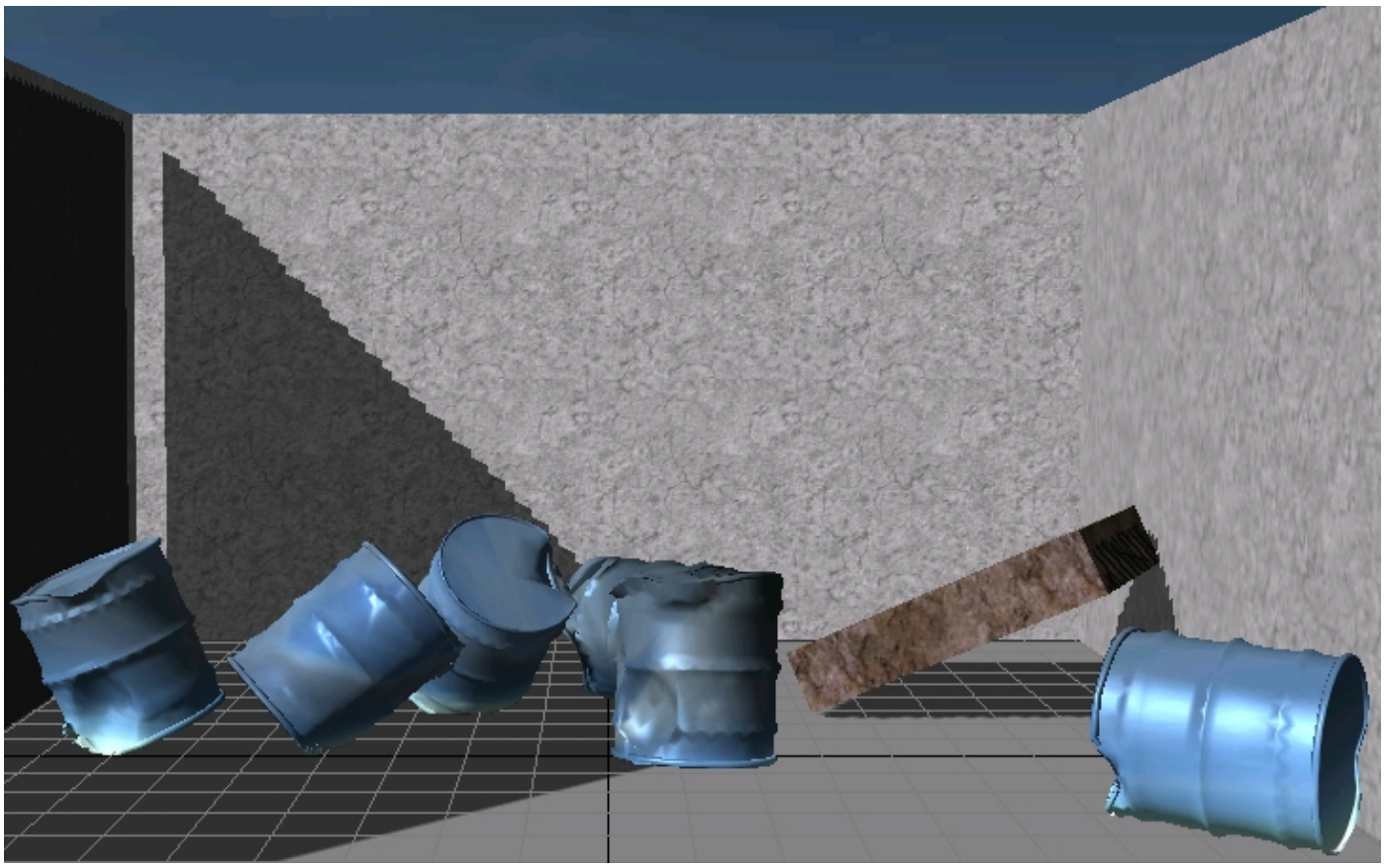
```
void NxCloth::wakeUp(NxReal wakeCounterValue = NX_SLEEP_INTERVAL);
```

As well as simply waking up the cloth, wakeUp() can be used to set the amount of simulation time until the cloth next goes to sleep. Setting a very high value can be used to force the cloth to always stay awake. Useful for performance testing or tuning of the cloth behavior.

API Reference

- NxCloth

Cloth Metal



Cloth Metal is a cloth mode which allows cloth to be used to simulate plastic deformation. For example dents in barrels, car bodywork or metal doors can be displayed when there is a large impact on the surface.

Cloth metal is created in a very similar way to other cloth types, first a cloth mesh is created (NxClothMesh) and instances of this cloth mesh are used in the form of NxCloths.

Then to enable the feature a rigid body should be attached to the cloth. The cloth acts as a deformable surface for the rigid body. The cloth automatically becomes rigid when the body is attached. The rigid body is used for simulation until an object hits the cloth with a sufficiently large impulse to cause deformation, at this point the cloth deforms. Following the impact the object is simulated again using the rigid body. Collision is made against cloth vertices, so objects smaller than the distance between cloth vertices might not interact fully with the metal cloth.

To attach the rigid body to the cloth use:

```
void NxCloth::attachToCore (NxActor *actor, NxReal impulseThreshold, NxReal penetrationDepth=
```

- **actor** - The actor/body to attach to the cloth. The actor should be dynamic. In version 2.6 only sphere shapes, box shapes, capsule shapes or compounds of spheres are supported for collision geometry.
- **impulseThreshold** - Threshold for when deformation is allowed.
- **penetrationDepth** - Can be used to increase the amount of deformation due to impacts.
- **maxDeformationDistance** - Maximum deviation of cloth particles from initial position.

API Reference

- NxCloth

Other Cloth Features

Vertex welding

You may wish to have multiple vertices at the same location of a mesh, for example in order to handle multiple texture coordinates. When simulating, such a mesh will fall apart, unless you use vertex welding. With welding, the cooker identifies close vertices (up to an epsilon specified by the user) and maps them to one single vertex internally.

In order to activate this feature, set the NX_CLOTH_MESH_WELD_VERTICES flag and the weldingDistance member (which is the aforementioned epsilon), both in the cloth mesh descriptor. When the mesh is created, vertices located at the same spot will internally be merged for purposes of simulation, and that merged vertex' data will be duplicated in the output for all the externally visible original vertices.

Raycasting

```
bool raycast(const NxRay& worldRay, NxVec3 &hit, NxU32 &vertexId);
```

In a similar way to rigid bodies, it is possible to test a ray against a cloth and to find out if and when the ray hits it, which is useful for dragging and editing. A raycast can be performed against the cloth which will provide the vertexId near where the ray hits. This ID can then be used to pin the cloth to the position of the mouse's cursor, allowing the cloth to be dragged to a different position.

Collision Filtering

```
NxCollisionGroup getGroup() const;
void setGroup(NxCollisionGroup collisionGroup);

void setGroupsMask(const NxGroupsMask& groupsMask);
const NxGroupsMask getGroupsMask() const;
```

Also similar to rigid bodies, collision filtering groups or masks can be applied to cloth. This allows the application to disable collision with a specific subset of the rigid bodies within the scene.

NOTE: Collision filtering for cloth shouldn't be changed after initial setup, otherwise the filtering may become inconsistent.

See Broad Phase Collision Detection and Contact Filtering for more information.

Overlap AABB Triangles

```
virtual bool overlapAABBTriangles(const NxBounds3 bounds, NxU32& nb, const NxU32*& indices)const;
```

The cloth triangles overlapping an axis aligned box can be retrieved using overlapAABBTriangles() (just like with a triangle mesh shape).

The triangle indices correspond to the triangles referred to by NxClothDesc.meshdata. Triangle i has the vertices 3i, 3i+1 and 3i+2 in the array NxMeshData.indicesBegin. See Cloth Rendering for more details.

Adding a Force at a Vertex

```
virtual void addForceAtVertex(const NxVec3& force, NxU32 vertexId, NxForceMode mode = NX_FORCE);
```

A force or impulse can be applied to a cloth vertex. The mode parameter allows the user to choose how the force is applied:

- NX_FORCE: velocity += force * invMass * dt
- NX_IMPULSE: velocity += force * invMass
- NX_ACCELERATION: velocity += force * dt
- NX_VELOCITY_CHANGE: velocity += force

In these modes, velocity is the velocity of the particle or vertex, invMass is the reciprocal mass of the particle (see Cloth Parameters) and dt is the timestep.

Adding a Force at a certain position

Instead of choosing a vertex in the cloth you may want to apply a force in and around a certain position in space. This may be done using the following methods:

```
virtual void NxCloth::addForceAtPos(const NxVec3& position, NxReal magnitude, NxReal radius, NxForceMode
```

```
virtual void NxCloth::addDirectedForceAtPos(const NxVec3& position, const NxVec3& force, NxReal radius,
```

The first of these applies a radial force in the vicinity of *position*, with a quadratic magnitude falloff up to the maximum *radius*. The second applies a force in an arbitrary direction, with the same quadratic magnitude falloff. The mode parameter is the same as described above.

See the API docs for more information.

API Reference

- NxClothMeshDesc
- NxCloth

Cloth and SoftBody interaction with Fluids

Introduction

Interaction between cloth/soft body and fluids is a new feature in 2.7.0 and is still in an experimental state. There are some things to think about when using this new feature:

- There are some types of scenarios that are difficult or impossible for the current algorithm to handle, e.g:
  - A piece of cloth or a soft body floating on a fluid.
  - A cloth bag filled with a fluid.
- The current implementation is not very efficient, and it is best to try and keep the number of cloths/soft bodies with the feature turned on at a minimum.
- Currently, each cloth / soft body with the fluid collision flag switched on collides against **all** fluids in the scene.
- Collision detection is performed for fluid particles against cloth / soft body particles, not against cloth triangles or soft body tetrahedra.
- The collision detection and response calculation is done in SW, even for HW cloth/soft bodies.

Usage

Even though the interaction feature affects both cloth/soft body and fluids, the feature is controlled through the API of cloth/soft body. The following section shows how to enable interaction for cloths, but the same scheme applies to softbodies. In the cases where there are differences, the soft body difference is noted in parenthesis.

To turn the interaction on, set the flag

```
NX_CLF_FLUID_COLLISION    (NX_SBF_FLUID_COLLISION)
```

in NxClothDesc.flags when creating the cloth. You can change the flag in runtime via the method

```
NxCloth.setFlags(NxU32 flags)
```

It is possible to control the intensity of the interaction on the cloth and on the fluid by specifying two response coefficients. It is possible to control the interaction so that for example the response of a collision is lower on the cloth than on the fluid.

```
//The intensity of the interaction on the cloth is controlled by
NxClothDesc.fromFluidResponseCoefficient

//The intensity of the interaction on the fluid is controlled by
NxClothDesc.toFluidResponseCoefficient
```

It is also possible to change these values in runtime using the following functions:

```
NxReal NxCloth.getFromFluidResponseCoefficient()
void NxCloth.setFromFluidResponseCoefficient(NxReal coefficient)

NxReal NxCloth.getToFluidResponseCoefficient()
void NxCloth.setToFluidResponseCoefficient(NxReal coefficient)
```

The interaction is always two-way, but you can simulate a one-way interaction by setting one of the coeffients to zero.

API Reference

- NxFluid
- NxCloth
- NxSoftBody

Soft Bodies

Motivation

The soft body feature of the NVIDIA PhysX SDK allows the simulation of volumetric deformable objects. It can also be used for items that are not usually thought of as classical soft bodies, such as plants or multiple layers of cloth. Here are some examples:

- Tires of a monster truck
- Big fat guys, monsters
- Curvy girls
- Objects with complex geometry such as trees, grass

To simulate complex geometry like a tree with many branches and leaves, a coarser volumetric mesh is first build around the object. The volumetric mesh is then simulated and moves the complex geometry with it (see soft body rendering). The advantage of the technique is that it makes the simulation of triangle soups possible, i.e. every possible mesh can be simulated.

The soft body feature is very similar to the cloth feature. Therefore, the two features have similar parameters and methods. Here are the main differences

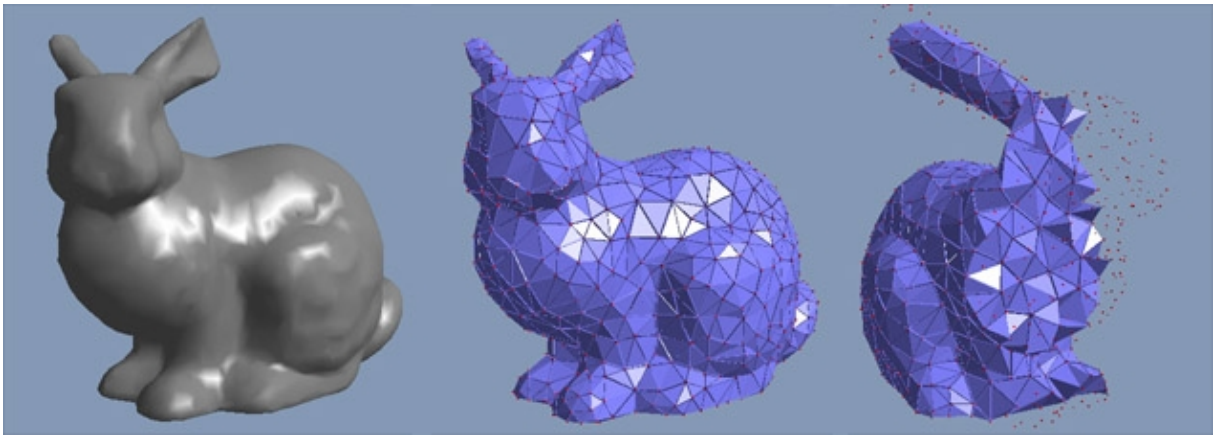
Cloth	Soft Body
Represented by a triangle mesh	Represented by a tetrahedral mesh
A cloth is a surface	A soft body is a volume
The triangle mesh is used for simulation **and** visualization	The tetrahedral mesh is used for simulation **only**
The artists triangle mesh can be used directly	A tetrahedral mesh has to be generated for the triangle mesh

Content Pipeline

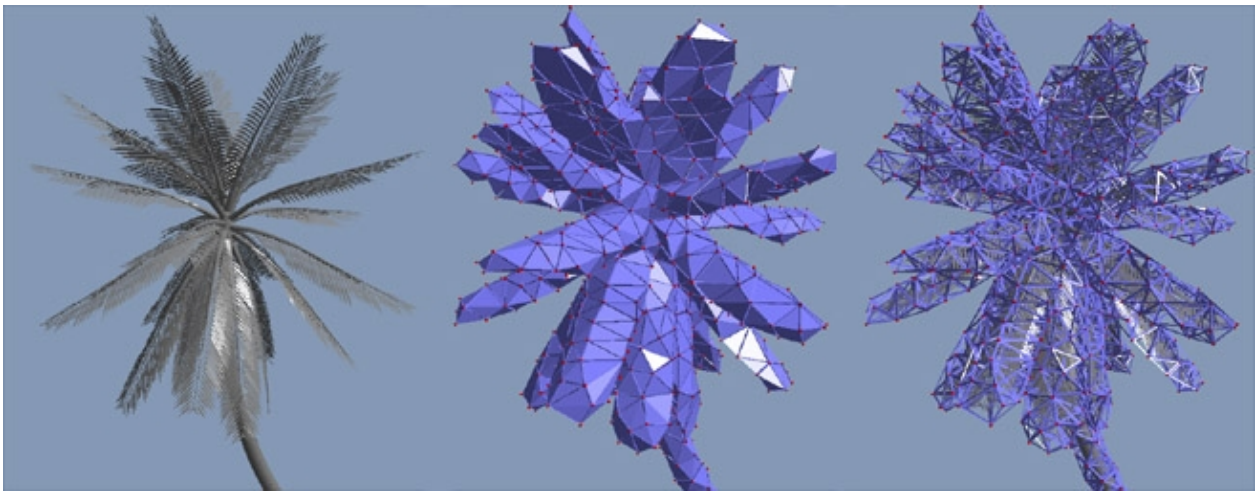
The content pipeline for soft bodies has some additional stages to the cloth pipeline.

1. The artist creates a detailed triangle mesh, possibly with textures, bump maps etc.
2. A tool is used to create a coarser tetrahedral mesh around the triangle mesh. (We provide tools for this purpose, which are illustrated in the PhysXViewer sample application)
3. The tetrahedral mesh is simulated using the NxSoftBody and NxSoftBodyMesh classes.
4. The original triangle mesh is deformed along with the tetrahedral mesh by the user.
5. The triangle mesh is displayed on the screen.

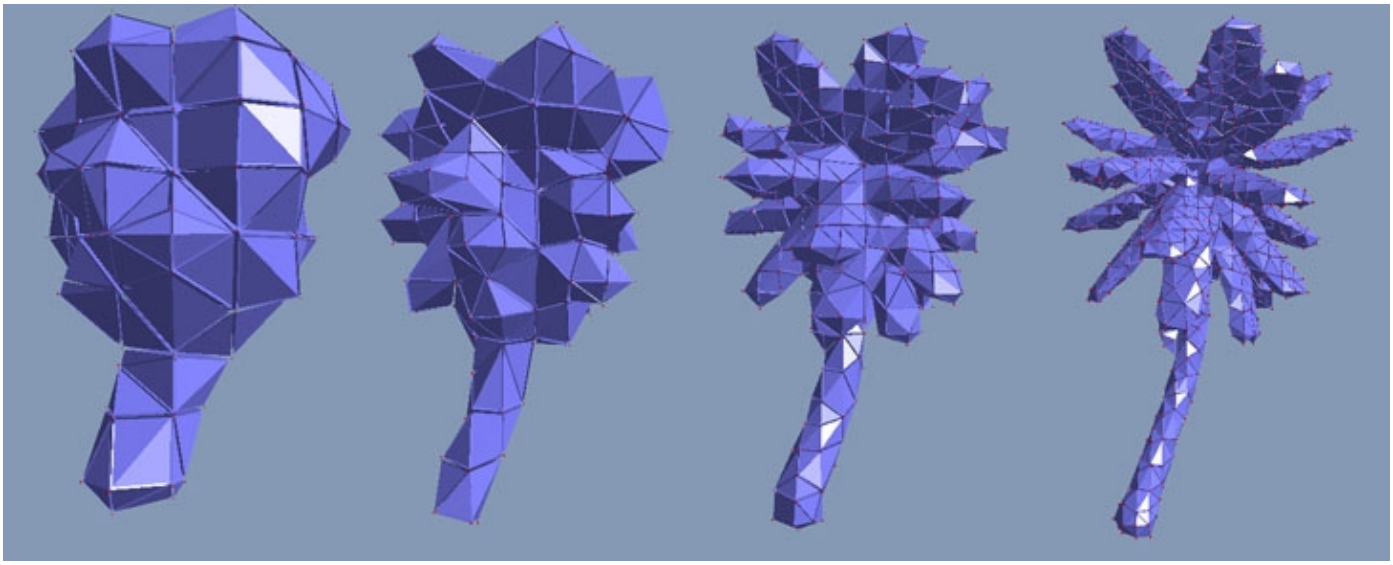
The following images show a tetrahedral mesh (middle, right) that was generated for a triangle mesh (left). The right most cut through the tetrahedral mesh shows that it covers the volume.



The process works for arbitrary triangle soups like the palm tree. The image on the right shows how the original triangle soup is embedded in the tetrahedral mesh which is shown in wireframe mode.



The resolution of the tetrahedral mesh can be chosen independently of the resolution of the triangle mesh. This allows for choosing a level of detail for the simulation without losing visual resolution. The following images show different resolutions of a palm tree mesh. With the coarser meshes, independent movement of individual leaves is not possible. To get this dynamic feature, more tetrahedra are needed as shown on the right:



API Reference

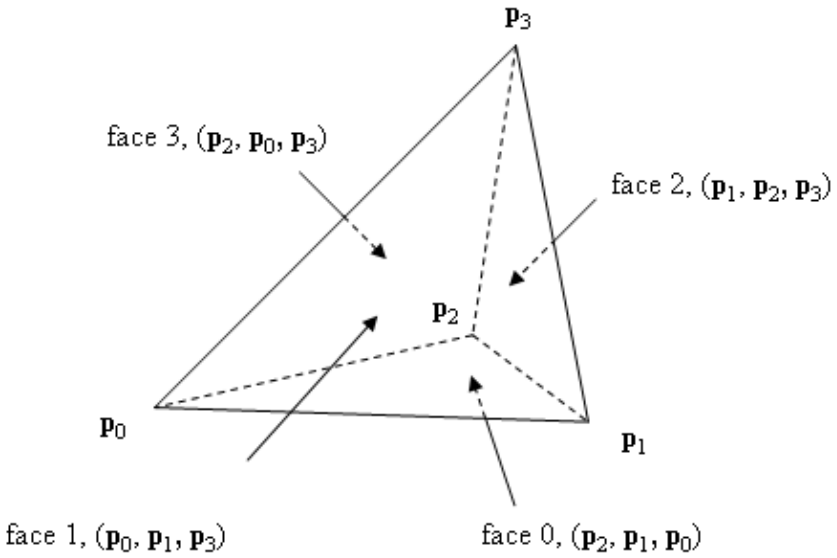
- NxSoftBody
- NxSoftBodyMesh

---

4 Soft Bodies

Soft Body Creation

A soft body is defined using a mesh which specifies the soft body's vertices and the constraints between them. Each constraint is associated with one tetrahedral volume element and is defined by specifying the 4 vertex indices of the tetrahedron's corner vertices.



tetrahedral volume element

The SDK expects the 4 vertices to be given in the sequence illustrated in the figure above, i.e., the 4 vertices should be given such that the four vertex triples defining the tetrahedron faces in the drawing appear in counter-clockwise order when being viewed at from the outside. (Our tetra-mesh generation tools take care of this automatically).

Creating a Soft Body Mesh

Similarly to how triangle meshes are handled in the cloth API, it is necessary to cook a soft body mesh into a suitable format for simulation. To do so, we need to fill in an NxSoftBodyMeshDesc structure and pass it to NxCookingInterface::NxCookSoftBodyMesh(). This function returns a stream of bytes that can either be saved and reused later or fed to NxScene::createSoftBodyMesh().

```
MemoryWriteBuffer wb;

if (!NxCookSoftBodyMesh(desc, wb)) return false;
```

The NxSoftBodyMeshDesc references a set of vertex positions and a list of indices. Groups of four indices are interpreted as tetrahedra. If soft body tearing is to be used, the user must raise the NxSoftBodyMeshFlags::NX_SOFTBODY_MESH_TEARABLE flag.

In a second step an NxSoftBodyMesh is instantiated using the stream

```
MemoryReadBuffer rb(wb.data);
```

```
softBodyMesh = scene->getPhysicsSDK().createSoftBodyMesh(rb);
```

Creating a Soft Body

Once the NxSoftBodyMesh has been created, it can be instanced in the form of NxSoftBody objects to go with the mesh. One soft body mesh may have many instances, for things like differentiating characters, or to illustrate certain conditions such as stiffness.

First, we need to fill in an NxSoftBodyDesc structure and call NxScene::createSoftBody() to create a soft body. Parameters for the soft body can be set either in the descriptor or later after the soft body has been created. See Soft Body Parameters for details.

```
NxSoftBodyDesc desc;

NxMeshData receiveBuffers;
//Fill in receive buffers...

desc.softBodyMesh = softBodyMesh;
desc.meshData = receiveBuffers;

mSoftBody = scene->createSoftBody(desc);
```

Examples

A Uniform 3D Block

```
/*
** Generate a regular 3d grid of vertices and connect them through tetrahedral
** constraints to generate a solid soft body block.
** One cube element in the grid is filled out with 5 tetrahedral constraints.
** numX, numY, numZ specify the number of cube elements in each dimension,
** h is the grid size
*/

int numX = 3;
int numY = 4;
int numZ = 5;
float h = 1.0f;

NxSoftBodyMeshDesc desc;

desc.numVertices= (numX+1) * (numY+1) * (numZ+1);
desc.numTetrahedra= numX*numY*numZ*5;
desc.vertexStrideBytes= sizeof(NxVec3);
desc.tetrahedronStrideBytes= 4*sizeof(NxU32);
desc.vertexMassStrideBytes= sizeof(NxReal);
desc.vertexFlagStrideBytes= sizeof(NxU32);
desc.vertices= (NxVec3*)malloc(sizeof(NxVec3)*desc.numVertices);
desc.tetrahedra= (NxU32*)malloc(sizeof(NxU32)*desc.numTetrahedra * 4);
desc.vertexMasses= 0;
desc.vertexFlags= 0;
desc.flags= 0;

int i,j,k;
NxVec3 offset(h * numX * 0.5f, h * numY * 0.5f, h * numZ * 0.5f);
NxVec3 *p = (NxVec3*)desc.vertices;
for (i = 0; i <= numX; i++) {
  for (j = 0; j <= numY; j++) {
    for (k = 0; k <= numZ; k++) {
      p->set(h*i, h*j, h*k);
```

```
        *p -=offset;
        p++;
      }
    }
  }

  int i1,i2,i3,i4,i5,i6,i7,i8;
  NxU32 *id = (NxU32*)desc.tetrahedra;
  for (i = 0; i < numX; i++) {
    for (j = 0; j < numY; j++) {
      for (k = 0; k < numZ; k++) {
        // compute the 8 corner vertices of the cube element
        i5 = (i*(numY+1) + j)*(numZ+1) + k; i1 = i5+1;
        i6 = ((i+1)*(numY+1) + j)*(numZ+1) + k; i2 = i6+1;
        i7 = ((i+1)*(numY+1) + (j+1))*(numZ+1) + k; i3 = i7+1;
        i8 = (i*(numY+1) + (j+1))*(numZ+1) + k; i4 = i8+1;

        // define 5 tetrahedral constraints per cube element,
        // by adding an index quadruple for each one
        if ((i + j + k) % 2 == 1) {
          *id++ = i1; *id++ = i2; *id++ = i6; *id++ = i3;
          *id++ = i6; *id++ = i3; *id++ = i7; *id++ = i8;
          *id++ = i1; *id++ = i8; *id++ = i4; *id++ = i3;
          *id++ = i1; *id++ = i6; *id++ = i5; *id++ = i8;
          *id++ = i1; *id++ = i3; *id++ = i6; *id++ = i8;
        }
        else {
          *id++ = i2; *id++ = i5; *id++ = i1; *id++ = i4;
          *id++ = i2; *id++ = i7; *id++ = i6; *id++ = i5;
          *id++ = i2; *id++ = i4; *id++ = i3; *id++ = i7;
          *id++ = i5; *id++ = i7; *id++ = i8; *id++ = i4;
          *id++ = i2; *id++ = i5; *id++ = i4; *id++ = i7;
        }
      }
    }
  }
```

API Reference

- NxSoftBody
- NxSoftBodyMesh
- NxScene

Soft Body Parameters

Soft body parameters may be specified during soft body creation, using members of the NxSoftBodyDesc, or after creation using members of NxSoftBody. Many of the soft body's properties have a similar, if not identical, interpretation as their NxCloth / NxClothDesc counterpart.

Flags

```
NxSoftBodyDesc::flags
```

or

```
NxSoftBody::setFlags()
```

The following flags may be used:

- NX_SBF_STATIC - Makes the soft body static.
- NX_SBF_DISABLE_COLLISION - Disable collision handling with the rigid body scene.
- NX_SBF_SELFCOLLISION - Enable/disable self-collision handling within a single soft body. Self-collisions are only handled inbetween the soft body's particles, i.e., particles do not collide against the tetrahedra of the soft body. The user should therefore specify a large enough particleRadius to avoid most interpenetrations. See NxSoftBodyDesc.particleRadius.
- NX_SBF_VISUALIZATION - Enable/disable debug visualization.
- NX_SBF_GRAVITY - Enable/disable gravity. If off, the soft body is not subject to the gravitational force of the rigid body scene.
- NX_SBF_VOLUME_CONSERVATION - Select volume conservation through NxSoftBodyDesc.volumeStiffness.
- NX_SBF_DAMPING - Enable/disable damping of internal velocities. Use NxSoftBodyDesc.dampingCoefficient to control damping.
- NX_SBF_COLLISION_TWOWAY - Enable/disable two way collision of the soft body with the rigid body scene. In either case, the soft body is influenced by colliding rigid bodies. If NX_SBF_COLLISION_TWOWAY is not set, rigid bodies are not influenced by colliding with the soft body. Use NxSoftBodyDesc.collisionResponseCoefficient to control the strength of the feedback force on rigid bodies.

  When using two way interaction care should be taken when setting the density of the attached objects. For example if an object with a very low or high density is attached to a soft body then the simulation may behave poorly.

  This is because impulses are only transfered between the soft body and rigid body solver outside the solvers.
- NX_SBF_TEARABLE - Defines whether the soft body is tearable. Make sure meshData.maxVertices and the corresponding buffers in meshData are substantially larger (e.g. 2x) than the number of original vertices since tearing will generate new vertices. When the buffer cannot hold the new vertices anymore, tearing stops. If this buffer is chosen big enough, the entire mesh can be torn into all constituent tetrahedral elements. (The theoretical maximum needed is 12 times the original number of vertices. For reasonable mesh topologies, this should never be reached though.)

  If the soft body is simulated on the hardware, additional buffer limitations that cannot be controlled by the user exist. Therefore, soft bodies might cease to tear apart further, even though not all space in the user buffer is used up.

  Note: For tearing in hardware, make sure you cook the mesh with the flag

NX_SOFTBODY_MESH_TEARABLE set in the NxSoftBodyMeshDesc.flags.

- NX_SBF_HARDWARE - Defines whether this soft body is simulated on the PPU. To simulate a soft body on the PPU set this flag and create the soft body in a regular software scene.
- NX_SBF_COMDAMPING - Enable/disable center of mass damping of internal velocities. This flag only has an effect if the flag NX_SBF_DAMPING is set. If set, the global rigid body modes (translation and rotation) are extracted from damping. This way, the soft body can freely move and rotate even under high damping. Use NxSoftBodyDesc.dampingCoefficient to control damping.
- NX_SBF_VALIDBOUNDS - Enable valid bounds for this cloth.

Volume And Stretching Stiffness

`NxSoftBodyDesc::volumeStiffness`

or

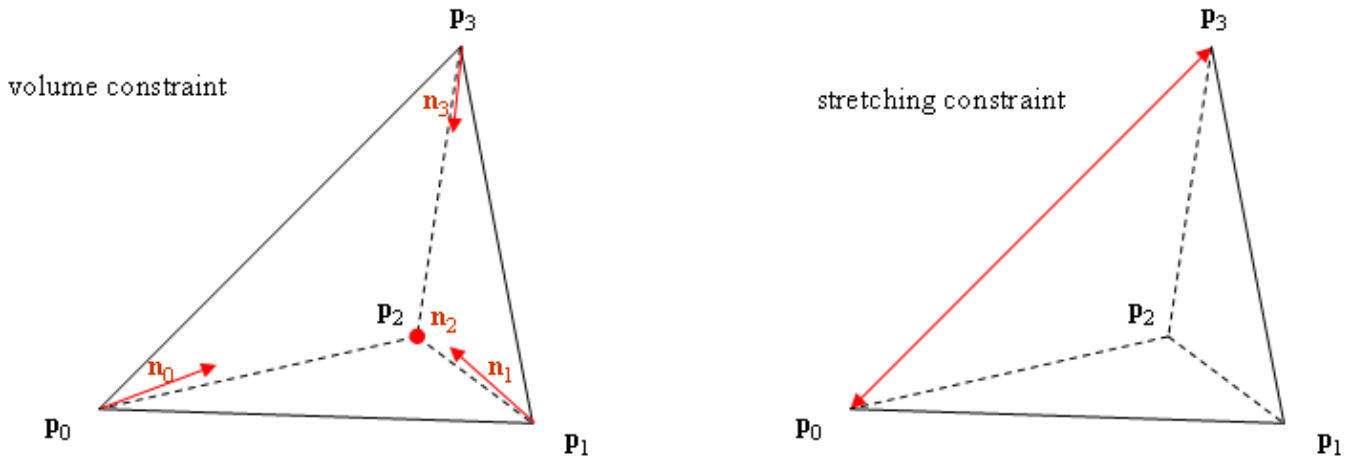
`NxSoftBody::setVolumeStiffness()`

`NxSoftBodyDesc::stretchingStiffness`

or

`NxSoftBody::setStretchingStiffness()`

Volume and stretching stiffnesses are used the specify how strongly the soft body responds to deviations from the rest state given by the input mesh. The soft body simulation manipulates the vertex positions to counteract deformations that change the volume and edge lengths of each tetrahedral element.



The stiffnesses of both constraint types can be set independently to a number in the range 0 to 1, similar to the bending and stretching stiffness values in the cloth simulation. The following combinations are typical:

	low volume stiffness	high volume stiffness

low stretching stiffness	The soft body vertices act as independently of each other as possible.	The soft body shape may now deviate strongly from the original configuration, but the preserved overall volume will still retain the original size.
high stretching stiffness	Only edge lengths are preserved. The simulation may still behave similarly as if a high volume stiffness were given. However, heavy deformations may cause tetrahedral elements to be inverted to their mirror image, causing the soft body to be trapped in an entangled state (from which it can usually recover though, once the volume stiffness is increased again).	The simulation maintains the original mesh configuration as closely as possible.

Note that even if the stretching stiffness is set very low, tetrahedron edges will cease to stretch further if their length exceeds a certain internal limit. This is done to prevent heavily degenerated tetrahedral elements which could occur if the vertices could move totally independent of each other.

Volume stiffness only has an effect if NX_SBF_VOLUME_CONSERVATION is set.

Density

`NxSoftBodyDesc::density`

NOTE: Density can only be specified when creating a soft body.

Density indirectly specifies the mass of the soft body particles. The mass of a particle is calculated as the sum of the following equation (for tetrahedra sharing a point):

1/4 * tetrahedronVolume * density

Particle Radius

`NxSoftBodyDesc::particleRadius`

or

`NxSoftBody::setParticleRadius()`

The particle radius specifies the radius of the particle's surrounding sphere used during collision detection, improving its robustness. Setting it very low is not recommended as this will lead to poor collision handling. This parameter is the soft body's analogue to the thickness parameter of the cloth.

Other Fields

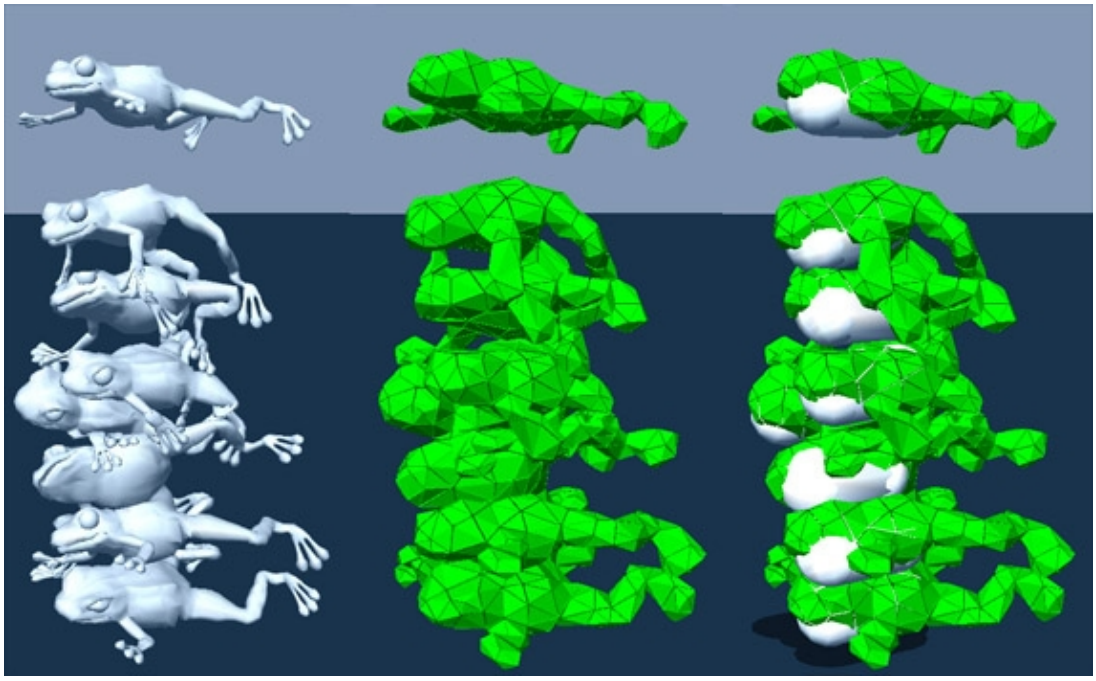
All other fields implement the same functionality as their cloth counterparts. E.g.

- Damping
- Solver Iterations
- Attachment Response Coefficent
- Collision Response Coefficient
- Friction
- External Acceleration
- Valid bounds

API Reference

- NxSoftBody
- NxSoftBodyMesh

Soft Body Attachments



Attaching Soft Bodies to Fixed Points

```
void attachVertexToGlobalPosition(const NxU32 vertexId, const NxVec3 &pos);
```

Soft body particles can be attached to fixed points specified in the global frame. The vertexId identifies the soft body particle to attach to the global position. The id corresponds to the index into the vertex position array passed to the API when cooking the soft body mesh, or, equivalently, the index of the vertex position as returned in the vertex position user buffer. The user may also attach vertices that get created during the tearing process.

A useful method for obtaining a vertex ID, for example when dragging with a mouse, is to cast a ray against the soft body. See Other Soft body Features for more information.

Attaching Soft Bodies to Shapes

```
void attachVertexToShape(NxU32 vertexId, const NxShape *shape, const NxVec3 &localPos, NxU32
```

Soft bodies can be attached to rigid body shapes in much the same way as attaching the soft body to a global position. But instead of specifying the global position, the position is relative to the shape (i.e., a position in the shape's coordinate frame). As the shape moves, this position will change in the global frame, pulling the soft body with it.

The attachment shape can be either static or dynamic (i.e., attached to a dynamic actor).

The attachmentFlags parameter specifies how the soft body interacts with the shape. The two possible values are shown below:

- NX_SOFTBODY_ATTACHMENT_TWOWAY - The object can pull the soft body and the soft body can move the object (only for dynamic shapes).

The default is one way interaction, i. e., the object can pull the soft body, but the soft body cannot move the object. If the object moves then the soft body will follow it. However, if the soft body is pulled, the object will behave as if it is fixed and not move.

NOTE: Care should be taken if objects with small masses (either through low density or small volume) are attached, as the simulation may easily become unstable. The NxSoftBodyDesc.attachmentResponseCoefficient field should be used to lower the magnitude of the impulse transfer from the soft body to the attached rigid body in such a situation. Other workarounds are simply increasing the density of the object or adding linear and angular damping to it.

- NX_SOFTBODY_ATTACHMENT_TEARABLE - When this flag is set the attachment can be torn. ie if enough force is exerted on the attachment then it will snap.

Attaching Soft Body to Shapes from Collisions

```
void attachToCollidingShapes(NxU32 attachmentFlags);

void attachToShape(const NxShape *shape, NxU32 attachmentFlags);
```

Since intermesh collisions are not supported, two soft bodies may freely penetrate each other. As a useful and fast workaround for this problem, rigid bodies can be used as cores inside soft bodies. As skeletons, they also give the soft body a firm interior and improve the global rubbery behavior of soft bodies. As an example, see the pile of frogs above. On the left you see the frog triangle meshes. The middle image shows the tetrahedral meshes used for simulation and on the right, the rigid cores are shown. In this case, capsules were used.

attachToCollidingShapes() attaches to any rigid body shape in the scene. attachToShape() is less indiscriminate and only attaches to a specified shape. attachmentFlags behaves in a similar way to the attachmentFlags parameter of attachVertexToShape().

It should be noted that this method of attachment will not work with general triangle meshes because the inside/outside of a general triangle mesh is not defined.

Detaching Soft Body

```
void detachFromShape(const NxShape *shape);

void freeVertex(const NxU32 vertexId);
```

Soft body particles can be detached as well as attached. detachFromShape() finds all the soft body particles that are attached to a particular shape and detaches them. freeVertex() detaches a specific vertex. The same vertex index that was used to attach a position or shape can be used again with attachVertexToGlobalPosition() or attachVertexToShape().

Tearable Attachments

When attaching a soft body to an actor the attachment can be flagged as tearable using NX_SOFTBODY_ATTACHMENT_TEARABLE, this allows the soft body to break the attachment if a sufficient force is applied to the attachment.

In fact the attachment is broken when the stretching constraint attached to the object exceeds a certain factor of its rest length. This factor is specified using:

```
NxSoftBodyDesc::attachmentTearFactor

void NxSoftBody::setAttachmentTearFactor(NxReal factor) ;
```

```
NxReal NxSoftBody::getAttachmentTearFactor();
```

The default is 1.5 so breakable attachments break when the constraint is stretched 50% more than its rest length.

API Reference

- NxSoftBody
- NxShape

Soft Body Rendering

Resulting soft body vertex positions are returned in a similar way as with cloth and fluids:

1. The user specifies a number of arrays to receive new soft body vertex positions.
2. The SDK updates these positions during simulation.
3. They are valid after fetchResults().

The SDK also updates counter variables to tell the user how many indices/vertices have been provided (this can change if the soft body is torn).

To provide the buffers to the SDK, the user should fill in an NxMeshData structure. This structure contains pointers to the appropriate arrays, along with the stride and counts for the positions, etc.. The mesh data can be specified when creating the soft body through the NxSoftBodyDesc::meshData member or updated later with the NxSoftBody::setMeshData (NxMeshData &meshData) function.

If the user does not require one or more of the arrays provided by the SDK, the appropriate pointer can be set to NULL. For example, if parent indices are not required, then the parentIndicesBegin field can be set to NULL and the SDK will not provide this information.

- verticesPosBegin - Pointer to the beginning of the array used to hold vertex positions. This array should be of at least sizeof(float) * maxVertices * 3. The vertex is stored as a 3 component float vector (xyz).
- verticesNormalBegin - Not used for soft body meshes. No normal data is generated for soft body vertices.
- verticesPosByteStride - The stride (number of bytes) from the start of one position to the next. This member allows application data to be interleaved with the positions.
- verticesNormalByteStride - Not used for soft body meshes.
- maxVertices - Maximum number of vertices which can be held by the array pointed to by verticesPosBegin.
- numVerticesPtr - Pointer to an integer value which is used by the SDK to communicate the actual number of vertices used (which may be less than the maximum available).
- indicesBegin - Pointer to an array of tetrahedron indices. Tetrahedra are specified as groups of 4 indices.
- indicesByteStride - The stride from the start of one index to the next.
- maxIndices - Max number of indices the index buffer can contain. This is four times the max number of tetrahedra.
- numIndicesPtr - Pointer to an integer which receives the number of tetrahedra indices in written to the buffer by the SDK.
- parentIndicesBegin - The pointer to the user specified buffer for vertex parent indices.
- parentIndicesByteStride - The stride from the start of one parent index to the next.

- maxParentIndices - Max number of parent indices the parent index buffer can contain.
- numParentIndicesPtr - Pointer to an integer which receives the number of parent indices that have been generated.
- flags
    - NX_MDF_16_BIT_INDICES - Specifies that indices are stored as 16 bit integers instead of 32 bit integers.

Parent indices are needed when vertices are duplicated by the SDK (e.g., soft body tearing). The parent index of an original vertex is its position in the verticesPos buffer. The parent index of a vertex generated by duplication is the index of the vertex it was copied from.

