

NovodeX Physics SDK Documentation



Table of Contents

NovodeX Physics SDK Documentation	1
1 Introduction	4
1.1 Other Resources	4
2 Installation	5
2.1 Setup	5
2.2 Directory Structure	5
2.3 DLL Configurations	6
2.4 Linux and Console Libraries	6
2.5 Compiler Setup	6
2.6 Dynamic Linking	7
3 API Basics	8
3.1 Architecture	8
3.2 Conventions	8
3.3 Data Types	8
3.4 SDK Initialization	9
3.5 Instancing	10
3.6 Upcasting	11
3.7 Downcasting	11
3.8 User Data	12
3.9 Object Names	12
3.10 User Defined Classes	12
3.11 Error Reporting	13
3.12 Saving the Simulation State	14
3.13 Units	15
4 Foundation SDK	16
4.1 Math	16
4.2 Memory Management	17
4.3 Error Reporting	17
4.4 Simple Shapes	17
4.5 Containers	18
4.6 Debug Rendering	18
4.7 Utilities	20
5 Dynamics	21
5.1 Scenes	21
5.1.1 Simulation	21
5.1.2 Asynchronous Simulation	22
5.2 Actors	23
5.2.1 Creating Actors	24
5.2.2 Reference Frames	25
5.2.3 Rigid Body Properties	27
5.2.4 The Inertia Tensor	28
5.2.5 Sleeping	29
5.2.6 Applying Forces and Torques	30
5.2.7 Gravity	30
5.2.8 Setting the Velocity	31

5.2.9	Fast Rotation	31
5.2.10	Static Actors	32
5.2.11	Kinematic Actors.....	32
5.3	Joints.....	33
5.3.1	Spherical Joint.....	34
5.3.2	Revolute Joint.....	34
5.3.3	Cylindrical Joint	35
5.3.4	Prismatic Joint	35
5.3.5	Point On Line Joint	36
5.3.6	Point In Plane Joint	36
5.3.7	Joint Frames	36
5.3.8	Joint Simulation Method	38
5.3.9	Breakable Joints	38
5.3.10	Joint Limits.....	38
5.3.11	Joint Motors, Springs, Projection, and Special Limits.....	40
5.3.12	Spring and Damper Element	40
5.3.13	Modeling with Joints.....	41
5.4	SDK Parameters	41
5.4.1	Min Separation For Penalty	42
6	Collision Detection.....	43
6.1	Shapes.....	43
6.2	Sphere.....	44
6.3	Box	44
6.4	Capsule	44
6.5	Triangle Mesh	44
6.5.1	Sphere-Mesh Collision Detection	47
6.5.2	PMaps.....	47
6.5.3	Convex Meshes	49
6.6	Plane	49
6.7	Height Field.....	49
6.8	Broad Phase Collision Detection	50
6.9	Shape Pair Filtering.....	51
6.9.1	Collision Groups	51
6.9.2	Disabling Pairs	51
6.10	Ray Casting	52
6.11	Triggers	53
6.12	Contact Reports	54
6.13	Materials.....	55
6.13.1	Materials per Triangle	56

1 Introduction

The NovodeX Physics SDK unites rigid body dynamics simulation and collision detection in a single easy to use package. The Physics SDK builds only on the Foundation SDK, a library of mathematics and utility functionality. It has no other dependencies. This document familiarizes the reader with both SDKs.

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.

1.1 Other Resources

In addition to this reference manual, the following resources may provide help with using the SDK:

- Annotated API reference manuals included with the SDK. These are good for looking up function parameters and other programming specifics once you have an overall idea of what you want to do.
- The source code of example programs that come with the SDK give you reusable programming demonstrations on how to do certain common things. In particular, example programs for the following functionality is available:
 - SampleBoxes – simplest demo, about creating piles of boxes
 - SamplePMap – shows collision detection between meshes and other shapes, including the use of pmaps.
 - SampleRaycast – demonstrates ray casting.
 - SampleTrigger – shows how to use trigger shapes.
 - SampleTerrain – demonstrates terrain height field collision detection.
 - SampleArticulateTruck – demonstrates joints and visualization.
 - SampleSerialize – shows how to save and load simulation data.
 - SampleMultipleScenes – a demo that creates multiple independent scenes
- NovodeX technical support available for licensees of the Professional Edition SDK. Contact NovodeX for details.
- High school level physics books and other third party sources are good for a more in-depth look at the theory.

2 Installation

2.1 Setup

Note: This setup procedure only applies to NovodeX Personal Edition licensees.

After downloading the NovodeX Personal Edition executable from the NovodeX web site, execute it and follow the direction of the installer program. After successful installation, the directory structure described in the next section should have been created in the location you have specified.

During installation you will be requested to register NovodeX Personal Edition. You will then receive the license key by email from NovodeX. You should then run the Purchase (System ID) application from the NovodeX program group in your start menu to enter the license key.

WARNING: Unless the license files are in place and contain the correct codes, the NovodeX DLLs will not function. For security reasons, an error message will only be displayed if your license is missing from the registry, but not if it is incorrect. If you have any doubts about the correct functioning of your SDK, please contact NovodeX.

2.2 Directory Structure

The SDK installer should create the following directory structure:

<User specified install directory>		
	BIN	SDK DLLs and demo executables.
	Graphics	include and library files for the graphics library used by the viewer.
SDKs		
	Foundation	The Foundation SDK. Provides basic functionality on which the other SDKs build. Full source code is included.
		compiler doc include lib src
	Physics	The Physics SDK.
		Documentation.
		SDK header files.
		SDK libraries.
		win32
		other platforms...

other SDKs...
Tools

Exporter plugins for various
authoring tools. See exporter
documentation.

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

2.3 DLL Configurations

The SDK currently ships with two SDK DLLs, called NxPhysics[*Personal*].dll, and NxFoundation.DLL, found in the BIN/win32 directory. This serves as our “default” configuration: It uses single precision floating point numbers, and no exception handling. We may be able to provide other configurations upon request.

The Physics SDK DLL comes in two flavors: NxPhysics.dll is the professional edition and NxPhysicsPersonal.dll the personal edition. They both have corresponding .lib (NxPhysics.lib vs. NxPhysicsPersonal.lib) and .h (/include/*.h vs /includePersonal/*.h) files which may not be interchanged.

If you receive a new DLL from NovodeX in the form of an update or a patch, you must make sure to recompile your program with the corresponding newly provided header files.

Note that even though the SDK may use the multithreaded C++ runtime, it is not thread safe. This means that you should not make calls to the DLL from more than one thread at any time. However, you may use the DLL in an otherwise multithreaded application.

2.4 Linux and Console Libraries

The SDK is available for the Linux operating system and console game platforms for Professional Edition licensees only. Please contact NovodeX for details.

2.5 Compiler Setup

In order to build the example programs, you should be able to open the provided MS Visual Studio workspace(s) for any example project, and issue a build command. No additional configuration should be needed. Note however that the created example programs will need to access the SDK DLL, which is located in the /bin/win32 directory.

If you are using a different compiler than the ones for which project files are provided, or you would like to create a new project using the SDK, you can easily set up a project as follows:

1. Specify the “SDKs\Foundation\include” and “SDKs\Physics\include” (or “SDKs\Physics\includePersonal” for the personal edition) as a directory in which include files are located.
2. Identify the NxFoundation.lib file in the “SDKs\Foundation\lib\win32\Release” and the NxPhysics.lib (or NxPhysicsPersonal.lib) file in the “SDKs\Physics\lib\win32\Release” as libraries to the linker.

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

2.6 Dynamic Linking

You may link to the SDK DLL dynamically if you wish. To do this you would load the DLL with the windows API function LoadLibrary() and then retrieve the entry point of the exported functions with GetProcAddress(). In this case you should not link with the static link stub libraries provided.

Dynamic linking has the following advantages:

- Windows will not automatically abort your application when the DLL is not found. You can handle the error in a customized way instead.
- You only need to load the DLL into memory when you really need it.

3 API Basics

3.1 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 as C functions.

3.2 Conventions

Each of the interface classes defines a set of methods. There are some coding conventions that are followed:

- 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 and having it blow up there.
- 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.

3.3 Data Types

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

The Foundation 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. In particular, note that the quaternion class is also ‘element order independent’ in the sense that all direct access routines need

to specify whether they are expecting to read or write data in x, y, z, w or w, x, y, z order. Because 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.

3.4 SDK Initialization

We will now start to walk the programmer through using the SDK, using the source code of the SampleBoxes demo application as an example. It can be found in the file /SampleBoxes/src/NxTest.cpp.

