SBGAMES 2006

TUTORIAL

AGEIA PHYSX

Authors:

Thiago Souto Maior Cordeiro de Farias

Daliton da Silva

Guilherme de Sousa Moura

Márcio Augusto Silva Bueno

Veronica Teichrieb

Judith Kelner

{mouse, ds2, gsm, masb, vt, jk}@cin.ufpe.br

Virtual Reality and Multimedia Research Group

Federal University of Pernambuco, Computer Science Center

CONTENTS

1. Introduction

This chapter introduces the newcommer concepts of the PhysX platform and its relation with application developing. The following sections will focus on motivation, what physics engines do, a basic architecture and related physics engines.

1.1. Motivation

In real world, objects are always interacting with others objects, each one showing different behaviors because the difference of matter like weight, smooth or rough surface, kind of material, among many others characteristics. For example, a person falling in the ground experiences a different sensation than falling in the water. Thus, the more an application incorporates physics the more realistic and interesting it will become.

Properties like friction, collision and torque added to a racing game are essential to make people feel like they are driving a real car. In first person shooter (FPS) games the simulation of people movements and realistic weapons and shoots creates a unique playing experience.

A game with embeded physics simulation requires lots of high complexity calculations, thus the easiest way available is making use of a physics engine. Because of the high demanding processing time for the calculations necessary to model real behavior, a good solution is to have a dedicated processor to perform this task; likewise, the graphical processing unit (GPU) does the task of rendering graphics.

There are two approaches addressing this matter, using a physics processing unit (PPU) from AGEIA or taking advantage of high end GPUs, like Havok FX does.

1.2. Physics Engines

A physics engine is a computer program that simulates Newtonian physics models, in order to predict what would happen in the real world in some specified situation. There are two types of physics engines, namely real time ones and high precision ones. High precision physics engines require lots of calculations to reflect real life faithfully. They are required by scientific applications and computer animated movies. For games, physics simulations need to be calculated in real time while a game is played, so the physics models applied are simplified to achieve that requirement.

There are open source, as well as commercial physics engines available to use jointly with a graphics engine, making games development easier with a good game play and a behavior of the actors and other objects near to real.

The basics beyond the physics engines are the modeling of physical behaviors using mathematical equations, building simulation algorithms to find out how the system changes over time, and executing it in a computer.

Usually physics engines available deal with collision detection and rigid body dynamics, because coping with fluids, cloths, particles and flexible body dynamics requires too much processing power, making real time execution unfeasible.

1.3. AGEIA PhysX SDK

Formerly known as NovodeX Physics SDK, the AGEIA PhysX SDK is a powerful middleware physics software engine for creating dynamic physical environments on all major game platforms and is the first asynchronous (multithreaded) physics API capable of unleashing the power of multiprocessor gaming systems. Its comprehensive tools and technology enable advanced gaming physics across all next-generation gaming platforms.

PhysX SDK supports game development life cycle – add physics to game objects through the PhysX Create which has plug-ins for both 3DSMax and Maya, preview and tuning of PhysX contents using PhysX Rocket and make use of the PhysX VRD, a real-time visual remote debugger, to interactively debug the code running in a PC, Xbox 360 or PS3. These PhysX components can be viewed in Figure 1.