After the user retrieves the data, the tetrahedral mesh could be simply rendered as a collection of triangles. This would mainly be of interest for debugging purposes, as the tetrahedral mesh is usually much coarser than the model it was created from and not suitable for actual game content. To render the orginal surface mesh associated to the volumetric mesh, the user himself must update the surface mesh vertices through the new tetrahedron mesh vertex positions generated by the simulation. The SampleSoftBody achieves this by assiociating each original surface mesh vertex with exactly one soft body tetrahedron.

Example code from SampleSoftBody:

(ObjMeshTetraLink stores the tetraNr per vertex and the barycentric coordinates of the surface vertex with respect to the tetrahedron of the original soft body mesh configuration)

```
NxMeshData tetraMeshData;

// (fill in, cook, call simulate / fetchResults )

NxU32 numVertices = *tetraMeshData.numVerticesPtr;

NxU32 numTetrahedra = *tetraMeshData.numIndicesPtr / 4;

const NxVec3 *vertices = (NxVec3*)tetraMeshData.verticesPosBegin;

NxU32* indices = (NxU32*)tetraMeshData.indicesBegin;

for (int i = 0; i < (int)mVertices.size(); i++) {

ObjMeshTetraLink &link = mTetraLinks[i];

const NxU32 *ix = &indices[4*link.tetraNr];

const NxVec3 &p0 = vertices[*ix++];

const NxVec3 &p1 = vertices[*ix++];

const NxVec3 &p2 = vertices[*ix++];

const NxVec3 &p3 = vertices[*ix++];

// compute new surface vertex position using the barycentric coordinates
relative to the tetrahedron
```

```
NxVec3 &b = link.barycentricCoords;

mVertices[i] = p0 * b.x + p1 * b.y + p2 * b.z + p3 * (1.0f - b.x - b.y -
b.z);

}
```

API Reference

- NxSoftBody
- NxSoftBodyMesh
- NxScene

Other Soft Body Features

overlapAABBTetrahedra

```
virtual bool overlapAABBTetrahedra(const NxBounds3 bounds, NxU32& nb,
const NxU32*& indices)const;
```

The tetrahedra of the soft body mesh overlapping a user specified axis aligned bounding box can be retrieved using overlapAABBTetrahedra() (just like the triangles in cloth).

The tetrahedra indices correspond to the tetrahedra referred to by NxSoftBodyDesc.meshData. Tetrahedron i has the vertices 4i, 4i+1, 4i+2 and 4i+3 in the array NxMeshData.indicesBegin.

Miscellaneous

Many of the features of the soft body API provide the very same functionality as their cloth counterparts. Please refer to the cloth API documentation for more details.

E.g.

- Sleeping
- Raycasting
- Collision Filtering
- Adding a Force at a Vertex

API Reference

- NxSoftBody

Cloth and SoftBody interaction with Fluids

Introduction

Interaction between cloth/soft body and fluids is a new feature in 2.7.0 and is still in an experimental state. There are some things to think about when using this new feature:

- There are some types of scenarios that are difficult or impossible for the current algorithm to handle, e.g:
  - A piece of cloth or a soft body floating on a fluid.
  - A cloth bag filled with a fluid.
- The current implementation is not very efficient, and it is best to try and keep the number of cloths/soft bodies with the feature turned on at a minimum.
- Currently, each cloth / soft body with the fluid collision flag switched on collides against **all** fluids in the scene.
- Collision detection is performed for fluid particles against cloth / soft body particles, not against cloth triangles or soft body tetrahedra.
- The collision detection and response calculation is done in SW, even for HW cloth/soft bodies.

Usage

Even though the interaction feature affects both cloth/soft body and fluids, the feature is controlled through the API of cloth/soft body. The following section shows how to enable interaction for cloths, but the same scheme applies to softbodies. In the cases where there are differences, the soft body difference is noted in parenthesis.

To turn the interaction on, set the flag

```
NX_CLF_FLUID_COLLISION    (NX_SBF_FLUID_COLLISION)
```

in NxClothDesc.flags when creating the cloth. You can change the flag in runtime via the method

```
NxCloth.setFlags(NxU32 flags)
```

It is possible to control the intensity of the interaction on the cloth and on the fluid by specifying two response coefficients. It is possible to control the interaction so that for example the response of a collision is lower on the cloth than on the fluid.

```
//The intensity of the interaction on the cloth is controlled by
NxClothDesc.fromFluidResponseCoefficient

//The intensity of the interaction on the fluid is controlled by
NxClothDesc.toFluidResponseCoefficient
```

It is also possible to change these values in runtime using the following functions:

```
NxReal NxCloth.getFromFluidResponseCoefficient()
void NxCloth.setFromFluidResponseCoefficient(NxReal coefficient)

NxReal NxCloth.getToFluidResponseCoefficient()
void NxCloth.setToFluidResponseCoefficient(NxReal coefficient)
```

The interaction is always two-way, but you can simulate a one-way interaction by setting one of the coeffients to zero.

API Reference

- NxFluid
- NxCloth
- NxSoftBody

Force Fields

For rigid body and cloth simulations, forces can be applied to individual objects (or points in a cloth), but they require calls to the API for every object at every time-step. For simulations running on the PPU, this is particularly inefficient.

To remedy this, you can use *force fields*. These are SDK objects akin to actors, which affect cloth, soft bodies, fluid and rigid bodies that enter their area of influence. Force fields allow you to implement for example gusts of wind, dust devils, vacuum cleaners or anti-gravity zones.

Force fields possess two important geometrical properties: their volume of activity, which is defined by groups of shapes (similar to actors), and the *force field kernel* function, which defines the strength of the field throughout the volume of activity. One can chose between the *linear kernel* which is corresponding to the 2.7.2 force field function or one can create its own *custom kernel* from a set of basic operations.

Force Field - Creation

To create a force field, you fill out a descriptor (NxForceFieldDesc) with the desired parameters and then call NxScene::createForceField:

```
NxForceFieldLinearKernelDesc linearKernelDesc;
linearKernelDesc.constant = NxVec3(...);

NxForceFieldLinearKernel* pLinearKernel;
pLinearKernel = gScene->createForceFieldLinearKernel(linearKernelDesc);

NxForceFieldDesc fieldDesc;
fieldDesc.kernel = pLinearKernel;
fieldDesc.actor = ...;

NxForceField *pForceField;
pForceField = gScene->createForceField(fieldDesc);
```

Note: NxForceFieldDesc does need a valid kernel pointer. Never release a kernel which is still used by a force field.

Next step is to define the volume where the force field is active. This volume is represented by force field shapes which are created inside a group.

```
NxBoxForceFieldShapeDesc boxDesc;
boxDesc.dimensions = ...;

NxBoxForceFieldShape *pBoxFFShape = pForceField->getIncludeGroup().createShape(boxDesc);
```

See Force Field Shapes and Force Field Shape Groups for more information.

As for the parameters in the force field descriptor:

- *actor*: Pointer to the NxActor that this force field should be attached to. If this is NULL, the force field is static (attached to the world frame). Detaching from the actor will cause the force field's pose to be relative to the world frame again.
- *clothType*: Type of scaling which is used for interactions with cloth.
- *coordinates*: Type of coordinate system used in the force field kernel function.
- *flags*: Force field flags. See force field scaling.
- *fluidType*: Type of scaling which is used for interactions with fluids.
- *forceFieldVariety*: Index to the force field scaling table. A variety index != 0 has to be created by calling NxScene::createForceFieldVariety().
- *group*: Collision group used for interaction filtering.
- *groupsMask*: Groups mask used for interaction filtering.
- *includeGroupShapes*: Array of force field shape descriptors which will be created inside the include group of this force field. This group moves with the force field and cannot be shared.
- *kernel*: Pointer to the force field kernel function.
- *name*: Possible debug name. The string is not copied by the SDK, only the pointer is stored.
- *pose*: Location and orientation of the force field. If *actor* is NULL, this is relative to the world frame, otherwise it is relative to the actor frame of the specified actor.
  The force field shapes of the include group are specified in this frame.
- *rigidBodyType*: Type of scaling which is used for interactions with rigid bodies.
- *shapeGroups*: Array of shape groups which define the volume of the force field. Those groups can be shared among other force fields within the same scene.
- *softBodyType*: Type of scaling which is used for interactions with soft bodies.
- *userData*: User can assign this to whatever, usually to create a 1:1 relationship with a user object.

Note that the force field function defines a force for every point in space. The actual volume affected is determined by the force field's groups.

The actual force working on an object can be further scaled depending on the type of object: cloth, fluid, soft body or rigid body and the 2 dimensional scaling table. See force field scaling.

If an object is affected by more than one force field, the forces are simply added together.

You can sample the force that would affect a body with a given velocity at a given point in space by using the NxForceField::samplePoints method.

API Reference

- NxForceField
- NxForceFieldDesc
- NxForceFieldShape
- NxForceFieldShapeDesc
- NxForceFieldShapeGroup
- NxForceFieldShapeGroupDesc
- NxForceFieldLinearKernel
- NxForceFieldLinearKernelDesc

Force Field - Shapes

Function

The shapes belonging to a force field group determine the volume of space in which the force field is active. As with actor shapes, force field shapes come in different types; for force fields, the available shapes are:

- Sphere
- Capsule
- Box
- Convex mesh

Unlike actors, you can specify some of the shape groups as *exclude* groups using the flag NX_FFSG_EXCLUDE_GROUP. Thus, the actual volume used is the union of all include shapes' volumes, minus the union of all exclude shapes' volumes. This provides a flexible means of shaping the force field just like you want it, with dead zones, irregular edges and the like.

Note that the shapes of a forcefield only determine *what* objects are affected, not *in what way*. The latter is the purview of the force field kernel function.

Creation

You create a shape in a force field by filling out a descriptor and calling NxForceFieldShapeGroup::createShape:

```
NxBoxForceFieldShapeDesc boxDesc;
boxDesc.dimensions = ...;

NxBoxForceFieldShape *pBoxFFShape = pForceField->getIncludeGroup().createShape(boxDesc);
```

There are four different descriptor classes, one for each force field shape type. Some properties are common while others are specific to the shape type:

- Common:
  - *name*: Optional name field.
  - *pose*: The shape's pose relative to the force field's frame.
  - *userData*: Optional user data pointer; will be copied to NxForceFieldShape::userData.
- Sphere:
  - *radius*: Radius of the sphere.
- Capsule:
  - *height*: Height of the capsule (length of cylindrical segment).
  - *radius*: Radius of the capsule.
- Box:
  - *dimensions*: Center-to-corner vector defining the size of the box.
- Convex:
  - *meshData*: Pointer to the NxConvexMesh to be used for the convex shape.

API Reference

- NxForceField
- NxForceFieldDesc
- NxForceFieldShape
- NxForceFieldShapeDesc

- NxForceFieldShapeGroup
- NxForceFieldShapeGroupDesc
- NxBoxForceFieldShape
- NxBoxForceFieldShapeDesc
- NxSphereForceFieldShape
- NxSphereForceFieldShapeDesc
- NxCapsuleForceFieldShape
- NxCapsuleForceFieldShapeDesc
- NxConvexForceFieldShape
- NxConvexForceFieldShapeDesc

Force Field - Shape Groups

Function

A shape group is simply a collection of force field shapes. The rationale for shape groups is that a given force field (say, one that causes particles to swirl around an actor) may be instanced multiple times, once for each of a collection of actors - but each field is likely to have the same set of exclude shapes. To free the application from having to add this collection to each field it creates, a group of exclude shapes may be created which may then be attached to several fields.

The pose of the shapes in a force field group is relative to the world frame. The group flag (NX_FFSG_EXCLUDE_GROUP) defines whether all shapes inside this group are exclude volumes or include volumes. There is one special include group per force field which cannot be shared among other force fields and which is moving with the force field. The pose of the shapes in this include group is relative to the force field pose and the shapes are include volumes only.

Creation

You create a force field group in a scene by filling out a descriptor and calling NxScene::createForceFieldShapeGroup:

```
NxBoxForceFieldShapeDesc excludeShapeDesc;
excludeShapeDesc.dimensions = NxVec3(...);
excludeShapeDesc.pose.t = NxVec3(...);

NxForceFieldShapeGroupDesc sgDesc;
sgDesc.flags = NX_FFSG_EXCLUDE_GROUP;
sgDesc.shapes.pushBack(&excludeShapeDessc);

NxForceFieldShapeGroup* pExcludeGroup;
pExcludeGroup = gScene->createForceFieldShapeGroup(sgDesc);

excludeShapeDesc.pose.t += 10.0f;
NxBoxForceFieldShape *pAnotherExcludeShape = pExcludeGroup->createShape(excludeShapeDesc);

...
pForceField->addShapeGroup(*pExcludeGroup);
```

As for the parameters in the force field shape group descriptor:

- *flags*: Force field shape group flags.
  - *NX_FFSG_EXCLUDE_GROUP*: Defines whether the shapes in this group are include or exclude volumes.
- *name*: Possible debug name. The string is not copied by the SDK, only the pointer is stored.
- *shapes*: A list of force field shape descriptors which will be added to the group.
- *userData*: User can assign this to whatever, usually to create a 1:1 relationship with a user object.

API Reference

- NxForceField
- NxForceFieldDesc
- NxForceFieldShape
- NxForceFieldShapeDesc
- NxForceFieldShapeGroup
- NxForceFieldShapeGroupDesc

Force Field - Coordinate Systems

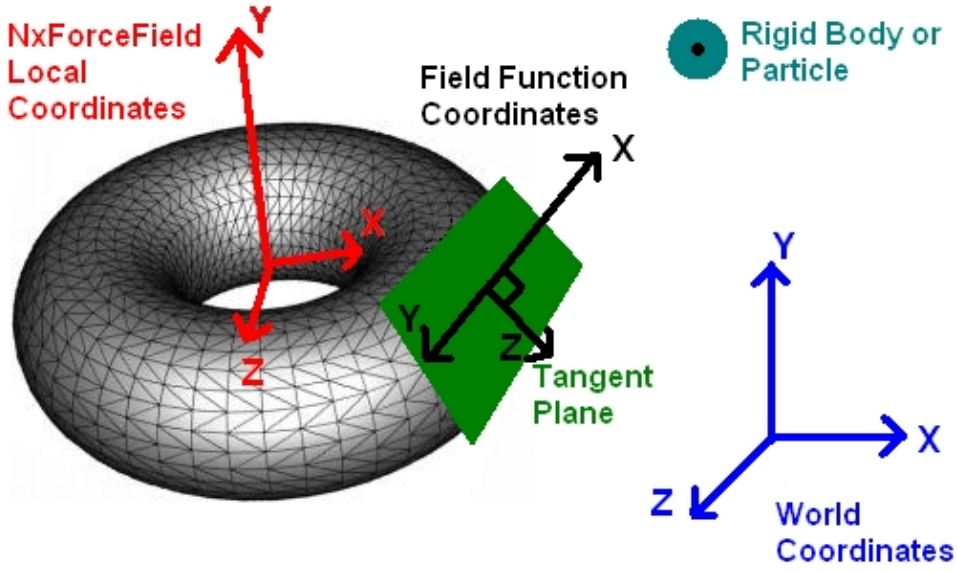
The force field does support different kinds of coordinate systems to help creating various effects with the linear kernel and to simplify custom kernel code. The force field class transforms the position and velocity of intersecting objects (rigid body, fluid, cloth, softbody) into the selected coordinate system and the force field's local frame. Those two variables are then passed to the force field kernel function as input parameters.

The variables are provided in the local, "tangential" coordinate system of the force field at the position of the object. For the coordinate type NX_FFC_CARTESIAN this is equivalent to the force field's own frame; for the other types the local axes shift depending on position:

- NX_FFC_SPHERICAL: The X axis is directed outwards from the center of the sphere; the Y and Z axes are the tangents of the spherical surface. The y and z components of the **p** and **v** vectors in the function are always 0, and target position/velocity in these directions is ignored.
- NX_FFC_CYLINDRICAL: The X axis is directed outwards from the cylinder axis. The Y axis is the cylinder axis, and the Z axis is the cylindrical surface tangent (directed to make XYZ a right-handed system). The z component of the **p** vector in the function is always 0, and the target position in this direction is ignored.
- NX_FFC_TOROIDAL: The X axis is directed outwards from the closest point on the torus' equator; the Y axis is the tangent of the equator at its closest point (directed in the clockwise direction as seen from the direction of the torus' axis), and the Z axis is normal to both forming a right-handed system XYZ. See the figure below. The y and z components of the **p** vector in the function are always 0 and the target position in these directions is ignored. Note that this coordinate system is only available for linear kernels!

Please note that there are potential singularities in the force function, at the axis of a cylinder or torus. You may wish to exclude these singularities from the force field, possibly using exclusion shapes.

Here is an illustration of the local coordinates for a toroidal force field:



The resultant force, **f**, is also in the local frame, with one exception: The constant part actually works in the global frame.

API Reference

- NxForceField
- NxForceFieldDesc
- NxForceFieldLinearKernel
- NxForceFieldLinearKernelDesc

Force Field - Kernels

The kernel is the function which computes a force and torque on a particle (or volume element or rigid body) with a particular position and linear velocity. The particle has to be inside the force field's activity volume defined by the shape groups. The force fields operate either with a *linear kernel* or a *custom kernel*.

Linear Kernels

Function

The *linear kernel* implements a function which was also used by the force fields in the SDK version 2.7.2. The resulting force in each dimension i is calculated through the following formula:
$\mathbf{f}_i = [\mathbf{K} + \mathbf{M}_p(\mathbf{P}-\mathbf{p}) + \mathbf{M}_v(\mathbf{V}-\mathbf{v}) + \mathbf{N}\,\mathbf{a}_{noise}]_i \,/\, [1 + \mathbf{l}_i|P_i-p_i| + \mathbf{q}_i(P_i-p_i)^2]$
Here,

- **f** is the resulting force (or acceleration, see below) on the particle
- **K** is the constant part, from NxForceFieldLinearKernelDesc.constant
- **p** is the position of the object, in the force field coordinate system
- $\mathbf{M}_p$ is the position dependence matrix, from NxForceFieldLinearKernelDesc.positionMultiplier
- **P** is the position target, from NxForceFieldLinearKernelDesc.positionTarget
- **v** is the velocity of the object, in the force field coordinate system
- $\mathbf{M}_v$ is the velocity dependence matrix, from NxForceFieldLinearKernelDesc.velocityMultiplier
- **V** is the velocity target, from NxForceFieldLinearKernelDesc.velocityTarget
- $\mathbf{a}_{noise}$ is the noise multiplier, from NxForceFieldLinearKernelDesc.noise
- **N** is a 3x3 matrix whose diagonal elements are three random numbers in the range [0, 1]
- **l** is the linear falloff term, from NxForceFieldLinearKernelDesc.falloffLinear
- **q** is the quadratic falloff term, from NxForceFieldLinearKernelDesc.falloffQuadratic

In words, the SDK calculates the difference between actual and target position/velocity, and uses the matrices $\mathbf{M}_p$ and $\mathbf{M}_v$ to calculate the response to these differences.
The falloff terms are used to attenuate the magnitude of the force with the distance to the target position. Note that the attenuation is done independently for each coordinate, and that none of the coefficients may be below zero.

Creation

You create a force field linear kernel in a scene by filling out a descriptor and calling NxScene::createForceFieldLinearKernel:

```
NxForceFieldLinearKernelDesc lKernelDesc;

lKernelDesc.constant = NxVec3(...);
lKernelDesc.positionTarget = NxVec3(...);
lKernelDesc.noise = NxVec3(...);

NxForceFieldLinearKernel* pLinearKernel;
pLinearKernel = gScene->createForceFieldLinearKernel(lKernelDesc);

NxForceFieldDesc ffDesc;
ffDesc.kernel = pLinearKernel;

NxForceField* pForceField;
pForceField = scene->createForceField(ffDesc);
```

As for the parameters in the force field linear kernel descriptor:

- *constant*: Location-invariant part of the force field function.
- *falloffLinear*: Linear falloff term, see the force field function.
- *falloffQuadratic*: Quadratic falloff term, see the force field function.
- *name*: Possible debug name. The string is not copied by the SDK, only the pointer is stored.
- *noise*: Random component of the force field function.
- *positionMultiplier*: Location dependent part of the force field function.
- *positionTarget*: Equilibrium point of location dependent force component.
- *torusRadius*: Radius of toroidal force field function coordinate system, if used.
- *userData*: User can assign this to whatever, usually to create a 1:1 relationship with a user object.
- *velocityMultiplier*: Velocity dependent part of the force field function.
- *velocityTarget*: Equilibrium velocity of velocity dependent force component.

Custom Kernels

Function

A custom kernel is a virtual machine definition which compiles for SW implementations using the platform compiler at build time, and compiles for HW using a runtime compilation system. Applications should not require anything special (or any special tools) in order to enable runtime compilation when HW is present, and the runtime must be able to compile any kernel (up to size limitations) - restriction which leads to a slightly unnatural syntax and limited functionality for kernel definition.

Custom Kernel Constants

Kernel constants are the parameters of a custom kernel function which the user can set with get/set accessors. Kernel constants are declared using the syntax (bool, float, NxVec3):
`NxBConst(name); | NxFConst(name); | NxVConst(name);`
For example a constant declared as `NxVConst(Offset)` will result in following member functions:

```
setOffset(const NxVec3 &val);
NxVec3 getOffset() const;
```

Custom Kernel Statements

A kernel consists of a sequence of statements, each of which is either an assignment or an early-out. Assignment syntax is as in C++:

Assignments are:

- `(NxBoolean)` *`var`* `(=)` *`boolean expression;`*
- `(NxFloat)` *`var`* `(= | += | -= | *=)` *`float expression;`*
- `(NxVector)` *`var`* `(= | += | -= )` *`vector expression;`*

Early-outs are:

- **`NxFailIf(`***`boolean expression`***`);`**
- **`NxFinishIf(`***`boolean expression`***`);`**

If the expression in a *NxFailif* statement is true, the kernel aborts, and no force is applied to the object If the expression in a *NxFinishIf* statement is true, the kernel finishes, and the current values of the force and torque variables are used.

Operators are:

- `NxBoolean: &, |, ^`

- NxFloat: **+, - (unary and binary), *, recip, sqrt, recipSqrt, NxSelect, ==, !=, , <=, >=**
- NxVector: **+, - (unary and binary), * (multiply by scalar), cross, dot, magnitude, NxSelect, .getX(), .getY(), .getZ(), .setX(), .setY(), .setZ()**

Inline constants are supported for NxBoolean and NxFloat types.
NxSelect is a ternary operator analogous to the C ?: operator: NxSelect(b,f1,f2) chooses f1 if b evaluates to true and otherwise f2.
recip and recipSqrt return zero if their operand is zero or smaller than the SDK parameter NX_FORCE_FIELD_CUSTOM_KERNEL_EPSILON (or in the case of recipSqrt if the operand is negative).
Short-circuiting is not supported: both arms of a select statement or boolean operator are always evaluated.
Kernels have two implicit input constants: *Position* and *Velocity*, and two output variables *force* and *torque* whose values are zero on entry.

Creation

You create a force field custom kernel in a scene by creating a .h file which is structured in the following way:

```
-------- SimpleKernel.h --------
"NxForceFieldKernelDefs.h"

// NxForceFieldKernelSimple will be the name of this kernel class
NX_START_FORCEFIELD(Simple)

// bindable constants - e.g. the height of your tornado
NxVConst(Foo);
NxFConst(Bar);
NxBConst(Orc);

NX_START_FUNCTION
// all kernel function code goes in here and have to be statements
// which were described in the Custom Kernel Statements section of the user guide.
// input are the vectors Position and Velocity transformed into the chosen coordinate system.
// output are the vectors force and torque

// sample instructions
NxFloat a = 2.0f;
NxFloat b = NxSelect(Orc, 2.0f, 15.0f);
NxVector x = Foo * Velocity;
NxFailIf(1.0f <a | b > x.getX());

force = Position * x.dot(x);
NX_END_FUNCTION

NX_END_FORCEFIELD(Simple)
"SimpleKernel.h"
...
NxForceFieldKernelSimple* pSimpleKernel;
pSimpleKernel = new NxForceFieldKernelSimple();
...
NxForceFieldDesc ffDesc;
ffDesc.kernel = pSimpleKernel;

NxForceField* pForceField;
pForceField = scene->createForceField(ffDesc);
```

Note: Do not release the kernels before all force fields which have a reference to this kernel are released.
Do not call any of the public methods of the custom kernel class other than the get/set accessors for the user defined constants.
You are responsible that the kernel is returning valid floating point results. To high scaling values or const

variables can produce bad floats (NaN, Inf).
There is a "checked build" of the SDK which tries to catch and report such bad data.

API Reference

- NxForceField
- NxForceFieldDesc
- NxForceFieldLinearKernel
- NxForceFieldLinearKernelDesc

Force Field - Scaling

Function

The force value obtained from the kernel are then scaled. Each kind of object (fluid, cloth, soft body, rigid body) has its own scaling properties. Conceptually scaling depends on two orthogonal criteria: resolution independence (a property of the simulation) and charge value (a property of the way the field acts upon the object). Hence there are two scaling properties:

- *Field Type*: An integer defining the field type per feature:
  - *NX_FF_TYPE_GRAVITATIONAL*: Scales the force by the mass of the particle or body.
  - *NX_FF_TYPE_OTHER*: Does not scale the value from the force field.
  - *NX_FF_TYPE_NO_INTERACTION*: Does disable force field interaction with a specific feature.

- *Volumetric Scaling*: A flag which indicates whether the force is scaled by the amount of volume represented by the object if NX_FF_TYPE_OTHER is selected. This is true by default for all features. This allows fields to act on cloth, fluids and soft bodies to be independent of the discretization.
  - *NX_FFF_VOLUMETRIC_SCALING_FLUID*: Flag to enable volumetric scaling for fluids.
  - *NX_FFF_VOLUMETRIC_SCALING_CLOTH*: Flag to enable volumetric scaling for cloths.
  - *NX_FFF_VOLUMETRIC_SCALING_SOFTBODY*: Flag to enable volumetric scaling for softbodies.
  - *NX_FFF_VOLUMETRIC_SCALING_RIGIDBODY*: Flag to enable volumetric scaling for rigid bodies.

- *Scaling Table*: In addition there is a scaling table where the force field's *variety* and the feature's *force field material* index are used to index the table.

Creation

You setup the force field scaling by filling out a descriptor and calling NxScene::createForceField:

```
NxForceFieldDesc ffDesc;

// enable volumetric scaling for all features but rigid body
ffDesc.fluidType     = NX_FF_TYPE_OTHER;
ffDesc.clothType     = NX_FF_TYPE_OTHER;
ffDesc.softBodyType  = NX_FF_TYPE_OTHER;
ffDesc.rigidBodyType = NX_FF_TYPE_OTHER;

ffDesc.flags =  NX_FFF_VOLUMETRIC_SCALING_FLUID |
                NX_FFF_VOLUMETRIC_SCALING_CLOTH |
                NX_FFF_VOLUMETRIC_SCALING_SOFTBODY |;

...

NxForceField *pForceField;
pForceField = gScene->createForceField(ffDesc);
```

To create an entry in the scaling table you have first to create the material and variety in the NxScene:

```
// create a new index (index 0 is already created)
NxForceFieldVariety var = gScene->createForceFieldVariety();
NxForceFieldMaterial mat = gScene->createForceFieldMaterial();

// change the scaling for the created pair, default is 1.0f
```

```
gScene->setForceFieldScale(var, mat, 0.5f);

// sets the scaling for all objects which have the default force field material 0
gScene->setForceFieldScale(var, 0, 3.0f);

// change the variety of a force field
pForceField->setForceFieldVariety(var);

// change the material of an object (cloth)
pCloth->setForceFieldMaterial(mat);
```

Note:
Torque values are never scaled.
Very big or small scaling values can produce invalid floating point results (NaN, Inf) in and after the kernel evaluation of the force field.
There is a "checked build" of the SDK which tries to catch and report such bad data.
2.8 scaling options correspond to the 2.7 force field options as follows:

NX_FORCE_TYPE_ACCELERATION	NX_FF_TYPE_GRAVITATIONAL
NX_FORCE_TYPE_FORCE	NX_FF_TYPE_OTHER
NX_FORCE_TYPE_FORCE with NX_FFF_LEGACY_FORCE	NX_FF_TYPE_OTHER without volumetric scaling

API Reference

- NxActor
- NxCloth
- NxFluid
- NxForceField
- NxForceFieldDesc
- NxScene
- NxSoftBody

Character Controller

The goal of the character controller SDK is to provide a default built on top of the NVIDIA PhysX SDK. Roughly it has to support the following:

- Character control
- Character interactions

This covers a very high number of features, which can be implemented in numerous ways. The goal is *not* to implement all of them (which would be a daunting task), but to offer a default implementation that people can use as a starting point. For example, the character's bounding volume could be anything, from a box to an inverted pyramid; therefore, in our initial implementation we support two common bounding volumes: an AABB and a capsule.

One might wonder why we didn't use the physics engine directly to implement the character controller. Here is the story.

Implementation Decisions

In the past, games didn't use "real" physics engines. However, they still used a character controller to move a player in a level. These games, such as *Quake* or even *Doom*, had a dedicated, customized piece of code to implement *collision detection and response*, which was often the only piece of physics in the whole game. It actually had little physics, but a lot of carefully tweaked values to provide a good *feeling* while controlling the player. The particular behavior it implemented is often called the " *collide and slide"* algorithm, and it has been tweaked for more than a decade. The result is that players expect to find the same well-known behavior in new games, and providing them with anything else is often dangerous (a few games come to mind but I'm not sure it's appropriate to name them). This is especially true if provided behavior is not as robust and stable as before, which is exactly what happens if you use a typical physics engine directly to control players.

In particular, here is a (non-exhaustive) list of typical problems when using a physics engine for character controllers:

- **(Lack Of) Continuous Collision Detection:** Typical physics engines use discrete collision checks, leading to the famous tunneling effect that has plagued various commercial & non-commercial physics packages for years, which leads to three main problems:
  - The tunneling effect itself - if the character goes too fast, it might tunnel through a wall.
  - As a consequence, the maximum velocity of the character might be limited (hence also limiting the game-play possibilities).
  - Even without tunnel, the character might jitter when pushed forward in a corner. For example, the engine keeps moving it back and forth to slightly different positions.

- **No Direct Control:** A rigid body is typically controlled with impulses or forces. It is nearly impossible to move it directly to its final position until you have converted the delta position vector to impulses or forces and applied them in hopes that the character will be where you wanted it to be as a result. Usually it doesn't work very well, in particular when the physics engine uses an imperfect linear solver.

- **Trouble with Friction:** When the character is standing on a ramp, you don't want it to slide. Infinite friction is desired. When the character is moving forward *on that same ramp*, or sliding against a wall, you don't want it to slow down, thus a null friction is needed. These are usually defined with either 0 or infinite. However, the friction model might not be perfect, and what you actually get is very little friction, so you can still feel the character slowing down, or a high-but-not-infinite friction, so the character slides very slowly on that ramp no matter how artificially high the friction parameters are. The conflicting requirements for ramps mean that usually there is simply no way to perfectly model desired behavior.

- **Trouble with Restitution:** Basically you don't want any restitution, ever. When the character moves fast and collides with a wall, you don't want it to bump against it. When the character falls from a height and lands on the ground, flexing his legs, you definitely don't want any bumps, which would visually look terrible. But once again, even when the restitution is exactly zero, you sometimes get a small bump nonetheless. This is not only related to the non-perfect nature of the linear solver, but also has to do with how typical penetration-depth-based engines recover from overlap situations, sometimes applying too high a force that repels objects more than desired.

- **Undesired Jumps:** It is often important that a character stick to the ground, no matter what the physical behavior should be. For example, characters in action games tend to move fast, at unrealistic speeds. When they reach the top of a ramp, the physics engine often makes them jump a bit, in the same way a fast car would jump in the streets of San Francisco. But that is often not desired: the character should stick to the ground regardless of its current velocity. This is sometimes implemented using fixed joints, which is a terrible, terrible solution to a very simple problem that has been solved for years without requiring all the modern complexity of a physics engine.

- **Undesired Rotations:** Finally, a character is always standing up and never rotating. However, a physics engine often has poor support for that kind of constraint, and a great deal of effort is put into just preventing a capsule around the character from falling (it should always stand up on its tip). This too is sometimes implemented using artificial joints, and the resulting system is neither very robust nor very fast.

In summary, a lot of time is spent in disabling the physics engine's features, one after the other, for the whole purpose of emulating an otherwise simple piece of customized code. This is not the correct approach.

Character Controller - Creation

To create a Character Controller, the first task is to assign a bounding volume. Currently, only boxes (*NxBoxController*) and capsules (*NxCapsuleController*) are supported.

Second, create a controller manager within the application to keep track of all created controllers and to allow your character to interact with other characters created by the same manager. This is a simple class so create the following variable:

```
NxControllerManager* gManager = NxCreateControllerManager(myAllocator);
```

Note: The controller manager is supposed to be a singleton. In other words, each application should take the appropriate steps to ensure only one instance of NxControllerManager is created.

Third, create one controller for each movable character using the following code:

```
NxScene* scene;//Your PhysX scene

NxCapsuleControllerDesc desc;

<fill the descriptor here>

NxController* c = gManager->createController(scene, desc);
```

When you no longer need to controller manager, you should release it:

```
NxReleaseControllerManager(gManager);
```

Note: the old class "ControllerManager" is still available, but it has been deprecated and will not be supported in the next version of PhysX.

API Reference

- NxCapsuleController
- NxCapsuleControllerDesc
- NxController
- ControllerManager

Character Controller - Update

In each frame, characters can be moved using the following function:

```
NxController::move(constNxVec3& disp, NxU32 activeGroups, NxF32 minDist, NxU32& collisionFlags
                   NxF32 sharpness=1.0f, const NxGroupsMask* groupsMask=NULL);
```

- **disp** is the displacement vector for the current frame. It is typically a combination of a vertical motion due to gravity and a lateral motion when your character is moving. Note that this is a displacement vector, i.e., a first order control. This is not an impulse (second order control) or a force (third order control) vector.
- **activeGroups** is a bit mask representing active collision groups, i.e., groups that should be included in the collision tests.
- **minDist** is a minimal length used to stop the recursive displacement algorithm early when remaining distance to travel goes below this limit.
- **collisionFlags** is a bit mask returned to users to define collision events that happened during the move. This is a combination of *NxControllerFlag* flags. It can be used to trigger various character animations. For example, your character might be falling while playing a falling idle animation, and you might start the land animation as soon as *NXCC_COLLISION_DOWN* is returned.
- **sharpness** is used to smooth motion with a feedback filter, having a value between 0 (so smooth it doesn't move) and 1 (no smoothing = unfiltered motion). Sharpness can ease the motion curve when the auto-step feature is used with boxes. Not needed with the capsule controller.
- **groupsMask** is used when you want to temporarily use an alternative mask for filtering the shapes that the controller can collide with, see NxScene::overlapAABBShapes().

Graphics Update

In each frame, keep the graphics object in sync with the position of the character controller. A controller never rotates so only its position can be accessed, using the following code:

```
const NxVec3& NxController::getPosition()const;
```

If you use this position in your world matrix and pass this to the renderer, the character should collide and slide smoothly against the world.

API Reference

- NxController

Character Controller - Volume

The character uses a bounding volume that is independent from already existing shapes in the SDK. Thus a dedicated collision volume for the character controller can be implemented (e.g., an ellipsoid), even if the corresponding *NxShape* doesn't exist.

Currently, the PhysX SDK supports two different shapes that may surround a character:

- An AABB, defined by a position and an extents vector. The AABB does not rotate. It always has a fixed rotation even when the player is (visually) rotating. This avoids getting it stuck in places not big enough to allow rotation.
- A capsule, defined by a position, a height and a radius. The capsule has a better behavior when climbing stairs for example. However it might be slightly more consumptive in terms of CPU time.

NOTE: In versions prior to 2.3 there was an *NxSphereController* - this has been removed since the *NxCapsuleController* is more robust and provides the same functionality (zero length capsule).

NOTE: Both types of controller use the Sweep API. This means that not all collidable shapes can be walked on; only those which can be swept against.

A small skin is maintained around the character's volume to avoid numerical issues that would occur if it were to touch other shapes. The size of this skin is user-defined. If rendering the character's volume for debug purposes, remember to expand the volume by the size of this skin to get accurate debug visualization.

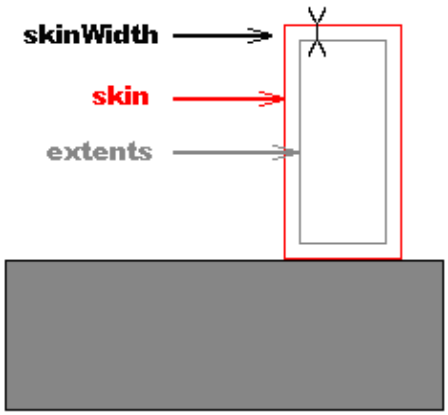
Related parameters for the AABB:

```
NxControllerDesc::position
NxControllerDesc::skinWidth
NxBoxControllerDesc::extents
```

Related parameters for the capsule:

```
NxControllerDesc::position
NxControllerDesc::skinWidth
NxCapsuleControllerDesc::radius
NxCapsuleControllerDesc::height
```

Visual image of AABB:

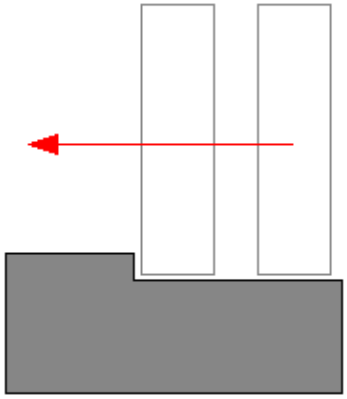


API Reference

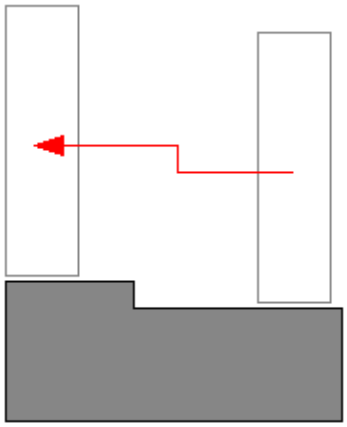
- NxController
- NxControllerDesc
- NxBoxControllerDesc
- NxCapsuleControllerDesc

Character Controller - Auto-Stepping

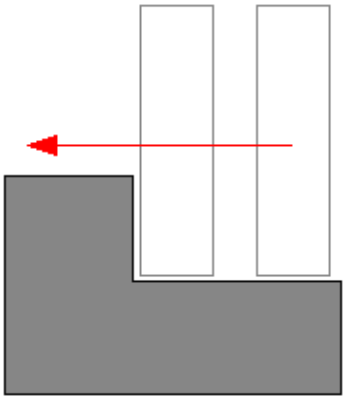
Without auto-stepping, it is easy for a box-controlled character to get stuck against slight elevations of the ground mesh. In the following picture, the small step would stop the character completely. It feels unnatural because in the real world you would just cross this small obstacle without thinking about it.



This is what auto-stepping enables us to do. Without any intervention from the player, the box correctly steps above the minor obstacle as shown below:



However, if the obstacle is too big (i.e., its height is greater than the *stepOffset* parameter), the controller can't climb automatically over, thus the character gets stuck (correctly this time):



Climbing (over this bigger obstacle, for example) may be implemented in the future as an extension of auto-stepping. A related parameter is given below:

```
NxControllerDesc::stepOffset
```

Up Direction

In order to implement the auto-stepping feature, the SDK needs to know about your up vector. The character controller only supports the axis-aligned axes listed below:

- NX_X => (1, 0, 0)
- NX_Y => (0, 1, 0)
- NX_Z => (0, 0, 1)

The SDK is not scheduled to support arbitrary up vectors at this time.

A related parameter is given below:

```
NxControllerDesc::upDirection
```

API Reference

- NxControllerDesc

Character Controller - Walkable Parts

By default, the character can go everywhere. This is not necessarily desirable since, in the real world, there are always limits. Thus it is important to be able to ban things like walking on polygons that have a high slope. The PhysX SDK can do this automatically thanks to a user-defined limit slope. All polygons whose slope is higher than the limit slope will be marked as non-walkable, blocking characters from access.

In the future, it may be possible to tag each triangle of a mesh as either walkable or not walkable.

A related parameter is given below:

```
NxControllerDesc::slopeLimit
```

The limit is expressed as the cosine of the desired limit angle. The slope limit in the example below is 45 degrees:

```
slopeLimit = cosf(NxMath::degToRad(45.0f));
```

If you use slopeLimit = 0.0f, the feature is automatically disabled (i.e., the character can go everywhere).

This feature is not always needed. A common strategy is to disable it and place invisible walls in the level to restrict player's movements.

API Reference

- NxControllerDesc

Character Controller - Volume Update

Sometimes it is useful to change the size of the character's volume at runtime. For example, if your character can crouch, you might want to reduce the height of its bounding volume so that it can move to places unreachable otherwise.

The character library supports runtime volume updates. However, volumes are directly modified without any extra tests, so it might happen that the resulting volume overlaps some nearby geometry. To avoid this, first use the already existing placement API from the SDK to check that the volume of space you want to occupy is actually empty. Only then can the character's volume be updated.

Related function for the AABB:

```
bool NxBoxController::setExtents(const NxVec3& extents) = 0;
```

Related functions for the capsule:

```
bool NxCapsuleController::setRadius(NxF32 radius) = 0;
bool NxCapsuleController::setHeight(NxF32 height) = 0;
```

Related placement API functions:

```
bool NxScene::checkOverlapSphere();
bool NxScene::checkOverlapAABB();
bool NxScene::checkOverlapCapsule();
```

API Reference

- NxCapsuleController
- NxCapsuleControllerDesc

- NxBoxController
- NxBoxControllerDesc
- NxScene

Character Controller - Callbacks

It is possible to define a user callback that will retrieve information about a controller's evolution (i.e., to get called when the character hits a shape or another character).

For both static and dynamic shapes, if a character hits a shape, on impact the following is retrieved:

- Current controller
- Touched shape
- A contact position
- A contact normal
- A feature identifier (for example, a triangle index)
- Extra motion data

These parameters can then be used in your application to do various things like playing sounds, rendering trails, applying forces, and so on. NOTE: Currently, not all parameters are supported - it depends on what shape the character is colliding with, etc. Eventually, all will be supported for all shapes.

If the character hits another character, for example, when a player collides with an NPC, the following information is retrieved:

- A pair of touching controllers

Character Interactions

Needed for game play, action codes from previous callbacks can be returned that lead to different behaviors (e.g., to prevent a character from pushing a particular box, even if the physics says the character is strong enough to push it).

The following are currently supported actions:

- NX_ACTION_NONE => disable physics (the character will not push the object)
- NX_ACTION_PUSH => enable physics (the character will push the object)

It is tempting to let the physics engine push objects naturally, applying forces to usual contact points. However, this can be bad for game play. The bounding volume around characters are artificial (boxes, spheres, etc.) and you don't want the pushing effect to change when you switch from a box controller to a capsule controller. The pushing effect should be dictated by game play. So it is often better to apply artificial forces to objects in the callback. Plus, it is difficult to push a box forward with a capsule. Since you never hit the box exactly in the middle, applied force tends to rotate it - even if all you want is to push it in a straight line.