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

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

If your application uses windows.h and it is 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:

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

Before the SDK can be initialized, the Foundation SDK has to be started up:

```
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 below for more information.

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

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

When you are done using the SDK, you must release it. Releasing the SDK should automatically release all of the SDK objects you created:

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

3.5 Instancing

You cannot create instances of the class interfaces with the C++ **new** operator, because they are pure virtual classes. The instance of the first class you create, NxPhysicsSDK, is returned by the NxCreatePhysicsSDK function, discussed above.

Many SDK interfaces contain create* methods that return instances of other types. You must use these methods to create SDK objects. To avoid overly long parameter lists and 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 in most cases not suffice to create an object. The user may then override these parameters:

```
sceneDesc.gravity.set(0,-9.8f,0);
sceneDesc.collisionDetection = true;
... etc
```

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 that it is important that the user check the return values of all create methods, since 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 need to destroy it. Again, you may not use the standard C++ **delete** operator. This will lead to memory corruption. Instead, you must ask the object which 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, be sure not to use the pointer or reference that used to point to it anymore. NOTE: 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 upcasting and down casting operators (see later) 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 that if you release an object which you have used to create other objects, these other owned objects will also be released automatically. For example, if you release the SDK, all the created scenes will be released.

3.6 Upcasting

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

There are two groups of related classes in the SDK which nonetheless do not form class hierarchies. In other words, the SDK does not use C++' built in interface inheritance.

This means that even though, for example, logically NxSphereShape is a kind of NxShape, the compiler building the user's application does not know this. For this reason explicit up and downcasting is necessary.

Suppose you created an instance of the NxSphereShape object, and then wanted to store the returned reference in an array of NxShape pointers. This is always possible because a NxSphereShape is a type of NxShape. The code below shows how you would do it:

```
NxShape * shapePointers[10];
NxSphereShape * mySphere = ... whatever ...;
shapePointers[0] = & mySphere.getShape();
```

The getShape() method never fails, and always returns a valid object. Note that the upcast is even simpler if you use references in your code:

```
NxSphereShape & mySphere = ...;
NxShape & shapeRef = mySphere;
```

Here, mySphere is automatically upcast because the overloaded cast operator

```
operator NxShape &()
```

it defines gets invoked.

3.7 Downcasting

Downcasting is the opposite of upcasting: Here you are trying to convert a reference to a subclass to one of its potential derived classes. Unlike upcasting, 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 a NxSphereShape object will successfully cast to a NxSphereShape reference, but the cast will fail when you try to cast it to a NxBoxShape. The below example 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 that 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 on which classes provide which up and downcast methods.

3.8 User Data

When you create a shape, you may want 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. You 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.

3.9 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.

3.10 User Defined Classes

As mentioned above, the SDK requires (in some cases this is optional) the user to implement some functionality. In particular, the user may implement the following tasks:

Class Interface To Implement	Functionality
NxUserAllocator	Memory management

NxUserDebugRenderer	Debug visualization
NxUserOutputStream	Error reporting
NxUserContactReport	Contact information
NxUserNotify	Misc. events, including joint breakage
NxUserRaycastReport	Ray casting information
NxUserTriggerReport	Trigger information

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 *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);
```

This creates a very simple memory allocator class which just uses the standard C memory allocation routines. The real purpose of a custom allocator is for you to use some custom memory allocator scheme, as described in the section 4.2 below.

3.11 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("SDK Error", message, MB_OK),
        exit(1);
}

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

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

} myOutputStream;

```

This object is then passed to either `NxCreatePhysicsSDK()`:

```

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

```

At the moment messages are only logged when an error occurs, with standard syntax.

For example:

```

c:\novodex\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. Then, if the DLL has been compiled with exception handling enabled (this is by default not the case though!) an exception (type `NxException`) will be thrown. This exception contains the same information as the error message.

3.12 Saving the Simulation State

Most of the simulation objects which are created using a descriptor have two methods called `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 state information that will be needed later will be written into the descriptor. Be sure to check the API documentation of `saveToDesc()` for the object at hand, because

sometimes some additional measures may be needed. The user is then responsible for serializing the data contained in the descriptor in some way. This sometimes involves the conversion of pointers to some 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, more optimally, create a new object by passing the descriptor to the `create*()` call.

3.12.1 Core Dumps

The SDK can automatically save its state to disc using the core dump functionality. The resulting file, using either a binary or a text file format, can then be loaded into a viewer application. This is tremendously useful for debugging because clients can send simulation state data to the support team without having to share source code or binaries.

A core dump can be requested using the `NxPhysicsSDK::coreDump(const char *fname, bool binary)` method.

3.13 Units

The SDK does not need to use any particular real world units. What is important is that you 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/second. Other such derived units apply to force, impulse, etc.

As with any numerics 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.

4 Foundation SDK

The NovodeX Foundation SDK provides basic support and mathematics functionality to the other NovodeX SDKs, including the Physics SDK. For that reason it is discussed here.

First, it is possible to use the Foundation SDK without using the Physics SDK, for example to develop new physics libraries! However, this is not the topic of this document, and won't be discussed here. Also, it is not a good idea to create your own Foundation SDK when also using the Physics SDK, because you might clobber some shared data.

The classes you will most likely use from the Foundation SDK are described below.

4.1 Math

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

The NxMath class contains both wrappers for the standard C floating point scalar math functions, such as sin() or fabs(), as well as additional related functionality 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 also inertia tensors. However, the SDK does not permit scaling or any other affine transformation that could in general 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

$$p' = H p = R p + t$$

The product of two transformations $A = [aR \ aT]$ and $B = [bR \ bT]$ is

$$A B = [aR \ bR \ aR bT + aT]$$

The storage of matrix elements in memory is meant to be opaque to the user and 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 (or similarly to Direct3D) is like this:

```
NxMat34 m; //NovodeX matrix
float glmat[16]; //OpenGL matrix
m.getColumnMajor44(glmat); //copy to glmat
glMultMatrixf(glmat); //send to OpenGL
```

Note that while Direct3D matrices are technically row major, they have an additional semantic difference to OpenGL matrices which cancels this out. So one should use the exact same code for Direct3D as well.

4.2 Memory Management

The memory management functionality of the Foundation SDK was introduced in section 3.9. This mechanism is not particularly useful for PC development, but was meant to be used on systems with special memory management requirements. However, it lets the user monitor the memory usage of the SDK, which may be useful in many cases. Note that currently parts of the mesh collision detection subsystem bypass this mechanism and allocates from the OS directly.

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. You can see an example implementation in section 3.9. Of course the point is for you to call your custom made memory allocator instead. The SDK does not guarantee any particular usage patterns that would let you make a really specialized allocator. In the many cases where this is possible, 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.

4.3 Error Reporting

The error reporting functionality was introduced in section 3.11. This mechanism actually belongs to the Foundation SDK and access to it is simply delegated to the user by the Physics SDK.

4.4 Simple Shapes

The Foundation 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 NxBounds3 (axis aligned box), NxBox (arbitrarily oriented box), NxCapsule, NxPlane, NxRay, NxSegment, and NxSphere.

4.5 Containers

The Foundation SDK provides with `NxArray` an array template and with `NxBitfield` a bit vector.

4.6 Debug Rendering

The debug renderer mechanism is intended to let the SDKs communicate potential problems to the user in a visual way, rather than just textual error messages. This is very useful because with such software many problems arise due to the incorrect spatial layout of objects. For example, a car will not roll if its wheels' axes are set up to be pointing in the wrong direction, but this reason for the problem will usually not be immediately obvious to the user, especially because while unintended, the configuration of the axes is not strictly illegal, and will thus not produce an error message at all.

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.

The first thing you must do to take advantage of this is to implement a subclass for the abstract base class `NxUserDebugRenderer`. This involves only implementing its single method, `renderData(const NxDebugRenderable& data)`. The data in question 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.

```
class DebugRenderer : public NxUserDebugRenderer
{
public:
    void renderData(const NxDebugRenderable& data) const;
};

void DebugRenderer::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();
}
}

```

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 a weird way, and you suspect this may be because the inertia tensors are set wrong. You decide that its best to have the SDK draw all the inertia tensors being used on screen. First, turn on visualization in general by setting a nonzero visualization scale:

```

NxReal myScale = 1.0f;
physicsSdk->setParameter( NX_VISUALIZATION_SCALE, myScale);

```

The scale parameter passed here scales all of the visualization elements used by the SDK, such as arrow lengths, uniformly. The scale you need to use depends on how large the objects are in your scene. Its best to bind it to some key or GUI slider widget so that you can interactively change the scaling of the elements. Note that turning this on will have a

slight performance cost even if you do not actually go on to visualize anything specific, so it's a good idea to only turn it on when debugging.