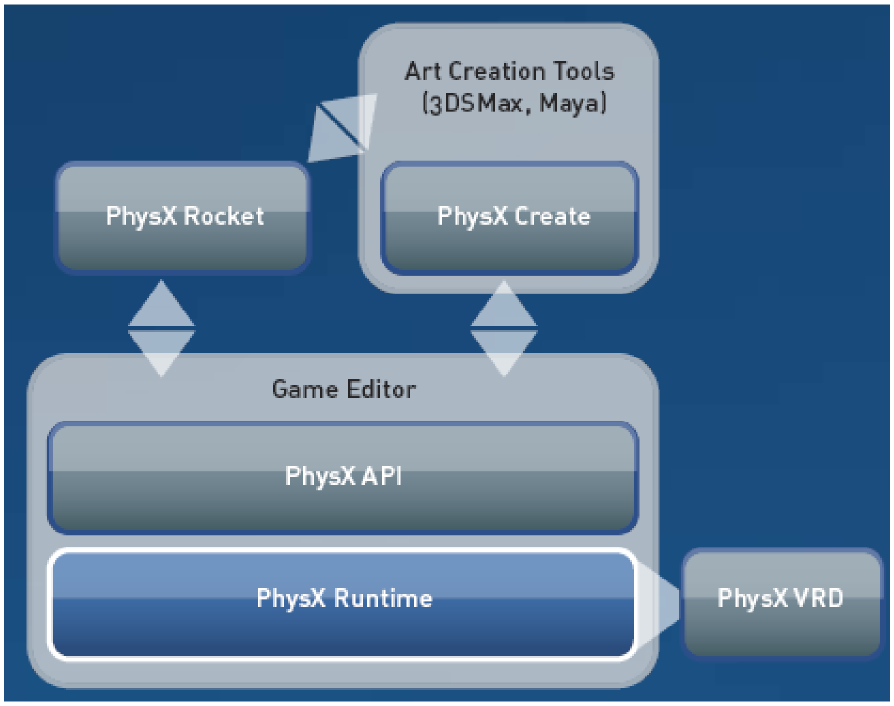

Figure 1. PhysX SDK components.

Figure 2 shows the architecture of a game application developed using PhysX SDK. There is integration with partner middlewares, like Unreal Engine 3.0, Emergent GameBryo and Natural Motion Endorphin, that allows the development of games making use of the PhysX SDK.

PhysX API delivers the most advanced simulation technology for games with support for an arbitrary number of physics scenes:

- Complex rigid body object dynamics and collision detection to realistic characters motion and interaction and allowing the developer to create very natural collisions and couple them to secondary effects, like the chinking noise when coins hit the floor, or skid marks where the driver hit the brakes;
- Joints and springs are useful tools for modeling complex mechanisms, going beyond character movements, doors and levers, the realm of vehicles, and the ability for the player to pick up and manipulate objects in real world;
- Volumetric fluid creation and simulation can enhance the visual effects of scenes, like waves on fluid's surface, and make possible the use of fluid weapons or fire hoses to gather the gamer attention to those effects;

- PhysX FX smart particle systems make it possible to simulate fire, smoke and fog in a natural way, like smoke contained in a room that rises to the ceiling, filling it completely until spilling out of the windows;
- Cloths boost the realism of a scene containing trembling curtains and cloths which mold to the characters bodies.

PhysX runtime is a completely integrated solver allowing fluids, particles, cloths and rigid bodies to interact correctly. It exploits parallelism at every level, from the core solver to the scene management, to achieve high performance simulation, and the hardware abstraction level eliminates the need to understand the internals of next-generation hardware. It is cross-platform and optimized for single and multi-core PC, Xbox 360, Playstation 3 and the PhysX processor (presented in sequence).

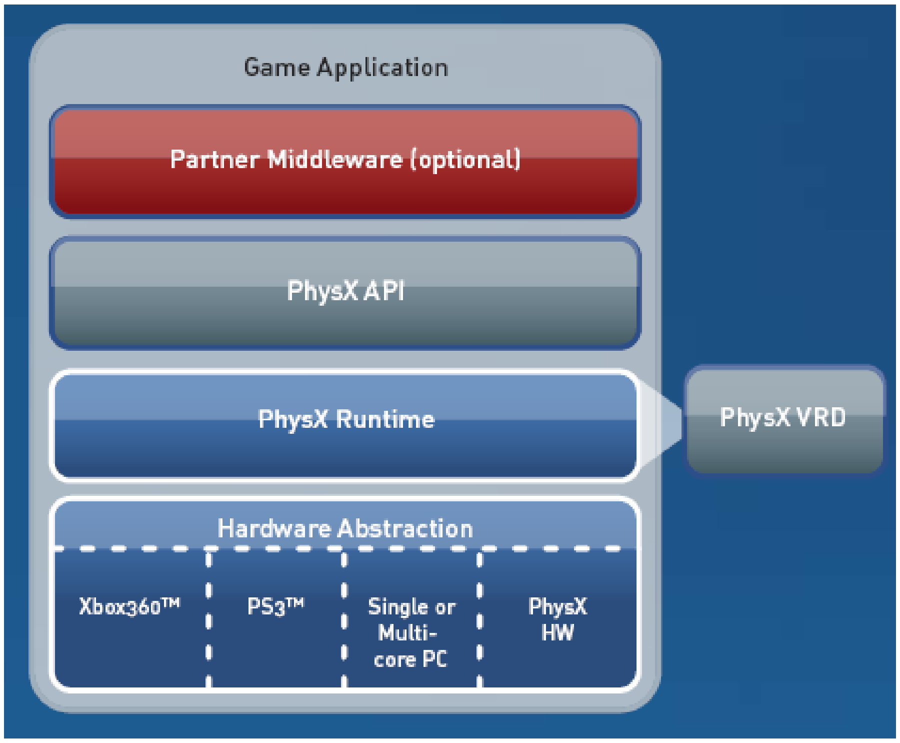

Figure 2. Application architecture developed using PhysX.