NOTE: *NX_ACTION_PUSH* is currently not implemented. In the sample, artificial (game play tweaked) forces are applied to objects in the callback, using provided impact data (position, etc.).

API Reference

- NxControllerDesc
- NxUserControllerHitReport

Character Controller - Hidden Kinematic Actors

The *NxCharacter* library creates a hidden kinematic actor for each controlled character. This sometimes causes confusion for a number of reasons, such as the total number of actors in the scene is not as expected, or unknown actors from scene collision queries may be received, etc.

When the move function to move an NxController in invoked, the underlying NxActor is also updated.

The kinematic actors can be retrieved using the following function:

```
NxActor* NxController::getActor() const;
```

However, you should avoid controlling or altering this actor directly, as the behavior is undefined. Purely information-gathering methods are OK.

API Reference

- NxController
- NxActor

Character Controller - Time Stepping

Actors used internally by the NxCharacter library follow the same rules as any common PhysX object. In particular, they are updated using fixed or variable timesteps. This can create trouble because the NxController objects are otherwise typically updated using variable timesteps (most of the time, the elapsed time between two rendering frames). Therefore, the NxController objects are not always perfectly in sync with their kinematic actors when using fixed timesteps.

API Reference

- NxController

PhysX Compartments

In version 2.6, the PhysX SDK introduced the concept of compartments, which allows hardware rigid bodies, fluids and cloth to be created within a single scene. Compartments provide a transparent mechanism of creating interacting rigid bodies, both in software and hardware, as well as cloth and fluids.

A compartment represents a sub-instance of one type of physics simulation, for example hardware fluid, hardware rigid body, or software rigid body. Several compartments may exist inside one scene, (including several of the same type) and interaction between them and the scene proper is taken care of automatically. This allows for transparent manipulation, simulation and interaction, without having to worry about the low-level nuts and bolts.
The SDK manages all links between the compartments. For example, when a material is created, it will be created for all rigid body compartments, or if a raycast is performed, the query will return all intersected hardware and software objects in all compartments.

A newly created scene, hardware or software, contains no compartments. Any object that is added to the scene will be inserted into the scene proper, unless either a compartment is specified in its descriptor, or that type of object cannot exist directly in the scene.
The latter applies to for example fluid in a software scene. In such a case, an appropriate default compartment will be created for the new object if none exists already.

Creation

Creating a compartment in a scene is simple; just create a NxCompartmentDesc descriptor, enter the relevant data into it and call the NxScene::createCompartment method:

```
NxCompartmentDesc compartmentDesc;
compartmentDesc.type = NX_SCT_RIGIDBODY;
compartmentDesc.deviceCode = NX_DC_PPU_0;
compartmentDesc.gridHashCellSize = 1.0f;
compartmentDesc.gridHashTablePower = 6;

NxCompartment *compartment = gScene->createCompartment(compartmentDesc);
```

The deviceCode defaults to NX_DC_CPU, which will simulate the objects on the CPU. You can also use NX_DC_PPU_AUTO_ASSIGN, which will assign new compartments to PPUs in a round-robin fashion, if more than one are available. When no PPU is available NX_DC_PPU_AUTO_ASSIGN will result in a CPU compartment. Alternatively, you can specify the PPU identifier explicitly.
The gridHashCellSize and gridHashTablePower parameters allow you to tweak the performance of the compartment, adjusting the parameters of the hash map that is used to mirror the scene's bodies. Larger cell sizes will mirror more geometry into the compartment; large hash tables will consume more memory but increase performance when many cells are touched.

The rigid bodies of the default software compartment are mirrored into the other compartments, where they function like kinematic actors. There is no communication in the opposite direction, except in the case of 2-way-enabled cloth or fluid.

Each compartment may also have its own settings for continuous collision detection as well as restricted scene mode (see Hardware Scenes). You set these modes using the flags member of the descriptor. By default, the flag NX_CF_INHERIT_SETTINGS is set. That means compartments default to using the modes of the master scene; however, you can clear this flag and manually specify NX_CF_RESTRICTED_SCENE and NX_CF_CONTINUOUS_CD before you call createCompartment.

Simulation

Simulation with compartments is as simple as with a single scene; you need only call simulate() for the master scene, and all its compartments will be simulated automatically.

By default, the master scene and the compartments are simulated in parallel (as far as possible). This utilizes the hardware resources optimally.
However, as synchronization with the master scene must wait until the compartments are done, this will cause a 1 frame lag in the interaction between e.g. cloth and rigid bodies. If this is unacceptable, there is the option to simulate the master scene after the compartments are finished (as was the case prior to release 2.6.2); to do this, simply set the scene flag NX_SF_SEQUENTIAL_PRIMARY.

Furthermore, it is possible to retrieve the results from the individual compartments/master scene, which may further enhance the efficiency of your application. To do this, call fetchResults on the individual compartments, and/or fetchResults with the NxSimulationStatus parameter NX_PRIMARY_FINISHED for the master scene. This will allow read-only access on the individual simulation results before the simulation as a whole is complete.
Note that you still have to call NxScene::fetchResults with NX_ALL_FINISHED before writing to the scene or simulating again.

Caveats

- Shape counting methods like getTotalNbShapes will report entities belonging to all compartments as well as the master scene. This includes mirrored entities, which means the count will sometimes be higher than might be expected at first glance.
- Meshes must not be dynamic (i.e., changed after creation).
- Meshes must be cooked with the hardware flag set even in SW scenes, if they're to be properly mirrored into HW compartments.
- Active Transform Notification for actors in compartments is not supported.
- Objects from different compartments do not interact directly.
- The sweep API is not yet supported by compartments.
- There may be a frame&"PhysicsSDK.chm::/classNxScene.html">NxScene
- NxCompartment
- NxCompartmentDesc

Fluid compartments

The SDK transparently handles cooking of triangle mesh geometry and mirroring of dynamic/static shapes into a hardware fluid scene. It cooks mesh data on the fly for the fluid scene as the particles from the fluid intersect the AABBs of rigid body geometry.

This makes it easy to integrate fluid support into an existing application. Simply call NxScene::createFluid() on your scene and the SDK will take care of the rest.

2-way interaction

PhysX by default allows objects of the software rigid body compartment to affect a fluid compartment, and not vice versa; however, this can be changed.
2-way interaction of fluids and rigid bodies are disabled on two levels by default:

- NxFluid
- NxShape

To enable the interaction, both the corresponding shape flag as well as the fluid flag must be set:

```
fluidDesc.flags |= NxFluidFlag::NX_FF_COLLISION_TWOWAY;

shapeDesc.flags |= NxShapeFlag::NX_SF_FLUID_TWOWAY;
```

The interaction is not necessarily symmetrical; the collision Response Coefficient defines which fraction of the collision impulse actually gets applied to the colliding rigid body:

```
fluidDesc.collisionResponseCoefficient = 0.8f;
```

The default collision response coefficient is 0.2.

Caveats

- Shape and actor properties other than pose and velocity are not updated during runtime, with the exception of collision/trigger/drain flags for dynamic shapes and actors. Flags for static shapes and actors don't get updated.
- The runtime creation and removal of triangle based static shape types is not mirrored correctly. If such a shape is created during simulation, it is not guaranteed that particles in the scene will collide with it. Also, if such a shape is removed during simulation, particles in the scene may still collide with it. This applies to the following shape types:

  ```
  static triangle mesh shapes

  static heightfield shapes

  static convex shapes
  ```

- Performance of the scene manager is dependent on the triangle density of objects within the bounding boxes of fluid. Thus, a heavily tessellated mesh may cause a stall while the mesh data is transformed into a form suitable for upload to the hardware.
- Because of the way the compartment interacts with the scene, fluids cannot interact directly with cloth, and indirectly (through interacting with rigid bodies in the scene proper) only with a delay.
- There's a limit of 8192 mesh triangles interacting with a fluid packet; packets exceeding this will exhibit particles penetrating solid objects. The packet size must thus not be too large compared to the

triangle density of meshes.

API Reference

- NxFluid
- NxFluidDesc
- NxScene
- NxSceneDesc
- NxCompartment
- NxCompartmentDesc

Cloth compartments

The SDK transparently handles cooking of triangle mesh geometry and mirroring of dynamic/static shapes into a cloth compartment. It cooks mesh data on the fly for the cloth as the particles from the cloth intersect the AABBs of rigid body geometry.

This makes it easy to integrate cloth support into an existing application. Simply call NxScene::createCloth() on your scene and the SDK will take care of the rest.

2-way interaction

PhysX by default allows objects of the software rigid body compartment to affect a cloth compartment, and not vice versa; however, this can be changed.
To enable the interaction the NX_CLF_COLLISION_TWOWAY cloth flag must be set:

```
clothDesc.flags |= NxClothFlag::NX_CLF_COLLISION_TWOWAY;
```

The interaction is not necessarily symmetrical; the collision Response Coefficient (a value between 0 and 1) defines which fraction of the collision impulse actually gets applied to the colliding rigid body:

```
clothDesc.collisionResponseCoefficient = 0.8f;
```

The default collision response coefficient is 0.2.

Caveats

- The PPU has a limit of 32 cloth compartments/scenes.
- Convexes up to 120 planes.
- Only cloth particle collision detection (no cloth triangle collision detection).
- Must have less than 1024 attached particles per cloth.
- Limit of less than 1024 colliding shapes.
- Wild movement of the cloth in connection with high bending stiffness can get the cloth into an entangled state. It looks like the cloth locally sticks to itself.
- Squeezing cloth between rigid bodies can cause jittering.
- For small meshes (< 256 vertices) it is more efficient to merge several of them into one cloth than creating a cloth instance for each individual mesh. These meshes do not need to be connected (e.g. multiple leaves of a plant).

API Reference

- NxCloth
- NxClothDesc
- NxScene
- NxSceneDesc
- NxCompartment
- NxCompartmentDesc

Rigid body compartments

PhysX provides a powerful mechanism to create rigid body actors that are simulated on hardware, with a minimum of hassle, by means of compartments.

Usage

In order to utilize rigid body compartments, simply create a hardware or software scene just as usual, then add one or more compartments of type NX_SCT_RIGIDBODY. Finally, insert new actors into the desired compartment by setting the compartment member of the NxActorDesc:

```
NxCompartmentDesc compartmentDesc;
compartmentDesc.type = NX_SCT_RIGIDBODY;
compartmentDesc.deviceCode = NX_DC_PPU_0;

NxCompartment *hwRBCompartment = gScene->createCompartment(compartmentDesc);

NxActorDesc actorDesc;
actorDesc.compartment = hwRBCompartment;
```

Leaving the compartment parameter NULL (the default) will create the actor in the scene proper and not in a compartment.
If the compartment can not support further actors, or it doesn't support the particular actor that you are trying to create, creation will fail. It does not automatically fall back to software mode.

Caveats

- Joint creation fails between actors in different compartments, or between an actor in the primary scene and one in a compartment.
- Effectors between hardware actors not supported.
- Actor or shape pair flags for hardware objects not supported.
- Hardware actors in compartments do not act on actors in the scene proper (no 2 way interaction).
- Only the dynamic actors of the compartment cause objects to be mirrored (static/kinematic actors do not).
- Just as in a normal scene there will be no interaction between static rigid bodies.
- Interaction is one-way, hence objects in the compartment will never affect objects in the master scene, in any way.
- Wheel shapes from the scene proper are not mirrored into the compartments.
- Shape filtering does not work for shapes mirrored into compartments.

API Reference

- NxActor
- NxActorDesc
- NxShape
- NxShapeDesc
- NxScene
- NxSceneDesc
- NxCompartment
- NxCompartmentDesc

Compartment interactions

Interactions

Below you find a comprehensive list of the different ways in which the PhysX SDK can be used, both with and without the use of PhysX hardware. The list has been divided into two main sections; PhysX hardware present/non-present, and each section into two more parts; with and without the use of compartments.

The tables specify the following properties of the objects that are created:

- Resides in - Which scene or compartment the object will be simulated in. Where an object "lives" affects the ways in which it is affected by and affects other objects.
- Runs on - Specifies whether the object is simulated in software or hardware.
- Affects - Other types of objects that are affected by the object.
- Affected by - What other objects affect the object?

The main rule for how objects interact and are affected by each other is that objects in the main scene affect and objects in compartments are affected. This behavior is possible to override for fluids, cloths and softbodies by the use of the NX_*_COLLISION_TWOWAY flags. When "rigid body" is specified, it also includes trigger testing, not only interactions with physical entities.

Force fields are always created in the master scene and will affect all other objects, without regard to where they are simulated. If some objects should be excepted from force field affection that needs to be done with filters. Note that if the force field should not affect a whole feature, e.g. cloth then you can set the scale parameter for cloth to 0 to stop it from interacting with the cloth.

No PhysX hardware available (software simulation)

When there is no PhysX hardware available, you need to specify that the simulation shall be done in software, which you do by specifying **NX_SIMULATION_SW** at scene creation.

With no compartment specified:

Item	Resides in	Runs on	Affects
Rigid bodies	Master scene	Software	Everything
Fluids	Default fluid compartment[2]	Software	**NX_FF_COLLISION_TWOWAY set:** Rigid bodies in master scene **NX_CLF_FLUID_COLLISION[1] set:** Cloth/SoftBody
Cloth/SoftBody	Master scene	Software	**NX_CLF_COLLISION_TWOWAY[1] se** Rigid bodies in master scene **NX_CLF_FLUID_COLLISION[1] set:** Fluids

[1] For softbodies, the corresponding flag names are NX_SBF_<flag>.
[2] Fluids without a specified compartment are created in the first fluid compartment that was created. If no compartment has been created before the first fluid, then a default fluid compartment is created to hold the fluid. The same applies to cloth and soft body as well.

With compartment specified in object descriptor:

Item	Resides in	Runs on	Affects	Affected by
Rigid bodies	Specified compartment	Software	Only rigid bodies in the same compartment	Other rigid bodies in the master scene[3] Not affected by Cloth/So collisions flags are set.
Fluids	Specified compartment	Software	**NX_FF_COLLISION_TWOWAY set:** Rigid bodies in master scene **NX_CLF_FLUID_COLLISION[1] set:** Cloth/SoftBody	Rigid bodies in master sc **NX_CLF_FLUID_COL** Cloth/SoftBody
Cloth/SoftBody	Specified compartment	Software	**NX_FF_COLLISION_TWOWAY set:** Rigid bodies in master scene **NX_CLF_FLUID_COLLISION[1] set:** Fluid	Rigid bodies in master sc **NX_CLF_FLUID_COL** Fluid

[3] Note that triggers in compartments do not react to bodies in the master scene.

PhysX hardware available (hardware simulation)

When there is PhysX hardware available for the application to use (can be checked through the PhysicsSDK object), the user can choose to use hardware simulation by specifying **NX_SIMULATION_HW** at scene creation.
Note that it is fully possible to use software simulation even when there is PhysX hardware present, and that you might choose to do so for a subset of your physics assets.

With no compartment specified:

Item	Resides in	Runs on	Affects	Affect
Rigid bodies	Master scene	As specified by user in scene type (SW/HW)	Everything	Other r **NX_F** Fluids **NX_C** **NX_S** Cloth (
Fluids	Default fluid compartment[3]	Software/Hardware	**NX_FF_COLLISION_TWOWAY set:**	Rigid b

			Rigid bodies in master scene **NX_CLF_FLUID_COLLISION[1] set:** Cloth/SoftBody
Cloth/SoftBody	Master scene (SW) or default cloth compartment[3] (HW)	Software/Hardware	**NX_CLF_COLLISION_TWOWAY[1] set:** Rigid bodies in master scene **NX_CLF_FLUID_COLLISION[1] set:** Fluids

With compartment specified in object descriptor:

Item	Resides in	Runs on	Affects	Affected by
Rigid bodies	Specified compartment	Software/Hardware[4]	Only rigid bodies in the same compartment	Other rigid in the maste
Fluids	Specified compartment	Software/Hardware[4]	**NX_FF_COLLISION_TWOWAY set:** Rigid bodies in master scene **NX_CLF_FLUID_COLLISION[1] set:** Cloth/SoftBody	Rigid bodie **NX_CLF_** Cloth/SoftB
Cloth	Specified compartment	Hardware[5]	**NX_CLF_COLLISION_TWOWAY[1] set:** Rigid bodies in master scene **NX_CLF_FLUID_COLLISION[1] set:** Fluids	Rigid bodie **NX_CLF_** Fluids

[4] Depends on the compartment setting.
[5] SW Cloth compartments are not allowed (but cloth can be run in SW if a compartment is not used, as noted in the SW compartment section above).

API Reference

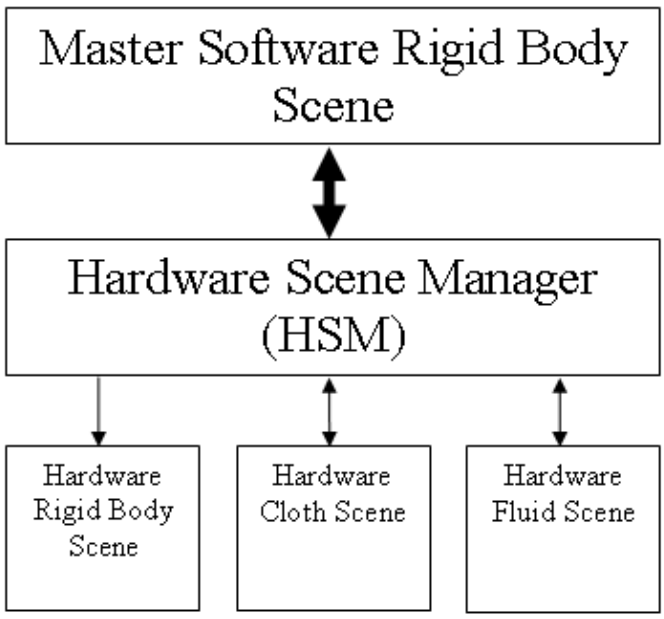
- NxScene
- NxSceneDesc
- NxCompartment
- NxCompartmentDesc
- NxCloth
- NxClothDesc
- NxFluid
- NxFluidDesc

PhysX Hardware Scenes

See the Dynamics section for a general description of scenes.

Creating a Software Master Scene

To take advantage of the PhysX processor the user needs to consider how they will be using the SDK. There are a couple of options with respect to hardware support. One option is to create a master software scene and have the *Hardware Scene Manager* (HSM) handle any hardware accelerated objects by way of compartments. Hardware objects can include rigid bodies cloth or fluids.

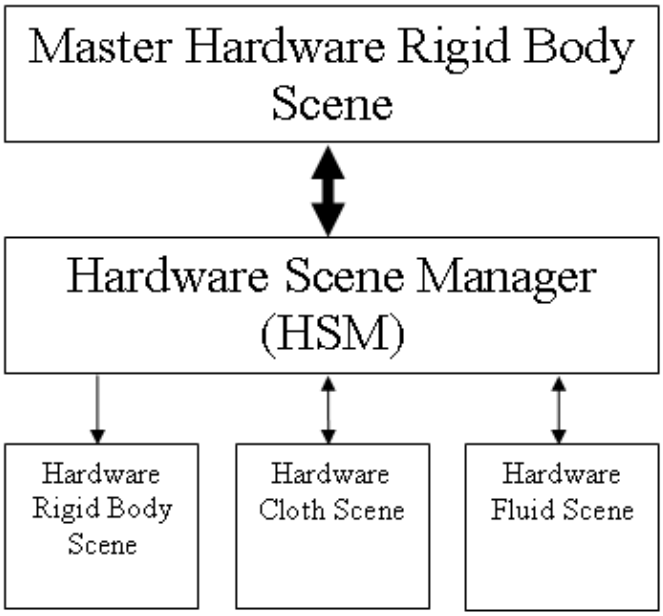


To create a scene of this type, use:

```
sceneDesc.simType                = NX_SIMULATION_SW;
```

Creating a Hardware Master Scene

Alternatively it is possible to create a hardware master, with compartments also running in hardware. This means that by default the simulation runs in hardware. But there is still the option of running cloth and fluids along with it in the same way as if the master scene is software. The rigid body scene may also be useful if the user wishes to run the scene in restricted mode (see below), since the rigid body compartment provides a way to extend the supported object counts.



To create a scene of this type, use:

```
sceneDesc.simType                = NX_SIMULATION_HW;
```

NOTE: It is no longer possible to create a pure hardware fluid scene. A fluid will always reside in a compartment.

See Hardware Support for details concerning limitations of hardware scenes.

See Hardware Detection for details concerning detection of hardware and disabling all hardware support.

Example - Hardware Master Scene

```
    gPhysicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION, 0, &gErrorStream);
    if(!gPhysicsSDK)
        return;


    // Create a scene
    NxSceneDesc sceneDesc;

    //...
```

```
"PhysicsSDK.chm::/classNxScene.html">NxScene
```

NxPhysicsSDK

NxSceneDesc

PhysX Hardware Support

In this release, the PhysX hardware supports a subset of the functionality supported by the software SDK. Exact details are provided in the API reference section. For a quick overview, see below.

Hardware Rigid Body Scenes

Supported

- Simple rigid body actors
- Compound shapes (multiple shapes attached to an actor)
- Convex, box, sphere and capsule (dynamic shapes)
- Convex, box, sphere, capsule, plane and non-convex triangle meshes (static shapes)
- Forces and impulses applied to dynamic shapes
- Joints, all types
- Continuous collision detection
- Start touch and end touch contact reporting
- A subset of visualizations, see the API reference for details (NxParameter)
- Contact streams
- Sub stepping
- Effectors

Not Supported

- D6 joints only support angular drives. They do not support linear drives or SLERP drives

Caveats

- Only one application can use the PhysX hardware at a time. The first application to access the hardware obtains exclusive access to it; however, it is possible to prevent the SDK from using the hardware even if it is present (see Hardware Detection for details).
- Each hardware scene can hold up to 64k (D6) joints and 4080 or 64k rigid bodies (depending upon if NX_SF_RESTRICTED_SCENE is set).
- There is a further limitation on the number of active rigid bodies. While a scene can contain 64k rigid bodies(if it is not a restricted scene), only 4k bodies, 4k D6 joints, 4k constraints and 8k shapes can be active.
- The hardware supports a maximum of 32 faces and 32 vertices for convex mesh objects. Larger objects will fall back to software.
- In addition, using large meshes with the PhysX hardware requires the application to perform mesh paging operations.
- Since the hardware uses a different solver, the behavior of joints and contacts differs from that of the software SDK. In particular, scenes using D6 joints may be stable in software but not hardware and vice versa depending on the configuration. Keep the following in mind when using the hardware:
    - ♦ As in software, over constrained/opposing joints lead to poor behavior and instability.
    - ♦ Hardware joints behave better if they are initially configured in a balanced/stable system. For example, in a fixed D6 joint system, such as a breakable tower, it is better to have all D6 joints initially in an un-biased pose; otherwise, the SW or HW has to perform calculations to balance the system.
    - ♦ In some cases, turning off pose projection may improve the stability of D6 joint systems.
    - ♦ There is a limit of 64 contexts; a scene will take 1 or 2 contexts depending on flags upon creation. A software scene (NX_SIMULATION_SW) will always take 1 context, a hardware scene will take 1 context if NX_SF_RESTRICTED_SCENE is set, otherwise 2 contexts.

PhysX Hardware Detection

An application can detect if PhysX hardware is present and available by querying for the version of the hardware using a method of NxPhysicsSDK. In the future, this method will also allow the application to differentiate between hardware versions.

Currently, getHWVersion() returns two values:

- *NX_HW_VERSION_NONE* - No usable hardware support is present.
- *NX_HW_VERSION_ATHENA_1_0* - PhysX hardware is available for use.

Example

```
bool IsPhysXHardwarePresent()
{
    return gPhysicsSDK->getHWVersion() != NX_HW_VERSION_NONE;
}
```

Disabling Hardware Use

In some circumstances, it is best to instruct the SDK not to use hardware support even if it is present (e.g., if a developer wishes to use a server and client application on the same machine). Since the PhysX hardware can only be used by one application at a time, the server application should use software only so the client application can make full use of the hardware.

This is achieved using the NX_SDKF_NO_HARDWARE which is passed during SDK creation in the flag's member of NxPhysicsSDKDesc.

Example

```
NxPhysicsSDKDesc sdkDesc;

sdkDesc.flags |= NX_SDKF_NO_HARDWARE;

gPhysicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION, gMyAllocator, gMyOutputStream, sdkDe
```


PhysX Hardware Crash Detection

In previous releases, when the PhysX firmware deadlocked or crashed, the application hung. Now fetchResults() takes a pointer argument which, if non-zero, points to a location where an integer return code will be stored. If the return code is also non-zero, an abnormal condition was detected on the hardware. The hardware remains in a stalled state until the Physics SDK object is destroyed and re-created.

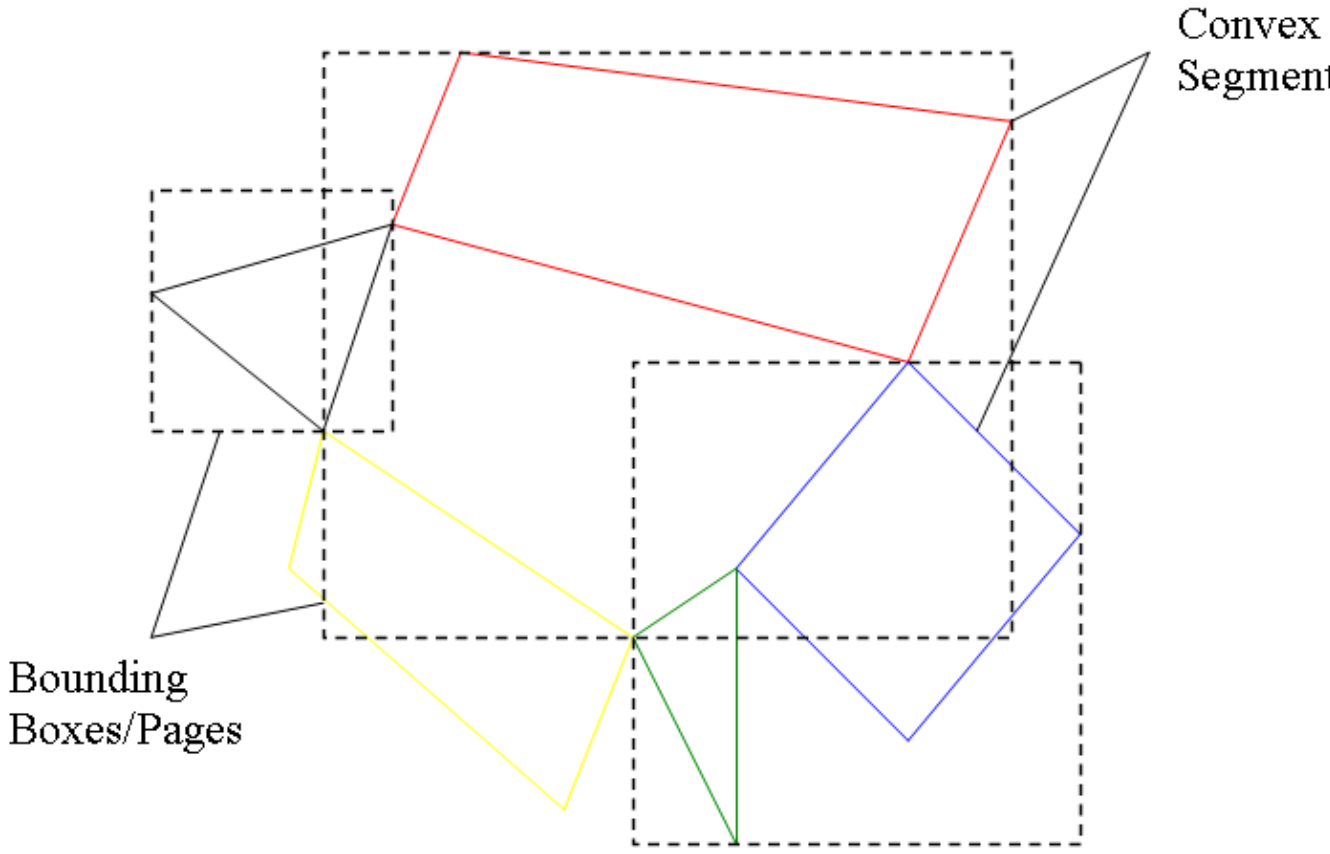
If the application ignores this return code and an abnormal termination condition is detected, instead of hanging, the application will continue; however, physics objects running on hardware will not move.

The return code currently provides no information about how or why the abnormal termination condition occurred.

Mesh Paging

To allow support for large static meshes when using the hardware, it is necessary to explicitly control mesh paging to the hardware. When a mesh is cooked the user must specify an additional flag to generate a pageable format which can be used with the PPU.

The flag NX_MF_HARDWARE_MESH is applied to the flags member of NxSimpleTriangleMesh/NxTriangleMeshDesc before cooking. Cooking splits the mesh into convex segments, which are then grouped into pages for the hardware.
Note that a mesh must be hardware cooked in order to perform collision detection with any hardware objects. Thus even a mesh in a master scene on software must set NX_MF_HARDWARE_MESH to be able to collide with objects in a hardware rigid body compartment (see Compartments).



When the mesh is loaded into a hardware scene, the user can retrieve a bounding box for each mesh page, and use this information to upload appropriate pages to the card to support their simulation.

```
NxU32 NxTriangleMesh::getPageCount() const;
NxBounds3 NxTriangleMesh::getPageBBox(NxU32 pageIndex) const;
```

To map pages for a particular shape:

```
bool NxTriangleMeshShape::mapPageInstance(NxU32 pageIndex);
void NxTriangleMeshShape::unmapPageInstance(NxU32 pageIndex);
bool NxTriangleMeshShape::isPageInstanceMapped(NxU32 pageIndex);
```

mapPageInstance() will return true if the page has been successfully uploaded to the PPU; otherwise, it will return false. In the case where a page cannot be uploaded to the hardware, the user should unmap a number of pages using unmapPageInstance() to make room for the new pages.

Automatic mesh paging

It is possible to have the SDK page meshes automatically, as needed, whenever broadphase overlaps indicate it. Use the NxTriangleMeshShapeDesc::meshPagingMode member to enable automatic paging. The member can take the following values:-

- NX_MESH_PAGING_MANUAL - Mesh paging is left to the user.
- NX_MESH_PAGING_FALLBACK - Automatic SW fallback when needed pages are not mapped.
- NX_MESH_PAGING_AUTO - Automatic mesh paging. Pages are mapped to hardware when broadphase overlaps are detected

Note that automatic mesh paging works in parallel to the normal manual scheme; this means that a mesh page is fetched when either the SDK or the user (through mapPageInstance) wishes it - and it is unmapped only when *both* the SDK and the user (through unmapPageInstance) wish it. The user can thus force a page mapping to occur before collision is imminent, in order to e.g. prevent performance spikes.

Alternatively, you can use the NX_MESH_PAGING_FALLBACK flag in order to have the SDK fall back to software mode for meshes that have not been paged into the hardware.

Care should be taken when using automatic mesh paging as the paging process may stall the simulation for a significant amount of time causing large spikes in the simulation time. ie User intervention to map appropriate pages ahead of time may be needed.

Also note that mesh paging may not be required for cases where the simulation falls back to software. For example heightfield meshes are not simulated in hardware and will always execute in software (version 2.5).

API Reference

- NxTriangleMesh
- NxTriangleMeshShape
- NxBounds3

Threading Interface

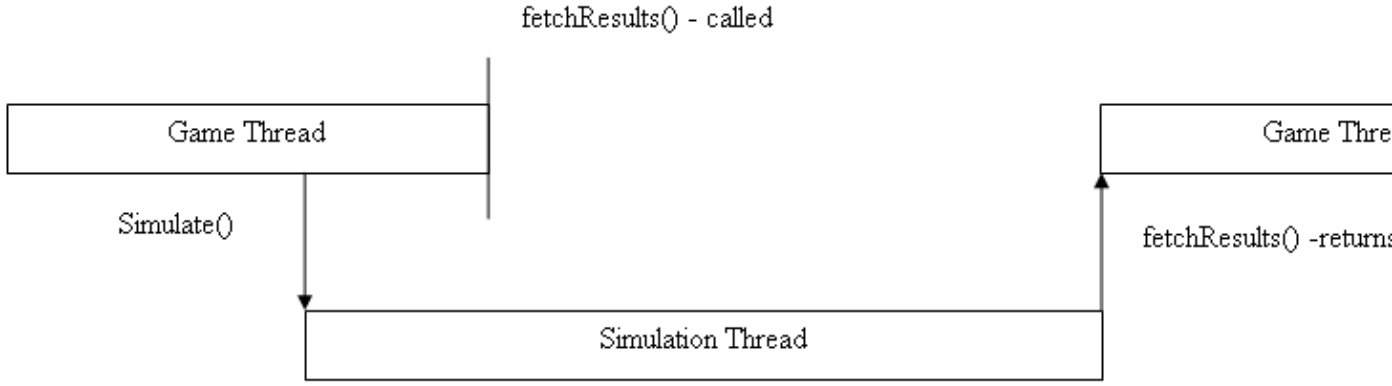
The PhysX SDK 2.4 and above gives the user more control over threading within the SDK. There are two main orthogonal features:

- The user can specify which thread the main simulation runs on.
- Fine grained threading of the simulation.
- Under some circumstances, the SDK can provide background work to the application (e.g., mesh cooking when the Hardware Scene Manager is in use).

Main Simulation Thread Control

The thread upon which the main simulation runs can be controlled with the NX_SF_SIMULATE_SEPARATE_THREAD flag. The flag, which is raised by default, is applied to the flag's member of the scene descriptor.

When the NX_SF_SIMULATE_SEPARATE_THREAD is raised, the SDK creates an internal thread upon which the simulation is performed (Simulation Thread). This allows the simulation to execute in parallel with the game code during the time between the simulate() call and the fetchResults() call.

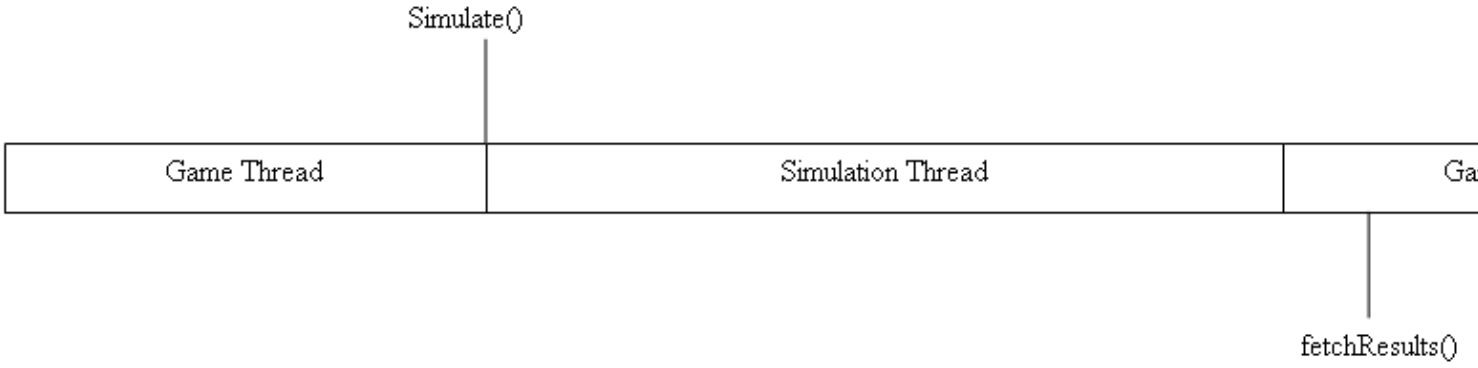


The alternative, when the NX_SF_SIMULATE_SEPARATE_THREAD is not raised, is to perform all the work associated with the simulation during the call to simulate(). This allows the application control over the thread upon which the simulation executes by choosing an appropriate thread to call simulate(). fetchResults() must still be called, but it will return immediately with the results and not be blocked if simulate() has finished or block as appropriate if it is still executing on another thread.

The ability to control which thread the user executes the simulation on is particularly important for platforms which statically allocate software threads to hardware threads(for example the XBox360). In these situations the user should take control of which thread the simulation executes on (by default the SDK runs the simulation on the hardware thread which creates the scene).

1. Create a software thread to run the simulation on.
2. Associate that thread with an appropriate hardware thread.
3. Create the SDK with NX_SF_SIMULATE_SEPARATE_THREAD **not** set.
4. Then during simulation:
    1. Update SDK state.
    2. Set an event to trigger the thread created above to call simulate().
    3. Do additional processing.
    4. Call fetchResults() which will block until simulate() finishes.

NOTE: Ensure that the SDK is called from a single thread at a time (i.e., provide synchronization which ensures all state modification is complete on another thread before the thread which calls simulate enters the SDK).



Example

```
//Create scene with simulation execute in the call to simulate()
NxSceneDesc mySceneDesc;

//Fill in scene desc...

mySceneDesc.flags         &= ~NX_SF_SIMULATE_SEPARATE_THREAD;

NxScene *newScene=gPhysicsSDK->createScene(sceneDesc);

/*************************************************************************/

//Create scene with simulation execute on another thread
NxSceneDesc mySceneDesc;

//Fill in scene desc...

mySceneDesc.flags         |= NX_SF_SIMULATE_SEPARATE_THREAD;

NxScene *newScene=gPhysicsSDK->createScene(sceneDesc);
```

Fine Grained Simulation Threading

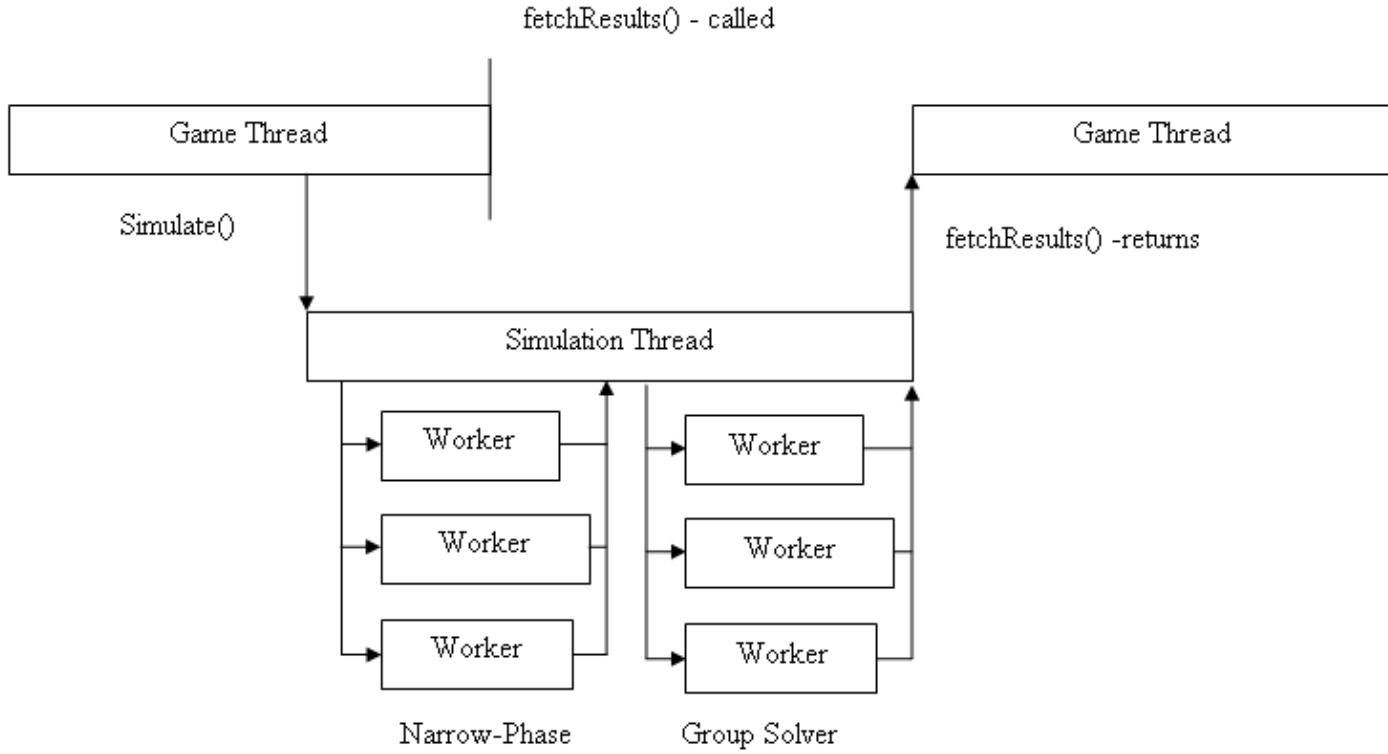
The SDK provides an option to split the simulation processing over a number of worker threads (on some platforms). For example, all the shape pairs which need contact point generation can be divvied up by the SDK and have multiple threads generating contact points at the same time.

By default, fine grained division of the simulation is not enabled. To enable fine grained threading, the user must specify the NX_SF_ENABLE_MULTITHREAD flag in NxSceneDesc.

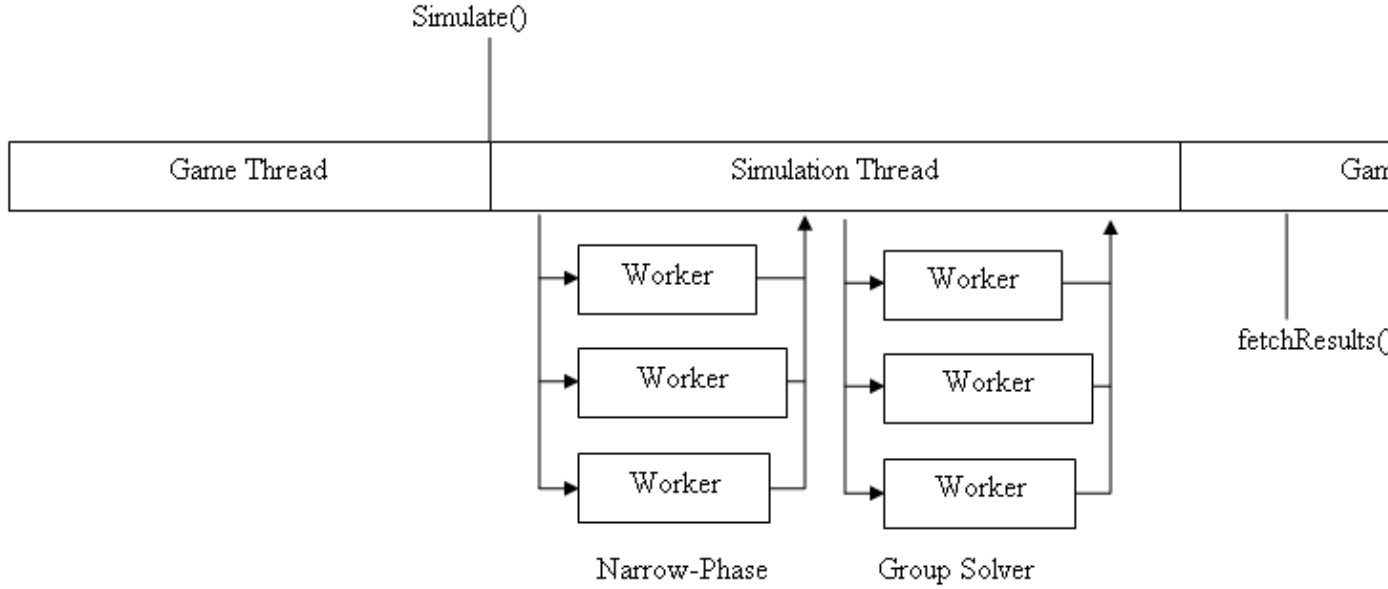
When performing each simulation using a number of threads, there is a main simulation thread which controls the division of work and performs tasks which must be executed in serial. This leads to two additional threading configurations when combined with the NX_SF_SIMULATE_SEPARATE_THREAD flag.