Next, you need to turn on the particular visualization option that you are interested in. In our case this is the 'mass axes', which draws, around each actor, a hypothetical box that would have the equivalent mass distribution as the actor does in the simulation.

```
physicsSdk->setParameter( NX_VISUALIZE_BODY_MASS_AXES, 1.0f );
```

This way it quickly becomes obvious, which objects, if any, have the mass properties set incorrectly.

4.7 Utilities

Utilities include NxProfiler, a performance profiler, NxQuickSort, an implementation of the quicksort algorithm, NxVolumeIntegration/ NxComputeVolumeIntegrals, a function for computing the inertia tensor of a triangle mesh, and NxUtilities.h, a collection of miscellaneous functions.

5 Dynamics

We finally come to the actual purpose of the SDK: rigid body dynamics simulation. The SDK's simulation functionality consists of the key concepts described next.

5.1 Scenes

Simulations take place in scenes (abstracted by the NxScene class). A scene is a container for a simulation's actors, joints and effectors. You may create several scenes and fill them with objects, and then simulate them in parallel. The simulation of two scenes is completely disjoint, so objects in different scenes do not influence each other. Of course you may add external forces which lead to a sort of communication between scenes.

Scenes don't 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 multiplayer game where a single process must perform the simulation for the server and a local client. This process would create two scenes to do this. Then the client scene could simulate only the immediate perceivable vicinity of the client's avatar, while the server scene would simulate the entire world.

We have briefly shown how to create a scene in section 3.5. 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 of course be changed at any time using `NxScene::setGravity()`.

5.1.1 Simulation

The main feature of a scene is its capability to actually perform the physics simulation. This results in various object properties being evolved over time, including for example bodies' positions and velocities.

Simulation is done one time step at a time, typically between $1/100^{\text{th}}$ of a second and $1/50^{\text{th}}$ of a second. For a real time simulation, if more than this time passes in reality, then the simulation must perform several of these time steps to keep up. How exactly time is subdivided is governed by the

```
NxScene::setTiming( );
```

method. These settings can also be set in the descriptor of the scene prior to construction. The default settings are appropriate for most simulations.

To actually compute the simulation for a certain amount of time, the following code is used:

```
scene->startRun(elapsedTime);
scene->flushStream();
scene->finishRun();
```

Here, `elapsedTime`'s worth of simulation is computed, eventually subdividing this time into several supsteps, in accordance with the timing settings discussed above.

Three commands are needed because some implementations of the SDK may run in a multithreaded or otherwise parallel environment. `NxScene::startRun()` is simply the command to perform the simulation, but on a parallel implementation it will return before the simulation is actually performed. In fact, it may happen that several commands, including `startRun()`, simply get buffered up into a command stream. This command stream is only sent to be executed when the buffer is full, for efficiency reasons. This implies that two things need to be kept in mind:

- The results of an operation such as `NxActor::setGlobalPose()` will not immediately become available, i.e. a following call to `NxActor::getGlobalPose()` may return the unchanged pose. Only when the next call to run completes (see `finishRun()` below) do settings definitely take effect.
- The command stream needs to be explicitly flushed using `NxScene::flushStream()` to avoid the simulation stalling when the buffer isn't quite full yet, but the application is waiting for a command to complete before issuing more commands.

The `NxScene::finishRun()` call blocks until the preceding `startRun()` call is complete, and makes the simulation's results available to the user's queries.

5.1.2 Asynchronous Simulation

If an implementation of the SDK is capable of performing the simulation in parallel with the user's program, the above method of calling `finishRun()` right after `startRun()` is inefficient, because the user's process will be forced to wait until the simulation completes. It is more efficient to try to use the time efficiently by performing tasks which do not depend on the simulation results:

```
/*submit command to start simulation */
scene->startRun(elapsedTime);

/*make sure that simulation computations are started */
scene->flushStream();

/*do some processing unrelated to physics*/
pollUserInput();
updatePositionalAudio();
etc();

/*wait for physics to complete*/
scene->finishRun();
/*render the objects at their new positions*/
```



```
renderActor(actor->getGlobalPoseReference());
```

It is even better if the application has some tasks it can optionally perform while waiting for physics to complete. In this case it can check whether the finishRun() function will block or not:

```
/*do some processing unrelated to physics*/
pollUserInput();
if (!scene->wait(NX_FENCE_RUN_FINISHED, false))
    updatePositionalAudio();

/*wait for physics to complete*/
scene->finishRun();
```

Note: The version 2.1 implementation of the SDK is not yet actually parallelized.

5.2 Actors


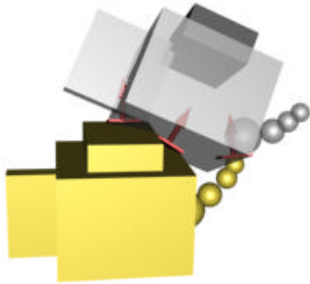
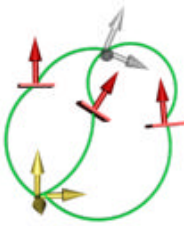
Actors are the protagonists of the simulation. In the current version of the SDK, actors have two basic roles: They can be either static objects, fixed in the world reference frame, or dynamic rigid bodies. We have not used the word “body” to describe all of them because later we plan on adding features such as fluids which will also be instances of class NxActor. Instead, we use body to refer to rigid bodies, i.e. all dynamic actors in the current version.

One important aspect of actors is that they can have shapes assigned to them. Collision detection insures that the shape(s) of one actor do not intersect with another. Shapes can also be used as triggers, to trigger various behaviors when another shape touches them. Shapes will be discussed in detail in chapter 5.4.

Static actors’ primary purpose is collision detection, so they should always have shapes assigned to them. Dynamic actors, on the other hand, may represent abstract point masses, which may be connected with joints to form pendulums and similar mechanisms. You may, but you do not have to add shapes to such bodies. Kinematic actors are a special kind of dynamic actor.

According to the laws of physics, any rigid object of any shape may be perfectly represented by a point mass located at the shape’s center of mass and an inertia tensor. This is the approach used by the NovodeX SDK and any other rigid body dynamics library. The SDK will optionally compute the inertia properties of the bodies using the assigned shapes, or the user can supply custom parameters.

The below images show three different ways that the same scene is represented: graphical, collision detection, and dynamics.

	
<p>The graphical representation of a scene with two stacked teapots. This is what the user would see. The graphics engine works with the mesh and texture data needed for such a representation. This graphical representation is optionally provided by the user and is not included in the Physics SDK.</p>	<p>The same scene, from the perspective of the SDK's collision detection functions. In this case the teapots are approximated with a set of bounding volumes. It is also possible to use a more accurate mesh representation. The bodies' inertia properties, and the contact points between the bodies are computed from this representation.</p>
	
<p>Finally, the representation of the scene used for dynamics. The contacts provided by collision detection are associated with the frames representing the bodies' masses.</p>	

5.2.1 Creating Actors

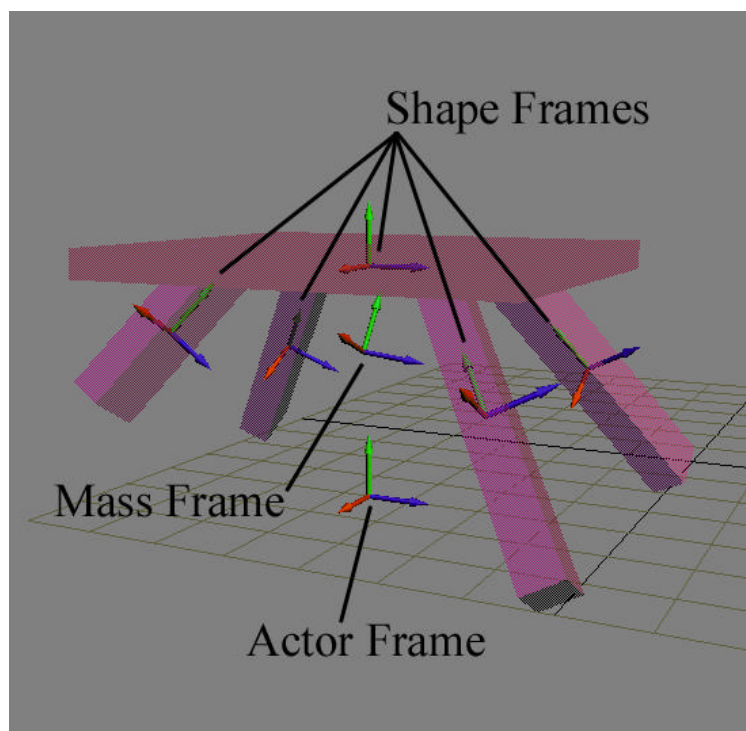
To create an actor, you must call the `createActor()` method of the scene object that is to contain the actor. You must, as usual, pass an actor descriptor. All there is to specify is the position and orientation of the actor in the scene (`NxActorDesc::globalPose`), an optional density, and its dynamics and surface material properties, and any eventual collision detection shapes. All these settings have working default values. As we discuss the features of the actor, keep in mind that many of the settings can be changed either via the descriptor or via a `set*()` method.