1.4. AGEIA PHYSX PROCESSOR

Enhancing the physics realism in a game highly increases the amount of mathematical and logical calculations needed. Therefore, the use of a dedicated processor concerned with physics simulation is a natural way to push out the load of the computer processor.

The AGEIA PhysX processor is the first, and the only one dedicated PPU built to enhance game play physics. Through the use of that PPU it is possible to have more realistic actions and behaviors in sophisticated games without degrading performance. A closer look in the PhysX PPU board may be taken at Figure 2.

Figure 2. Ageia PhysX PPU board.

To make use of the powerful physics enhancements it is necessary to have a good CPU processor and a high performance GPU because the realistic physics demands more detailed object rendering like cloth's movement, besides the quantity of debris in a scene simulating high realistic explosions.

1.5. OTHERS PHYSICS ENGINES

There are others physics engines available to develop applications, each one having its own features and licensing. Below are presented some of those physics engines, along with their respective brief explanation.

Open Dynamics Engine (ODE) and Bullet Physics Library are open source and free for commercial use engines that offer rigid body dynamics and collision detection.

TOKAMAK Game Physics and Newton Game Dynamics are free for commercial use engines. Like ODE and Bullet, the features available are rigid body dynamics and collision detection.

Havok Physics SDK is a commercial product that offers rigid body dynamics, collision detection, continuous physics, and visual debugger for in-game diagnostic feedback, among other features. Havok FX is a new technology that makes use of GPU to enhance the performance of physics calculations.

Among the physics engines mentioned above, only Havok FX delivers features near to those available on PhysX, with the tradeoff that they cannot be used without buying a license. PhysX licensing details are described in Chapter 2.

2. ENVIRONMENT CONFIGURATION

The PhysX SDK is available according to some constraints that are described in its license. This chapter explains how to download, install and consult the documentation, and gives some information about the PhysX's license.

2.1. PHYSX SDK AND SYSTEM SOFTWARE DOWNLOAD

In order to download the PhysX SDK an account registration is required, which approval can take 1-3 business days. The registration occurs through the AGEIA Developer Support Site (**http://devsupport.ageia.com**), clicking the **Sign in** link at the top right corner of the page, and the **Request a new account** button; user needs to fill up the required field and click again the button **Request a new account**.

Once registered and signed in, user should follow some steps: click **Online Support**, **Downloads**, then on the left panel click the **desired PhysX SDK**, and in the main panel click **PhysX SDK Core** to download the PhysX SDK that enables the development using its API.

To run any game or application developed with PhysX it is necessary to install the AGEIA PhysX System Software, which is available on the AGEIA Site (**http://www.ageia.com**): click **Driver & Support** link and then download the latest PhysX Drivers. It offers support for the AGEIA PhysX PPU and includes the latest versions of PhysX System Software.

2.2. License

Below, some information about PhysX's license are given, which have been extracted from the AGEIA Developer Support Site:

At present, the following two options are available: AGEIA will likely never provide any sort of royalty-based license. Open-source usage is limited only to the most middleware-friendly license model--in other words, it retains full ownership and rights to its own IP, and what you ship of AGEIA's will still be specified by contract.

- Free license:
  - Non-commercial use;
  - PS3 platform (through Sony pre-purchase);
  - Through some of our middleware partnerships, such as UE3, Gamebryo 2.2, and others - often limited to non-commercial use;
  - PC platform (if the game makes significant use[1] of PhysX HW).
- $50k per platform:
  - All other uses;
  - Fee may be waived at AGEIA's discretion for multi-platform developers providing PC HW support;
  - Fee may be waived at AGEIA's discretion for some Tools & Middleware providers.

2.3. Installation

System requirements for AGEIA PhysX System Software are a PC running Microsoft Windows XP or Windows Vista RC-1 (32-bit & 64-bit) with the most recent service pack installed, a minimum of 512MB of system memory and at least 50MB of free hard disk space. For PPU, it is necessary also a vacant PCI expansion slot and an additional 4 pin molex power connector.