NX_SF_SIMULATE_SEPARATE_THREAD is specified and fine grained threading is enabled displayed in

the diagram below:



When NX_SF_SIMULATE_SEPARATE_THREAD is not specified, it is displayed as follows:



The SDK provides the following two distinct ways to control how work is scheduled between worker threads:

- SDK Managed Work Queue
- User Managed Work Queue

These methods are mutually exclusive; either the SDK maintains an internal work queue and executes work items with internal threads plus user polling, or the user takes over complete control of the work item scheduling.

SDK Managed Work Queue

To enable SDK managed threading, the user must specify the NX_SF_ENABLE_MULTITHREAD flag and not supply a custom scheduler. In addition, the SDK can be instructed to create threads internally to process work items by setting the internalThreadCount member of NxSceneDesc to the number of threads which should be managed by the SDK.

The SDK supports user control over which processor an internal thread is allocated to on systems which statically allocate threads to processors, such as the X Box 360. This is accomplished through the NxSceneDesc member threadMask.

When the SDK is allocating internal threads to logical processors, it scans from the least significant bit of threadMask, allocating the first internal thread to the processors corresponding to the first set bit, to the processors corresponding to subsequent bits. For example, 0xffffffff would allocate internal threads to every processor starting at 0 and continuing to 31. If more threads are specified than set bits, the SDK wraps around to the beginning and allocates a second thread to the set bits. Setting the threadMask to 0 causes the SDK to pick an appropriate setting.

Example

```
NxSceneDesc mySceneDesc;

//Instruct the SDK to use its multi threaded core.
mySceneDesc.flags                    |= NX_SF_ENABLE_MULTITHREAD;

//Instruct the SDK to allocate internal threads to each logical processor, skipping the first.
mySceneDesc.threadMask=0xfffffffe;

//The SDK creates two additional worker threads.
mySceneDesc.internalThreadCount   = 2;

NxScene *newScene=gPhysicsSDK->createScene(mySceneDesc);
```

Polling for Work

To execute work items from your own threads, call the pollForWork() member of NxScene as shown below:

```
NxThreadPollResult pollForWork(NxThreadWait wait);
void resetPollForWork();
```

When calling pollForWork(), specify how long to wait for a work item to become available. The function can return immediately if there are no work items to execute, or it can block until a work item becomes available:

```
enum NxThreadWait
    {
    /* The poll function will return immediately if there is no work available.*/
    NX_WAIT_NONE,

    /* The poll function will wait until the end of the simulation tick for work.*/
    NX_WAIT_SIMULATION_END,

    /* The poll function will wait until the shutdownWorkerThreads() member of NxScene is called or
    a work item has been processed. */

    NX_WAIT_SHUTDOWN
    };
```

When the user instructs pollForWork() to wait using NX_WAIT_SIMULATION_END, it blocks until the simulation is complete or it is able to execute a single work item. When the simulation step completes, subsequent calls to pollForWork() will return immediately (even if it is instructed to wait) until resetPollForWork() is called. This avoids a race condition when the scene completes a step. Without resetPollForWork(), the simulation could start a new step before threads which have called pollForWork() have the chance to return.

Alternatively, if NX_WAIT_SHUTDOWN is specified, threads will wait past the end of a simulation step until a work item is executed or NxScene::shutdownWorkerThreads() is called. shutdownWorkerThreads() must be called before the scene is destroyed so that all user threads can exit the SDK safely (i.e., call shutdownWorkerThreads and block until all user created threads have left the SDK).

pollForWork() returns a status code describing the reason for its return as shown below:

```
enum NxThreadPollResult
    {
    /* There is no work to execute at the time the function was called. */
    NX_THREAD_NOWORK,

    /* There may be more work waiting for execution. */
    NX_THREAD_MOREWORK,

    /* The function returned because the simulation tick finished. */
    NX_THREAD_SIMULATION_END,

    /* The function returned because the user call shutdownWorkerThreads().  */
    NX_THREAD_SHUTDOWN,
    };
```

The status returned by pollForWork() should be considered more of a hint than a guarantee that the scene is in a specific state, since the state could change at any point after pollForWork() releases its internal lock. For example, another thread could execute all items remaining in the work queue or another thread could call resetPollForWork().

Example

```
threadFunctions()
{

    while(!quit)
    {
        //Wait for simulation to begin...
        do
            {

            pollResult=scene->pollForWork(NX_WAIT_SIMULATION_END);

            }while((pollResult==NX_THREAD_MOREWORK)||(pollResult==NX_THREAD_NOWORK));
    }
}

/************************************************/

//Main thread


//Simulation has ended...
```

```
//Wait for all threads to block...

scene->resetPollForWork();

//Signal threads that simulation has started

simulate()
```

User Managed Work Queue

For complete control of the way work items are executed, the user must implement their own scheduler interface to provide to the SDK:

```
class UserScheduler : public  NxUserScheduler
    {
    public:

    virtual void addTask(NxTask *task);
    virtual void addBackgroundTask(NxTask *task);
    virtual void waitTasksComplete() ;
    };
```

Then when creating the scene, provide the scheduler interface using the customScheduler member of NxSceneDesc. NOTE: When specifying a user scheduler, the internalThreadCount and backgroundThreadCount members should be set to zero and the pollForWork()/resetPollForWork() functions should not be called.

The SDK will call NxScheduler::addTask() to queue up a work item to be executed by the application. At points where the SDK must synchronize, it will call waitTasksComplete(). On a call to waitTasksComplete() the application will block until all tasks which were submitted using addTask() have been completed (i.e., the application has called the execute() method of each task and then returned).

A typical implementation of waitTasksComplete() will reuse the calling thread (simulation thread) to execute work items. When the queue is drained it will block until all in progress work items are also complete.

waitTasksComplete() will always be called in the context of the main simulation thread. However, the application should not assume that addTask() will only be called in that context, as, in future versions, alternative threads may add work to the scheduler.

addBackgroundTask() allows the SDK to submit tasks to the user which are not time critical. For example, when using the Hardware Scene Manager the SDK may submit a task to cook mesh data into a form which is suitable for fluid simulation on the hardware. Tasks which are submitted with addBackgroundTask() need not be completed when waitTasksComplete() returns.

To execute work items, the SDK provides the simple NxTask interface below:

```
class NxTask
    {
    public:

    virtual void execute() = 0;
    };
```

Example

```
class CustomScheduler : public NxUserScheduler
    {
    CustomScheduler(){}
    ~CustomScheduler(){}

    virtual void addTask(NxTask *task)
        {
        //Add task to work queue.
        }

  virtual void addBackgroundTask(NxTask *task)
        {
        //Add task to background work queue.
        }

    virtual void waitTasksComplete()
        {
        //Wait for all tasks in the work queue to complete.
        }
    };

CustomScheduler gCustomScheduler;

// .....

NxSceneDesc mySceneDesc;

//Instruct the SDK to use its multi-threaded core.
mySceneDesc.flags                    |= NX_SF_ENABLE_MULTITHREAD;
mySceneDesc.customScheduler          = &gCustomScheduler;

NxScene *newScene=gPhysicsSDK->createScene(mySceneDesc);
```

Background Tasks

In addition to regular simulation tasks, the SDK can also submit background tasks to the user work queue or the SDK managed work queue. These tasks are not time critical (i.e., do not need to be completed within a specific simulation step). Processing of these tasks is handled through the addBackgroundTask() member of the user supplied scheduler or through the pollForBackgroundWork() member of NxScene.

The user can specify SDK managed threads to execute background work using the NxSceneDesc::backgroundThreadCount and backgroundThreadMask members. These operate in a similar way to NxSceneDesc::internalThreadCount; however, threads are created for the sole purpose of executing background tasks.

When using pollForBackgroundWork(), the NX_WAIT_SIMULATION_END does not make sense since background tasks are not associated with a particular simulation step.

Performance

Performance for fine grained threading (i.e., splitting the simulation step into multiple work items) is highly dependant on the configuration of the scene. In some cases, the threaded version may run more slowly than the non-threaded version because there are insufficient opportunities for parallel execution and the synchronization imposes a overhead.

However, in scenes which consist of many shape pairs (preferably with non-trivial contact routines, e.g., convex vs. mesh) and many equally sized islands, the internal threading is able to achieve speed-ups to 40%.

Memory Usage

- Memory utilization, when using NX_SIMULATE_SEPARATE_THREAD, should not change (though there may be a small memory reduction when NX_SIMULATE_SEPARATE_THREAD is not specified due to the lack of an additional thread).
- Memory utilization, when running each simulation step in parallel, is non-trivial.
- Memory utilization for the near phase computation requires memory proportional to the number of vertices in the largest triangle mesh for each additional thread.
- Memory utilization for the joint/contact solver requires memory proportional to the size of the largest body group per additional thread.
- Memory is required to describe each work unit. This should not be as significant as the other overheads, but it is dependant on the number of threads and the rate at which they process work units. Because of the additional memory cost associated with entering the SDK with each additional thread, the user should try and make sure as few separate threads as possible enter the SDK. If a thread executes a work item on the first frame and never executes another work item, then that thread will still incur a memory overhead until the scene is destroyed (future versions of the SDK may support deleting the additional memory associated with a thread).
- The SDK uses dynamic stack allocations in certain places in order to improve performance. In certain rare situations (e.g. large numbers of actors placed in the same position at once) these allocations may be too big to fit on the stack, and should fall back to heap allocation. There is an automatic threshold that decides when to do this, equal to half the smallest SDK-allocated thread stack size (set through NxSceneDesc::simThreadStackSize and ::workerThreadStackSize) for one allocation.
  You may override this using the NxFoundationSDK::setAllocaThreshold() method. Take care; too small a threshold will impact performance, too large may precipitate stack overflows.

Caveats

- When running a simulation using fine grained threading, the simulation will be non-deterministic as a result of the dynamic allocation of work units between the threads. At each step, the work units can be allocated in a different order with different results due to the order of floating point operations changing.
- The SDK requires two Thread Local Storage (TLS) slots per scene, with one additional TLS slot required when the profiler is enabled. The number of TLS slots available is platform specific, but generally in the region of 64.

Samples

Sample Threading

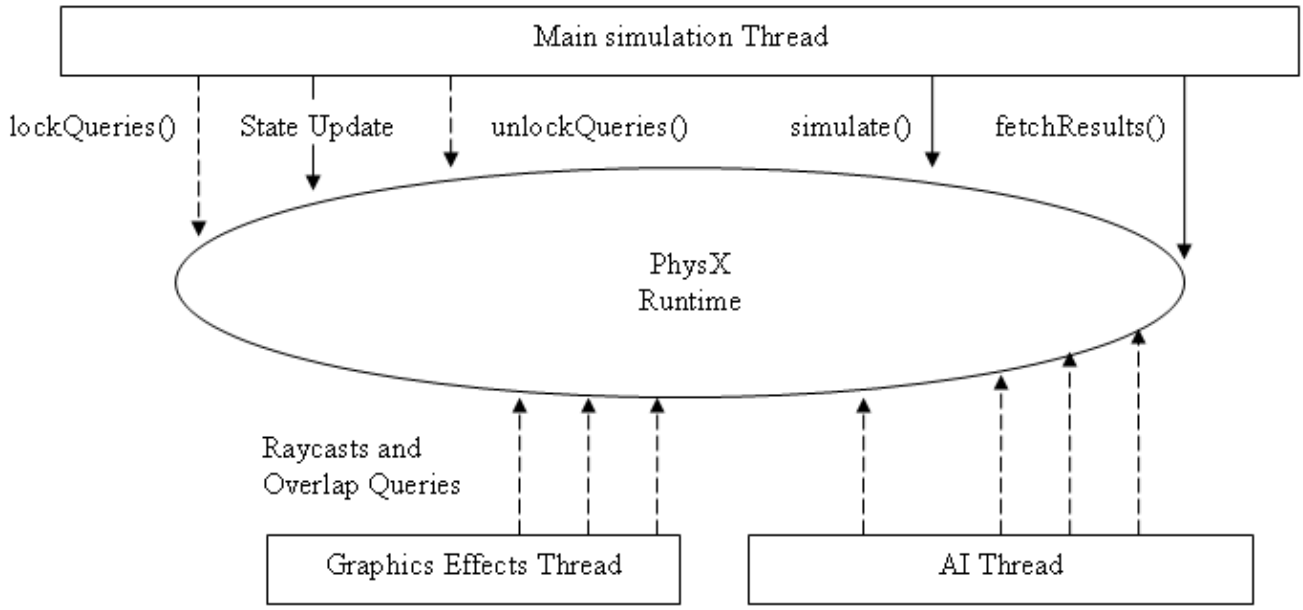
API Reference

- NxUserScheduler
- NxTask
- NxSceneDesc
- NxScene

Thread Safe Raycasting, Sweep and Overlap Queries

In general, it is not safe to use the NVIDIA PhysX API for more than one thread. However, there are a few exceptions, in particular the APIs designed to support simulation across multiple threads (see the Threading Interface page for details) and a subset of the raycasting/sweep/overlap queries. It is also safe to retrieve the user pointer and names associated with shapes as long as they are read-only.

The motivation for making these tests thread safe is that they are often used for AI and graphics effects, which may be performed on separate threads. See Raycasting, Sweep API, and Overlap Queries for more details concerning these functions.



The following queries are thread safe:

- NxScene::raycastAnyBounds()
- NxScene::raycastAnyShape()
- NxScene::raycastAllBounds()
- NxScene::raycastAllShapes()
- NxScene::raycastClosestBounds()
- NxScene::raycastClosestShape()

- NxScene::linearOBBSweep()
- NxScene::linearCapsuleSweep()
- NxActor::linearSweep()

- NxScene::overlapSphereShapes()
- NxScene::overlapAABBShapes()
- NxScene::overlapOBBShapes()
- NxScene::overlapCapsuleShapes()

- NxScene::cullShapes()
- NxScene::checkOverlapSphere()
- NxScene::checkOverlapAABB()
- NxScene::checkOverlapOBB()
- NxScene::checkOverlapCapsule()

Queries which are NOT thread safe are listed below:

- NxShape::raycast()
- NxShape::checkOverlapSphere()
- NxShape::checkOverlapOBB()
- NxShape::checkOverlapAABB()

- NxTriangleMeshShape::overlapAABBTriangles()
- NxTriangleMeshShape::getTriangle()

NOTE: Be careful when changing states which will affect the results of a raycast or overlap query. For example, if filtering constants are modified half way through a query, the results returned will be inconsistent. To facilitate safe modification of state, NxScene exposes a pair of functions to prevent raycasts from running while state is being modified:

- NxScene::lockQueries()
- NxScene::unlockQueries()

By default, the SDK acquires a mutex/critical section with the lockQueries() method, which is released when unlockQueries() is called. The number of calls to lockQueries() must match a corresponding number of calls to unlockQueries().

Example

```
gScene->lockQueries();

gMyShapeA->setGroupsMask(newMaskA);
gMyShapeA->setFlag(NX_SF_DISABLE_RAYCASTING, true);
gMyShapeB->setGroupsMask(newMaskB);

gScene->unlockQueries();
```

The following list contains state updates which must be protected by lockQueries():

- Shape groups
- Shape flags
- Shape groups mask
- Shape geometry changes
- Shape pose changes

NOTE: It is not recommended to change the state while running a simulation, as changes may lead to unexpected physical behavior. For example, disabling collisions between a box and plane when the box is resting on the plane will not cause the box to fall through it.

Overlap queries with individual shapes were not made thread safe as the overhead associated with doing so can become significant.

Performance

There is a memory overhead per additional thread which enters the SDK to perform a raycast, since each thread must allocate working buffers. Some of this overhead can be shared if the thread also takes part in multi threaded simulation (see the Threading Interface page).

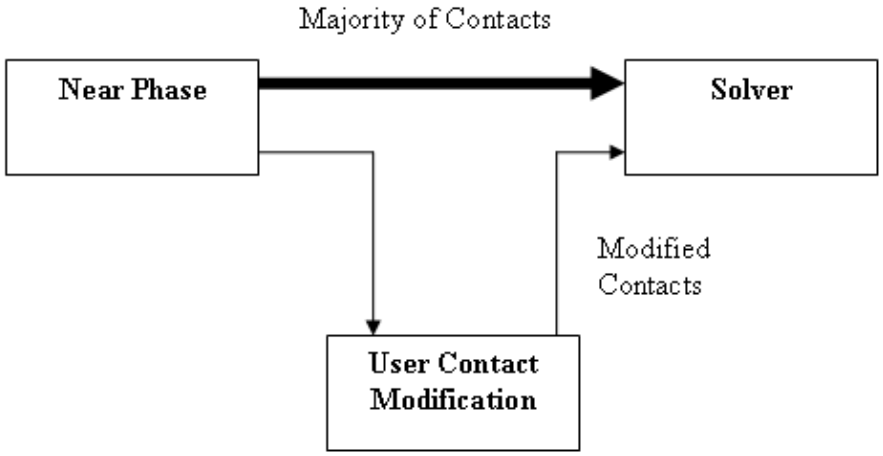
A mutex was chosen to protect raycasting because, in most cases, it yields better performance and less overhead than a reader/writer lock. The exception occurs if very heavy raycasting is performed from many threads. Customers with source can instead select the reader/writer lock on platforms which are statically linked, such as the XBox 360 (see NxReaderWriterLock.h).

API Reference

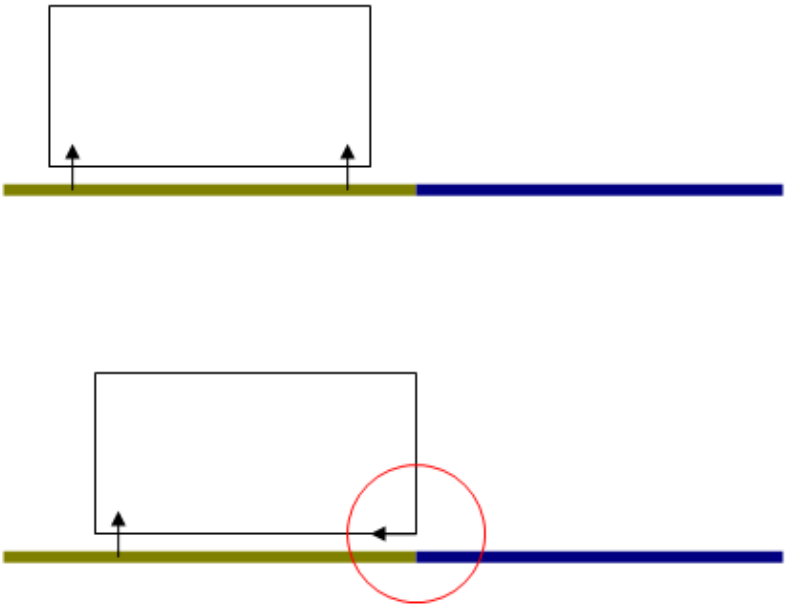
- NxScene

Contact Modification

In version 2.5 and later of the PhysX SDK, it is possible to intercept contacts between the near phase and solver to allow the user to modify the contact information. This ability should not be used unless absolutely necessary as there is a significant performance impact, especially when running on hardware. However, the ability to modify contacts can be very useful when dealing with important game play objects.



For example, it might be necessary to modify contacts when a car crosses a seam between two mesh objects. At this point, with default contact generation, incorrect contact constraints can be generated due to collision not being performed across both objects at once.



Contact modification is accomplished by providing a callback class which receives notifications when contacts are available to be modified.

```
class NxUserContactModify
{
```

```
        virtual bool onContactConstraint(
            NxU32& changeFlags,
            const NxShape* shape0,
            const NxShape* shape1,
            const NxU32 featureIndex0,
            const NxU32 featureIndex1,
            NxContactCallbackData& data) = 0;

    };
```

There are a couple of special requirements for onContactConstraint() due to the callback coming from deep inside the SDK. In particular, the callback should be thread safe and reentrant. In other words, the SDK may call onContactConstraint() from any thread and it may be called concurrently (i.e., asked to process 2 or more constraints simultaneously).

The contact modification callback class can be set using the userContactModify member of NxSceneDesc or the setUserContactModify() methods of NxScene.

```
NxUserContactModify* NxSceneDesc::userContactModify;

virtual void NxScene::setUserContactModify(NxUserContactModify* callback) = 0;
virtual NxUserContactModify* NxScene::getUserContactModify() const = 0;
```

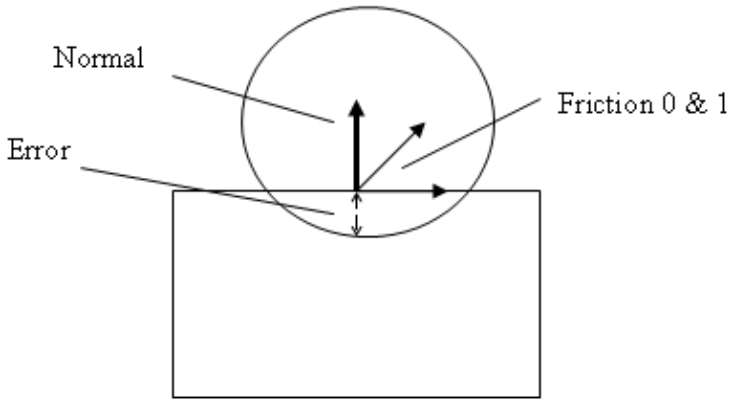
Below is a list of the onContactConstraint() parameters:

- **changeFlags** - Set the change flags based upon which members of NxContactData have been changed. Modifying the members of NxContactData without setting the appropriate flags can cause unexpected results. Bit wise OR together members of NxContactConstraintChange to signify the set of changes.
- **shape0** - First shape involved in the contact.
- **shape1** - Other shape involved in the contact.
- **featureIndex0** - Feature index associated with shape0. Feature indices are only defined for triangle mesh and heightfield shapes. A feature index for a triangle mesh shape is the pre cooked triangle index. For a heightfield shape, a feature index is a triangle index as specified on creation, including holes in the index.
- **featureIndex1** - Feature index associated with shape1.
- **data** - Contact data (see NxContactCallbackData for more information).

Contact Data:

- **minImpulse** - Minimum impulse value that the solver can apply. Normally this should be 0, negative amount gives sticky contacts.
- **maxImpulse** - Maximum impulse value that the solver can apply. Normally this is FLT_MAX. If you set this to 0 (and the min impulse value is 0) then you will void contact effects of the constraint.
- **error** - Error vector. This is the current error that the solver should try to relax.
- **target** - Target velocity. This is the relative target velocity of the two bodies.
- **localpos0** - Constraint attachment point for shape 0. If the shape belongs to a dynamic actor, then localpos0 is relative to the body frame of the actor. Alternatively it is relative to the world frame for a static actor.
- **localpos1** - Constraint attachment point for shape 1.
- **localorientation0** - Constraint orientation quaternion for shape 0 relative to shape 0s body frame for dynamic actors and relative to the world frame for static actors. The constraint axis (normal) is along the x-axis of the quaternion. The Y axis is the primary friction axis and the Z axis the secondary friction axis.
- **localorientation1** - Constraint orientation quaternion for shape 1.

- **staticFriction0** - Static friction parameter 0. (Note: 0 does not have anything to do with shape 0/1, but is related to anisotropic friction, 0 is the primary friction axis).
- **staticFriction1** - Static friction parameter 1.
- **dynamicFriction0** - Dynamic friction parameter 0.
- **dynamicFriction1** - Dynamic friction parameter 1.
- **restitution** - Restitution value.



Contact modification can be enabled on a per actor basis using the NX_AF_CONTACT_MODIFICATION actor flag. This tells the SDK to call the contact modification callback for each contact between the actor and any other actor.

```
NxActorDesc::flags
```

Alternatively, the NX_NOTIFY_CONTACT_MODIFICATION flag can be set for a pair of actors, causing the contact modification callback to be active only for a specific pair.

```
virtual void setActorPairFlags(NxActor& actorA, NxActor& actorB, NxU32 nxContactPairFlag) = 0
```

API Reference

- NxScene
- NxUserContactModify

Profiler and Scene Stats

Basic Profiling

Basic profiling information can be retrieved programmatically through the readProfileData() member of NxScene. This returns a NxProfileData object, which contains an array of NxProfileZone-s. The exact list of the zones exposed may vary from release to release, but certain zones defined by the NxProfileZoneName enum can be easily searched for using NxProfileData::getNamedZone().

Detailed Profiling

For customers with a source code release, the NVIDIA PhysX SDK can be compiled with profiling support for the software core. To control this there is a define in NxProfiler.h called NX_ENABLE_PROFILER. When enabled, timing information is gathered during the simulation and can be retrieved either through the Visual Remote Debugger or programmatically through the readProfileData() member of NxScene.

Profiling data is gathered as a series of nested zones. For example, there might be a zone which provides the timing for the simulation, nested within timing for collision detection, constraint solving, etc. Profiling zones are presented to the user through the NxProfileData and NxProfileZone classes. A profile zone contains the following:

- The name of the zone (name)
- The number of times the zone was entered (callCount)
- The time it took to execute the zone, including sub zones (hierTime)
- The time it took to execute the zone, minus that for nested zones (selfTime)
- Recursion level, i.e., the number of parent zones (recursionLevel)
- The percentage time the zone took of its parent zone

When the scene is executed across multiple threads, the timing results for all zones on all threads are concatenated in the array of zones provided by readProfileData().

Since the profiling data is dependant on the internal structure of the SDK, the zones defined will vary between releases. For best results, new zones should be defined when required. See Scene::Scene() and Profiler.h for details.

In addition to the above information, it is possible to enable an additional counter for the zones using the NX_ENABLE_PROFILER_COUNTER define. This counter has specific uses on some platforms. For example, on XBox 360 it is possible to modify ProfileClock.h (when source code is available) to provide level 2 cache misinformation.

NOTE: There is a significant performance penalty for enabling the profiler, in the order of 10-20%, so it should only be enabled when needed (and not using an external profiler).

NOTE: When retrieving profiler data, make sure the SDK executes a consistent number of substeps, or at least one; otherwise, profiler results may be misleading.

Scene Stats

The NVIDIA PhysX SDK provides a simple way of retrieving statistics about the current scene, such as body count, constraint count, etc. This information is accessed through the Visual Remote Debugger or through the getStats() member of NxScene.

The following list contains some scene statistics:

- Current number of contacts and maximum number of contacts
- Current number of constraints and maximum number of constraints
- Number of shapes
- Number of joints
- etc.

See the API reference for specific details.
There is also an extended scene stat class that may be used to retrieve more in-depth information on the simulation.

API Reference

- NxScene
- NxProfileZone
- NxProfileData

Extended Scene Stats

For purposes of both optimizing performance and troubleshooting physics artifacts, PhysX version 2.6.1 adds an extended scene statistic API with a large amount of data that can be extracted from a scene in order to ascertain what it is doing beneath the hood.

The information is stored in the NxSceneStats2 class. Using the NxScene::getStats2, you can access the statistics for the last simulated frame in the scene. You may either process this data on a frame-per-frame basis, or extract pertinent subsets of the data and store it for later inspection.
Note that if you wish to store the data each frame, you will need to copy it out of the NxSceneStats2 structure yourself, as it refers to an internal data structure which is overwritten during each simulation step.

The data items consist of simple counters, enumerating important numerical quantities of the scene, including e.g. different categories of shapes as well as certain warnings that may arise when, for example, internal limits are exceeded. The maximum value of each counter over the lifetime of the scene is also kept on record. Furthermore, some counters are considered subordinate to others; this hierarchy can help in sorting and filtering data items. A counter will not exceed the count of its parent (but the total of the parent's children may).

Here follows a list of the current set of scene stat counters. Note, however, that items may be added to this list in the future, and that its contents are not considered part of the API (though its contents are guaranteed to remain the same while a program is running).

```
TotalPairs - Total number of pairs of actors that exist for near-phase collision detectio
        TotalFastPairs - Of those, how many are "fast"
        TotalContactNotificationPairs - How many generate notifications
        TotalContactModificationPairs - How many generate modification callbacks
TotalFilteredPairs - How many pairs are filtered out (this may include lightweight pairs
ActivePairs - Total number of active pairs of actors that exist for near-phase collision
        ActiveFastPairs - Of those, how many are "fast"
        ActiveContactNotificationPairs - How many generate notifications
        ActiveContactModificationPairs - How many generate modification callbacks

TotalContacts - Total number of contact points between rigid bodies

TotalActors - Total number of actors in scene
    TotalDynamicActorsInAwakeGroups - Of those, how many are dynamic and part of a group
        TotalDynamicActors - How many are dynamic
        TotalCompoundActors - How many are compounds
                TotalCompoundActors3Plus - How many have above 3 shapes
                        TotalCompoundActors10Plus - How many have above 10 shapes
        TotalKinematicActors - How many are kinematic

ActiveDynamicActors - Total number of active actors in scene
        ActiveCompoundActors - Of those, how many are compounds
                ActiveCompoundActors3Plus - How many have above 3 shapes
                        ActiveCompoundActors10Plus - How many have above 10 shapes

TotalShapes - Total number of shapes in scene
    TotalStaticShapes - Of those, how many are static
TotalMirroredShapes - How many are mirrored into compartments
    TotalTriggerShapes - How many are trigger shapes
    TotalCCDEnabledShapes - How many have Continuous Collision Detection enabled
    TotalBoxShapes - How many are boxes
    TotalConvexShapes - How many are convexes
            TotalBigConvexShapes - How many are convexes too big to simulate on hardware
    TotalSphereShapes - How many are spheres
    ... (etc. for all shapes)

    TotalDynamicShapes - How many are dynamic
    ActiveDynamicShapes - Total number of shapes belonging to active dynamic actors in sc
```

```
                ActiveTriggerShapes - How many are trigger shapes
                ActiveBoxShapes - How many are boxes
                ActiveConvexShapes - How many are convexes
                        ActiveBigConvexShapes - How many are convexes too big to simulate on hardwar
                ActiveSphereShapes - How many are spheres
                ... (etc. for all shapes)

    TotalColliders - Total number of contact managers between rigid body shapes in the scene
        TotalSwBoxBoxColliders - Of those, how many are between box and box, and run in software
        TotalSwBoxSphereColliders - How many are between box and sphere, and run in software
        TotalSwBoxCapsuleColliders - How many are between box and capsule, and run in software
        ...

        TotalHwBoxBoxColliders - Of those, how many are between box and box, and run on hardware
        TotalHwBoxSphereColliders - How many are between box and sphere on hardware
        TotalHwBoxCapsuleColliders - How many are between box and capsule on hardware
        ...

SwForcedColliders - Number of contact managers that use software fallbacks.

    ActiveColliders - Total number of contact managers between active rigid body shapes in the scene
        ActiveSwBoxBoxColliders - Of those, how many are between box and box, and run in software
        ActiveSwBoxSphereColliders - How many are between box and sphere, and run in software
        ActiveSwBoxCapsuleColliders - How many are between box and capsule, and run in software
    ...

        ActiveHwBoxBoxColliders - Of those, how many are between box and box, and run on hardware
        ActiveHwBoxSphereColliders - How many are between box and sphere on hardware
        ActiveHwBoxCapsuleColliders - How many are between box and capsule on hardware
    ...


    TotalTriggers - Total number of trigger pairs between active rigid body shapes in the scene
        SwBoxBoxTriggers - Of those, how many are between box and box, and run in software
        SwBoxSphereTriggers - How many are between box and sphere, and run in software
        SwBoxCapsuleTriggers - How many are between box and capsule, and run in software
        ...

        HwBoxBoxTriggers - Of those, how many are between box and box, and run on hardware
        HwBoxSphereTriggers - How many are between box and sphere, and run in hardware
        HwBoxCapsuleTriggers - How many are between box and capsule, and run in hardware
        ...

    TotalMeshPageInstances - Total number of mesh page instances in the scene
    MappedMeshPageInstances - Total number of mappings of mesh pages
    AutoPageMappings - Number of automatically initiated page mappings this frame
    AutoPageUnmappings - Number of automatically initiated page unmappings this frame
    FailedPageMappings - Number of failed mapping attempts this frame

    TotalIslands - Total number of rigid body islands in the scene

    TotalJoints - Total number of joints in the scene
            TotalD6Joints - Of those, how many are D6 joints
            TotalSphericalJoints - How many are spherical
            TotalRevoluteJoints - How many are revolute
            ...

    ActiveJoints - Total number of active joints in the scene
            ActiveD6Joints - Of those, how many are D6 joints
            ActiveSphericalJoints - How many are spherical
            ActiveRevoluteJoints - How many are revolute
            ...
    DeadJoints - Number of joints that are in the dead joint list (from being broken or having one a

    TotalRaycasts - Number of raycasts performed in the scene since last simulate call
    TotalOverlapTests - Number of overlap tests performed in the scene since last simulate call
    TotalSweepTests - Number of sweep tests performed in the scene since last simulate call
```

```
        TotalFluidEmitters - Number of fluid emitter shapes in the scene
        TotalFluids - Total number of fluids in the scene
        TotalFluidParticles - Total number of fluid particles in the scene
        TotalFluidPackets - Total number of fluid packets in the scene
        ActiveFluidParticles - Total number of fluid particles being simulated in the scene

        TotalCloths - Total number of cloths in the scene
        ActiveCloths - Total number of cloths being simulated in the scene
        TotalTearableClothPieces - Number of cloths that are set as tearable
        ActiveTearableClothPieces - Number of active cloths that are set as tearable
        TotalClothVertices - Total number of vertices in cloths in the scene
            TotalAttachedClothVertices - Of those, how many are attached to rigid bodies
        ActiveClothVertices - Total number of vertices in active cloths in the scene
        ActiveAttachedClothVertices - Of those, how many are attached to rigid bodies

        TotalForceFields - Total number of force fields
        TotalForceFieldShapes - Total number of force fields' shapes
            TotalBoxForceFieldShapes - Of those, how many are box shapes
            TotalSphereForceFieldShapes - How many are sphere shapes
            TotalCapsuleForceFieldShapes - How many are capsule shapes
            TotalConvexForceFieldShapes - How many are convex mesh shapes

    WarnConvexShapesIgnored - Number of convex shapes ignored in cloth collision due to too many
    WarnClothShapesDropped - Number of shapes ignored in cloth collision due to limit to collidin
    WarnAxisConstraintsFailed - Number of joint constraints that could not be processed in hardwa
    WarnJointDrivesIgnored - Number of unsupported joint drive types that are ignored (SLERP or a
    WarnPacketLimitedFluids - Number of fluids currently at their maximum packet capacity - furth
```

API Reference

- NxScene
- NxSceneStats2
- NxSceneStatistic

XBox 360 Notes

[Only available when installing the XBox 360 version of the SDK]

See XBox 360 Notes

Visual Remote Debugger

Using the VRD it is possible to interact with, record the physics behavior of, and study the settings on all objects in the Physics scenes. To enable a connection from an application to the visual remote debugger, after starting the VRD, call the following method:

```
gPhysicsSDK->getFoundationSDK().getRemoteDebugger()->connect("localhost", 5425);
```

As of version 2.7.0 of the SDK, it is possible to connect to the VRD at any time during the life of the application. The only rule is that the Physics engine should not be in a state of simulation when the connection is made.

For further details, see the Visual Remote Debugger Documentation and the SampleVRD code, and see the XBox 360 Notes for information concerning creating a connection from an application running on the XBox 360 to the VRD.

API Reference

- NxFoundationSDK

Serialization Class Library

The serialization class library provides a method of saving a complete scene into a portable binary format (ascii is supported for write-only to aid debugging).

In addition, the serialization library is flexible enough to write out parts of a scene (e.g., individual actors) and interleave this data into the user's file format. The advantage of using the serialization class library is that the user does not have to write tedious serialization code and deal with the many special cases associated with serializing the objects.

Collections

The serialization library accomplishes all reading and writing by means of collections, NXU::NxuPhysicsCollection.
A 'physics collection' is a set of related scenes, meshes, actors, constraints, cloth, and fluids. If can be everything in an entire game level or something as simple as just one 'chair'. Once a physics collection has been instantiated on the SDK it can be released without memory leaks or dangling pointers.

Note: Though the collection has a set of public methods and members, these are for internal use and you should not rely on them when using NxuStream; instead, interact with collections through the global NXU:: methods (see the NXU namespace). The members of NxuPhysicsCollection are not considered public API and may change without notice.

A collection is saved or retrieved using the functions NXU::loadCollection and NXU::saveCollection (all the serialization interface is located within the namespace "NXU"). The file format is specified using the enum NXU::NXU_FileType:

- NXU_FileType::BINARY - Binary stream. Efficient but not human-readable nor, in general, cross-version compatible.
- NXU_FileType::XML - Human-readable XML stream.
- NXU_FileType::COLLADA - Collada stream.

Of the above, the binary and xml formats are reflective of the SDK; that is, they store all relevant information. The COLLADA format conforms to the COLLADA industrial standard and doesn't contain all features in PhysX; on the other hand, it contains graphics information that is ignored by PhysX.
Collections should be released with NXU::releaseCollection() when they are no longer needed:

```
/* Read a collection from a binary serialized file:*/

NXU::NxuPhysicsCollection* collection = NXU::loadCollection(pFilename, NXU::NXU_FileType::BIN

//...

/* Create an XML output file */

NXU::saveCollection(collection, pFilename2, NXU::NXU_FileType::XML);

NXU::releaseCollection(collection);
```

Saving Physics State

To save a scene or objects within a scene, the user must create a NxuPhysicsCollection object; subsequently, the user can call the NXU::add... function to save individual components or the entire scene.

After adding objects to the collection, call NXU::saveCollection() to serialize the added contents to a file. It is also possible to serialize into a memory buffer using NXU::saveCollectionToMemory.

Alternatively, you may use the convenience functions NXU::extractCollectionSDK and NXU::extractCollectionScene to automatically create collections containing the entire SDK or a whole scene, respectively.
In addition, there's one more convenience function, NXU::coreDump, which entirely hides the interface to collections and simply writes the entire SDK to a file of the specified type.

Loading Physics State

Loading physics using the serialization library is a simple matter of retrieving the collection using NXU::loadCollection, and then instantiating it.
When instantiating a collection using NXU::instantiateCollection, you have the option of providing a NxScene pointer. If you do, the instantiation will not create a new scene, but insert all its contents into the provided scene. You may also specify a spatial transform when doing this, which allows you to place the instantiated objects wherever you wish in the existing scene.

```
NxMat34 instanceFrame;
instanceFrame.t = NxVec3(10.0f, 0, 0); // Moves the instance 10 units in the positive x direction
NXU::instantiateCollection(collection, gPhysicsSDK, existingScene, &instanceFrame, NULL);
```

Instances and hierarchy

NxuStream supports a hierarchical scheme which allows you to save a great deal of storage space using instantiation. The basic idea is simple: A scene may be stored once but instanced multiple times in different places. See Hierarchies for more information.

User Callbacks

The serialization class library provides a couple of callback interfaces to allow the user to control scene creation and associate their own data with physics objects as they are loaded.

NXU_userNotify provides one set of callbacks which are executed just after an object is created and all its state has been initialized, and another set which is invoked before creation, allowing you to modify the descriptors after they're read from the collection, and abort creation by returning "false".
The first set is prefixed "NXU_notify..." and the second "NXU_preNotify...". Overload any or all of these methods in order to receive the particular notifications.
The scene pre-creation callback is special in that it allows you to specify an existing scene to put the contents in, rather than just modifying the descriptor; returning NULL will create a new scene from the descriptor.

```
class MyUserNotify : public NXU::NXU_userNotify
 {
     public:

     //Notify the application of a scene creation.
     virtual void NXU_notifyScene(NxU32 sceneNo, NxScene *scene, const char *userProperties) {...}
     //Notify application of a joint creation.
     virtual void NXU_notifyJoint(NxJoint *joint, const char *userProperties) {...}

     //Control creation of scenes
     virtual NxScene * NXU_preNotifyScene(NxU32 sceneNo, NxSceneDesc &desc, const char *userPropertie
     {
         desc.userData = ...;
         return NULL;
```

```
        }
        //Control creation of joints
        virtual bool NXU_preNotifyJoint(NxJointDesc &desc, const char *userProperties)
        {
            desc.userData = ...;
            return true;
        }
    };


    ...
    MyUserNotify gUserNotify;
    NXU::instantiateCollection(collection, gPhysicsSDK, NULL, NULL, &gUserNotify);
```

Compartments

Being part of a scene, compartments require some special consideration when serializing scenes that contain them. If such a scene is instantiated "as is", each compartment will be recreated with its original contents inside it as expected.

However, if a scene with compartments is instantiated inside another scene rather than by itself, NxuStream will attempt to match source and destination compartments so as not to create unnecessarily many of them. Each compartment in the source scene will be matched to the first compartment with *the same descriptor* found in the destination scene, and its contents placed in that destination compartment. If no matching compartment is found a new one is created.

For example, if a scene with a fluid compartment is instantiated in an existing scene with an identically (parameter-wise) defined fluid compartment, the fluids will be instantiated inside the existing compartment instead of a new one. If there are two identical fluid compartments in the source, only the first will be absorbed into the existing fluid compartment, while a new one is created to accept the contents of the second.

You can of course always override this behavior by overloading the pre-create callbacks for the relevant content types and manually setting the compartment to place them into.

Cloth and soft bodies

Cloth and soft bodies contain a certain amount of internal state information that is not normally visible to the user (it is not part of the descriptor). By default, this internal state is not saved by NxuStream. Though greatly reducing memory costs, this means that a cloth or soft body will revert to its initial shape when it is instantiated.

In particular, cloth attachments are not serialized on a per-particle basis by default. This means that when instantiated the best NxuStream can do to recreate the attachment is call NxCloth::attachToShape, which may attach different particles than originally, if the shape has moved relative to the cloth.

If this is not acceptable, you can elect to save the active state when serializing. To do this, you call:

```
NXU::setUseClothActiveState(true);
```

or

```
NXU::setUseSoftBodyActiveState(true);
```

before serializing that asset.

Note that active state serialization of metal cloth is not currently supported.

API Reference

- NXU namespace
- NXU_userNotify

Serialization Hierarchies

When creating a collection using NxuStream, you can organize assets into a hierarchical structure using the NXU::addSceneInstance() and NXU::addToCurrentSceneInstance() functions. This allows you to to create instances of assets as many times as you like without having the collection redundantly store all data for each instance.

The basic building block for this hierarchy is the *scene instance*, which acts both as a placeholder for another asset (scene or scene instance) that already exists in the collection, and as a container for subordinate scene instances. This allows for a tree structure, where each node can be either abstract (referring only to other scene instances) or concrete (referring to an existing scene).
When constructing a collection, NxuStream keeps a pointer to the *current* scene instance. The addSceneInstance function inserts a new scene instance as a child of the current instance; addToCurrentSceneInstance does the same, but also makes the newly created instance current, effectively moving down the tree one level. To return to the previous level, call NXU::closeCurrentSceneInstance().