5.2.2 Reference Frames

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

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

We assume the common practice that both high detail geometry and either hand placed or automatically generated bounding boxes for the table top and legs are exported from the 3d software. The physics SDK does not concern itself with the graphical representation, but the bounding boxes will be instantiated in the SDK as shape objects. The below image shows what the table may look like in the game editor. The reference frames that we would like to discuss are also shown.



Note that to make things more interesting, we have made a table where someone has cut off a large chunk 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 world frame is represented by the image by the ground plane grid.

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 it is placed at zero height in the world, the legs of the table touch the floor. This makes 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 arbitrarily placed as far as the physical behavior is concerned, and are usually chosen with other practical or artistic considerations in mind.

The table is made up, as far as collision detection is concerned, out of five bounding box shapes. These have fixed poses relative to the actor frame. Each box has at its origin (defined to be its center) its own local frame called the shape frame, oriented along the primary axes of the box. Other types of shapes also 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 the shapes, the SDK computes the center of mass of the actor, assuming it is dynamic. (Or you may specify a center of mass yourself.) The center of mass is the point around which real physical objects rotate by nature, unless they are constrained in some way. The center of mass may also 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, and this is 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 also 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& getGlobalPoseReference() const
```

is available for fast retrieval of an actor's transform for rendering purposes. This can then be sent to OpenGL (or similarly to Direct3D) like this:

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

The methods

```
void setCMassLocalPose(const NxMat34&);
```

```
void setCMassLocalPosition(const NxVec3&);
void setCMassLocalOrientation(const NxMat33&);
```

set the pose of the mass frame relative to the actor frame, or in other words, determine the center of mass of the actor. This is computed and set by default by the SDK to the correct values based on the mass distribution of the actor's shapes.

To set the pose of the component boxes (or other shapes) relative to the actor frame, one may use the below methods of NxShape:

```
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 positions of all the bodies involved, the simulation would continue without problem. However, if you changed the position of only one of the bodies, 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 well lead to the simulation failing or the scene 'exploding'. To summarize, setting the pose of objects directly is not necessarily a bad thing, you just need to watch out that you don't accidentally pull joints apart or put bodies into each other – which may not be appreciated by your collision detection system.

Note that there is no performance difference for using either quaternion or matrix representation to specify the orientation, because the SDK uses both formats internally.

5.2.3 Rigid Body Properties

Dynamic actors have a set of special properties that are used for rigid body simulation. First, they have a center of mass position and orientation, as discussed in the last section. At this point a certain mass is concentrated, and its spatial distribution is expressed using the inertia tensor. The rate of change of the position of the center of mass (and because this is fixed inside the actor, also of the actor itself) is expressed via the body's velocity. Finally, forces may act on the body to produce the motion, as long as this is not prevented by colliding shapes or joints.

Note that all 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)

position (3-vector) the position of the center of mass of the body relative to the actor's frame. velocity (3-vector) force (3-vector) the amount of force that is being exerted on this body.	describes the mass distribution of the body along its three dimensions. orientation (quaternion or 3x3-matrix) the orientation of the principal moments relative to the actor's frame. angular velocity (3-vector) torque (3-vector) the amount of force that is being exerted on this body.
---	--

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

5.2.4 The Inertia Tensor

Being the most unusual aspect of a body for those without dynamics experience, the inertia tensor warrants some additional explanation. As we stated above, its purpose is to describe the mass distribution of the body, which may not be available in any other way, since we do not always store the actual shape. Even if shapes are available, sometimes we wish to use mass settings which are not strictly compatible with the shapes, perhaps 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 in the middle, it would be much more difficult to swing it around (especially if you push it near the middle and not at the ends, which gives you a great deal of leverage).

You will find formulas in high school physics books that 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 actually decomposed into a rotation and a vector. This is both faster and more intuitive to use than the dense matrix because the rotation is used to give 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 is to assign collision detection shapes to the actor and 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 eventually precomputed them earlier with the previous method), which is more efficient. The SDK contains a wide selection of utility functions to compute and manipulate inertia tensors. In particular see the files `NxUtilities.h`, and

NxVolumeIntegrals.h in the Foundation SDK, and NxInertiaTensor.h in the Physics SDK.

The NxActor methods

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

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, and as a result this tensor changes every time the actor moves.

5.2.5 Sleeping

When a body does not move for a period of time, it is assumed that it will not move in the future either until some external effect acts on it that throws it out of equilibrium. Until then it is no longer simulated in order to save time. This state is called sleeping. There are actually two flavors of sleep: individual and group. (A group is a set of mutually connected bodies, such as those in a big pile or tower.) If a body is not asleep, its group won't be either, but if it is asleep, its group may not be. You can query a body's sleep state with the methods:

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

A Body goes to sleep when its angular and linear velocities are below certain thresholds for a certain time. These thresholds can be set using the methods:

```
void      NxActor::setSleepLinearVelocity(NxReal);
NxReal    NxActor::getSleepLinearVelocity();
void      NxActor::setSleepAngularVelocity(NxReal);
NxReal    NxActor::getSleepAngularVelocity();
```

If for a body these settings are not changed, they default to the global parameter settings of NX_DEFAULT_SLEEP_LIN_VEL_SQUARED and NX_DEFAULT_SLEEP_ANG_VEL_SQUARED. These can be set and retrieved using the methods NxPhysicsSDK::setParameter() and getParameter() respectively.

Because the object automatically wakes up when an awake object either touches it, or the user changes one of its properties, the entire sleep mechanism should be transparent to the user. If for some reason you need to explicitly wake up a sleeping object, or force an object to sleep, use:

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

5.2.6 Applying Forces and Torques

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

$$f = m a \quad (\text{force} = \text{mass} * \text{acceleration})$$

directly control a body's acceleration, but its velocity and position only indirectly. For this reason control by force may be inconvenient if you need immediate response. The advantage of forces is that regardless of what forces 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 a force to bodies.

Unfortunately applying large forces to articulated bodies at the resonance frequency of the system may lead to ever increasing velocities in the system, 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 in this case.

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 what they all do:

```
void setForce(const NxVec3 &);
void setTorque(const NxVec3 &);
void addForceAtPos(const NxVec3 & force, const NxVec3 & pos);
void addForceAtLocalPos(const NxVec3 & force,
    const NxVec3 & pos);
void addLocalForceAtPos(const NxVec3 & force,
    const NxVec3 & pos);
void addLocalForceAtLocalPos(
    const NxVec3 & force, const NxVec3 & pos);
void addForce(const NxVec3 &);
void addLocalForce(const NxVec3 &);
void addTorque(const NxVec3 &);
void addLocalTorque(const NxVec3 &);
```

5.2.7 Gravity

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

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

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

5.2.8 Setting the Velocity

If you need to immediately get a body moving in a certain direction, you can set its velocity. It is also possible to set its momentum, which may be more convenient if you don't know what the mass of the body is, but want its speed to depend on it.

If you set the velocity of a body, and then run the scene it is in, you may notice that after the simulation for the time frame is complete, the velocity of the body is no longer what you have specified. This is because the simulation overrides your settings if they are in conflict with a constraint that you have specified. For example, if a ball is resting on a table and you give it a downward velocity, this will be clamped back to 0. If you set the velocity of a chain link that is jointed to a bunch of other chain links, the velocity of the chain link you set will be diminished, and the other chain links' velocity will be increased, to permit the chain to stay together.

The methods of `NxActor` that help you 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 that in the current version, setting the velocities / momenta of bodies joined together with joints of simulation type `Reduced` does not have an effect. In this case you should use a joint motor to get the desired relative velocity.

5.2.9 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, which is then converted to a very high rotational velocity when they start to rotate 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 an per body basis:

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

5.2.10 Static Actors

Static actors are actors which do not have any of the dynamic properties discussed above. You create a static actor by letting the body member be NULL in the Actor descriptor when creating the actor.