[1] This will be specified in the contract, and may depend on the title; but, general rules of thumb are: (1) grandmother test, that defines that if the developer shows the game running on PhysX HW side-by-side with the non-HW version, his/her grandmother would be able to easily point out the differences; (2) gamer test, that defines that if the gamer is going to want to run his/her game with PhysX HW, AGEIA helps him/her make you're his/her game (inexpensively, but with a world-class SDK), so he/she is helping AGEIA sell HW (by having a great game supporting PhysX HW).

First, the PhysX Driver has to be installed that allows the execution of PhysX based applications. After the installation it is possible to run a demonstration to test the PhysX System Software, following some steps: click Start Menu, All Programs, AGEIA and AGEIA PhysX Properties to launch the application; in the **Demonstration** tab, double-click the AGEIA PhysX Boxes Demo to run and view a sample application using PhysX runtime.

Finally, the PhysX SDK Core has to be installed that provides the header and lib files, some sample projects, and the documentation and training programs to teach the features of PhysX SDK.

2.4. Documentation

There are lots of documents available to help the developer of PhysX based games and applications. After installation of the PhysX SDK Core will be available in the AGEIA PhysX SDK Start Menu folder the following folders:

- Docs – containing the AGEIA PhysX Documentation, a help windows file with a detailed documentation of PhysX SDK and PhysX API;
- Samples – containing source code and executable of a lot of examples projects covering most of PhysX features;
- Training Programs – a complete training program that includes documents, overview and the projects source code used in the training.

In the AGEIA Developer Support Site is available the Developer Knowledge Base in a form of FAQ, covering several topics from general issues to PhysX SDK hardware specific questions.

3. Architecture and Components

This chapter woks as an overview of the more important aspects and classes of the PhysX engine. The main concepts to understand this engine and general information about physics simulation are treated in the following sections.

3.1. World

PhysX SDK is implemented in C++, and internally organized as a hierarchy of classes. Each class containing functionality accessible by the user implements an interface, which is in fact an abstract base class. Furthermore, some stateless utility functions are exported as C functions.

The interface classes follow some coding conventions:

- All classes have a header file with the same file name as the class name;
- Types and classes start with a capital letter;
- Interface classes always start with "Nx";
- Methods and variables start with a lowercase letter;
- Return values and parameters in places where a NULL value is acceptable are coded with pointer (*) syntax;
- Return values and parameters in places where a NULL value is unacceptable are coded with reference (&) syntax;
- If some functionality is user dependent for implementation, he/she needs to implement an interface defined for this purpose (i.e. memory management).

Figure 3 presents a simplified architecture diagram, showing the main classes of the PhysX engine.