Both addSceneInstance and addToCurrentSceneInstance take the following arguments:

- *c* - the NxuPhysicsCollection that is being added to
- *instanceId* - the name to give the newly created scene instance
- *sceneName* - the name of the existing scene or scene instance for which the new instance is a placeholder
- *rootNode* - pose offset of the instance; relative to its parent if applicable, otherwise to the global frame.
- *ignorePlane* - whether or not to ignore ground planes upon instantiation. True by default.

Instantiating the hierarchy

The following is the procedure NxuStream follows when instantiating a collection:

- If there are no scene instances, instantiate each scene in the collection.
- otherwise, for each scene instance:
    - If the instance points to a scene, instantiate that scene.
    - Otherwise (the instance points to a scene instance) recurse.
    - Finally, recurse for each child instance - but *only* if this is not a top-level scene instance (see below).

Note that NxuStream does not recurse over children for top-level nodes. This is to allow you to create abstract scene instances - i.e. instances that will not automatically be instantiated unless they're referenced somewhere in the hierarchy. For example, you may want to set up a "room" template, composed of instances of "chairs" and "tables", without having a room automatically created in your scene at the origin.
Also note that the scenes themselves are automatically made abstract in this sense as soon as you add an instance to the collection; no actual scenes will be instantiated unless referenced by a scene instance (that is not itself abstract).

The instance scene

It's important to realize that all the objects created by a call to NXU::instantiateCollection will go into the *same* NxScene. If you do not supply a defaultScene with the call, the first scene actually created by the above procedure will become the instance scene and everything else will be created in it.
Thus, the scene instance hierarchy is not used to create multiple scenes, but rather to populate one scene with assets in a structured and data-efficient way. The scenes in the collection are viewed not as independent NxScenes but as containers for physics assets.

Example

As an illustration of the above, picture a collection containing a game level that consists of a street with cars. The cars are all the same physical model and so lend themselves to instancing. Furthermore, they contain drivers - ragdolls - which are also all the same. There are ragdolls in other parts of the scene as well.

To create this setup, you create a scene in PhysX for each of the assets (street, car, ragdoll) and add each to the collection using NXU::addScene, with a descriptive identifier string: "Street", "Car", "Ragdoll".

Then, you create the abstract person-in-car instance:

```
NXU::addToCurrentSceneInstance(c, "CarAndDriver", 0, NxMat34());
```

and populate it with the car and the ragdoll:

```
NXU::addSceneInstance(c, "CarBody", "Car", NxMat34());
NXU::addSceneInstance(c, "Driver", "Ragdoll", NxMat34());
```

Finished with CarAndDriver, we close it:

```
NXU::closeCurrentSceneInstance();
```

Now you can create the master level instance and begin to populate it.

```
NXU::addToCurrentSceneInstance(c, "LevelInstance", 0, NxMat34());
NXU::addSceneInstance(c, "StreetInstance", "Street", NxMat34());

NxMat34 carOffset = ...;
NXU::addSceneInstance(c, "CarInstance1", "CarAndDriver", carOffset);
carOffset = ...;
NXU::addSceneInstance(c, "CarInstance2", "CarAndDriver", carOffset);
...
NxMat34 dollOffset = ...;
NXU::addSceneInstance(c, "Pedestrian1", "Ragdoll", dollOffset);
...
NXU::closeCurrentSceneInstance();
```

Finally, create the concrete instance - a top-level scene instance referring to "LevelInstance" and with no children:

```
NXU::addSceneInstance(c, "Level", "LevelInstance", NxMat34());
```

On disk, this collection will be compact, containing the data for the car, ragdoll and street only once, plus the hierarchy information - roughly the size of the code used to generate it. When instantiated, one scene will be populated with street, cars and ragdolls; the scene used (as far as scene parameters and flags are concerned) will be that of the original "Street" NxScene, since that is the first one encountered in the concrete scene instance "Level". Of course, you can also put everything in an existing scene using the defaultScene argument of NXU::instantiateCollection.

API Reference

- NXU namespace
- NXU_userNotify

NVIDIA PhysX SDK FAQ

How do I save a core dump?

The NVIDIA PhysX SDK no longer supports explicit core dump functionality. However, the same functionality is now provided by the sample code in the form of a helper class which serializes the state of all objects.

See Saving the Simulation State for a more detailed description.

I have objects at the beginning of my scene that start out inter-penetrating slightly and then fly apart when I begin the scene. How do I prevent this? What is the NX_SKIN_WIDTH parameter?

If at all possible, do not start your scene with objects inter-penetrating. Find a way to push them apart before the scene begins. If you can't, adjust the NX_SKIN_WIDTH parameter so the objects can overlap to the depth that the parameter specifies.

The NX_SKIN_WIDTH parameter is the depth which the physics solver allows for inter-penetration between objects (a certain "grace depth" of penetration before collision). Notice how far your objects are penetrating each other at the beginning of the scene. You want to set the NX_SKIN_WIDTH to a value just bigger than this.

You can set the bounding volume's size (e.g., a box's dimensions or a sphere's radius) to the graphically appropriate size plus half of NX_SKIN_WIDTH. In this way the objects will come to rest with the graphical representations barely touching. Setting NX_SKIN_WIDTH too small will lead to jitter/instability, while setting it too large will make inter-penetration too obvious. You can reduce the NX_SKIN_WIDTH that is needed for good simulation by reducing the timestep size.

What types of shapes should I use for my objects to get collision results?

For speed and robustness use the following:

- box/box
- convex mesh/convex mesh
- (pmap/pmap)

If at all possible, just use boxes, spheres or capsules. Box collisions are highly optimized and robust. If you want to use a collision object that is not a box, but not concave, use a convex mesh. If your object is concave, decompose it into convex parts. PMaps are legacy objects and should not be used.

What is the difference between using limit planes and limit values for joints?

Limit planes allow you to specify a point on one of the jointed bodies that gets stopped by the limit planes when the joint rotates or translates. Limit values allow you to specify an axis and the minimum and maximum angles around the axis that the joint can rotate, or the minimum and maximum distances on the axis that the joint can translate.

What is an example in code using limit planes and limit points?

The following code demonstrates a basic example of joint limits, using a revolute joint and two actors, a door and a bigger, heavier wall it is attached to.

```
NxActor *door, *wall;
NxRevoluteJoint *hinge;

NxActor *CreateBox(const NxVec3& pos, const NxVec3& boxDim, const NxVec3& localPos, float density)
{
    NxBodyDesc BodyDesc;
    NxBoxShapeDesc BoxDesc;

    BoxDesc.dimensions.set(boxDim.x,boxDim.y,boxDim.z);
    BoxDesc.localPose.t = localPos;

    NxActorDesc ActorDesc;
    ActorDesc.shapes.pushBack(&BoxDesc);
    ActorDesc.body = &BodyDesc;
    ActorDesc.density = density;
    ActorDesc.globalPose.t = pos;

    return gScene->createActor(ActorDesc);
}

void CreateDemo()
{
    //Create door
    NxVec3 doorDim = NxVec3(4,9,1);
    NxVec3 doorLocalPos = NxVec3(0,10,0);
    NxVec3 doorStartPos = NxVec3(0,0,0);

    door = CreateBox(doorStartPos, doorDim, doorLocalPos, 2);

    //Create wall
    NxVec3 wallDim = NxVec3(8,10,2);
    NxVec3 wallLocalPos = NxVec3(0,10,-1);
    NxVec3 wallStartPos = NxVec3(-12,0,0);

    wall = CreateBox(wallStartPos, wallDim, wallLocalPos, 15);

    //BEGIN LIMIT PLANE CODE
    NxRevoluteJointDesc revJointDesc;
    NxVec3 doorPivotPos(-4,9,1);

    revJointDesc.setToDefault();
    revJointDesc.actor[0] = door;
    revJointDesc.actor[1] = wall;

    revJointDesc.setGlobalAnchor(doorPivotPos);
    revJointDesc.setGlobalAxis(NxVec3(0,1,0));

    NxJoint* joint = gScene->createJoint(revJointDesc);

    //Create hinge limits
    NxVec3 limitPoint = NxVec3(4,9,1);
    NxVec3 limitNormal1 = NxVec3(0,0,1);
    NxVec3 limitNormal2 = NxVec3(0.707,0,-0.707);

    joint->setLimitPoint(limitPoint, false);
    joint->addLimitPlane(limitNormal1, doorPivotPos);
    joint->addLimitPlane(limitNormal2, doorPivotPos);
```

```
//END LIMIT PLANE CODE

hinge = joint->isRevoluteJoint();

//No sleep for jointed bodies

door->wakeUp(1e10);
wall->wakeUp(1e10);
}
```

The crucial bit is contained in that last part with the hinge limits. The door is hinged to the wall at (-4,9,1), the doorPivotPos, and swings around the y-axis (0,1,0). The limit point (4,9,1) is a point on the door on the other side of the hinge (i.e., where the doorknob is). The two limit planes are the xy-plane (normal (0,0,1)) and a plane rotated 45 degrees from the first around y, both with the same point-in-plane (the doorPivotPos).

So the result is a door that swings around the y-axis and gets stopped when the doorknob hits the xy-plane or when it hits a plane rotated 45 degrees from the xy-plane. The door stops when swung closed or 45 degrees open. The normals of the limit planes point inward, toward the area through which the door can swing.

Notice the code difference between using limit values and limit planes. The following code snippets produce the same results, one using limit planes and one using limit values. Remove the first snippet from the above example and replace with the second to use limit values instead of limit planes.

Limit Planes:

```
//BEGIN LIMIT PLANE CODE

NxRevoluteJointDesc revJointDesc;

NxVec3 doorPivotPos(-4,9,1);
revJointDesc.setToDefault();
revJointDesc.actor[0] = door;
revJointDesc.actor[1] = wall;

revJointDesc.setGlobalAnchor(doorPivotPos);
revJointDesc.setGlobalAxis(NxVec3(0,1,0));

NxJoint* joint = gScene->createJoint(revJointDesc);

//Create hinge limit point and planes

NxVec3 limitPoint = NxVec3(4,9,1);
NxVec3 limitNormal1 = NxVec3(0,0,1);
NxVec3 limitNormal2 = NxVec3(0.707,0,-0.707);

joint->setLimitPoint(limitPoint, false);
joint->addLimitPlane(limitNormal1, doorPivotPos);
joint->addLimitPlane(limitNormal2, doorPivotPos);

//END LIMIT PLANE CODE
```

Limit Values:

```
//BEGIN LIMIT VALUE CODE

NxRevoluteJointDesc revJointDesc;

NxVec3 doorPivotPos(-4,9,1);
revJointDesc.setToDefault();
```

```
revJointDesc.actor[0] = door;
revJointDesc.actor[1] = wall;

revJointDesc.setGlobalAnchor(doorPivotPos);
revJointDesc.setGlobalAxis(NxVec3(0,1,0));

//Enter hinge limit values

revJointDesc.limit.low.value = -NxPi/4.0f;
revJointDesc.limit.high.value = 0.0f;
revJointDesc.flags = NX_RJF_LIMIT_ENABLED;

NxJoint* joint = gScene->createJoint(revJointDesc);

//END LIMIT VALUE CODE
```

My creature keeps twitching and popping after it falls to the ground.

Check the masses of the character's limbs; they may be fairly small relative to the other parts. This may be contributing to the twitchiness. Try to normalize the masses of the body parts so that the heaviest is not more than 10 times as heavy as the lightest. Also, remember that you can raise the iteration count of the bodies that make up the creature using the solverIterationCount field in NxBodyDesc.

Other Resources

In addition to this User Guide, the following resources may provide help using the SDK:

- API Reference Manuals
  - Physics SDK
  - Cooking SDK
  - Character SDK
- Source code (e.g., programs that come with the SDK and give you reusable programming demos on how to do certain common things
  - Tutorials and Samples
- PhysX technical support is available through the NVIDIA developer website: http://developer.nvidia.com/object/physx.html

External Resources

This user guide and API reference provide information for using the PhysX SDK in an application. However, it does not attempt to teach many basic concepts which are necessary to fully utilize physics within a game or application.

The references listed below, while not endorsed or supported, may be useful for providing background:

Rigid Body Dynamics

- Chris Hecker's Rigid Body Dynamics Information
- Game Physics, by David H. Eberly
- Physically Based Modeling: Rigid Body Simulation, by David Baraff

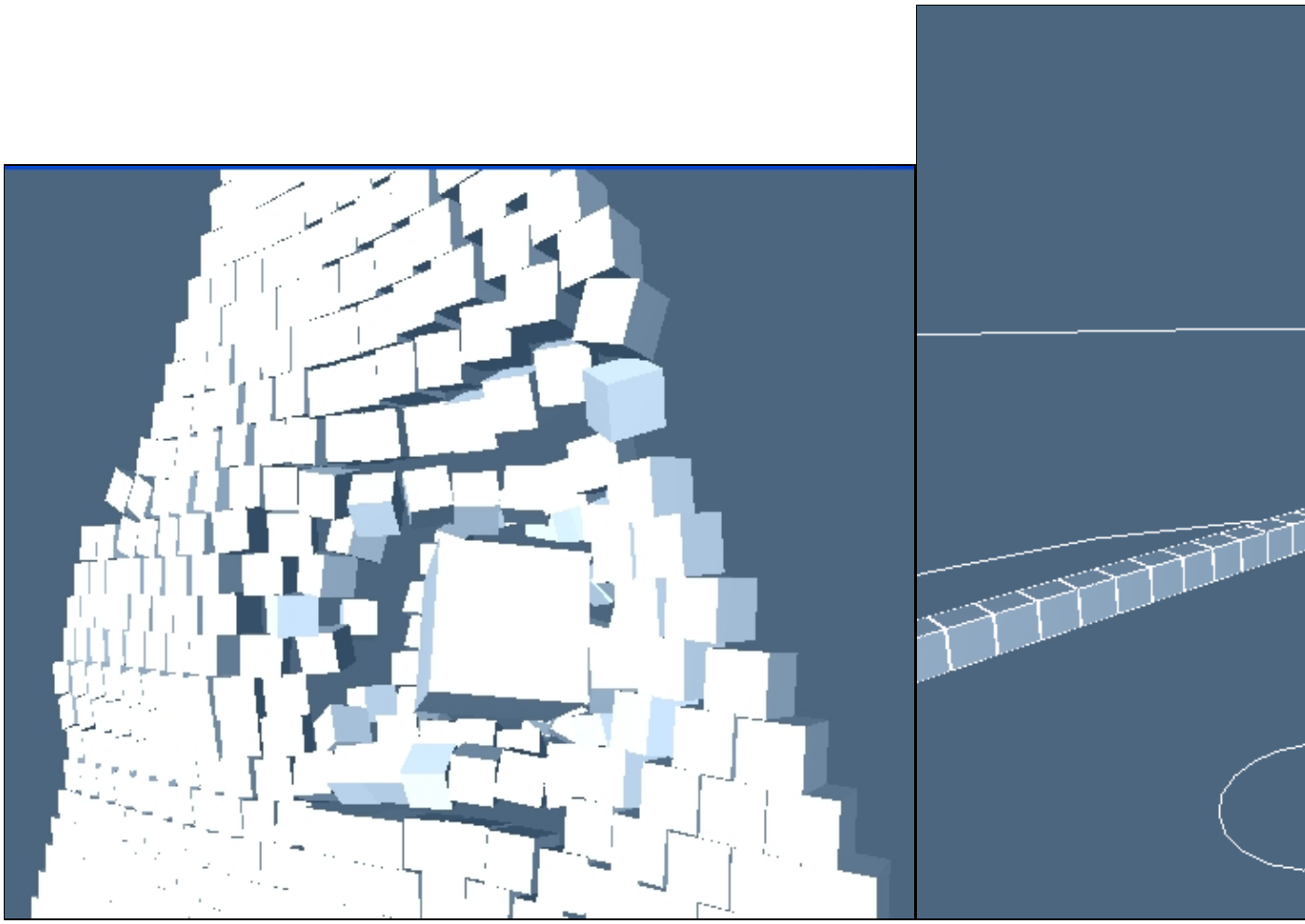
Collision Detection

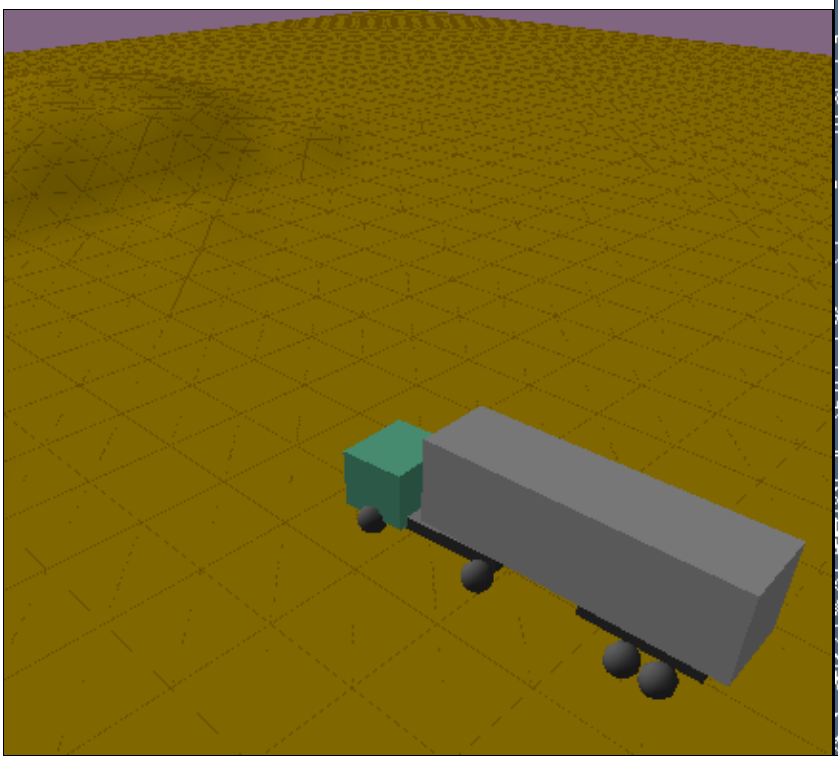
- Real Time Collision Detection, by Christer Ericson
- Collision Detection in Interactive 3D Environments, by Gino Van Den Bergen

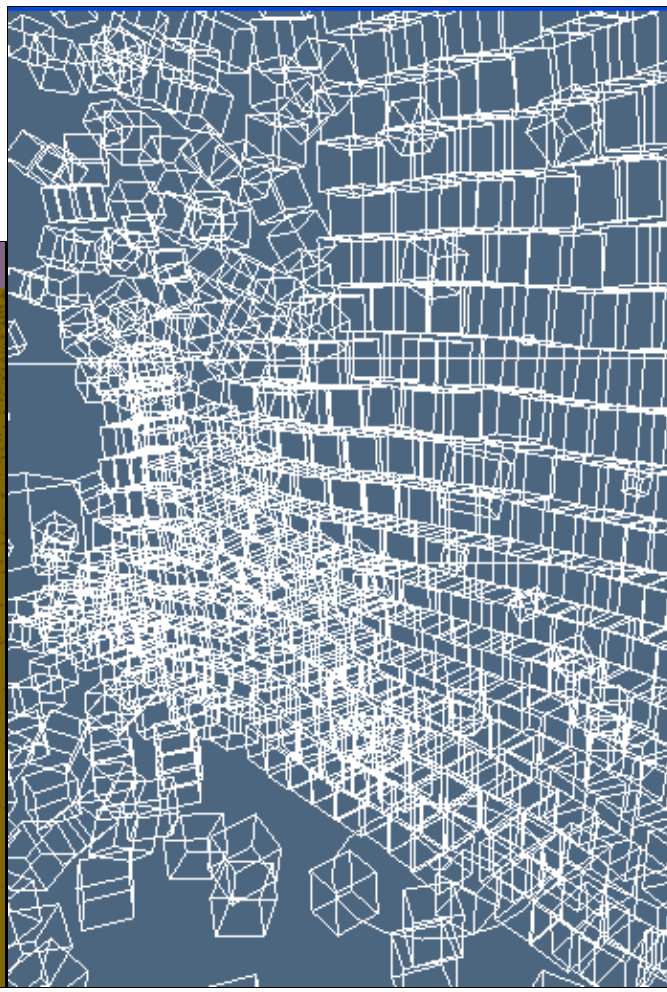
Fluids

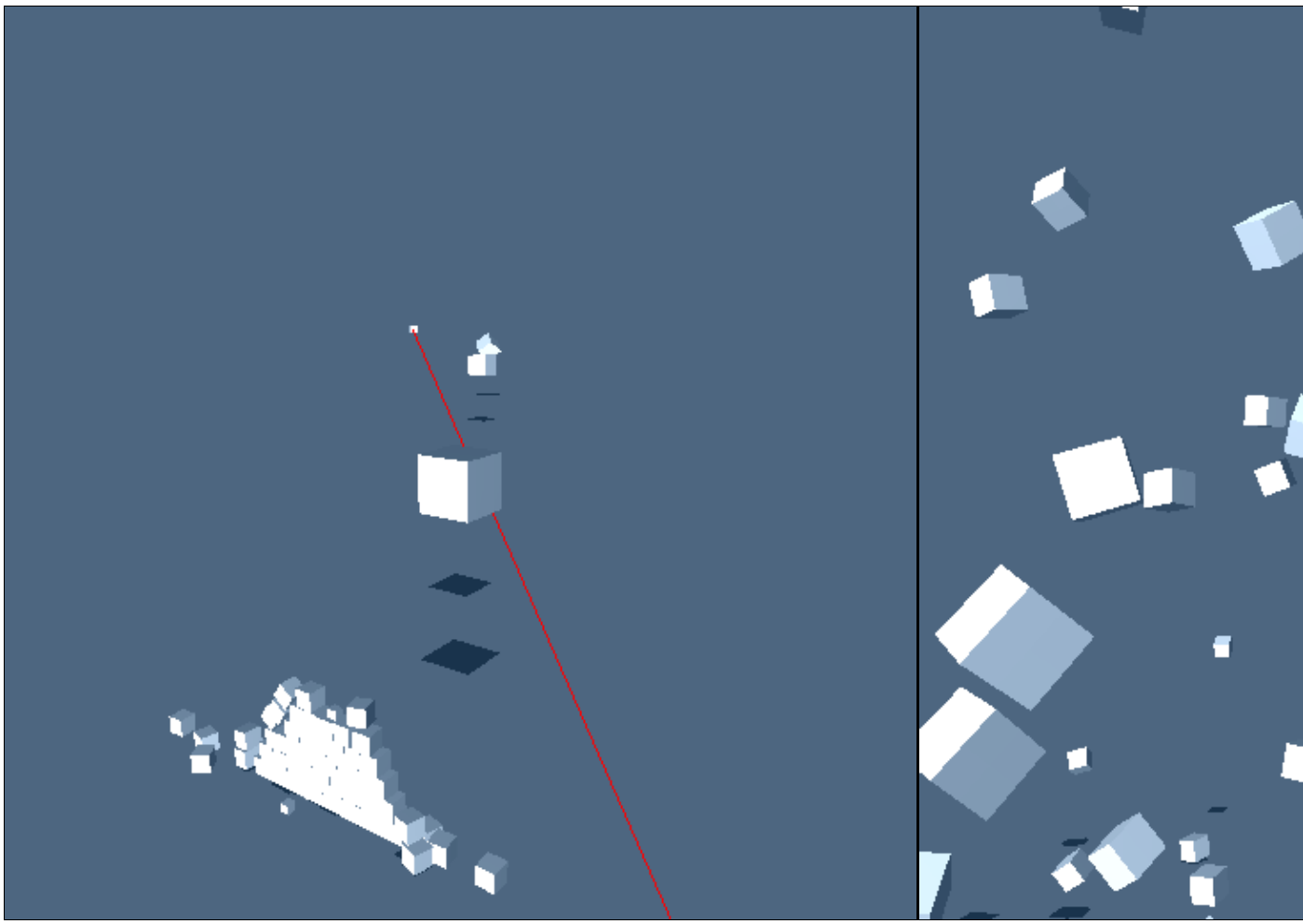
- Particle-Based Fluid Simulation for Interactive Applications, by Matthias Muller, David Charypar and Markus Gross

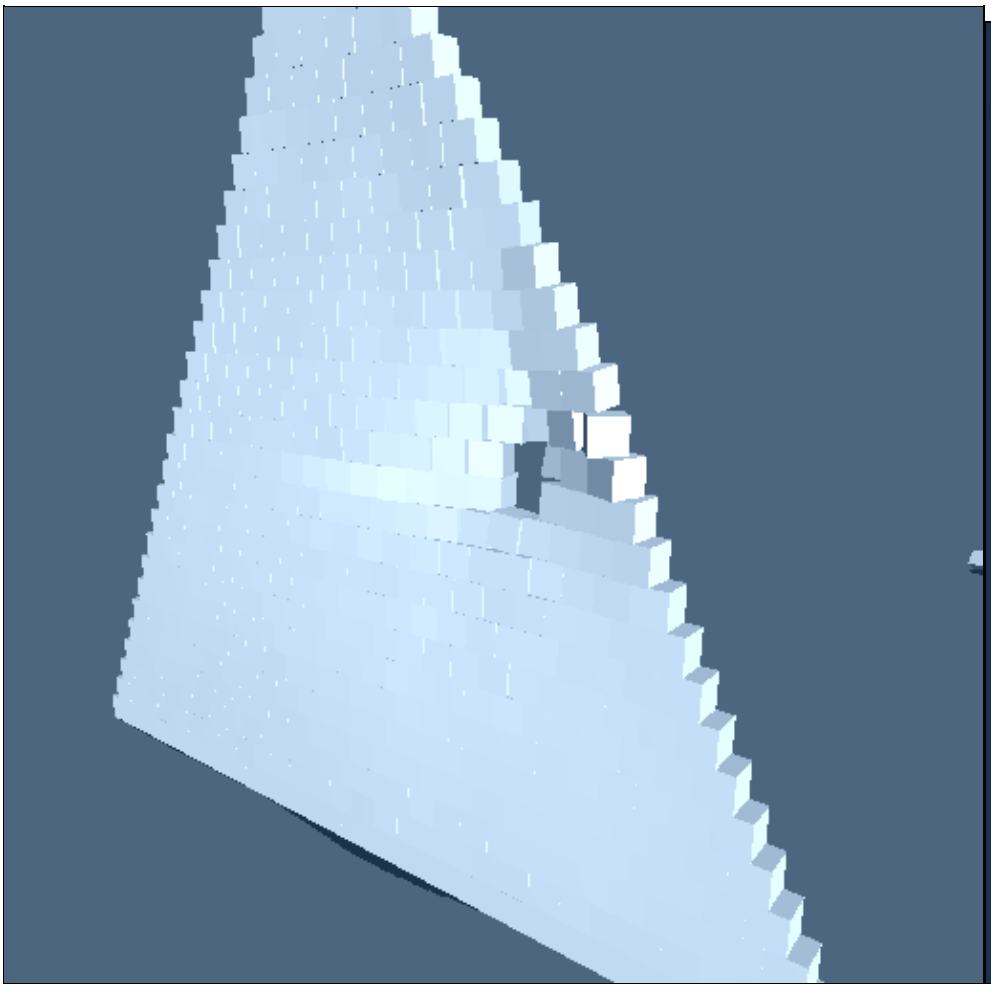
Tutorials and Samples

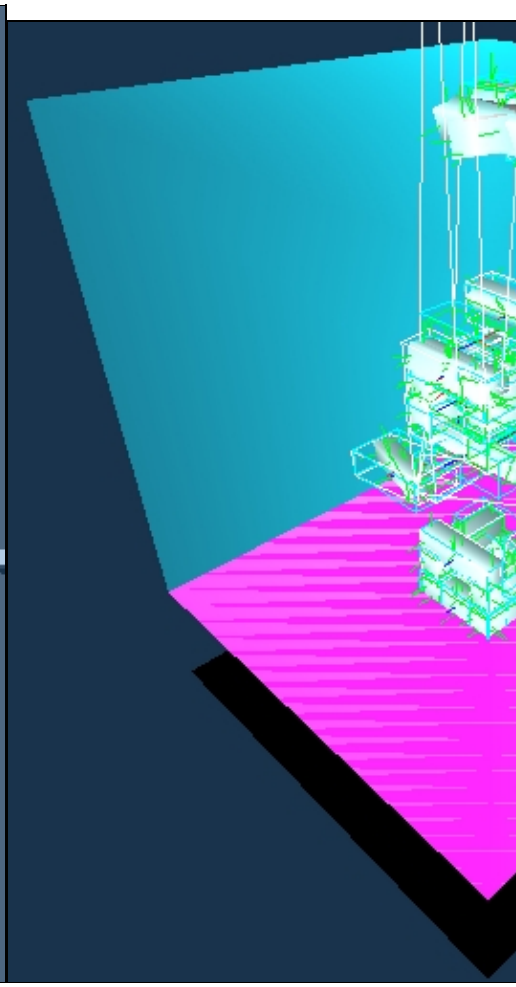


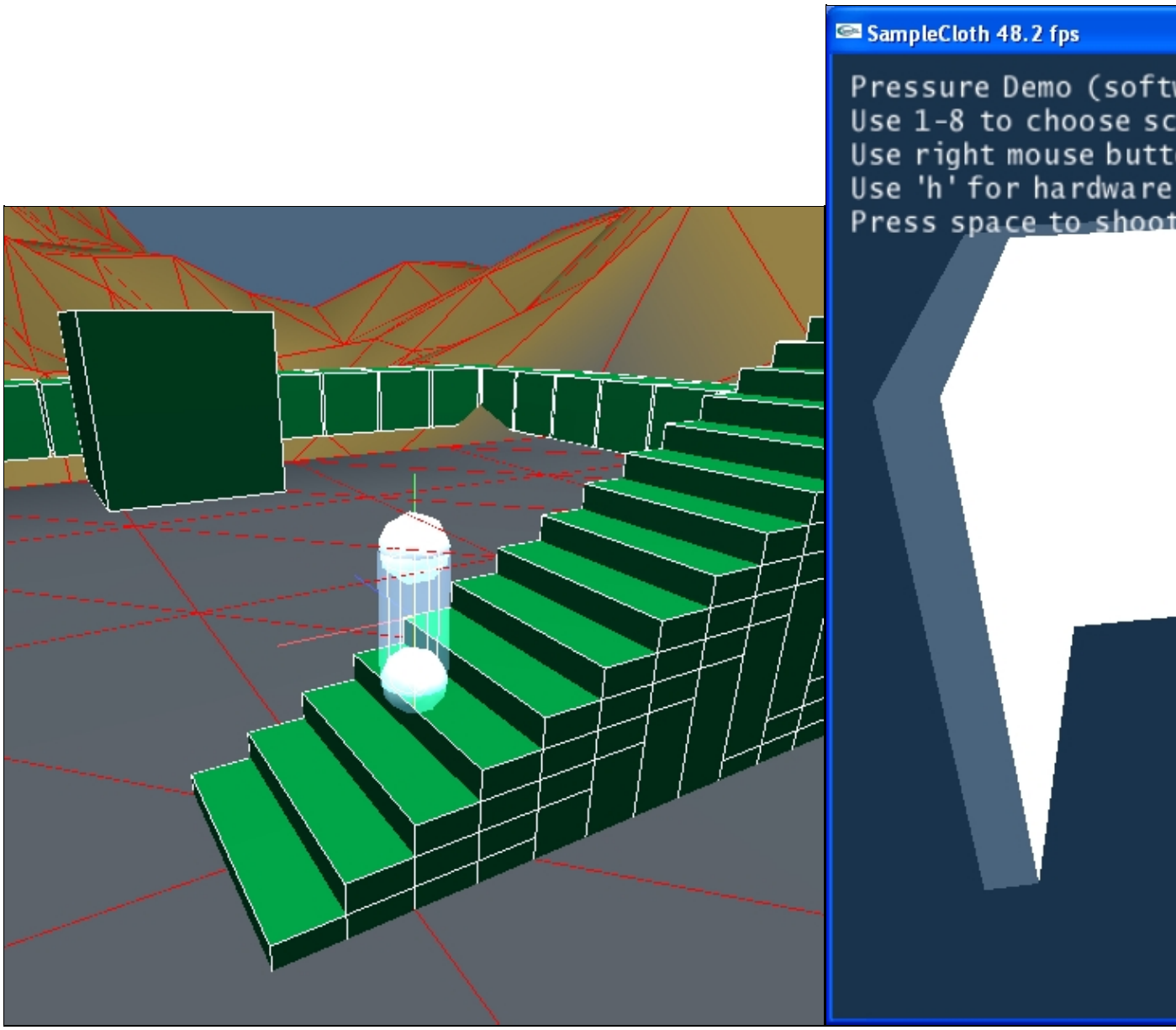


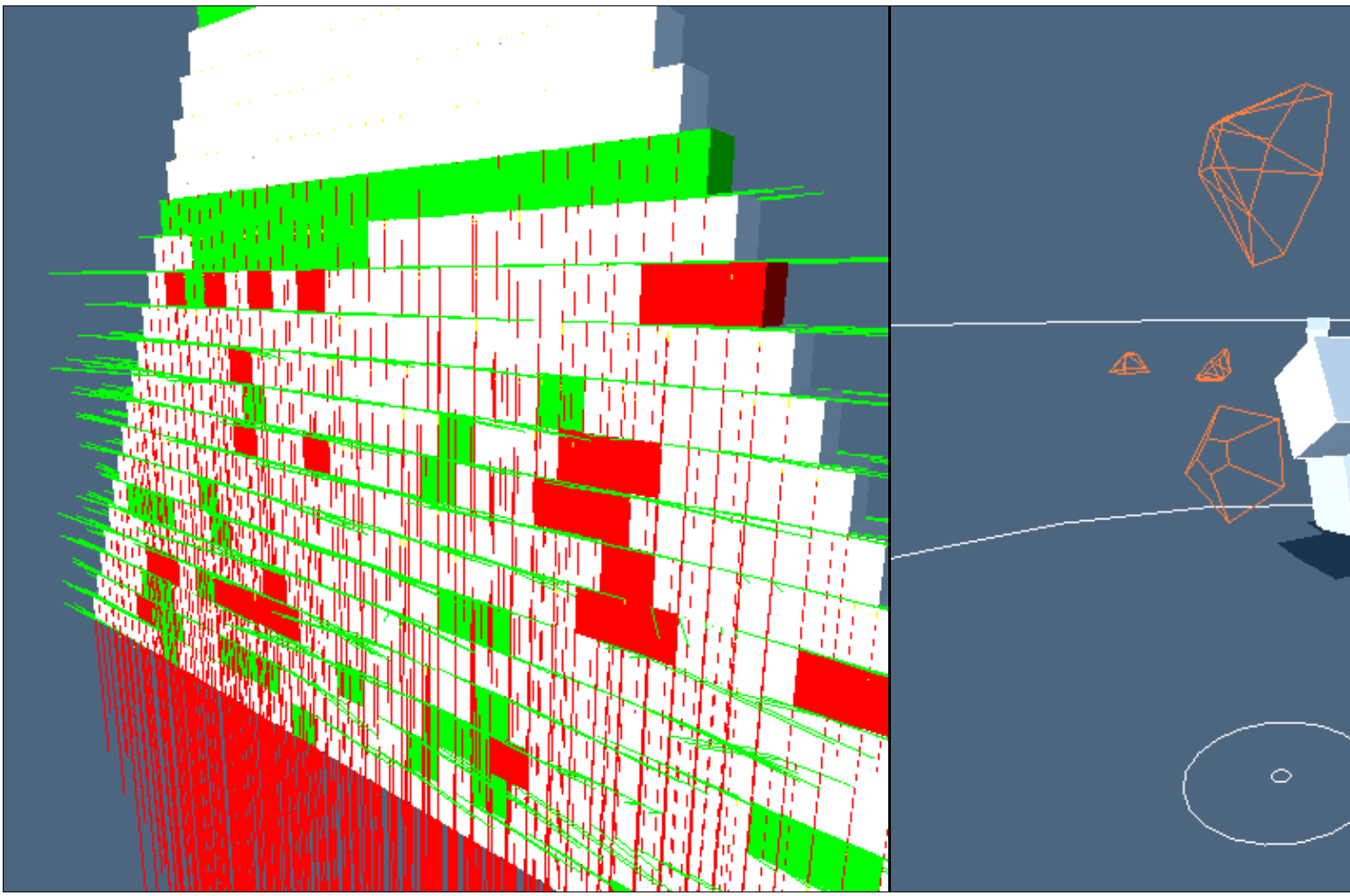


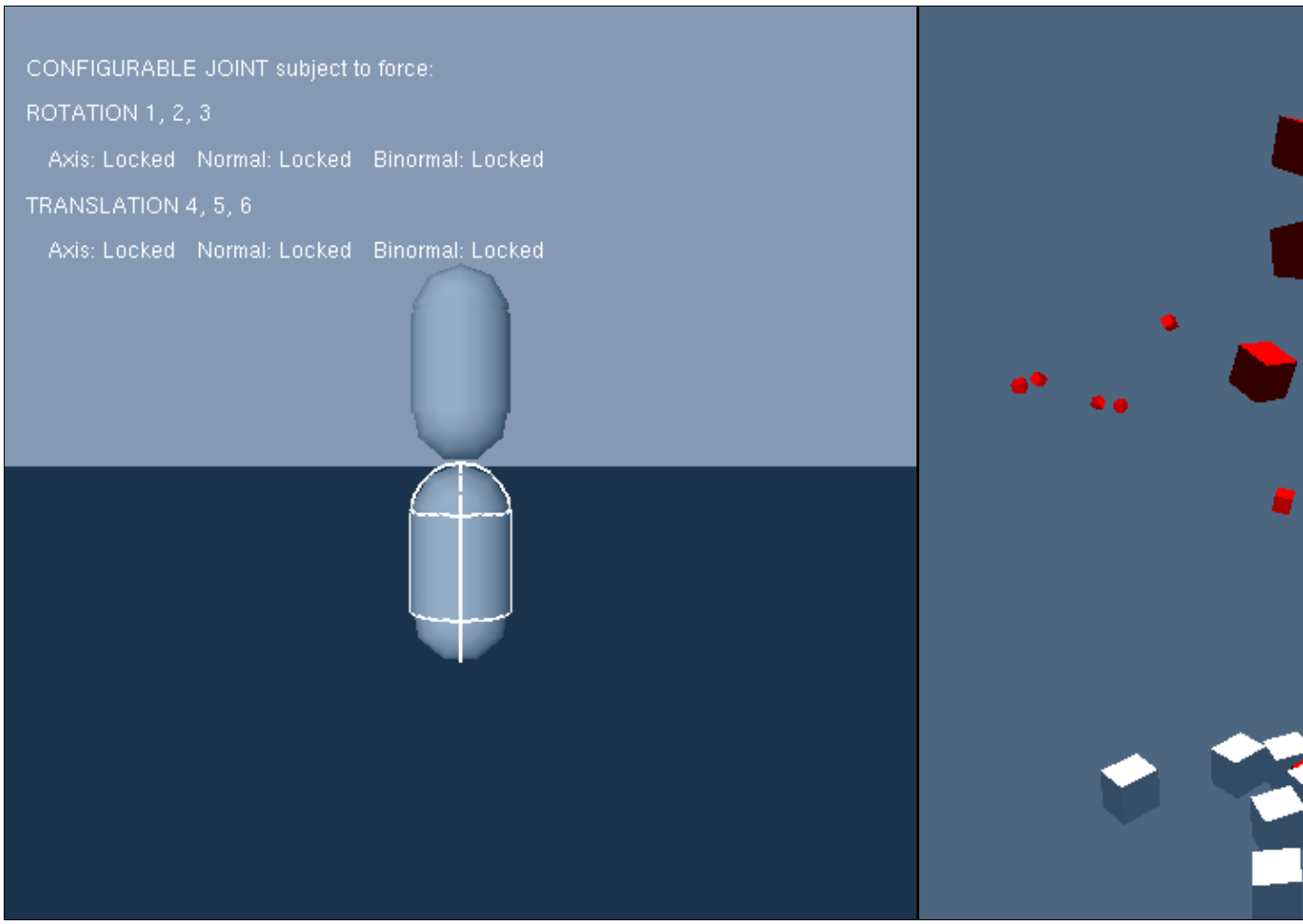


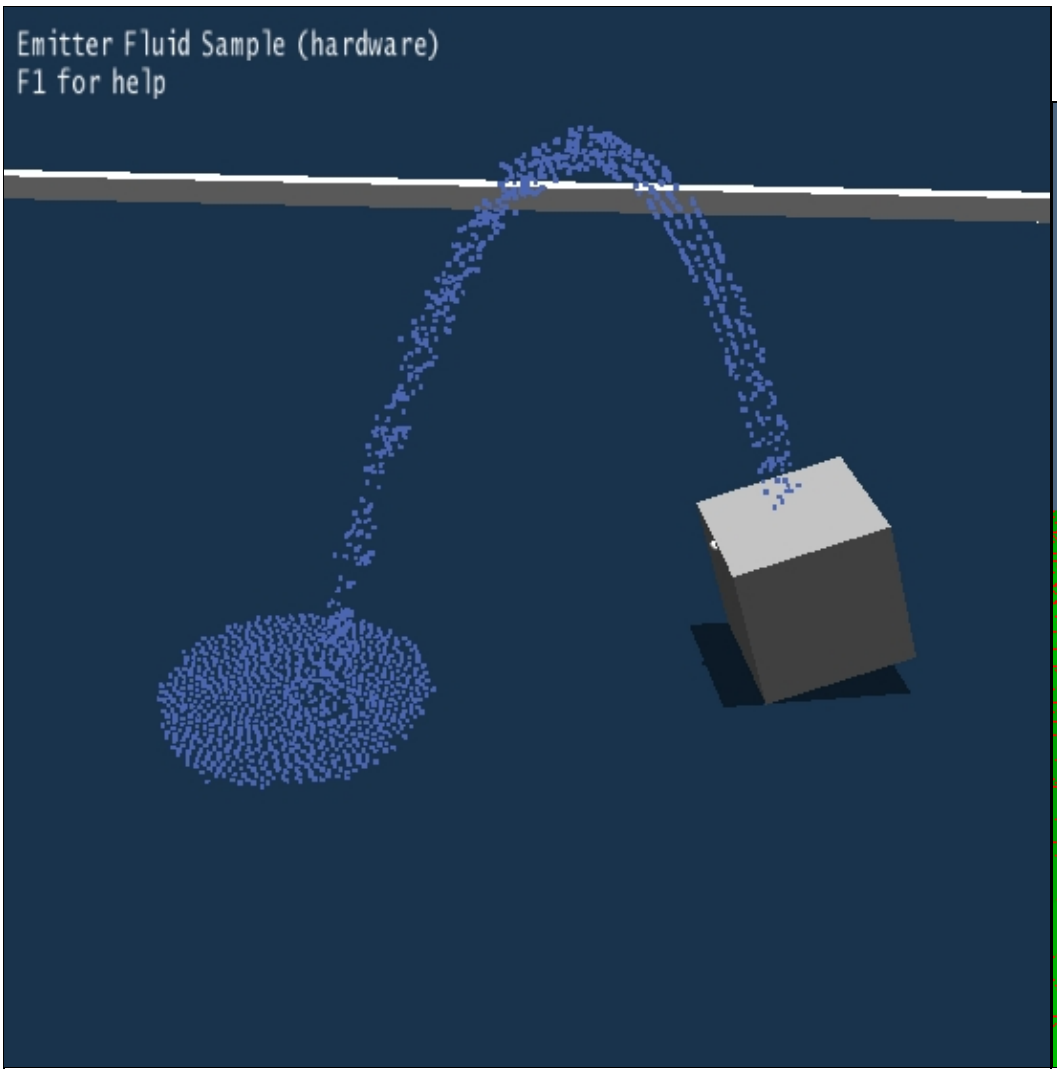


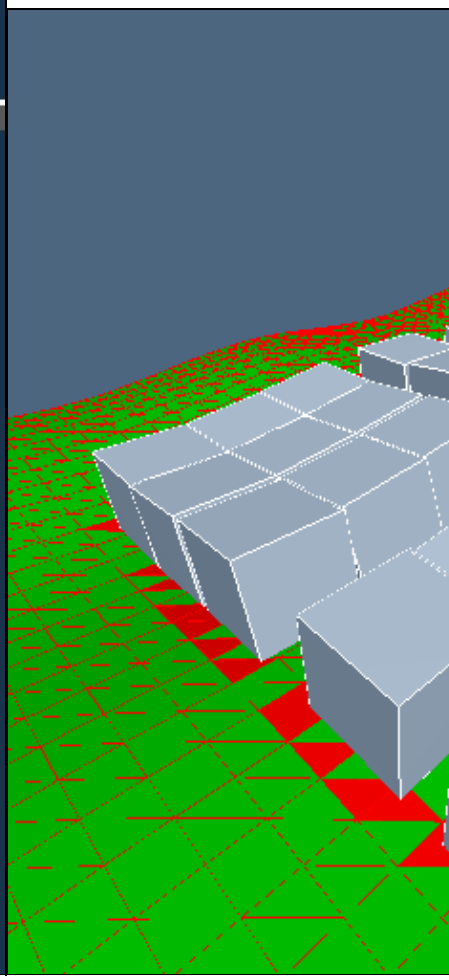


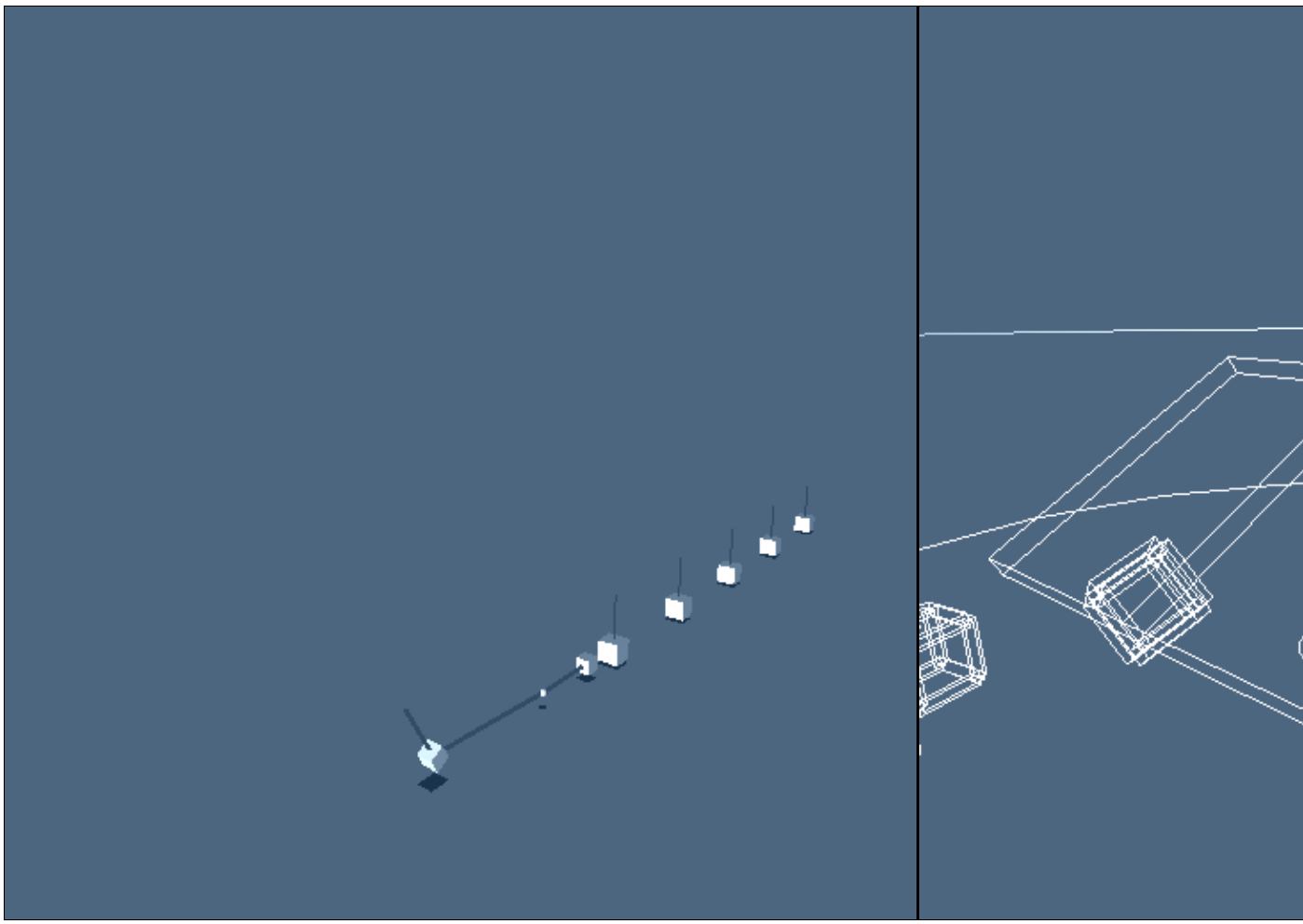

SampleCloth 48.2 fps
Pressure Demo (softu
Use 1-8 to choose sc
Use right mouse butt
Use 'h' for hardware
Press space to shoot