Once the static actor is created, you should 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: First, 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 recomputation of such data structures. Second, the code driving dynamic actors and joints is also written with the assumption that static actors will not move or be deleted for the duration of the simulation, which also 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, and they will levitate in midair. Even if they are woken up, the collision response between moving statics and dynamic actors is of low quality.

This doesn't mean that you cannot have all the behaviors you'd expect to have if movable static actors were supported. We have provided so-called kinematic actors that do just what you would expect. Read about them in the next section.

5.2.11 Kinematic Actors

A kinematic actor is a special kind of dynamic actor that is not moved directly by the user and not by the SDK. This means that it indeed does not move in response to user forces, gravity, collision impulses, or if it is tugged by joints. They also don't have a real velocity or momentum.

Instead, the user can move it around the scene by setting a slightly different position using the `moveGlobal*()` functions. (Note: don't use the `setGlobal*()` functions for this, that will lose you many of the advantages that kinematic objects provide.) They appear to have infinite mass and will push regular dynamic actors out of the way.

The move command will result in a velocity that, when successfully carried out inside `run*()` (i.e. the motion is not blocked due to joints or collisions), will move the body into the desired pose. After the move is carried out during a single time step, the velocity is returned to zero. Thus, you must continuously call this in every time step for kinematic actors so that they move along a path.

The move functions simply store the move destination until `run*()` is called, so consecutive calls will simply overwrite the stored target variable.

To create a kinematic actor, simply create a dynamic actor, and 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.

Note that if a dynamic actor is squished between a kinematic and a static or two kinematics, then it will have no choice but to get pressed into one of them. Later we will make it possible to have the kinematic motion be blocked in this case. Also, kinematic actors will only collide with true dynamic actors, but not with statics or dynamics. This may also change in the future.

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

You can not connect Reduced joints to kinematic actors. Lagrange joints work ok if the platform is moving with a relatively low, uniform velocity.

5.3 Joints

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

Joints and contacts (see below) are both called constraints, because they constrain the motion of a body. The SDK only supports pair wise constraints. In other words, each joint or contact is between exactly two actors. However, one of these actors may be the implicit “world” actor, which represents the unmovable 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. The reason is that connecting two static actors which can’t move anyway would be senseless.

Joints are created by calling the `NxScene::createJoint()` method. This method takes as a parameter a joint descriptor of a certain type that contains all the information needed to create the joint.

The simulation cost of a joint is determined by the number of degrees of freedom it removes from the system. (See a physics book for a definition on degrees of freedom.)

Below we describe the different types of joints, and give the number of degrees of freedom (DOFs) they remove.

5.3.1 Spherical Joint



A sphere joint is the simplest kind of joint. It constrains two points on two different bodies to coincide. This point, specified in world space (this guarantees that the points coincide to start with) is the only parameter that has to be specified.

An example for a common spherical joint is a person's shoulder. Of course it is quite limited in its range of motion. (See limits below.)

DOFs removed: 3 DOFs remaining: 3

5.3.2 Revolute Joint

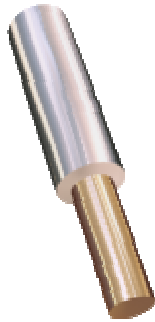


A revolute joint removes all but a single rotational degree of freedom from two objects. The axis along which the two bodies may rotate is specified with a point and a direction vector. The point along the direction should not matter in theory, but it should be preferably close to the area where the bodies contact.

An example for a revolute joint is a door hinge.

DOFs removed: 5 DOFs remaining: 1

5.3.3 Cylindrical Joint



A cylindrical joint permits relative translational movement between two bodies along a single axis, and also relative rotation along that axis.

An example for a cylindrical joint is a telescopic radio antenna.

DOFs removed: 4 DOFs remaining: 2

5.3.4 Prismatic Joint

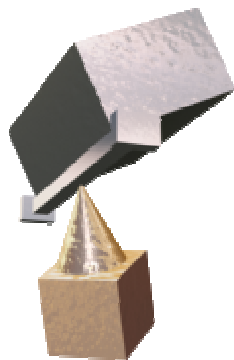


A prismatic joint permits relative translational movement between two bodies along an axis, but no relative rotational movement at all.

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

DOFs removed: 5 DOFs remaining: 1

5.3.5 Point On Line Joint

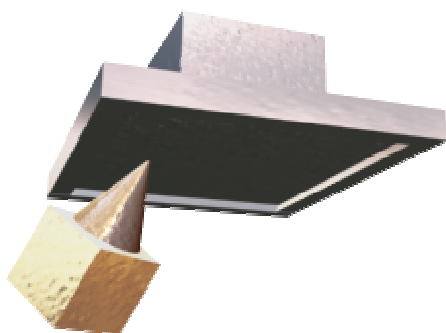


A point on line joint constrains a point on one actor to only move along a line attached to another. The starting point of the point is defined as the anchor point. The line through this point is specified by its direction (axis) vector.

An example for such a joint is a curtain hanger widget.

DOFs removed: 2 DOFs remaining: 4

5.3.6 Point In Plane Joint



A point in plane joint constrains a point on one actor to only move inside a plane attached to another. The starting point of the point is defined as the anchor point. Its normal vector specifies the plane through this point.

An example is a magnet on a refrigerator.

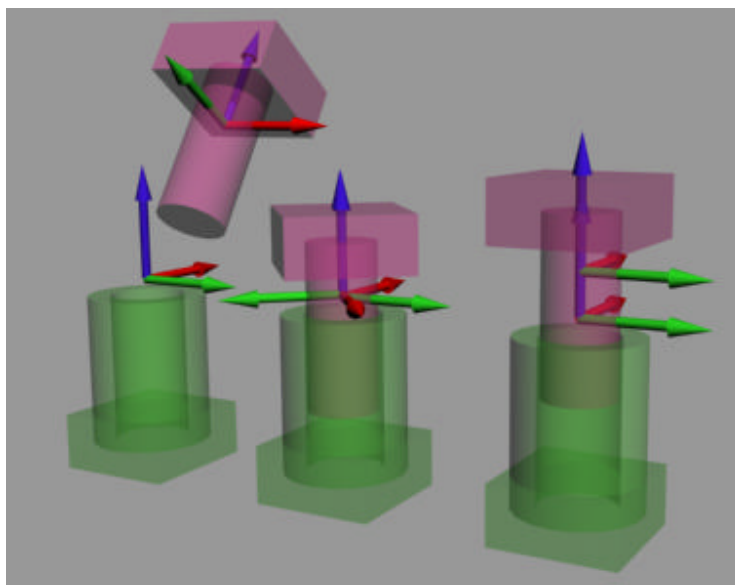
DOFs removed: 1 DOFs remaining: 5

5.3.7 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) has a so called joint frame that is specified relative to its actor frame. The two frames are each stored in `NxJointDesc` as two basis vectors and point:


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

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



These two frames uniquely identify the geometrical placement of the joint at any point in the simulation. These frames never change because they are fixed relative to the respective actors, but rather the joint ‘moves’ by the connected actors moving and thus these frames, when transformed to the world space via the actor’s transform, end up in different places. The above image shows joint frames for a cylindrical joint. In the leftmost 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.

The joint frame may actually be specified in world space, which is often the most intuitive. Use the NxJoint or NxJointDesc methods `setGlobalAxis()` / `setGlobalAnchor()` to specify the frames in world space. They will simply be transformed to the local spaces of the actors internally.

The local spaces 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 do not match up in world space anymore. It is important to let the user have access to this error, like all internal state, so that he is for example able to save it and restore it at a later time, and have the simulation proceed exactly at the state it has left off.

Not all joint types would really need an entire frame to specify their geometry. For example, a simple spherical joint would only need 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 do use 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 for the different joint types about how they interpret the basis vectors.

5.3.8 Joint Simulation Method

Currently the SDK implements two methods for simulating joints: Reduced mode and Lagrange mode. Reduced mode was introduced to make the simulation of joints that remove many degrees of freedom, such as prismatic and revolute joints, more accurate.

The reduced method has no joint error, but it has some disadvantages such as the slower speed and some instability when the joints build long chains. The user is encouraged to try both joint implementations for these types of joints.

The joint mode can be set independently for each joint using the `NxJointDesc::method` member.

5.3.9 Breakable Joints

Joints simulated using the Lagrange method can set to be breakable. This means that if the joint is tugged on with a large enough force, it will snap apart. Setting up a joint to be breakable is very simple: Just 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` member is the resistance to linear pulling forces, while the `maxTorque` resists twisting. The exact behavior depends on the type of joint. It is also 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 such an object using the `NxScene::setUserNotify` method. Further, the joint is put into a 'broken' state, which basically deactivates it. At this point the user may reactivate the joint (for example with a `loadFromDesc()` call) or, more probably, eventually just ask for it to be deleted. The state of a joint may be retrieved using `NxJoint::getState()`.

5.3.10 Joint Limits

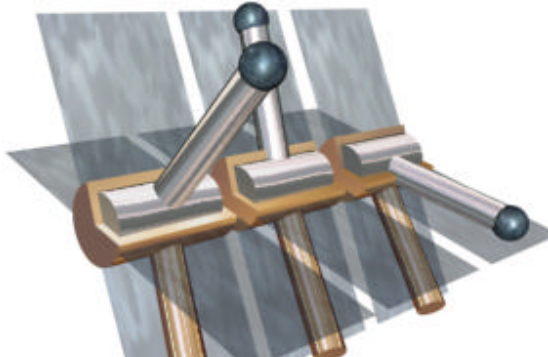
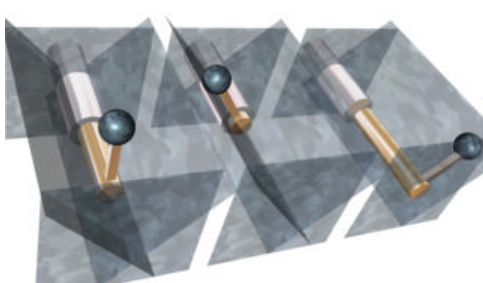

In the real world, most joints can only move in a certain range, usually because the hinges' own construction gets in the way. For example, a door hinge can only move in one direction until it 'straightens out' at the 180-degree position, and can perhaps reach the 20-degree mark in the other. A more complex anatomic joint like a person's shoulder, which we have mentioned is a sort of 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 is general works with all the different joint types. The disadvantage of the generality is that it is less intuitive to use than the other, joint specific

approach, which is provided only for spherical and revolute joints. We plan on adding custom joint limits to the other joint types in the future.

The special methods simply consist of having to specify a minimum and maximum rotational angle around a certain rotation axis. For revolute joint this is the joint axis. For spherical joints there are two different limits: A twist limit and a swing limit.

Now we describe the general method for creating joint limits. You may elect a point attached to one of the two bodies to act as the limit point. You may also specify 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. This model is actually very simple once you understand how it works. It is best to look at the following pictures:

	
<p>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.</p>	<p>Slider joint with four limit planes, and two possible extreme positions. The planes are specified to move with the silver actor.</p>
<div style="text-align: center;">  </div> <p>A point on line joint limited with two planes which move with the silver actor.</p>	

See the API reference documentation of 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);
```

5.3.11 Joint Motors, Springs, Projection, and Special Limits

The spherical joint and the revolute joint now include some new properties. You can see the descriptions of these methods in their respective descriptors' API documentation. To summarize:

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 just control the speed that is to be maintained. The monster truck demo uses revolute joint motors to drive all four wheels.

Joint springs are similar to joint motors. Here you can set up a spring and a damper constant that resist the motion of the joint DOF that they are applied to.

Joint projection is a method to combat the joint drift of joints solved using the Lagrange method. You can basically specify the maximum joint error you wish to allow before projection kicks in.

Finally, special angular limits may be defined for these joints that are much easier to use than the general limits described in the previous section. We plan to add these functions also to the other joint types.

5.3.12 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 class `NxSpringAndDamperEffector`. This element exerts a force between two bodies, proportional to the relative positions and/or the relative velocities of the bodies.

You 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 you need to 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, position1, actor2, position2);
```

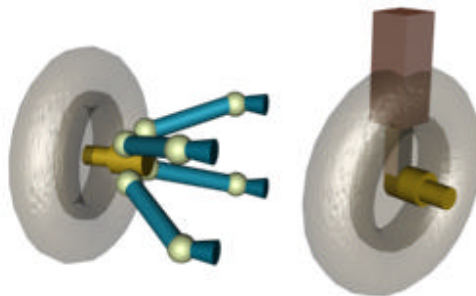
When the above call is made, the effector reads the bodies' poses so that it can remember what the relative position is when the spring is relaxed. At this initial position, no spring force is applied. When you then stretch or compress the spring by moving the bodies relative to each other, a force will 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.

Note that this effector element is not as numerically stable as the `NxSpringDesc` spring elements built into certain types of joints.

5.3.13 Modeling with Joints

When you are looking for the appropriate joint to model a real world machine, making an exact copy of the real thing may be possible, but it is almost never the most optimal way in the simulation. This is because the engineers who design real world things have constraints you do not have in the 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 create a workaround and use additional joints to lead the suspension mechanism around the wheel. In the virtual simulation this is not a problem. Also, in the real world, perhaps a particular asymmetric construction is impractical because it leads to added friction or less stability. In the simulation, you don't have friction if you don't want it. On the other hand, 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 next picture of a real world car suspension assembly, versus a virtual approximation. They are quite different. The real thing also looks much more “realistic” naturally. For this reason you may find it useful to create the real assembly as a graphical prop which is displayed and animated in your application in the place of the simplified dynamics scheme, which 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.

5.4 SDK Parameters

The Physics SDK has a set of centrally managed, global parameters. These parameters 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. The specific set of parameters are documented in the API reference. We describe an important one here in detail.

5.4.1 Min Separation For Penalty

The simulation deals with inaccuracy when stacking objects by letting them slightly interpenetrate each other. The amount of permitted interpenetration can be regulated using the NX_MIN_SEPARATION_FOR_PENALTY parameter. Clearly, a lot of interpenetration is visually undesirable. On the other hand, forbidding interpenetration 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 interpenetration that is best permitted depends on many things such as the size of the objects involved (so that the interpenetrating 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.) Note that the NX_MIN_SEPARATION_FOR_PENALTY constant is provided to the SDK as a separation value rather than a penetration, which is why this should be negative. (This is just an arbitrary convention.)

6 Collision Detection

Many possible applications of dynamics are not concerned with the forces that occur when bodies collide or touch. Examples are the simulation of a rocket taking off, or a cannon ball flying in an arc due to gravity. On the other hand, most interesting applications need to concern themselves with collision detection. For example, if you want the cannon ball to bounce on the ground or an articulated body to fall down stairs, you need to provide the SDK with the shapes of the bodies that can touch, and also the properties of the surfaces – for example how slippery they are. This collision detection functionality can range from very simple (in the case of the cannon ball, you would only need to test if the ball is above or below ground level or not) to quite complex (in the case of the articulated body falling down the stairs).

6.1 Shapes

The SDK lets you create shapes – these are objects with occupy a well-defined volume in space. In section 5.2.2, we have already introduced, with the help of the table example, 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;
//we will 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); //the head's position relative
to the mace's origin
handle.radius = 0.01f; //handle has a radius of 1 cm
handle.height = 0.50f; //the handle is 50 cm long along its y
axis
handle.localPose.t.set(0,15,0); //the handle's position
relative to the mace's origin

desc.globalPose = ... //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.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 that we set above can also be accessed and changed using the NxShape::setLocalPose, etc, methods.

Next we discuss the different types of shapes.

6.2 Sphere

Spheres (class NxSphereShape) are the simplest types of shapes. Their only property is their radius.

6.3 Box

Boxes (class NxBoxShape) have a dimensions property. The elements of this vector are the volume's half width, half height, and half depth, respectively.

6.4 Capsule

Capsules (class NxCapsuleShape) are also called line swept spheres. They are like cylinders, only with rounded ends. Capsules are great to approximate round shapes like body parts because the collision detection with them is very fast and robust, even more so than boxes. The below image shows a character whose bounding boxes are made using capsules.



6.5 Triangle Mesh

With a triangle mesh you can represent complex shapes which are either too complex to represent with a bunch of boxes and spheres, or the people who author your simulation data do not want to spend time to tediously fit these bounding boxes.

Our triangle mesh code has no limitations: the mesh doesn't have to be convex, or even closed. The mesh-mesh collision query is also surprisingly fast. The common practice of storing a single triangle mesh, and then creating several differently posed instances of it is mirrored in the SDK.

The following programming example is taken from the "samplePMap" demo. The first step is to make your mesh data accessible to the SDK. The source data for our bunny triangle mesh is shown below. You can use a somewhat different format as well.

Differences may include different vertex and index strides, clockwise or counterclockwise face winding, and 16 or 32 bit indices.

```
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,
    ...
};
```

You provide this basic triangle data to the SDK by using the NxTriangleMeshDesc structure:

```
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;

NxTriangleMesh * bunnyTriangleMesh;
bunnyTriangleMesh = gPhysicsSDK->createTriangleMesh(bunnyDesc);
```

Finally the triangle mesh is created using createTriangleMesh(). Note that the flags member above is zero, but it can be used to indicate a special formatting, such as:

NX_MF_FLIPNORMALS – Your triangle's winding is counterclockwise. (see below)
 NX_MF_16_BIT_INDICES – The triangle's vertex indices are 16 instead of 32 bits.
 NX_MF_CONVEX – The mesh is convex. This permits certain optimizations.