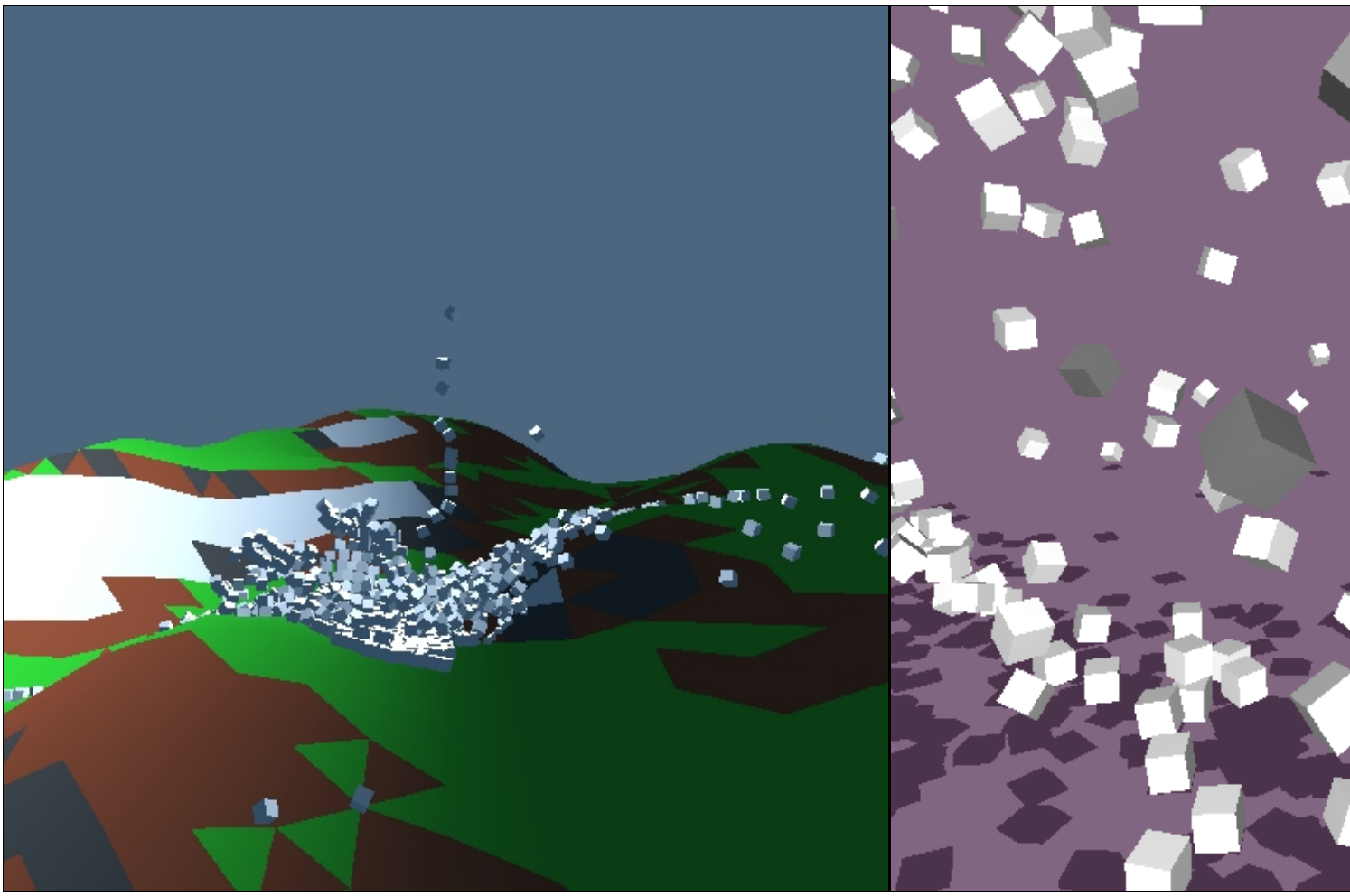


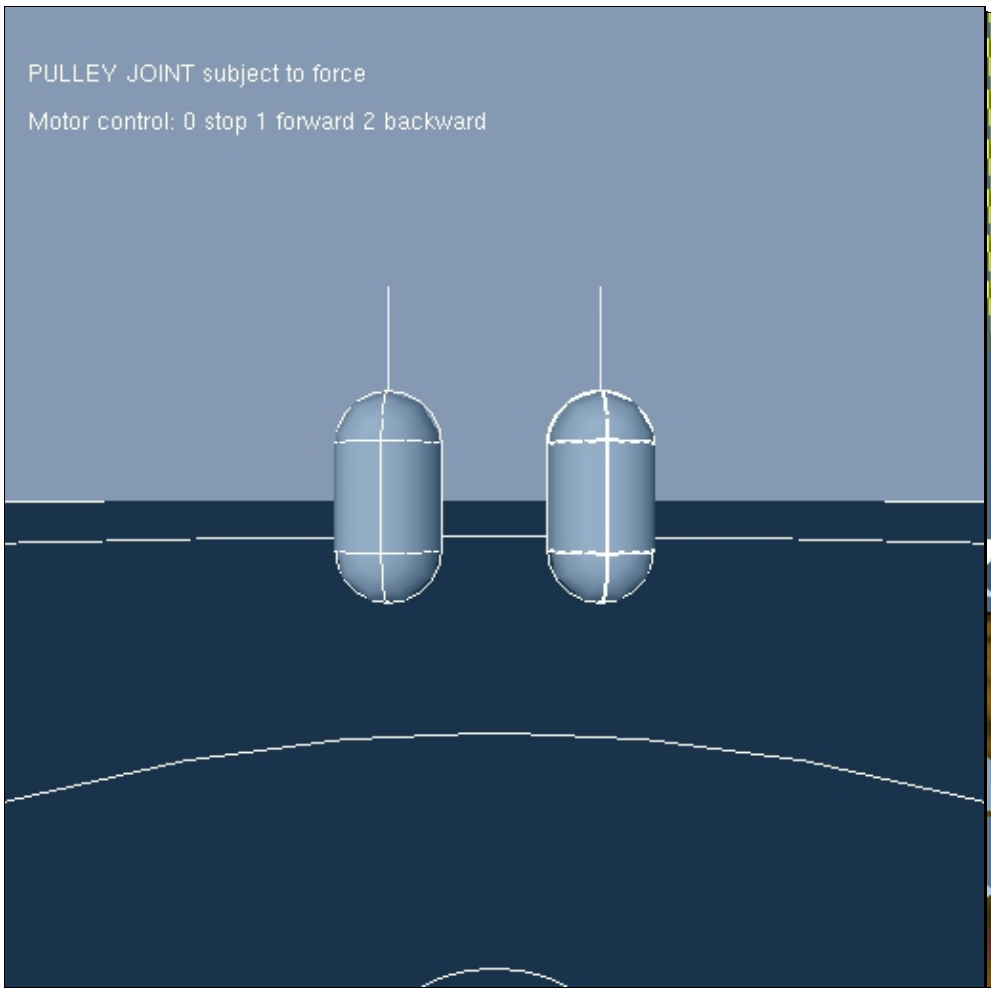

CONFIGURABLE JOINT subject to force:
ROTATION 1, 2, 3
Axis: Locked Normal: Locked Binormal: Locked
TRANSLATION 4, 5, 6
Axis: Locked Normal: Locked Binormal: Locked

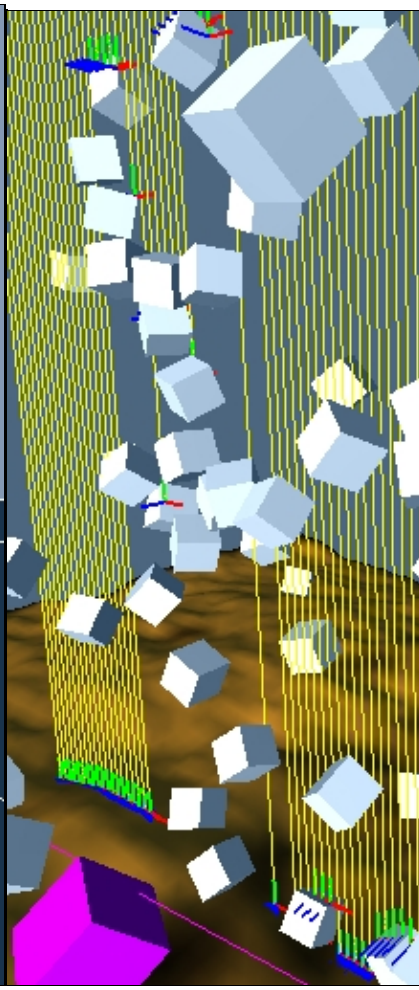

Emitter Fluid Sample (hardware)
F1 for help

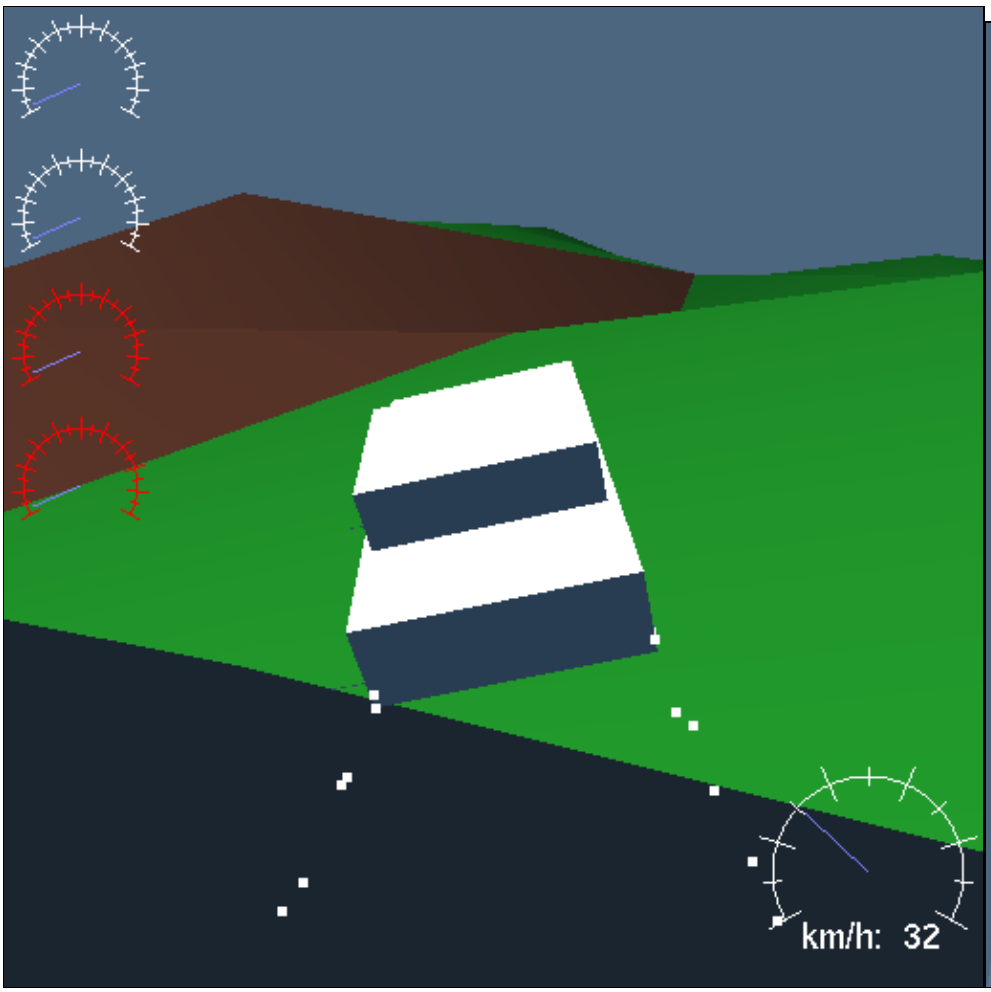


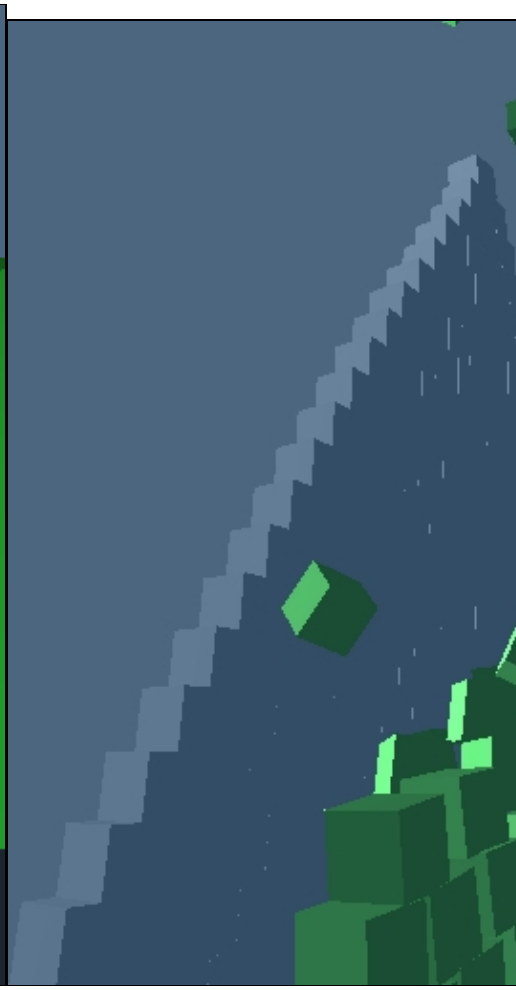


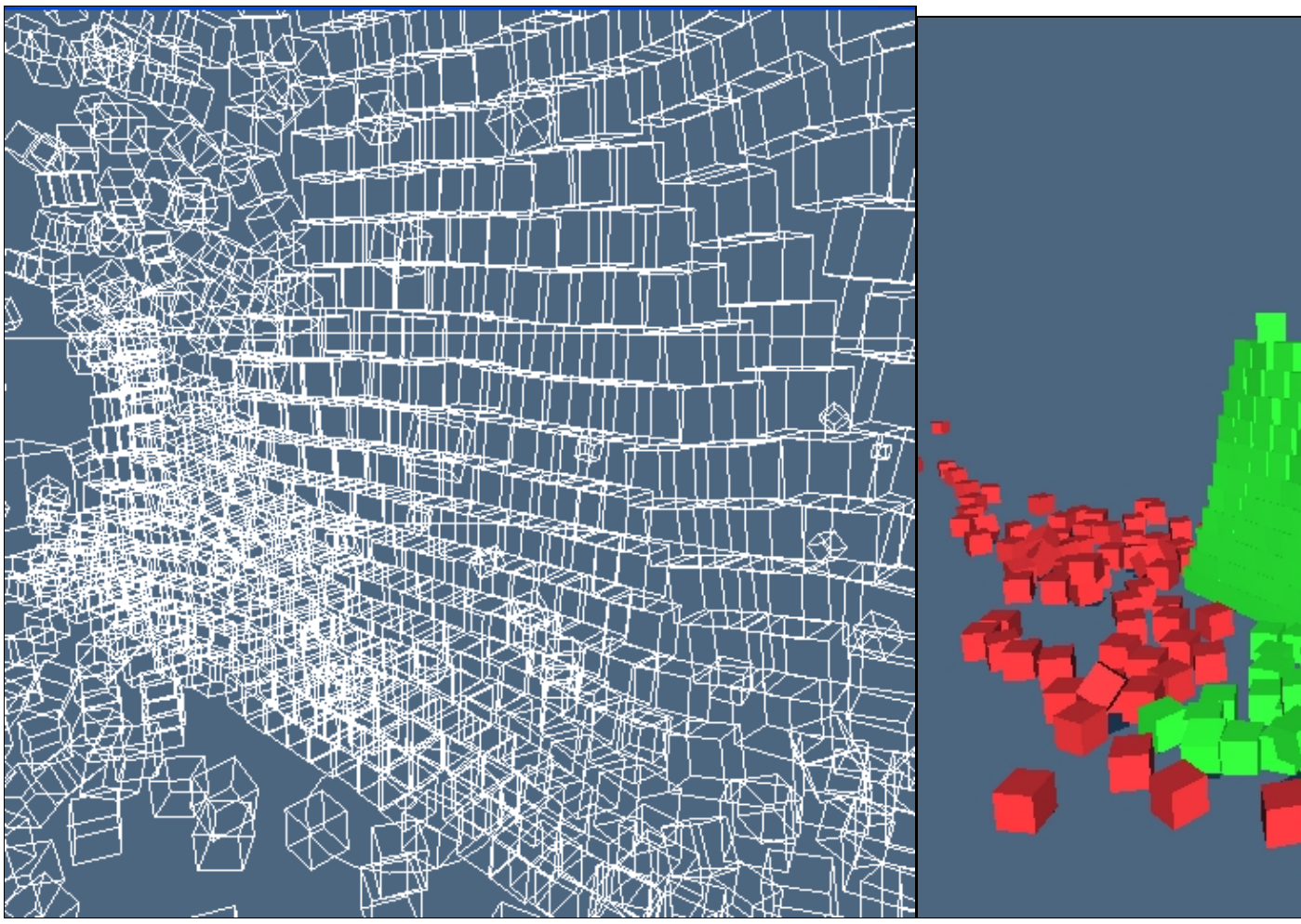


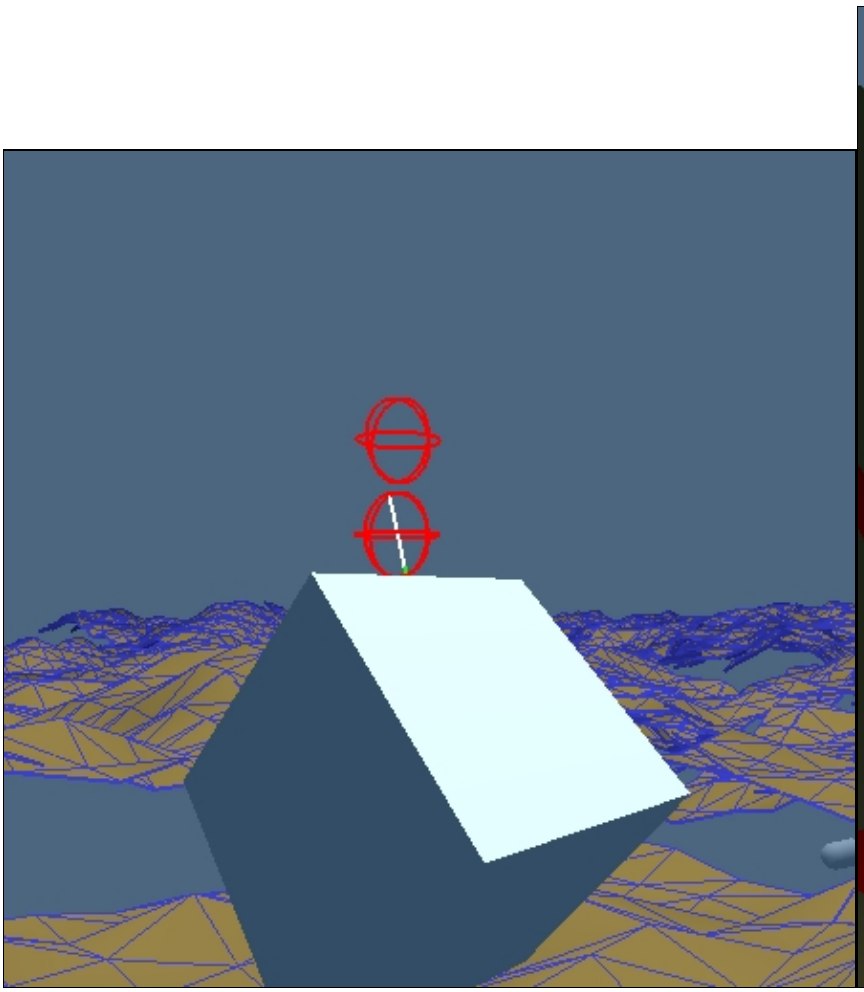

PULLEY JOINT subject to force
Motor control: 0 stop 1 forward 2 backward

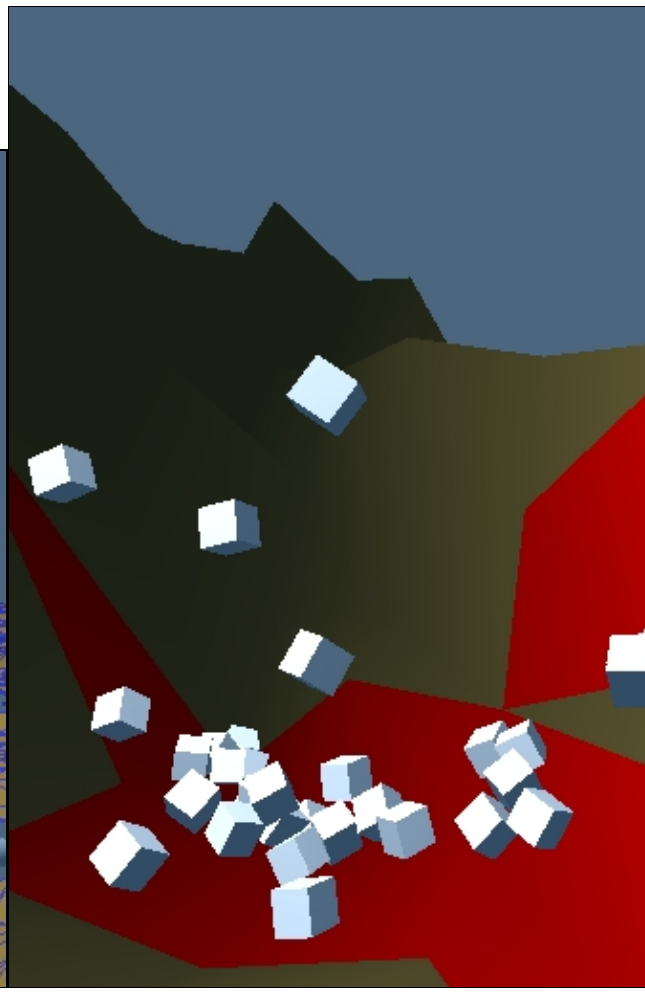


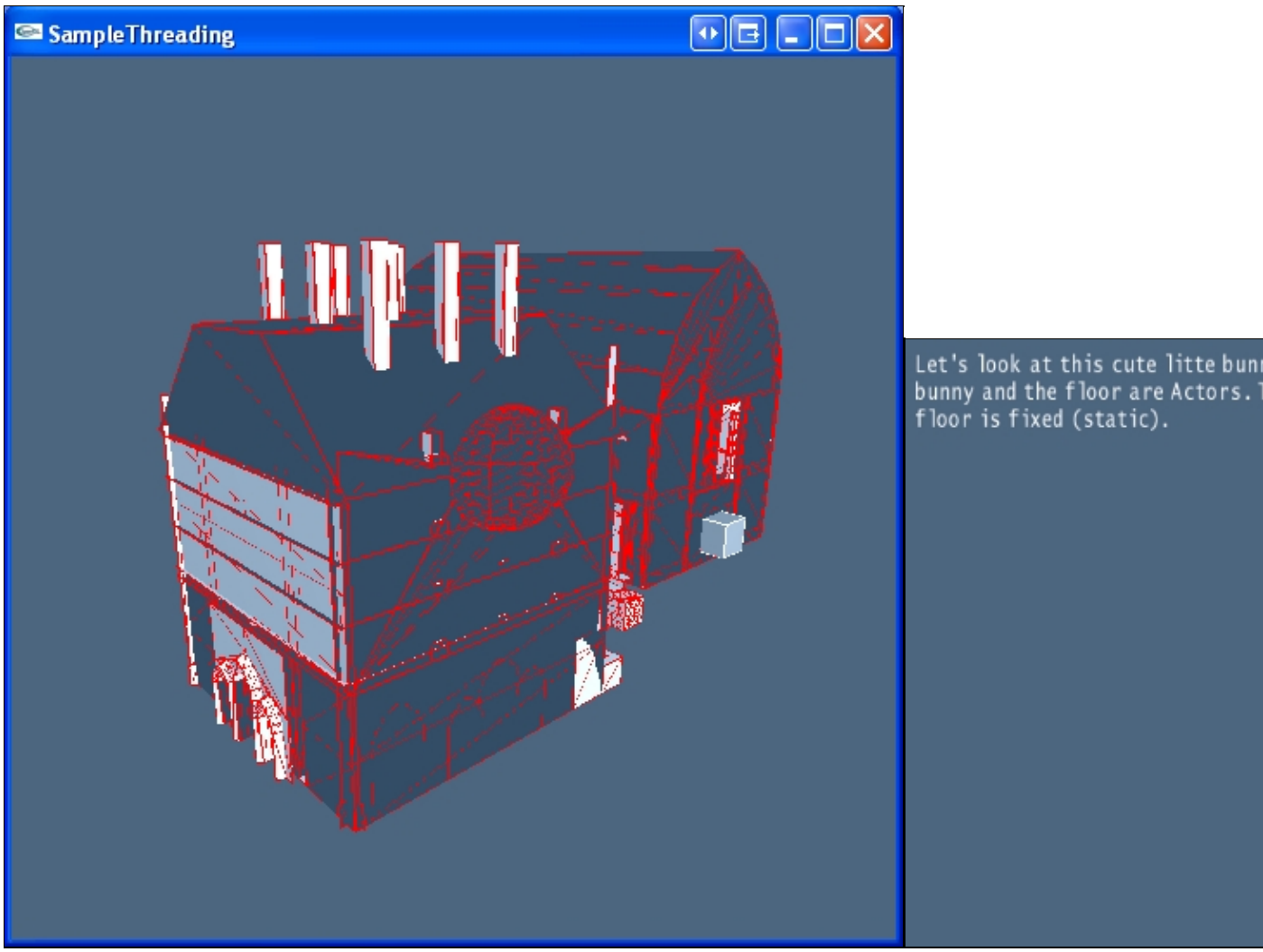

km/h: 32

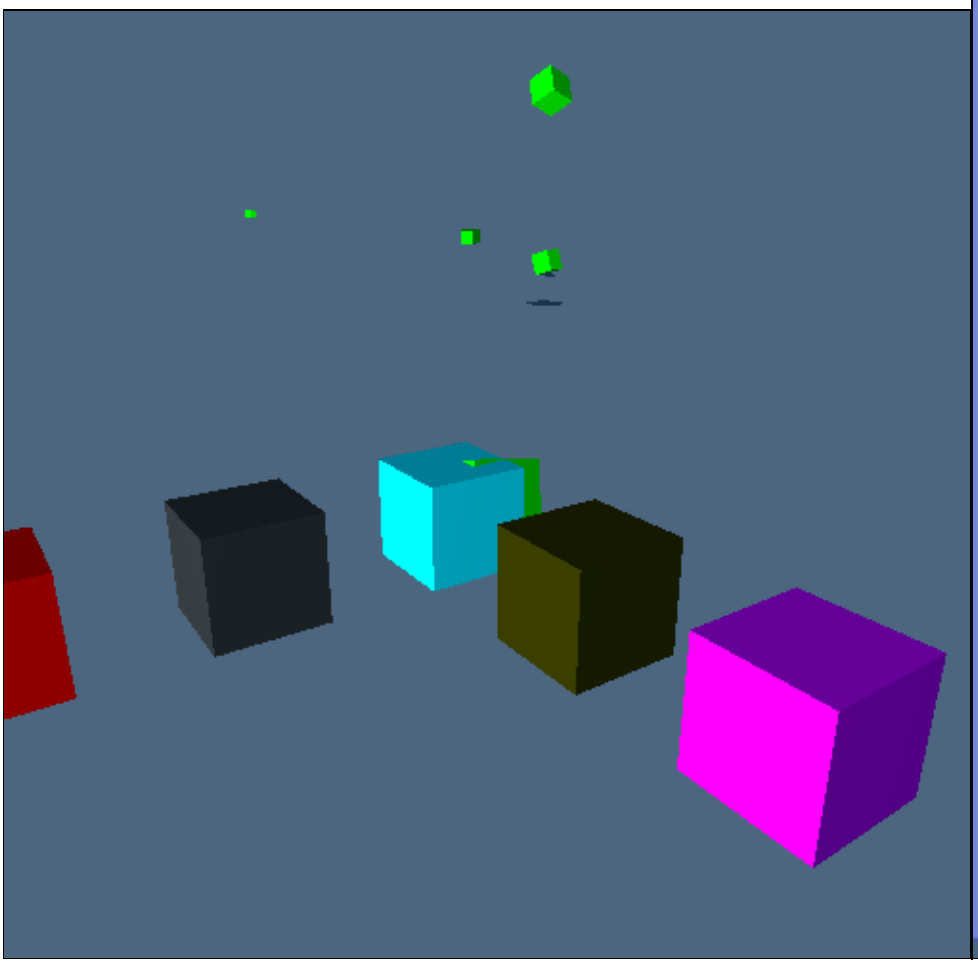


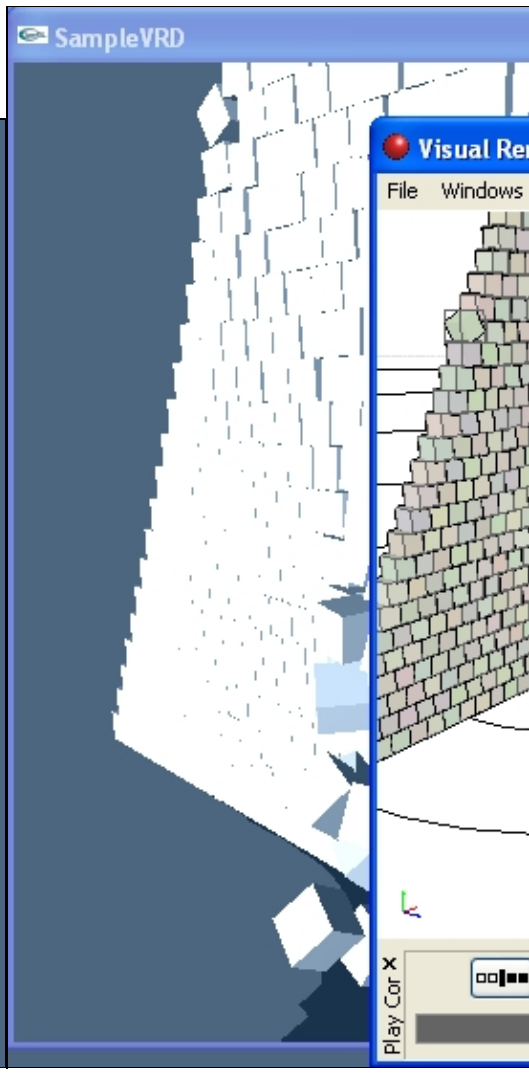


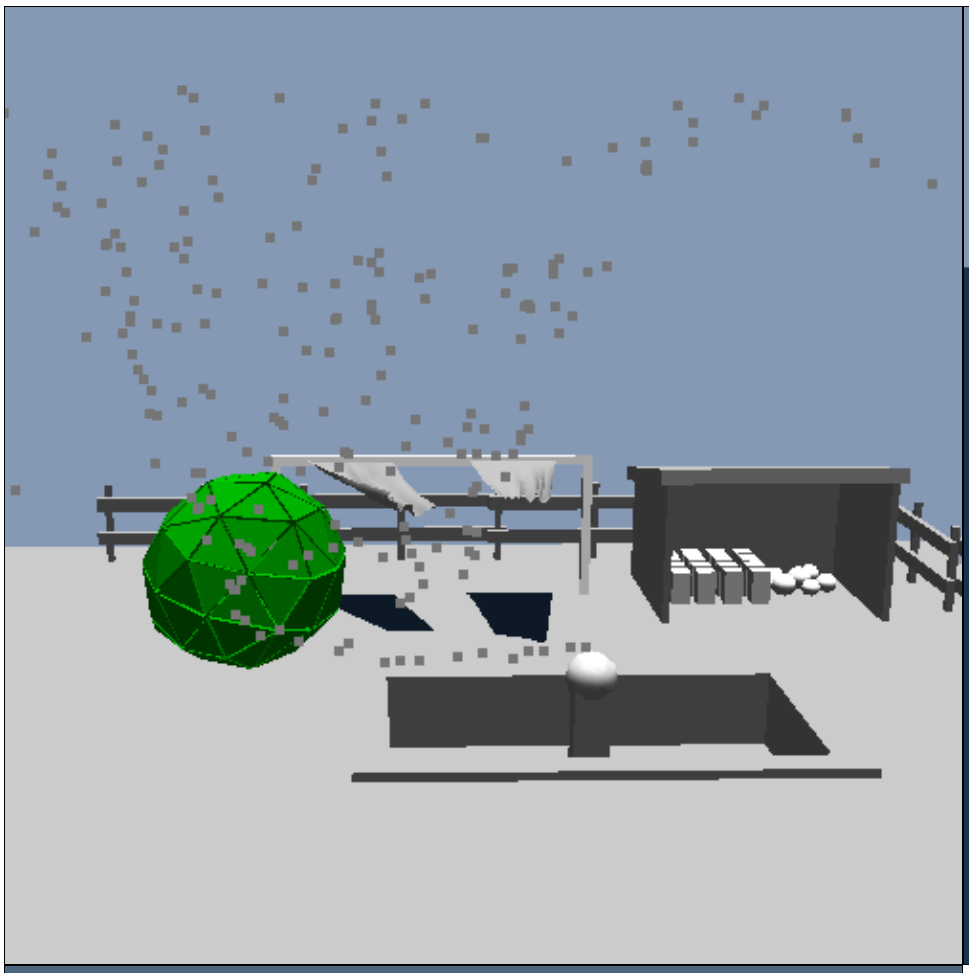


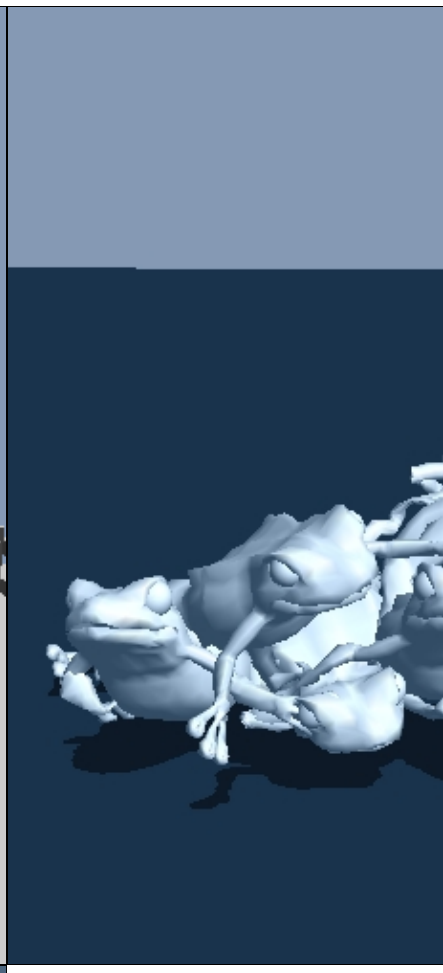



SampleThreading
Let's look at this cute litte bunn
bunny and the floor are Actors.
floor is fixed (static).