Note the triangles' normals are not passed explicitly, and are assumed to be:

$$(\mathbf{v1} - \mathbf{v0}) \times (\mathbf{v2} - \mathbf{v0})$$

where v0, v1, and v2 are the vertices of the triangle, indexed in that order. Note that this is equivalent to ‘clockwise’ triangle winding in a graphics system that has the camera’s direction axis pointing forward into the scene. If this is not how you store your triangles, you should set the NX_MF_FLIPNORMALS flag:

```
meshDesc.flags |= NX_MF_FLIPNORMALS;
```

One feature important enough to at least mention right here is the pmap. This is a rather heavy weight data structure that the SDK uses for mesh-mesh collision detection. Basically, if you only intend to collide simple shapes like planes, boxes, spheres, and capsules against the triangle mesh, you don’t need a pmap. However, you will most likely want to have a pmap if the mesh is going to be colliding against other meshes – pmap-less mesh-mesh collision detection works, but is not very robust. In this case you need to assign the pmap to the triangle mesh object either via the pmap member of the mesh descriptor, or after creating the mesh like this:

```
bunnyTriangleMesh->loadPMap(gBunnyPMap);
```

The creation and management of pmaps is discussed in section 6.5.2.

When creating the NxTriangleMesh, the vertex and index data is copied, so the user is free to delete its own data. It is not possible for the SDK to share mesh data with the user because the internal representation is optimized and converted to a format that is more suited to collision detection than the classic layout used for graphics. Thus, if your mesh data changes for any reason, you should have it reloaded using the NxTriangleMesh::loadFromDesc() call. Note that because pmaps can take quite some time to compute, they are not suitable for changing meshes.

We are now ready to start creating 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 just 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.material = gDefaultMaterial;
actorDesc.density = 1.0f;
NxActor* newActor = gScene->createActor(actorDesc);
```

Some notes about 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 you have two triangles 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 a decreased performance and unreliable results.
- Also avoid t-joints and non-manifold edges for the same reason. (A t-joint is a vertex of one triangle that is placed right on top of an edge of another triangle, but this second triangle is not split into two triangles at the vertex, as it should. A non-manifold edge is an edge (a pair of vertices) that is referenced by more than two triangles.)
- To precompute the inertia tensor, mass, and center of mass of a mesh, you can use the `NxComputeVolumeIntegrals` function in the Foundation SDK. This is the same function that also gets used internally if you do not provide an inertia tensor. Note that 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!
- It is possible to assign a different material to each triangle in a mesh. See section 6.13.1 for details.

6.5.1 Sphere-Mesh Collision Detection

When performing collision detection between spheres and meshes, you can choose between two algorithms: 'normal' and 'smooth'. The 'normal' algorithm assumes that the mesh is composed from 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 to smooth the triangles' vertex normals (which in this case are particularly important). This way the rolling sphere's velocity will change smoothly over time, instead of suddenly. We recommend this algorithm for simulating car wheels on a terrain. You can 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.

6.5.2 PMaps

We already mentioned that pmaps are a supplementary data structure for triangle meshes that serve to make the mesh-mesh collision detection more robust. Basically, they voxelize the mesh so that inside-outside queries and the like, which are time consuming to perform on a boundary representation, become fast and efficient. This voxelization process, however, has the downsides that its time consuming, and takes up quite a bit of memory. The resolution of the pmap can be controlled by the user.

The meshes used for pmap generation should be closed volumes. A ray coming from the outside is assumed to enter into the mesh volume when it hits a triangle, and exit the volume when it later hits another. Also, meshes should not contain any very fine geometry, such as thin spikes, because these may be missed by the raycasting process used to create the pmap.

Because pmap generation is time consuming, it is best to precompute and save them, and then load them from disc when needed. Below is an example from the SamplePmap demo showing how a pmap can be created for the bunny mesh:

```
NxPMap gBunnyPMap;
FILE* fp = fopen("bunny.pmap", "rb");
if(!fp)
{
    // Not found => create PMap
    printf("please wait while precomputing pmap...\n");
    if(NxCreatePMap(gBunnyPMap, *bunnyTriangleMesh, 64))
    {
        // The pmap has successfully been created,
        // save it to disk for later use.
        fp = fopen("bunny.pmap", "wb");
        if(fp)
        {
            fwrite(gBunnyPMap.data, gBunnyPMap.dataSize, 1, fp);
            fclose(fp);
            fp = NULL;
        }
        // Release data and go on
        //(we have to release the memory allocated in the SDK)
        NxReleasePMap(gBunnyPMap);
    }
    printf("...done\n");
}
```

First we try loading the pmap from disk. The format of the pmap is only known to the user as a block of binary data of a certain size. This can be read and written to disc as desired. If the file on disc is not found, the SDK is asked to create the pmap from the bunnyTriangleMesh object (introduced in section 6.5). The resolution of the pmap voxel block is given as 64^3 . Note that you only have to call the NxReleasePMap() function if you need to release a pmap created by the SDK using the NxCreatePMap() call. If you instead 'created' the pmap object yourself by loading it from disc, you are the owner of it, and you are responsible for releasing its memory. The below code shows how you could load the pmap data from disc:

```
fp = fopen("bunny.pmap", "rb");
if(fp)
{
    gBunnyPMap.dataSize = getFileSize("bunny.pmap");
    gBunnyPMap.data = new NxU8[gBunnyPMap.dataSize];
    fread(gBunnyPMap.data, gBunnyPMap.dataSize, 1, fp);
    fclose(fp);
}
}
```

Using the pmap for collision detection is quite trivial: All you need to do is assign it to the triangle mesh that it was created from:

```
if(gBunnyPMap.data)
    bunnyTriangleMesh->loadPMap(gBunnyPMap);
```

6.5.3 Convex Meshes

It is often practical to use shapes that are more complex than a box, yet the data and preprocessing overhead of PMaps is often inconvenient for small, simple meshes. Furthermore, PMap based mesh-mesh collision detection is optimized for smooth, highly tessellated meshes, but inaccurate for meshes that are on the order of a few dozen polygons.

The convex mesh feature is meant to satisfy this need. The only requirement for convex meshes is that they should be convex! If you know that this is the case, then all you need to do is tell the SDK:

```
meshDesc.flags |= NX_MF_CONVEX;
```

If you omit the pmap, you will still get robust collision detection between convex meshes. Note that a robust algorithm for nonconvex mesh-convex mesh collisions is not yet in place, unless a pmap is used for both shapes.

6.6 Plane

Planes (class `NxPlaneShape`) are useful for simple demos where we need a ground plane, but little else. A plane is characterized in the world space plane equation $ax + by + cz = d$. Planes are static and cannot be assigned to track an object; the plane equation is always in world space and the shape's relative transform is ignored. You can move the plane 'manually' by changing the plane equation though.

The plane represents a half space rather than an infinitely thin shape. Thus, all points behind a plane are treated as contacting it. This way when you drop an object onto a plane, it will never pop through it and continue falling.

Because planes are assumed to be static objects, they never report contacts with other planes.

6.7 Height Field

A height field is like a bumpy plane. It is also infinite in two dimensions, and defines an elevation along the third. All values below the elevation (down until a user definable depth) are treated as being inside the height field volume. The height field implementation of the SDK is based on the triangle mesh implementation. It is possible for the height field triangulation to be irregular. In fact the only difference between height fields and general triangle meshes is that a height field is not a closed shape, but rather it is assumed to be the boundary of a volume that extends downward for a certain distance. This way fast moving objects will not fall through the terrain if they travel across its surface in a single time step, which could happen for general triangle meshes.

You provide the triangles for the height field in exactly the same way as for 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;

```

Height field geometry must be 'flat' in the sense that the projections of all triangles onto the height field plane must be disjoint. (If the height field vertical axis is Y, the height field plane is spanned by X and Z.) You need to set which local (vertex-space) axis should be the height axis, and how far along this axis the implicit height field volume should extend:

```

terrainDesc.heightFieldVerticalAxis = NX_Y;
terrainDesc.heightFieldVerticalExtent = -1000.0f;

```

In this way even objects which are under the surface of the height field but above this cutoff are treated as colliding with the height field. The heightFieldVerticalExtent has to be outside of the vertex coordinate range of the mesh along the heightFieldVerticalAxis. You may set this to a positive value, in which case the extent will be cast along the opposite side of the height field. You may use a smaller finite value for the extent if you want to put some space under the height field, such as a cave.