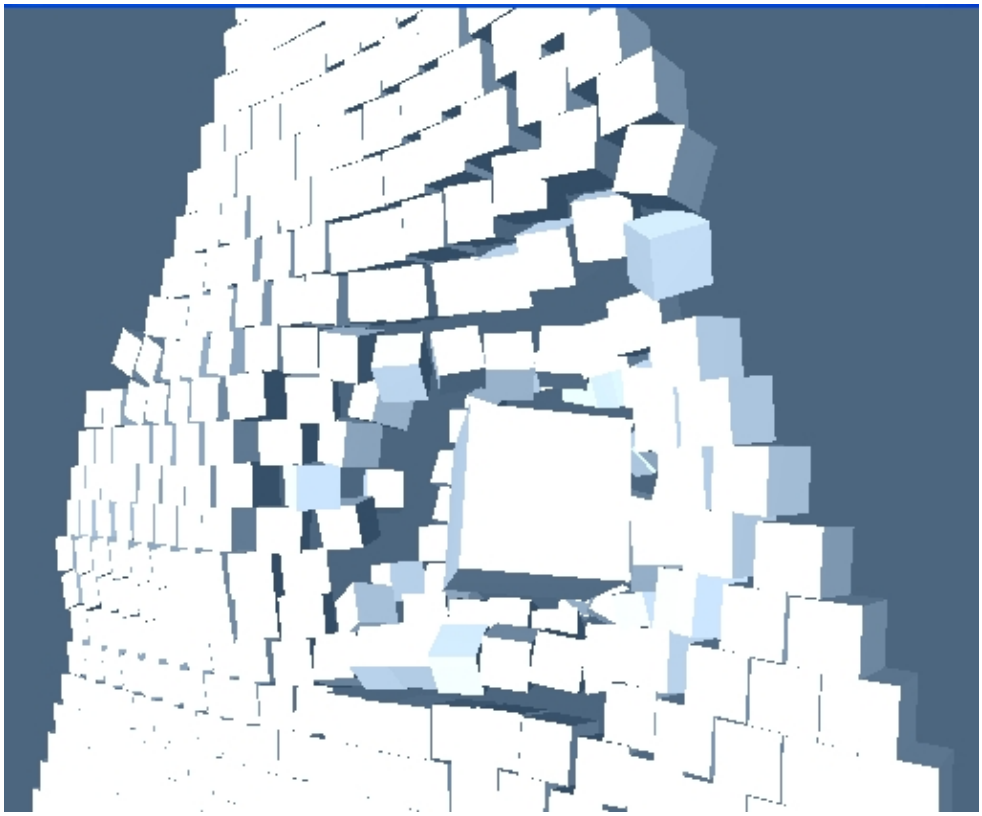



SampleVRD
File
Windows
Play Cor





Sample Active Transforms



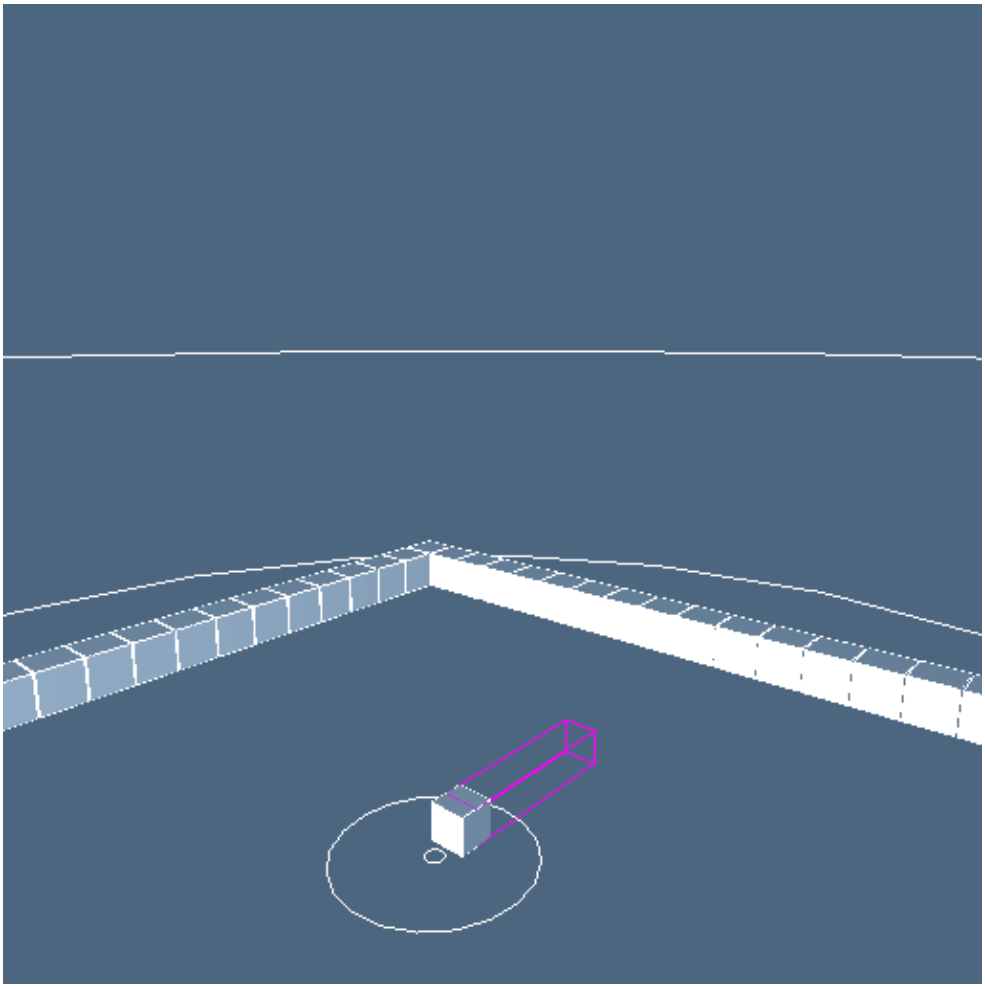
Overview

Demonstrate the use of Active Transform Notification.
An array of custom "GameObject" objects is kept up-to-date using ATN.

Path

Source:	(SDK Path)\Samples\SampleActiveTransforms\src
Executable:	(SDKPath)\Bin\...

Sample AI Sensor



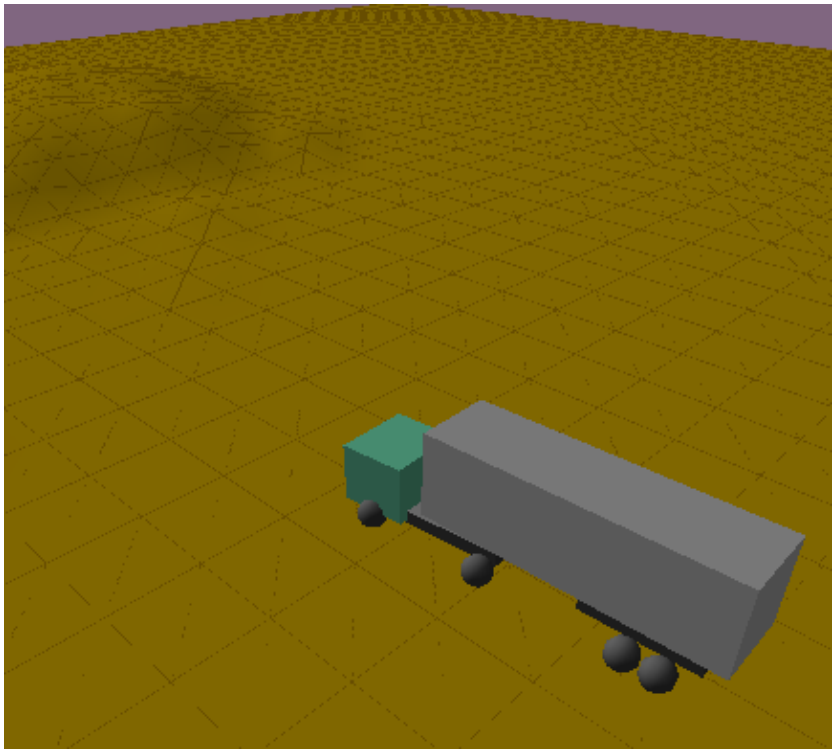
Overview

Demonstrate the use of triggers as sensors for use with AI.

Path

Source:	(SDK Path)\Samples\SampleAISensor\src
Executable:	(SDKPath)\Bin\...

Sample Articulate Truck



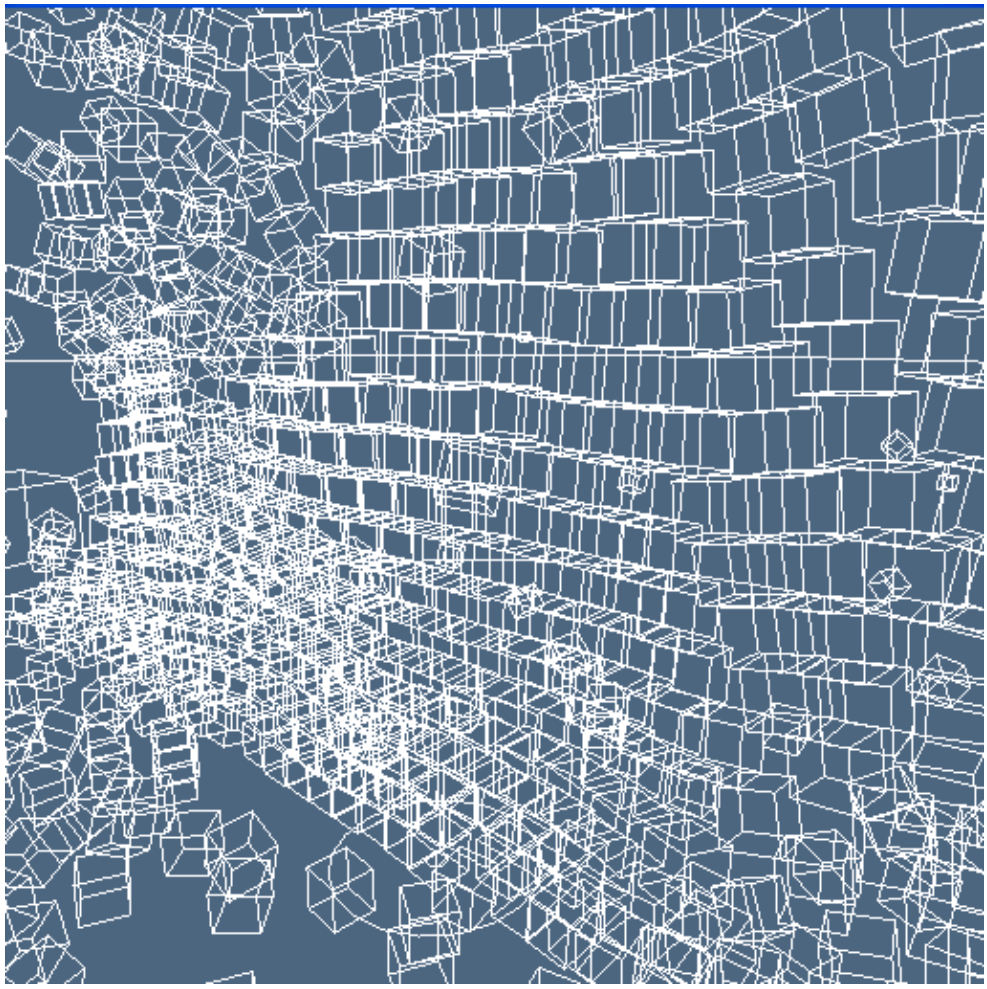
Overview

Demonstrate a vehicle constructed with joints.

Path

Source:	(SDK Path)\Samples\SampleArticulateTruck\src
Executable:	(SDKPath)\Bin\...

Sample Asset Export



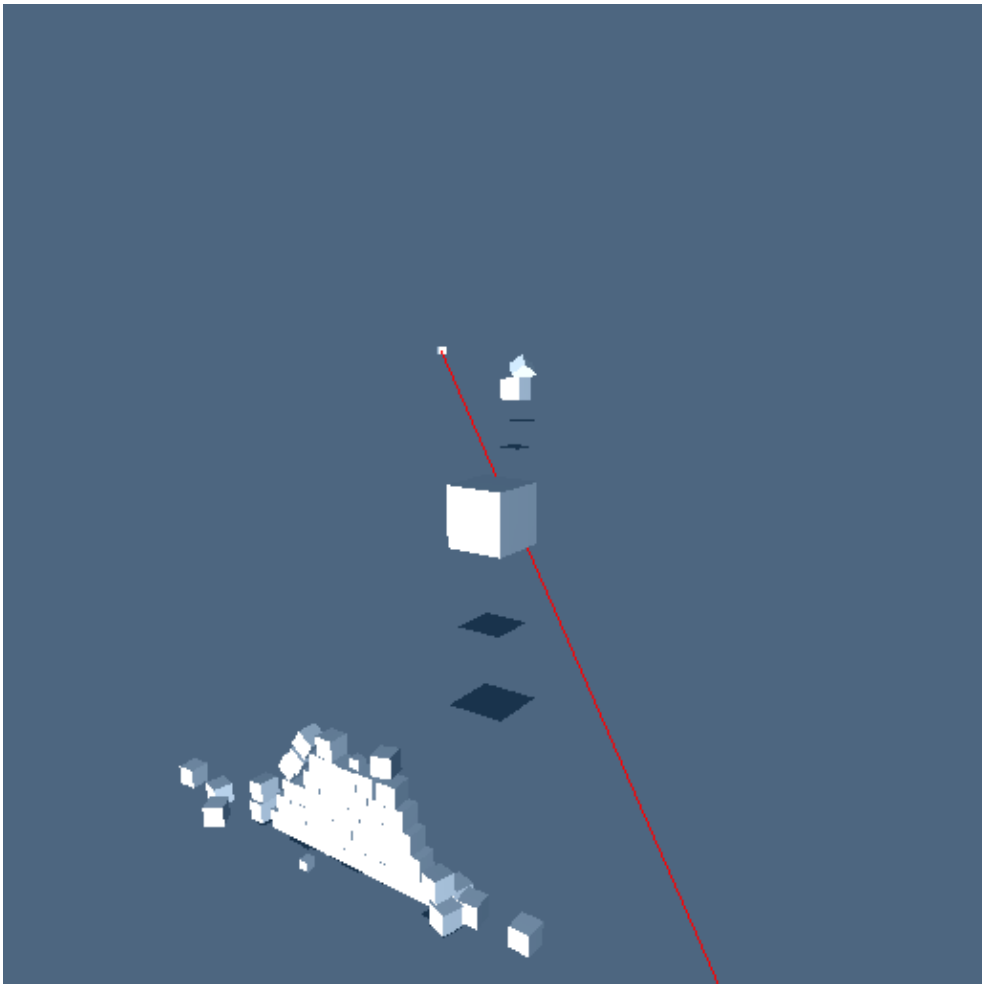
Overview

Demonstrate exporting objects to a custom file format and reading them back in.

Path

Source:	(SDK Path)\Samples\SampleAssetExport\src
Executable:	(SDKPath)\Bin\...

Sample Async Boxes



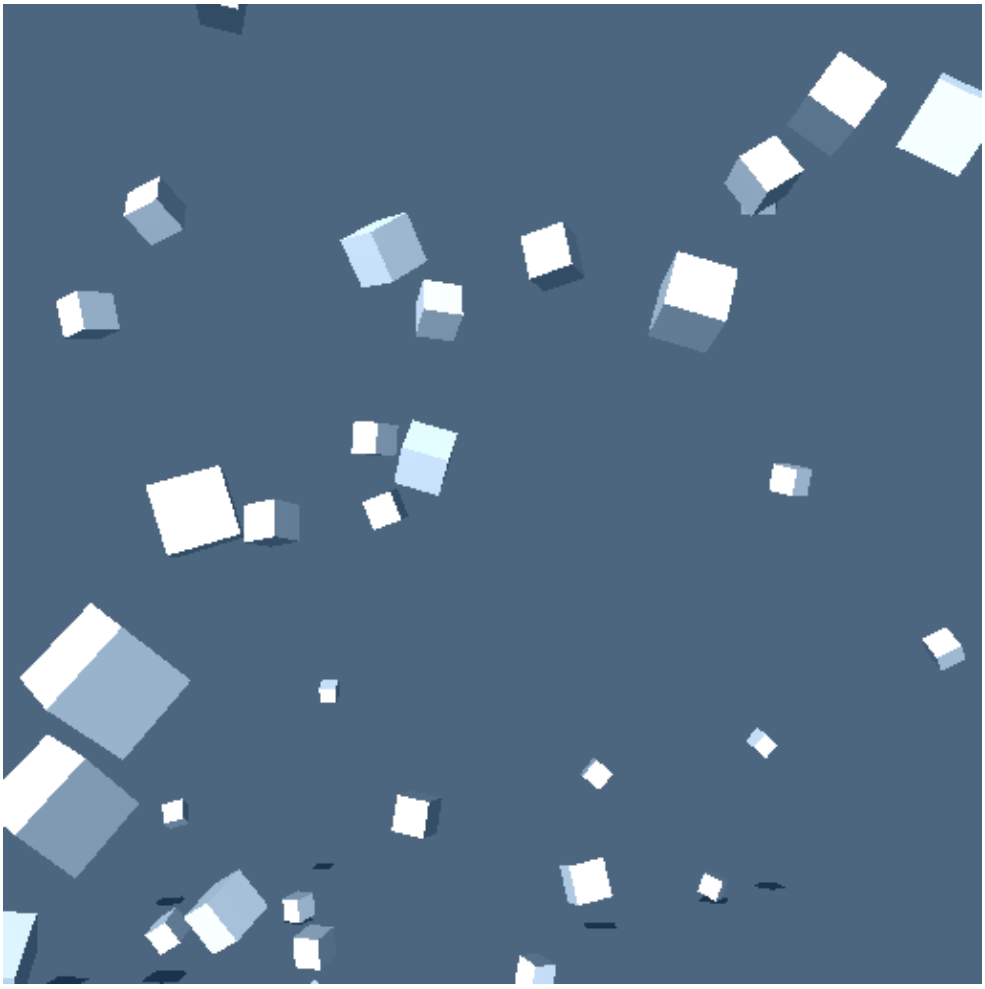
Overview

Simple demonstration of writing an async game loop.

Path

Source:	(SDK Path)\Samples\SampleAsyncBoxes\src
Executable:	(SDKPath)\Bin\...

Sample Boxes



Overview

A minimal PhysX test application.

Path

Source:	(SDK Path)\Samples\SampleBoxes\src
Executable:	(SDKPath)\Bin\...

Sample CCD Dynamic

Overview

Demonstrate continuous collision detection by shooting a very fast object onto a stack of stationary objects.

Path

Source:	(SDK Path)\Samples\SampleCCDDynamic\src
Executable:	(SDKPath)\Bin\...

Sample CCD Explosion

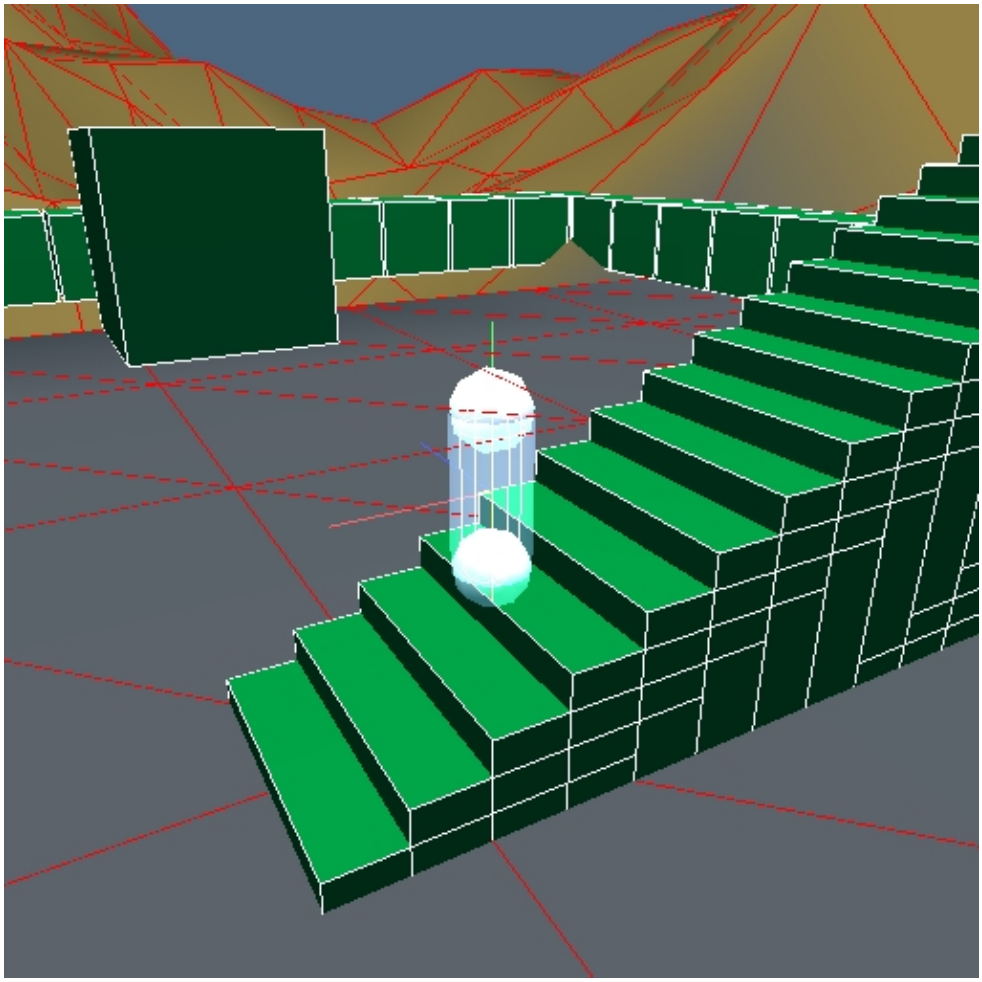
Overview

Sample showing an exploding stack of blocks, with continuous collision detection applied.

Path

Source:	(SDK Path)\Samples\SampleCCDExplosion\src
Executable:	(SDKPath)\Bin\...

Sample Character Controller



Overview

Sample showing how to setup a character controller.

Path

Source:	(SDK Path)\Samples\SampleCharacterController\src
Executable:	(SDKPath)\Bin\...

Sample Cloth

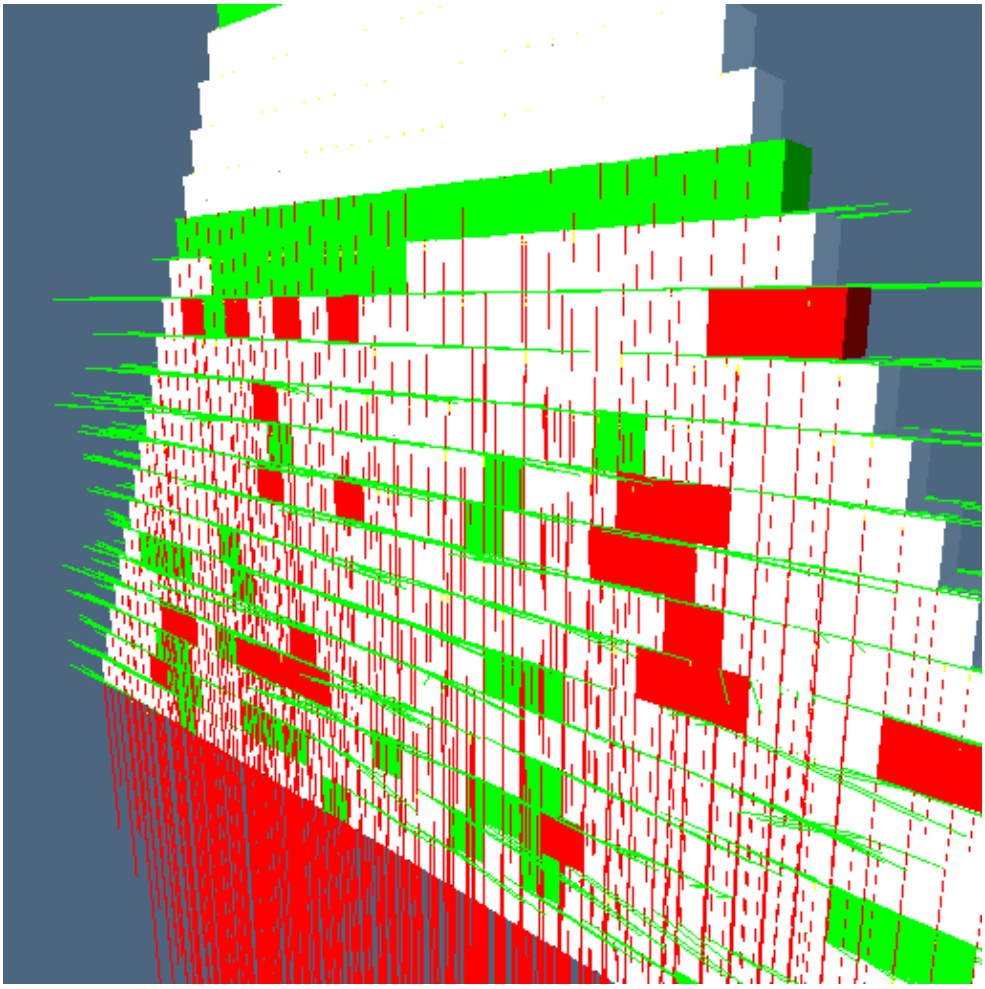
Overview

Several examples of the use of cloth, including tearing and pressurized cloth.

Path

Source:	(SDK Path)\Samples\SampleCloth\src
Executable:	(SDKPath)\Bin\...

Sample Contact Stream Iterator



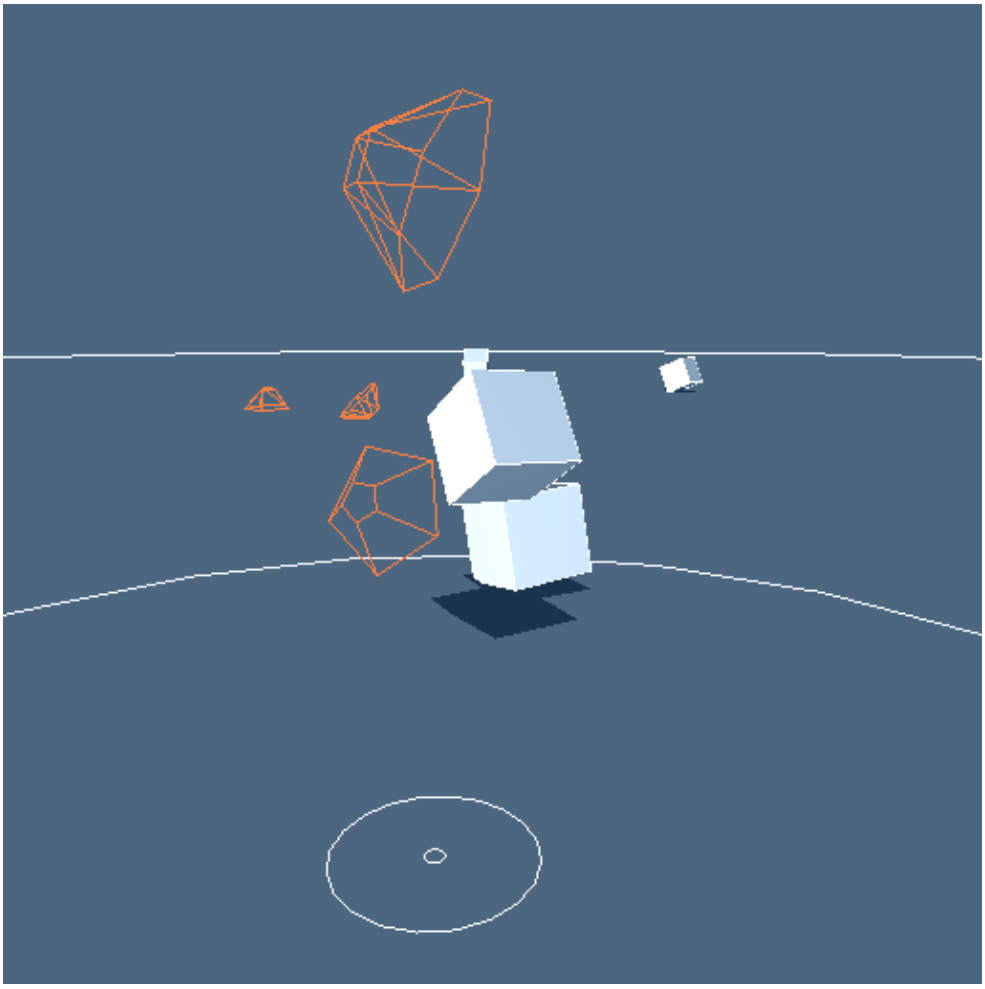
Overview

Sample showing how to create a contact stream iterator and extract the contact information.

Path

Source:	(SDK Path)\Samples\SampleContactStreamIterator\src
Executable:	(SDKPath)\Bin\...

Sample Convex



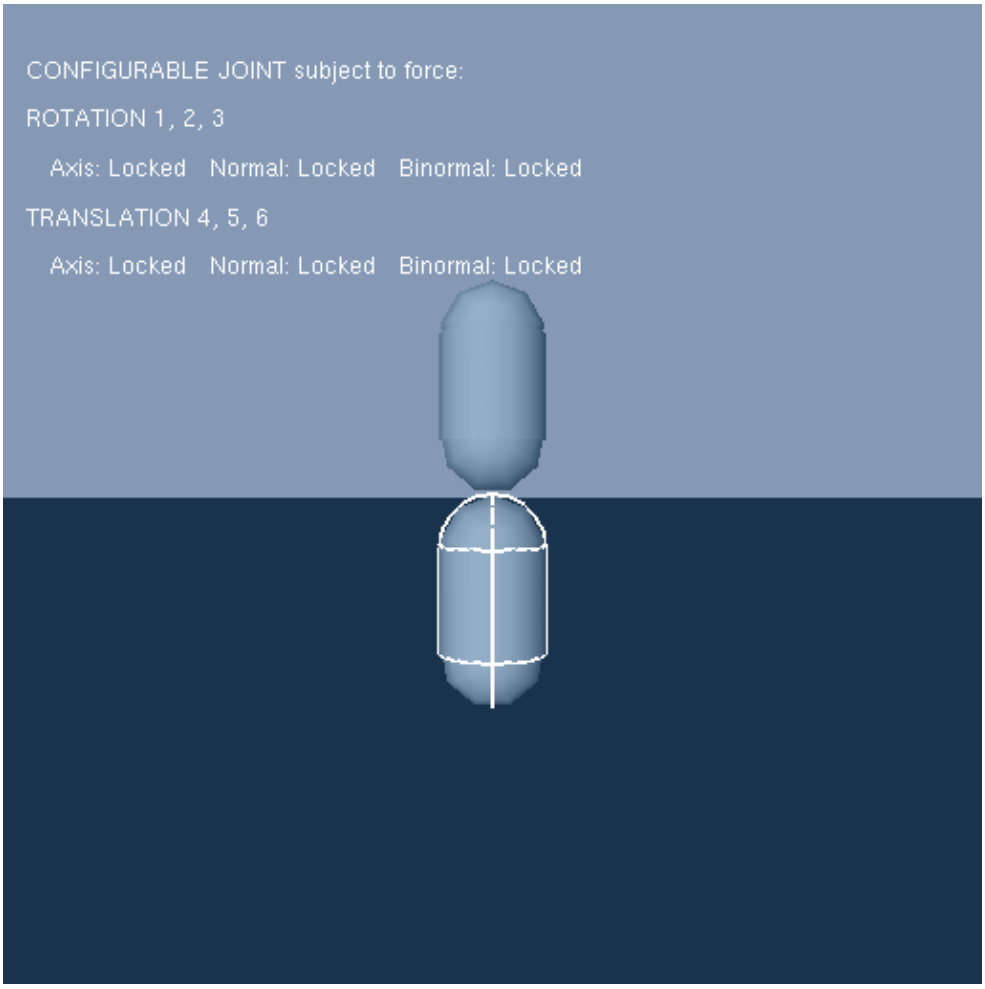
Overview

Sample showing how to create convex mesh objects.

Path

Source:	(SDK Path)\Samples\SampleConvex\src
Executable:	(SDKPath)\Bin\...

Sample D6 Joint



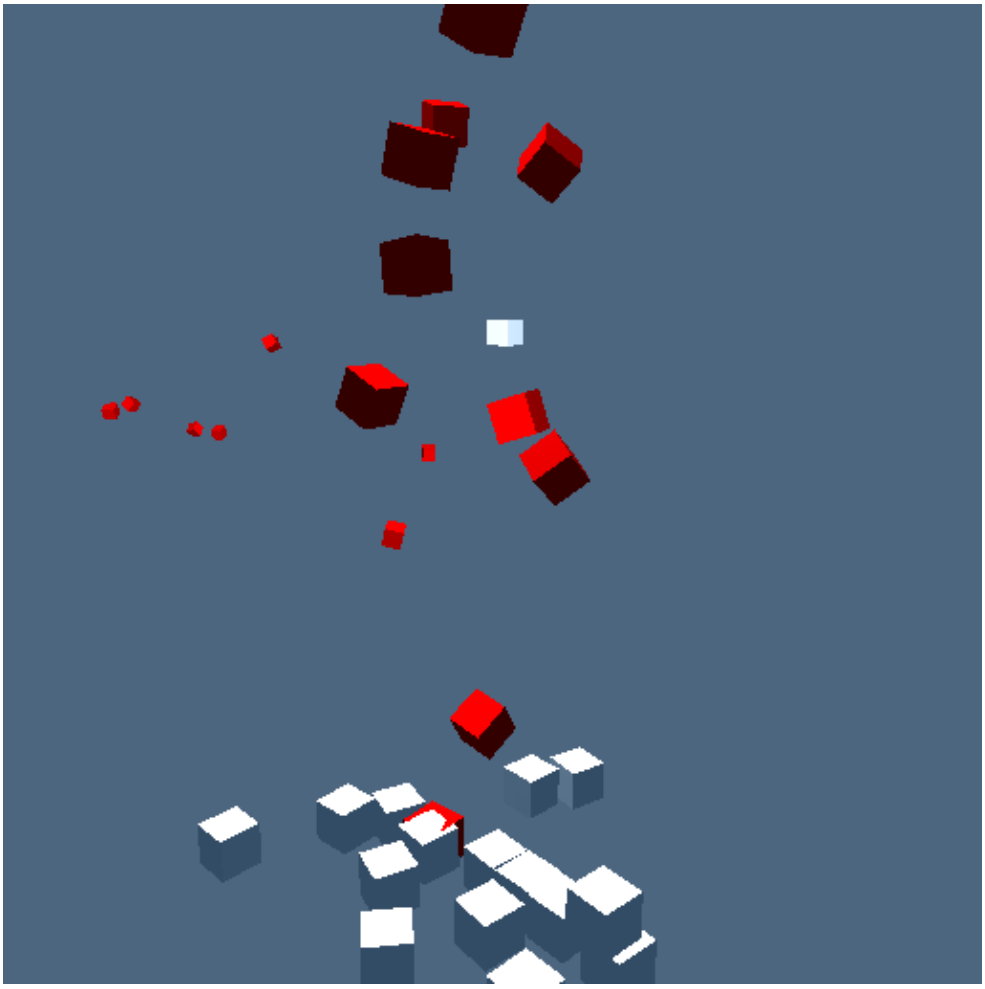
Overview

Sample showing how to setup a D6 joint and switch it from one joint type to another.

Path

Source:	(SDK Path)\Samples\SampleD6Joint\src
Executable:	(SDKPath)\Bin\...

Sample Filtering



Overview

This sample shows how to use the new filtering API. The arbitrary filtering rule in this sample is "shapes sharing at least one group should collide". The filtering equation is:

(G0 op0 K0) op2 (G1 op1 K1)==b

So, we set it up like this:

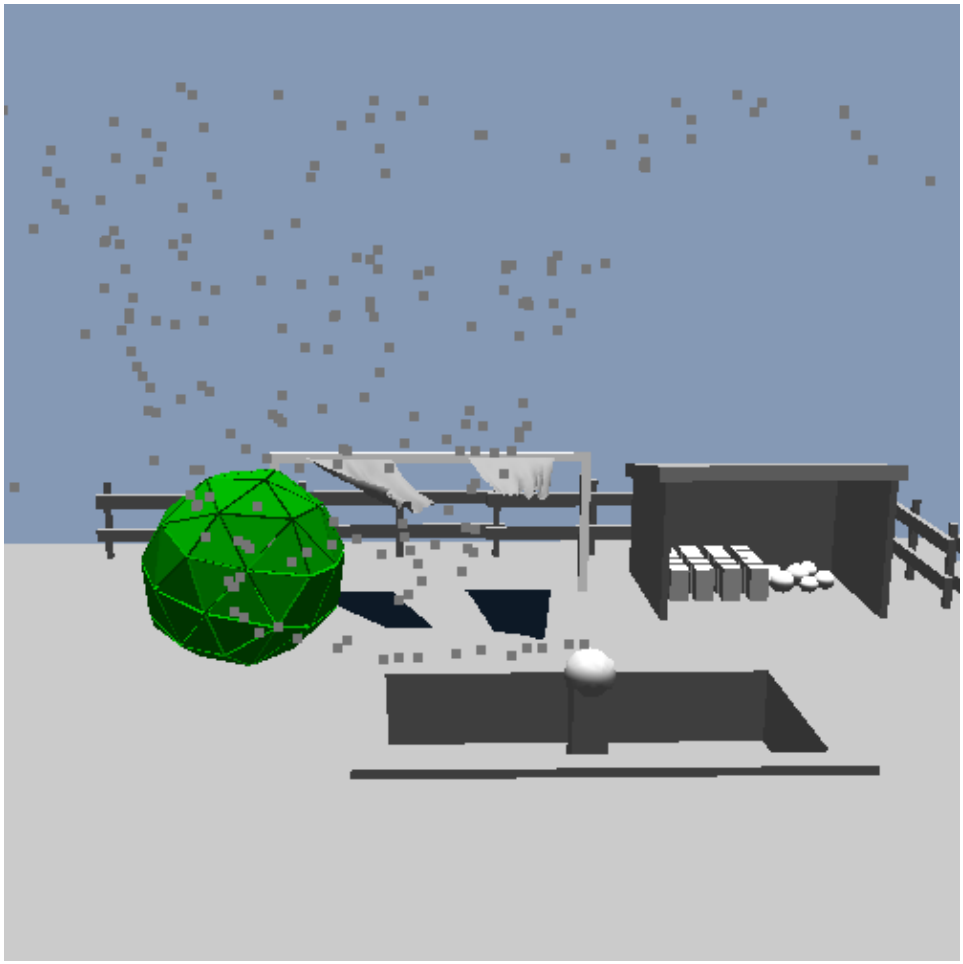
K0 = K1 = 0
op0 = op1 = NX_FILTEROP_OR
op2 = NX_FILTEROP_AND
b = true;

That way we end up with the following filtering equation: collision enabled <=> (G0 & G1)==true In other words, if the group bitmasks have at least one common bit, we collide.

Path

Source:	(SDK Path)\Samples\SampleFiltering\src
Executable:	(SDKPath)\Bin\...

Sample Force Field



Overview

Demonstrates the use of the Force Field features in the PhysX SDK.

The sample consists of several scenes that demonstrates various uses of the PhysX SDK force field API:

1. Create a spinning tornado (also inclusion and exclusion shapes)
2. Create an animated wind
3. Make a simple explosion effect

Path

Source:	(SDK Path)\Samples\SampleForceField\src
Executable:	(SDKPath)\Bin\...

Sample Heightfield

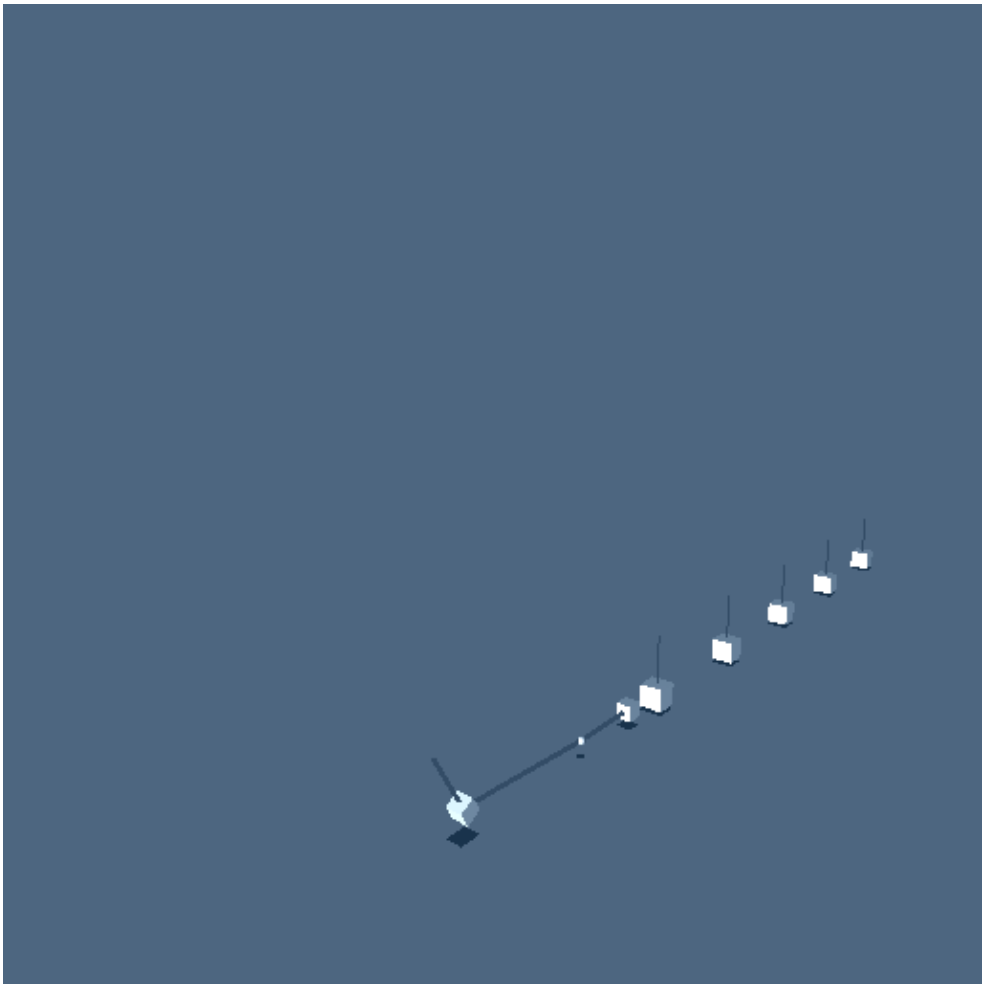
Overview

Demonstrate the heightfield shape and interaction with a variety of dynamic objects.

Path

Source:	(SDK Path)\Samples\SampleHeightfield\src
Executable:	(SDKPath)\Bin\...

Sample Joints



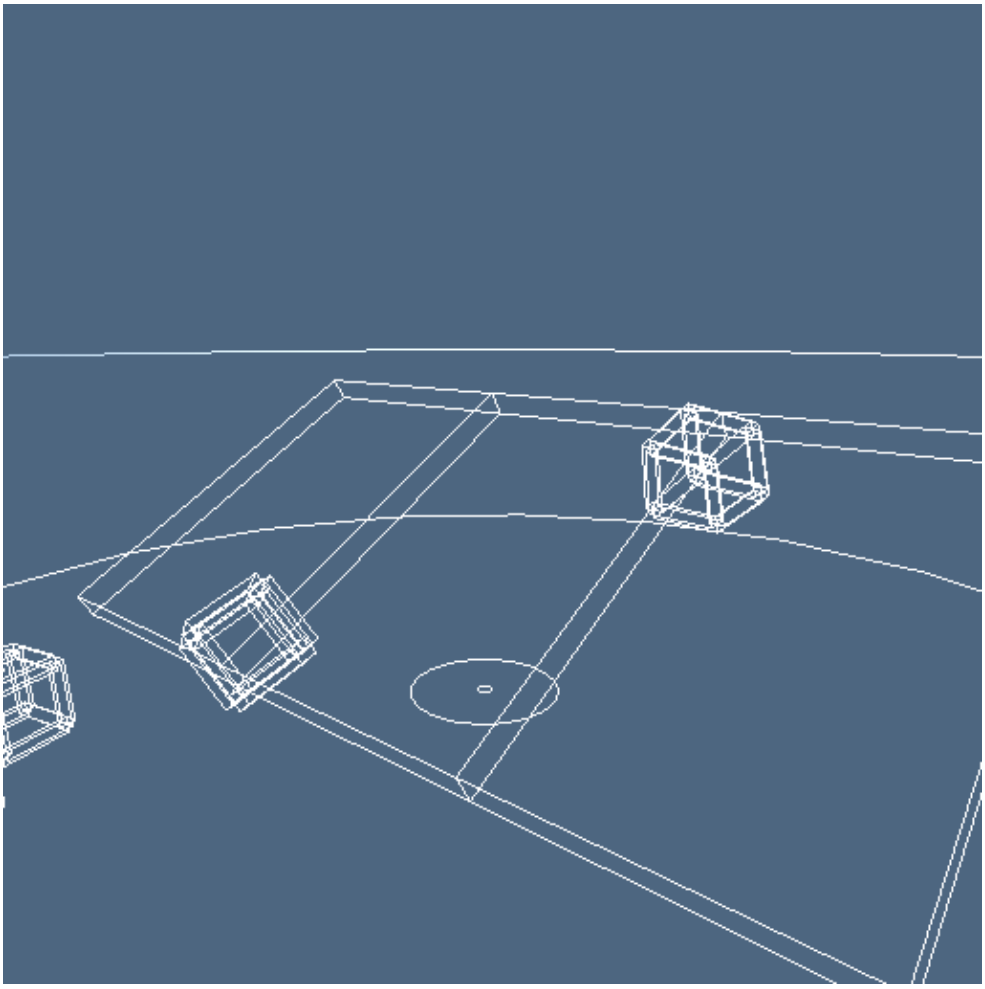
Overview

Demonstrates the use of a number of different joints.

Path

Source:	(SDK Path)\Samples\SampleJoints\src
Executable:	(SDKPath)\Bin\...

Sample Materials



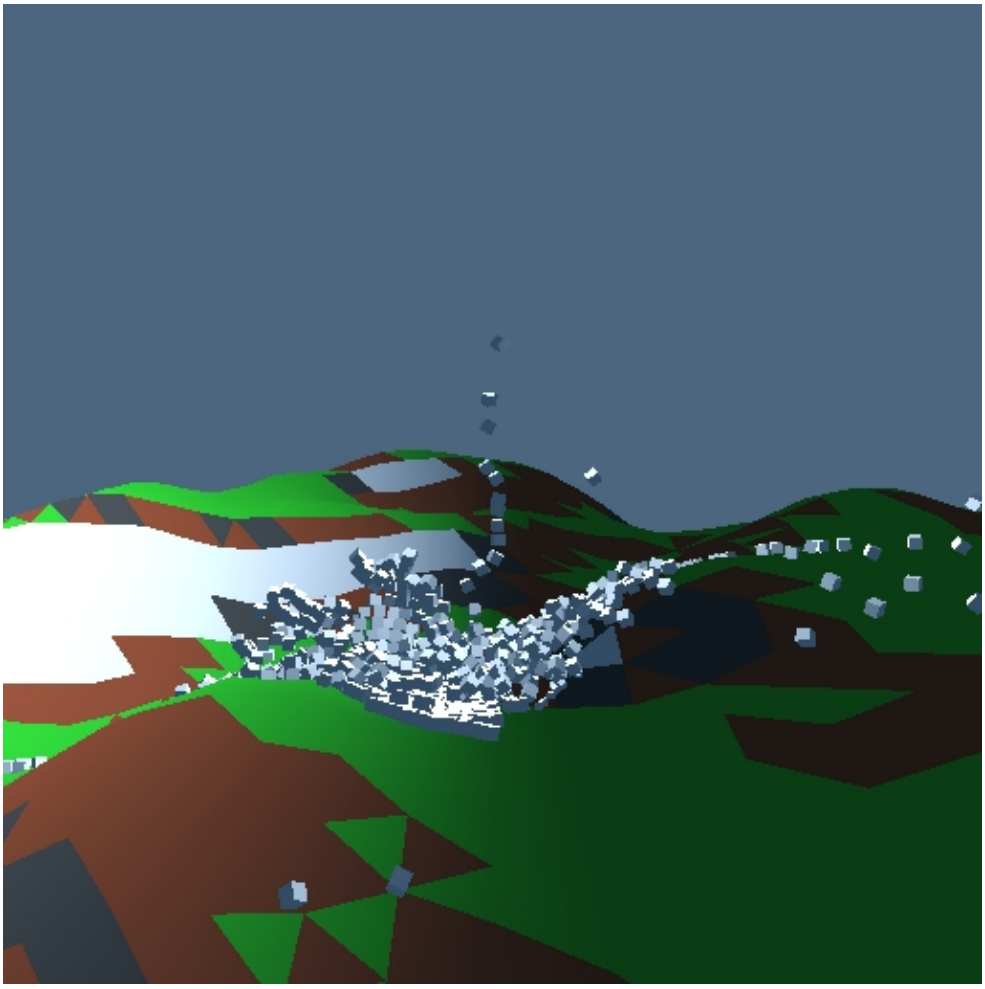
Overview

A programming sample showing the use of materials. We create a compound of several shapes, each shape having different material properties. We also create several ramps with different friction properties.

Path

Source:	(SDK Path)\Samples\SampleMaterials\src
Executable:	(SDKPath)\Bin\...

Sample Mesh Materials



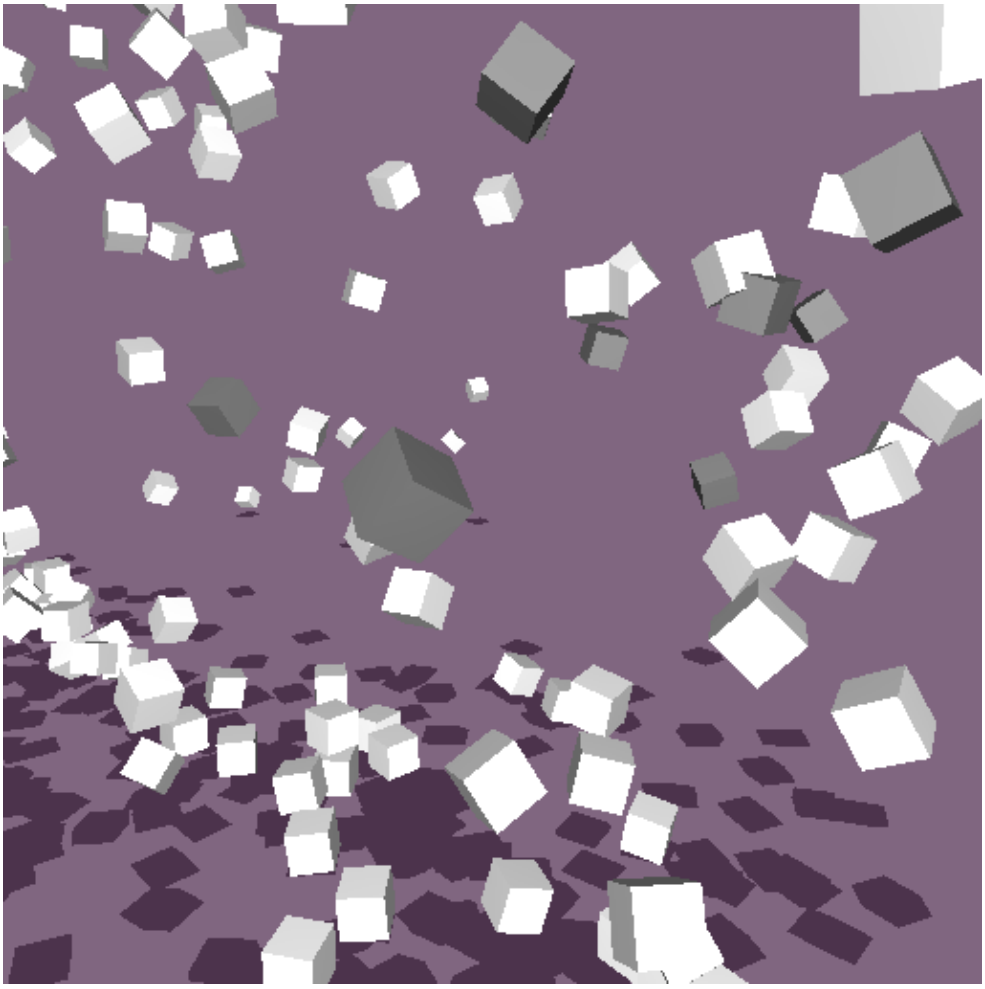
Overview

Shows how to apply different materials to individual triangles of a mesh.

Path

Source:	(SDK Path)\Samples\SampleMeshMaterials\src
Executable:	(SDKPath)\Bin\...

Sample Multiple Scenes



Overview

Shows how to create multiple scenes at once.

Path

Source:	(SDK Path)\Samples\SampleMultipleScenes\src
Executable:	(SDKPath)\Bin\...

Sample One-way Interactions

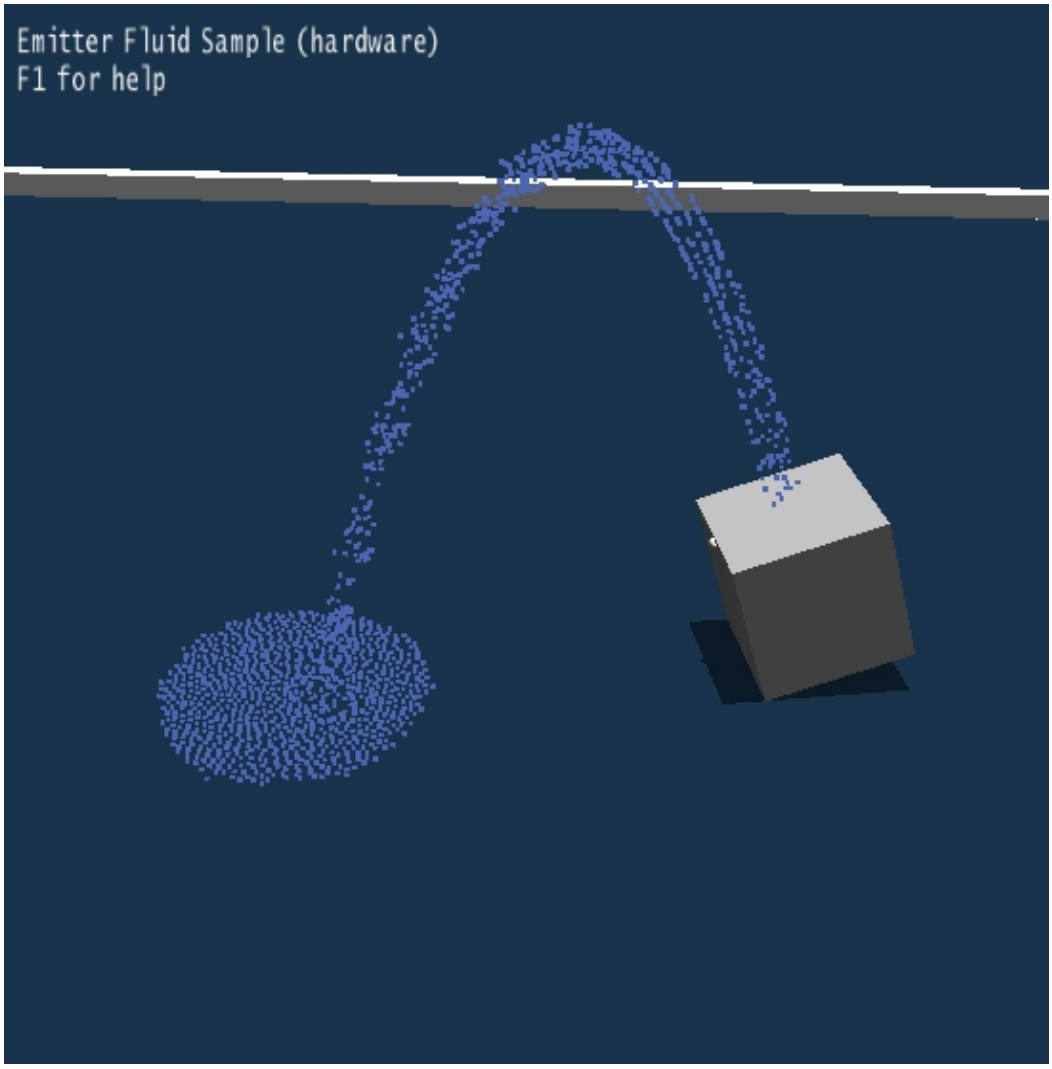
Overview

Demonstrates the use of 'dominance groups' (which can be used for creating one-way interactions between objects). This sample demonstrates this feature through a small 'game' where you fly on a hoverboard across a set of platforms. There are one-way interactions between the player and the debris on the floor, and internally in the player between the board and the body, and between the body and the arms.

Path

Source:	(SDK Path)\Samples\SampleOneWayInteractions\src
Executable:	(SDKPath)\Bin\...

Sample Particle Fluid



Overview

Demonstrates simulation of fluids in hardware and software.

The sample consists of several scenes that demonstrates various features of the PhysX SDK fluid API:

1. Create a fluid
2. Create a fluid through an emitter
3. Fluid vs. Rigid Body collision
4. Fluid particle forces
5. Fluid emitter events
6. Fluid particle userdata
7. Fluid packets visualization

Path

Source:	(SDK Path)\Samples\SampleParticleFluid\src
Executable:	(SDKPath)\Bin\...

Sample Pulley Joint

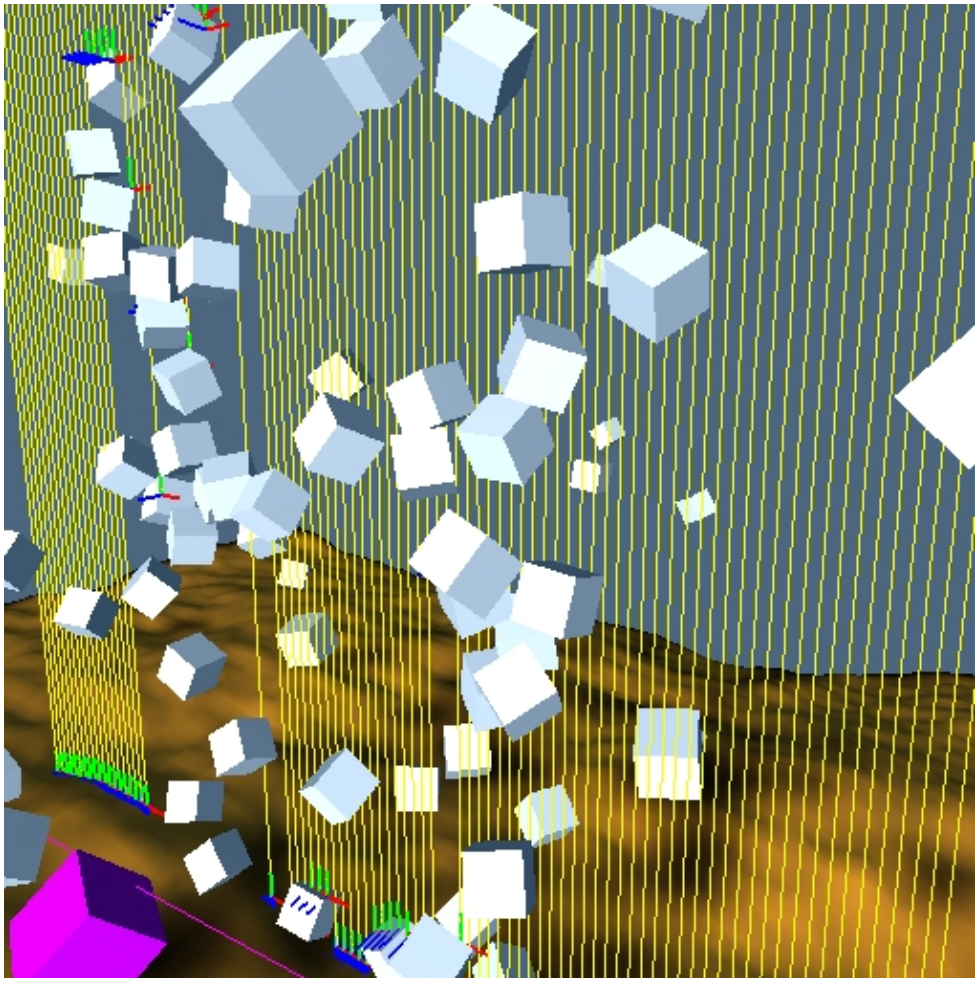
Overview

Shows how to setup a pulley joint.

Path

Source:	(SDK Path)\Samples\SamplePulleyJoint\src
Executable:	(SDKPath)\Bin\...

Sample Raycast



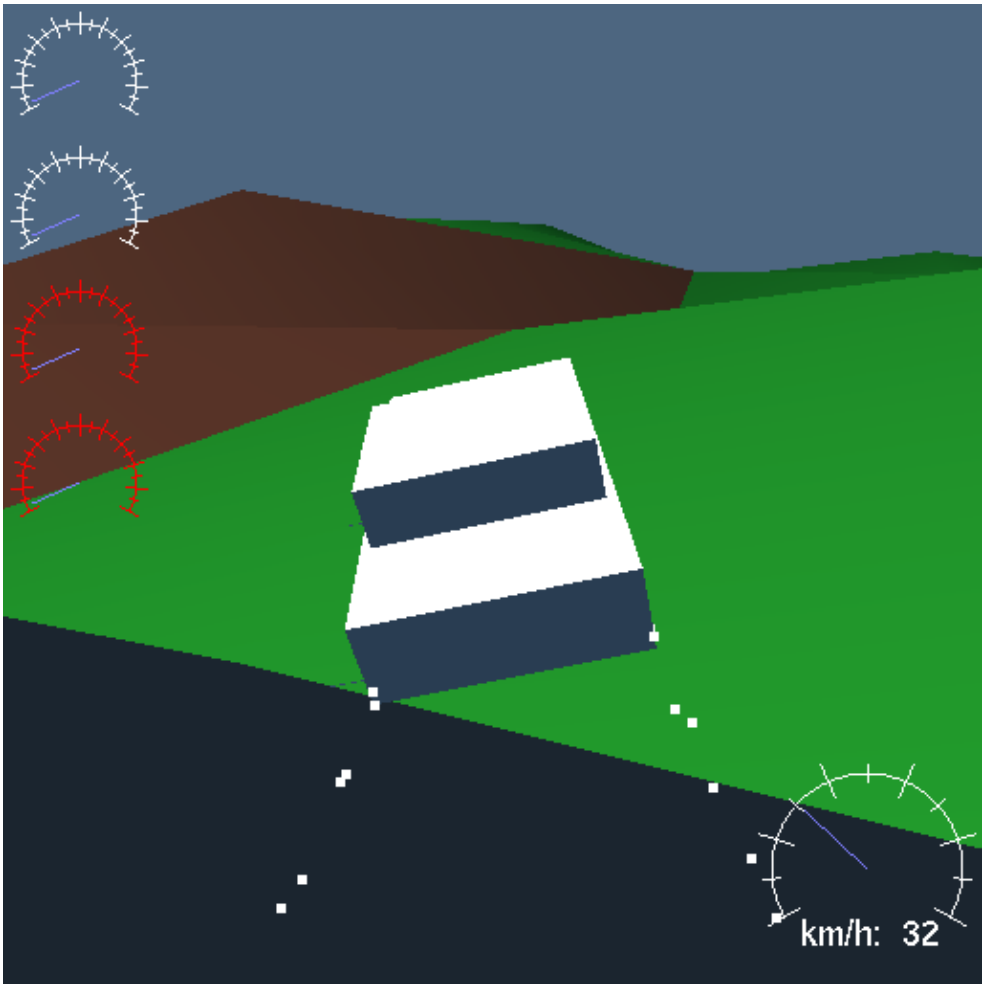
Overview

Shows how to perform raycasts against a terrain.

Path

Source:	(SDK Path)\Samples\SampleRaycast\src
Executable:	(SDKPath)\Bin\...

Sample Raycast Car



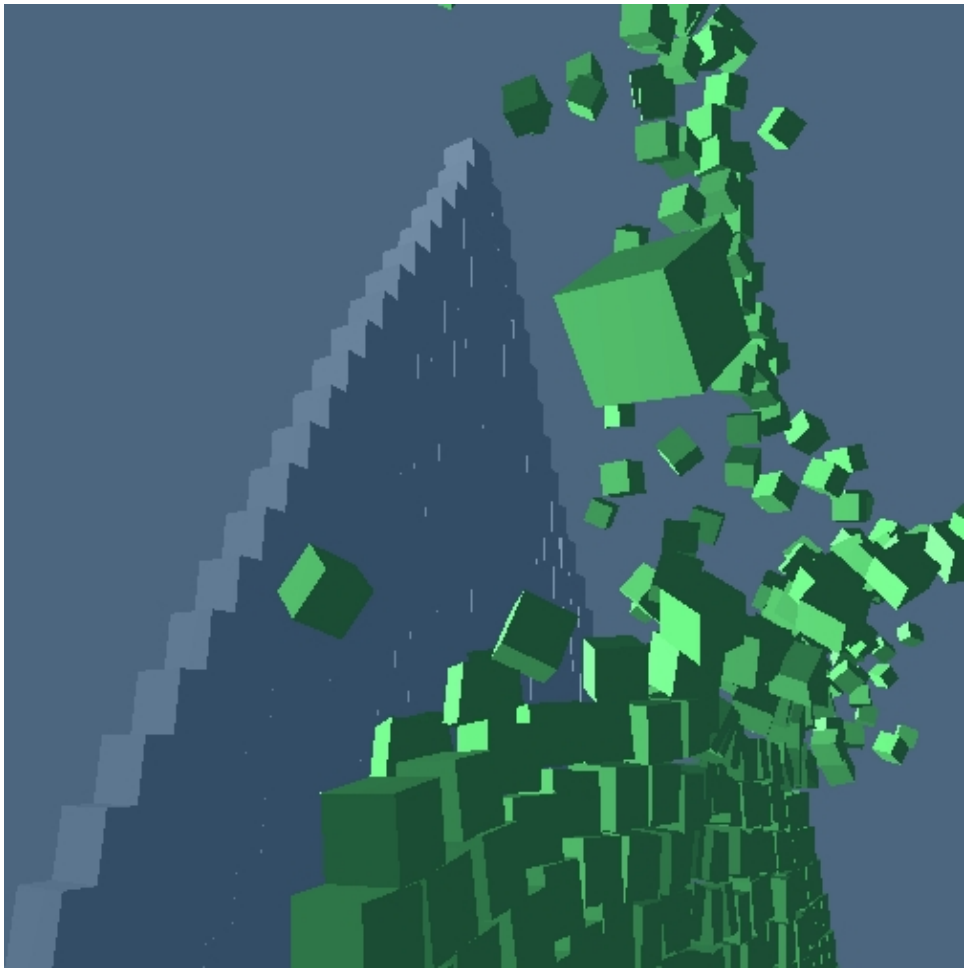
Overview

Shows how to create a raycast car.

Path

Source:	(SDK Path)\Samples\SampleRaycastCar\src
Executable:	(SDKPath)\Bin\...

Sample Rigid Body HSM



Overview

Demonstrate how software and hardware rigid body scenes interact using compartments, by means of the Hardware Scene Manager.

Path

Source:	(SDK Path)\Samples\SampleRBHSM\src
Executable:	(SDKPath)\Bin\...

Sample Scene Export

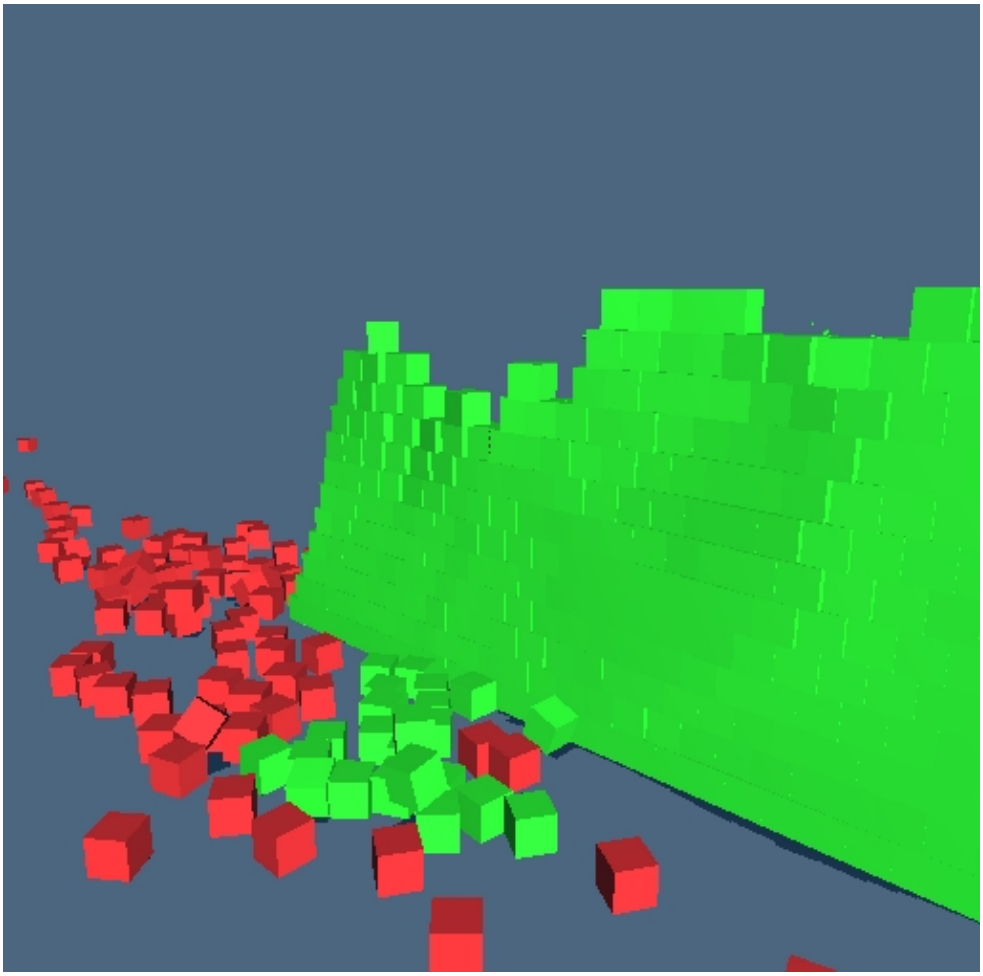
Overview

Demonstrate how to export and import a scene to and from a special file format.

Path

Source:	(SDK Path)\Samples\SampleSceneExport\src
Executable:	(SDKPath)\Bin\...

Sample Sleep Callback



Overview

Demonstrate the use of Sleep Callbacks.
Graphical objects have their color updated as their physical counterparts go to sleep or wake up.

Path

Source:	(SDK Path)\Samples\SampleSleepCallback\src
Executable:	(SDKPath)\Bin\...

Sample SoftBody

Overview

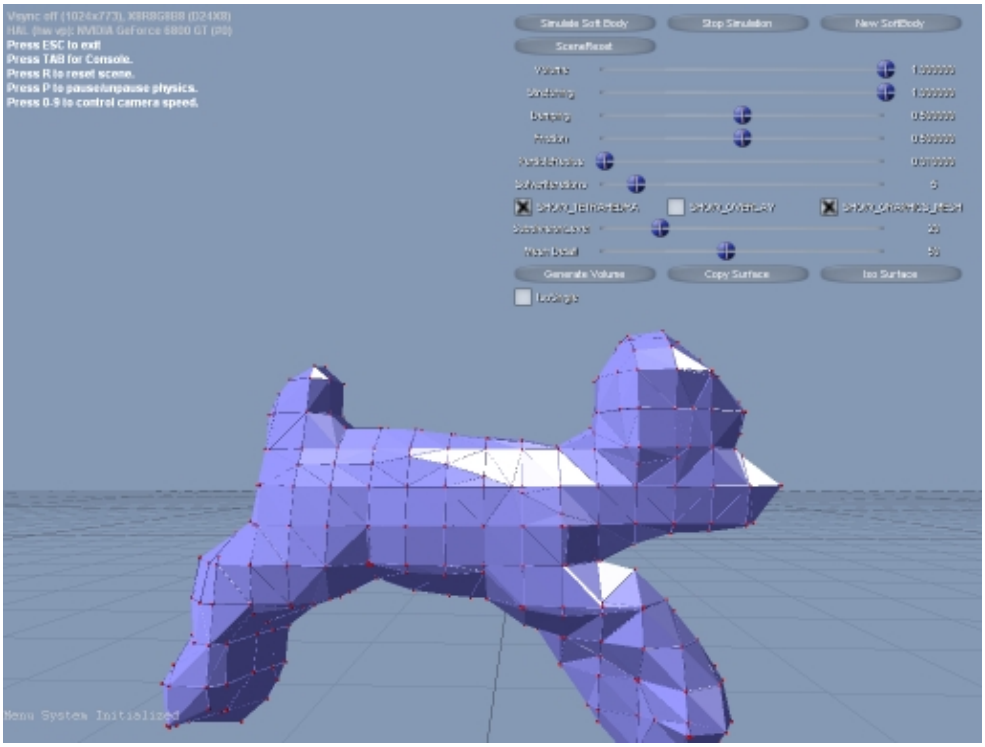
Demonstrates the use of SoftBody through a selection of soft body demos:

1. A bunch of frogs falling down on top of each other.
2. Big soft wheels attached to a simple car body.
3. A swaying palm tree.
4. Grass that moves when hit by objects.
5. Soft boxes and spheres interacting.
6. A soft bunny getting squeezed between two plates.
7. Pumpkins (with different settings) falling down a set of stairs.

Path

Source:	(SDK Path)\Samples\SampleSoftBody\src
Executable:	(SDKPath)\Bin\...

PhysXViewer



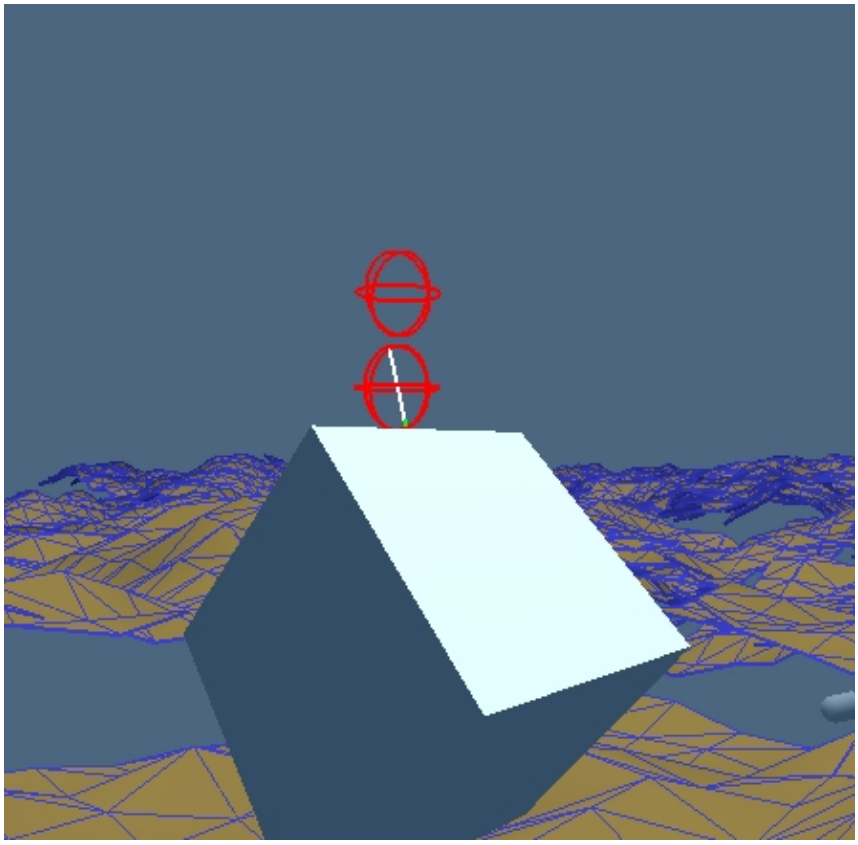
Overview

Supports authoring of tetrahedral models for soft body simulation.

Path

Source:	(SDK Path)\Samples\PhysXViewer\src
Documentation:	(SDK Path)\Samples\PhysXViewer\documentation
Executable:	(SDKPath)\Bin\...

Sample Sweep Tests



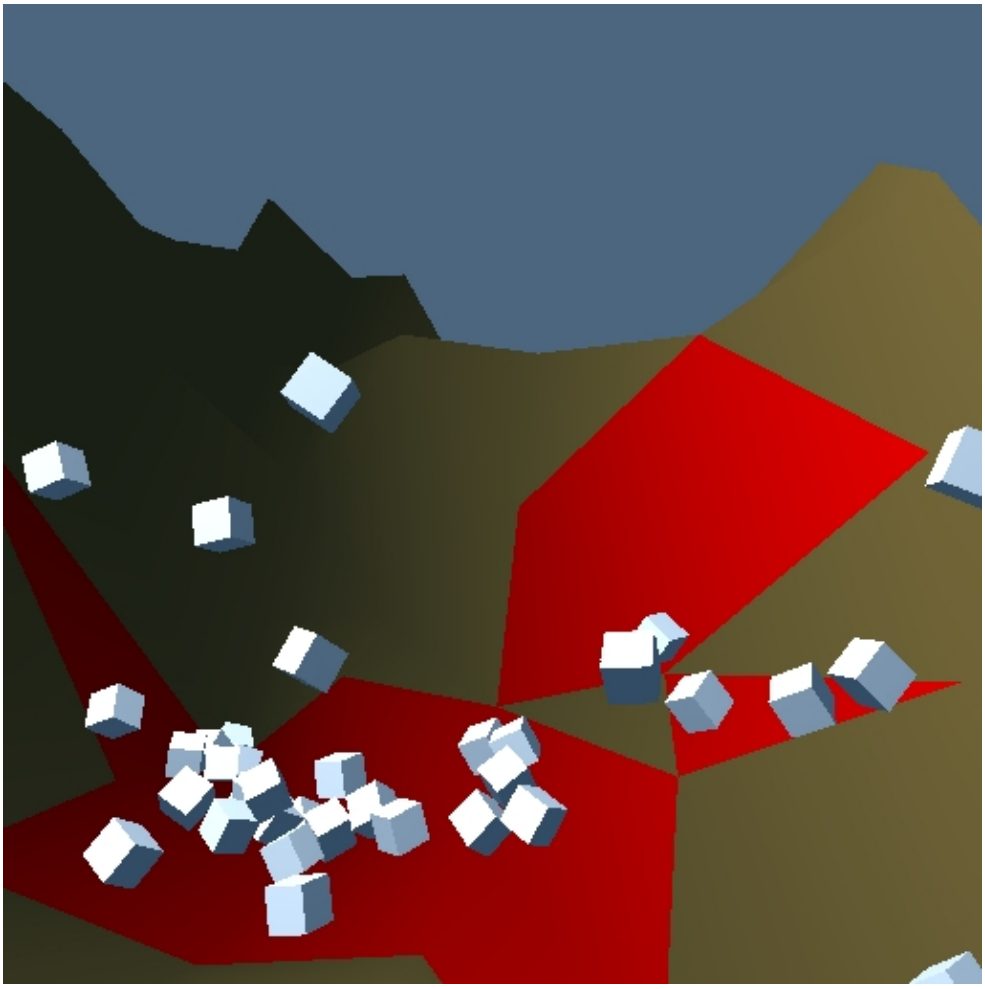
Overview

Demonstrate sweep functionality.
Use differently shaped sweeps to find the intersection to static meshes or dynamic objects.

Path

Source:	(SDK Path)\Samples\SampleSweepTests\src
Executable:	(SDKPath)\Bin\...

Sample Terrain



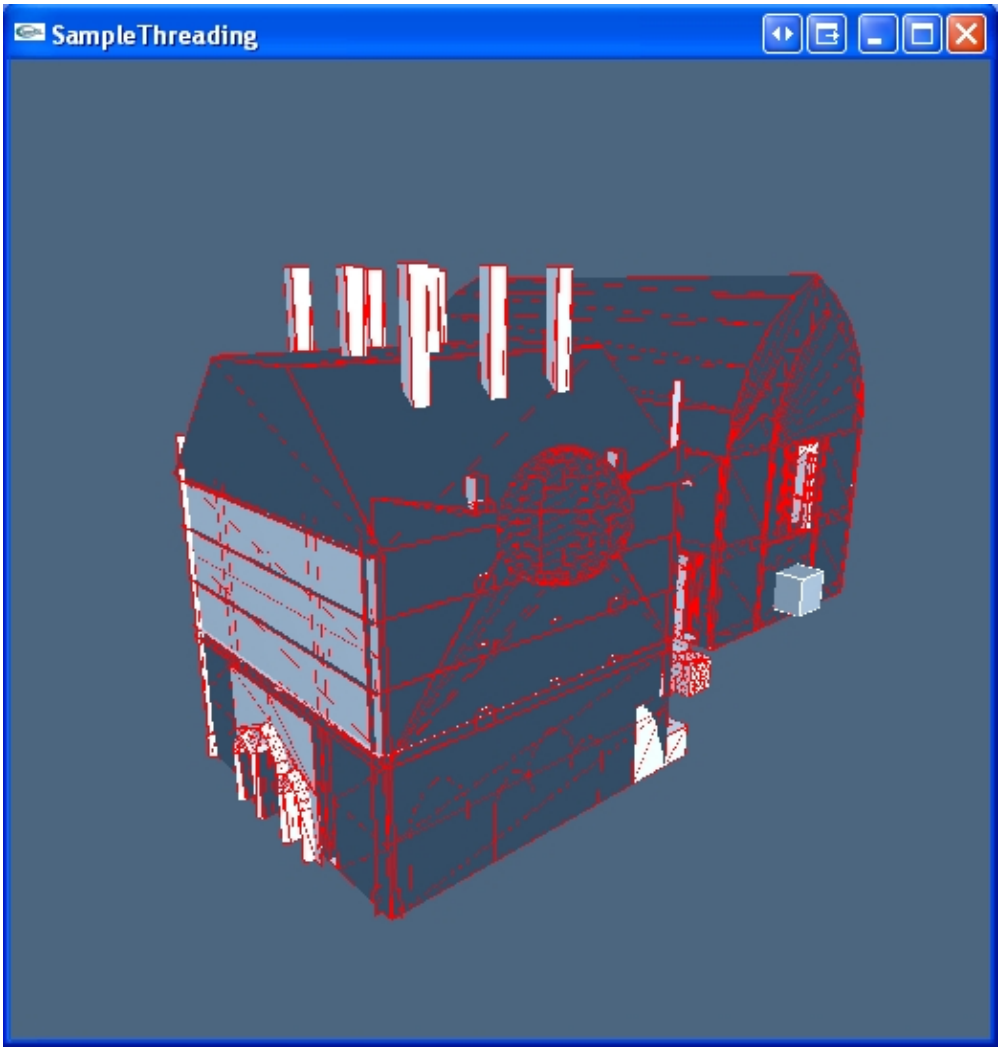
Overview

Shows how to create a terrain mesh with the PhysX SDK.

Path

Source:	(SDK Path)\Samples\SampleTerrain\src
Executable:	(SDKPath)\Bin\...

Sample Threading



Overview

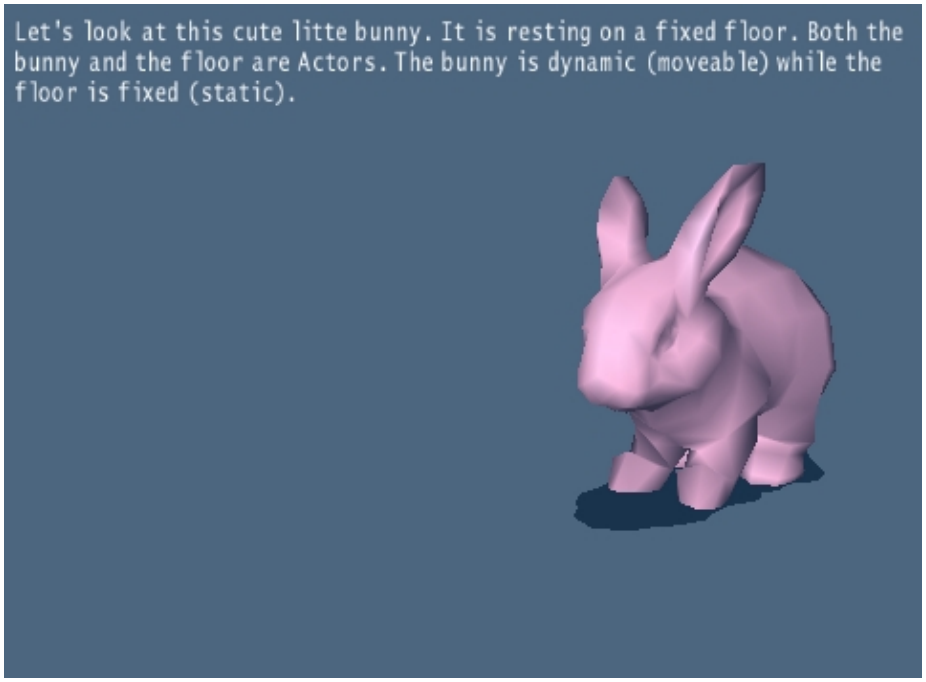
Demonstrate 4 different approaches to threading:

- Application supplied scheduler
- Polling, i.e. the SDK manages the work queue and the application can poll a function to perform work
- SDK managed threading (At present this is likely to yield the best performance)
- No threading

Path

Source:	(SDK Path)\Samples\SampleThreading\src
Executable:	(SDKPath)\Bin\...

Sample Transforms



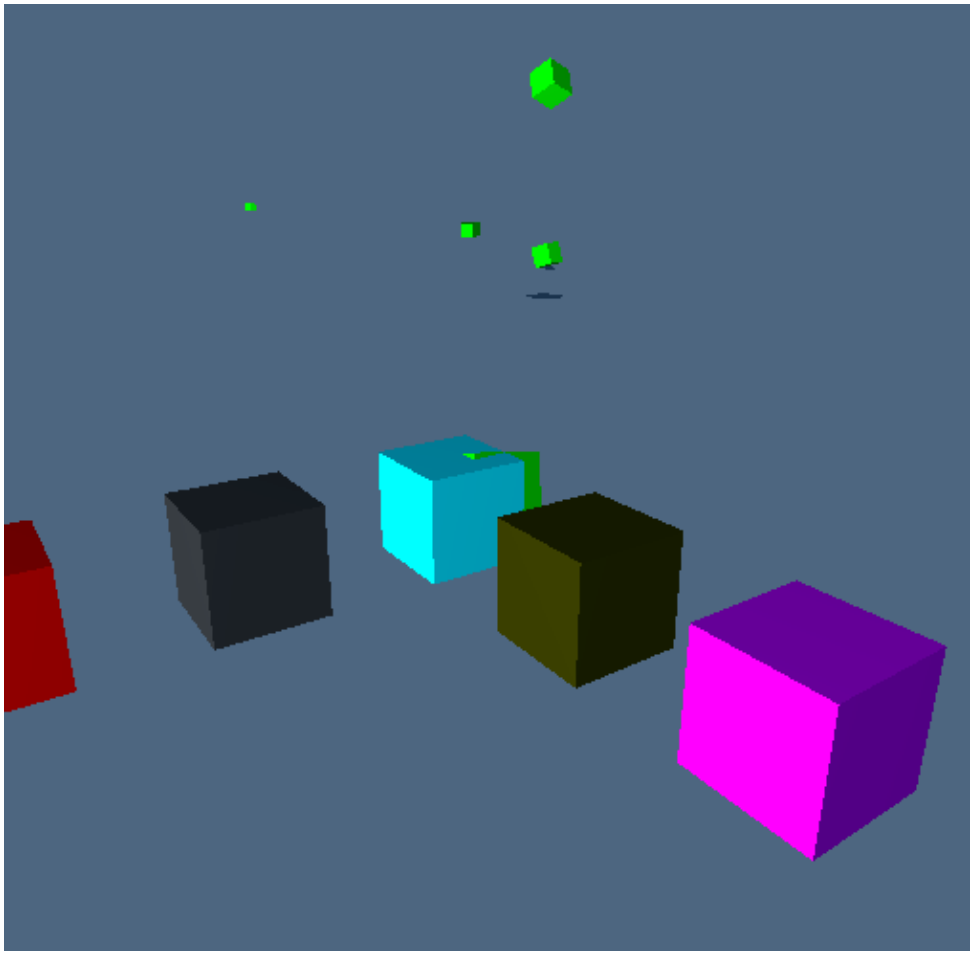
Overview

Provides a tutorial for the use of transforms.

Path

Source:	(SDK Path)\Samples\SampleTransforms\src
Executable:	(SDKPath)\Bin\...

Sample Trigger



Overview

Demonstrates the use of boxes as triggers.

Path

Source:	(SDK Path)\Samples\SampleTrigger\src
Executable:	(SDKPath)\Bin\...