Above we picked the Y axis, other options being NX_X and NX_Z. Note that we have used a negative value for the extent, because the positive Y axis is 'up', and here we need the 'down' direction. Once this is done, the rest of the code is the same as usual:

```

NxTriangleMesh * terrainMesh;
terrainMesh = gPhysicsSDK->createTriangleMesh(terrainDesc);

NxTriangleMeshShapeDesc terrainShapeDesc;
terrainShapeDesc.meshData = terrainMesh;

NxActorDesc ActorDesc;
ActorDesc.shapes.pushBack(&terrainShapeDesc);
terrainActor = gScene->createActor(ActorDesc);

```

6.8 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 the 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. The SDK implements three different broad phase algorithms, which can be selected independently for each scene using the `NxSceneDesc::broadPhase` variable. Note however that the default setting is the best, so there is no reason to change this.

6.9 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 two shapes a and b occurs if:

```
(a->getActor()->isDynamic() || b->getActor()->isDynamic())
&& NxPhysicsSDK::getGroupCollisionFlag
    (a->getGroup(), b->getGroup())
&& (!(NxScene::getShapePairFlags(a,b) & NX_IGNORE_PAIR))
```

The first test simply means that static objects which can't move anyway do not collide against each other. We look at the last three conditions in order in the following sections.

6.9.1 Collision Groups

The first condition checks the shapes' collision group membership. All shapes can be assigned to a collision group with the 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 1, but this doesn't really come with any built in meaning. The SDK maintains a 32×32 symmetric Boolean matrix which says if 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 by calling:

```
NxPhysicsSDK::setGroupCollisionFlag(CollisionGroup g1,
    CollisionGroup g2, bool enable);
```

For example, the below call disables collisions between group 11 and group 0:

```
sdk->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.

6.9.2 Disabling Pairs

The second broad phase condition checks if a pair that is about to be collision tested has been disabled or not. You can disable any pair of actors or shapes by calling:


```
NxScene::setActorPairFlags(NxActor&, NxActor&, NxU32 flags);
NxScene::setShapePairFlags(NxShape&, NxShape&, NxU32 flags);
```

The second method is useful if you have actors made up of multiple shapes, and you only want to disable collisions between some of these shapes. In this case the only flag of interest is `NX_IGNORE_PAIR`. (We will discuss other possible flags for the `setActorPairFlags()` call later.) For example, the below call disables collisions between the two actors passed:

```
sdk->setActorPairFlags(a1, a2, NX_IGNORE_PAIR);
```

6.10 Ray Casting

Ray casting 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 objects with the mouse to checking if there is a line of sight between two characters.

There are several different functions for ray casting, each one doing something slightly different. They are all members of `NxScene` called `raycast*()`. We look at each one in turn.

- `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 in the first case, or the shape itself in the second.
- `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 returns all of the shapes stabbed by the ray, including distances and intersection points. To use these, you must provide a `NxUserRaycastReport` interface, which will be called with the information for each of the stabbed shapes.

We give an example for implementing `NxUserRaycastReport` for use with the `raycastAllShapes()` function:

```
class myRaycastReport : public NxUserRaycastReport
{
    virtual bool onHit(NxShape& shape, const NxVec3*
worldImpact, const NxF32* distance)
    {
        //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_SHAPES);

```

Note that the shapes are not guaranteed to be passed to onHit() in the order they are geometrically laid out along the ray.

6.11 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 them.

You can create triggers from any shape. Note that triangle mesh triggers count as hollow surfaces for collision detection, not volumes. 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.triggerFlags = NX_TRIGGER_ENABLE;

NxActorDesc actorDesc;
actorDesc.shapes.pushBack(&boxDesc);
NxActor * triggerActor = gScene->createActor(actorDesc);

```

Above, NX_TRIGGER_ENABLE means that all possible trigger events should be sent to the user. Alternatively, it would be possible to only select a subset of the possible flags NX_TRIGGER_ON_ENTER, NX_TRIGGER_ON_LEAVE, and NX_TRIGGER_ON_STAY.

To be able to actually receive trigger events, you need to tell the SDK where these are to be sent. Trigger events are passed to a user defined object of type NxUserTriggerReport. Below is an example implementation, plus the code to assign it to the scene containing the trigger:

```

class TriggerCallback : public NxUserTriggerReport
{
    void onTrigger(NxShape& triggerShape, NxShape& otherShape,
    NxShape::TriggerFlag 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--;
    }
    //Shouldn't go negative
    NX_ASSERT(gNbTouchedBodies>=0);
}
}myTriggerCallback;

gScene->setUserTriggerReport(&myTriggerCallback);

```

6.12 Contact Reports

The `NxUserContactReport` class has functions for passing information to the user about collisions that happen. This information can be used, for example, for playing 3d sounds or other effects in response to simulated collisions. We demonstrate this using an example:

First, we 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;

```

You then have to register this object with the SDK as before:

```

scene->setUserContactReport(&myReport);

```

The `MyContactReport::onContactNotify()` function receives the contact information for each pair of actors for which we request notifications. Currently this can only be done explicitly for each pair using `setActorPairFlags()`. The possible notify flags are `NX_NOTIFY_ON_START_TOUCH`, `NX_NOTIFY_ON_END_TOUCH`, and `NX_NOTIFY_ON_TOUCH`, besides the `NX_IGNORE_PAIR` flag already mentioned:

```

scene->setActorPairFlags(actor0, actor1, NX_NOTIFY_ON_START_TOUCH
| NX_NOTIFY_ON_END_TOUCH | NX_NOTIFY_ON_TOUCH);

```

Note how we passed a combination of all the possible notify flags. This creates a behavior similar to triggers. However, when using a trigger, one receives events when a shape passes into or leaves the volume of another shape. Here we are instead dealing with shapes which start or stop touching each other's surface. We plan on adding more contact status flags in the future to better support sounds for simulation.

The contact information provided to `onContactNotify()` is in the form of the `NxContactPair` structure, and the events argument is the type of event that has happened. This is a combination of one or more of the flags that we have set with `setActorPairFlags()`. The most interesting member of `NxContactPair` is the contact stream (type `NxConstContactStream`). This is a compressed data structure that contains detailed information about the contacts between the pair. You can use the `NxContactStreamIterator` class to read it. Please refer to the API documentation and the code for the sample `SampleContactStreamIterator` for details on these classes.

6.13 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 surfaces will bounce, or not, 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 will follow.

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:

```
NxMaterial          myMaterial;

myMaterial.restitution      = 0.7f;
myMaterial.staticFriction  = 0.5f;
myMaterial.dynamicFriction = 0.5f;
```

Here `NxMaterial` is used just like a class descriptor. Its constructor initializes the material's fields to defaults, which may then be overridden. All these members are explained in the API documentation for `NxMaterial` in detail. Once the material is specified, it is copied to the SDK's material table, in one of two ways:

```
NxMaterialIndex bouncyMatIndex = 5;
gPhysicsSDK->setMaterialAtIndex(bouncyMatIndex, & myMaterial);
```

– or –

```
bouncyMatIndex = gPhysicsSDK->addMaterial(myMaterial);
```

The first call overwrites entry five in the array with the provided material. If the array had less than five entries, then the entries between the last entry and the one specified are filled in with default materials. This way there are never any holes in the array.

The second method simply appends the material to the array, and returns the index where it was stored (the index of the last entry).

Because materials store surface properties, they need to be assigned to shapes. This is best done via the shape descriptor at creation time:

```
NxActorDesc actorDesc;
NxBoxShapeDesc boxDesc;
boxDesc.dimensions.set(4,4,5);
boxDesc.materialIndex = bouncyMatIndex;
actorDesc.shapes.pushBack(&boxDesc);
...
NxActor * myActor = gScene->createActor(actorDesc);
```

But 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.

6.13.1 Materials per Triangle

While in most cases it is sufficient to assign a single material to an entire shape, it can happen that one triangle mesh shape is used to model a large environment. In this case it may be practical to assign different materials to certain regions of the mesh.

To facilitate this it is possible to associate each triangle in a mesh with a material index. To do this, we review the mesh creation procedure as outlined in section 6.5. We have now added a material array.

```
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, as
for example returned by NxPhysicsSDK::addMaterial().

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;

//create the triangleMesh object as usual:
NxTriangleMesh * bunnyTriangleMesh;
bunnyTriangleMesh = gPhysicsSDK->createTriangleMesh(bunnyDesc);
```