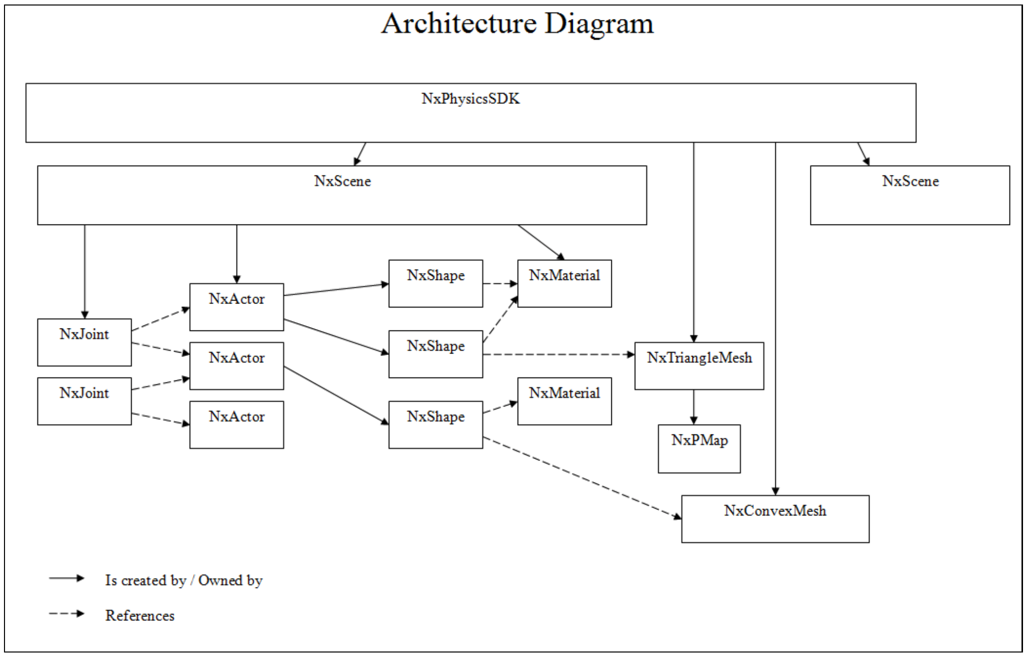

Figure 3. PhysX architecture.

The NxPhysicsSDK class is an abstract singleton factory class used for instancing objects and setting global parameters that will affect all scenes.

In order to get an instance of this class, the NxCreatePhysicsSDK() method should be called, as shown in the example bellow:

```
NxPhysicsSDK * myWorld = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION);
```

The NX_PHYSICS_SDK_VERSION constant ensures that the application is using the same header version as the library was built with.Optionally, programmers can specify the Memory Allocator and the Output Stream, using user defined interfaces that inherit from NxUserAllocator and NxUserOutputStream interfaces, respectively. However, it's common to see the former simplified definition.

3.2. *SCENE*

A scene is a collection of bodies, constraints, and effectors, which may interact.

The scene simulates the behaviour of these objects over time. Several scenes may be created at the same time, but its bodies or constraints cannot be shared among other scenes. For example, developer can't create a joint in one scene and then use it to attach bodies from a different scene.

A scene can manipulate environment elements (such as gravity, corpses, joints, materials etc.) and it is internally represented by the NxScene class. Implementation details will be shown on Chapter 5.

3.3. Actor

An actor (NxActor) is the main simulation object in the physics SDK. The actor is created by a specific scene, which owns it. The actor can be dynamic, or static, fixed in the world. Dynamic/Static actors will be detailed on Chapter 6.

Bellow is presented an example of how to create an actor:

```
// Create a static 'ground plane' actor.
NxPlaneShapeDesc planeDesc;

NxActorDesc actorDesc;
actorDesc.shapes.pushBack(&planeDesc);

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

This code branch inserts a plane actor into the scene.

3.4. Shapes

Shapes are used to detect and control collisions. The NxShape abstract class encapsulates the subclasses presented in Figure 4.

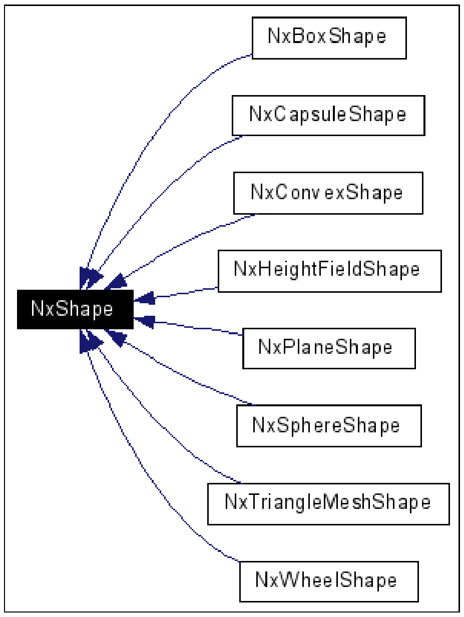

Figure 4. NxShape subclasses.

3.4.1. Box Shape

The NxBoxShape is the class that implements the box shaped collision detection primitive. Each shape is owned by the actor that it is attached to.

An instance of the class can be created by calling the createShape() method of the NxActor object that will own it. The shape properties are defined in an NxBoxShapeDesc description object, which contains all the information about the shape and is passed as a parameter to the createShape() method.

```
NxBoxShapeDesc boxDesc;

boxDesc.dimensions.set(0.5f,0.5f,0.5f); // A 1m cube.

NxBoxShape *boxShape=actor->createShape(boxDesc)->isBox();
//Note isBox() is a safe way to cast.
```

The shape is deleted by calling the NxActor::releaseShape() method on the owning actor.

3.4.2. *CAPSULE SHAPE*

The NxCapsuleShape is defined by a "radius", which is the size of the capsule's hemispherical ends, and a "height", which is the distance between the two ends of the capsule. Figure 5 illustrates a Capsule Shape.

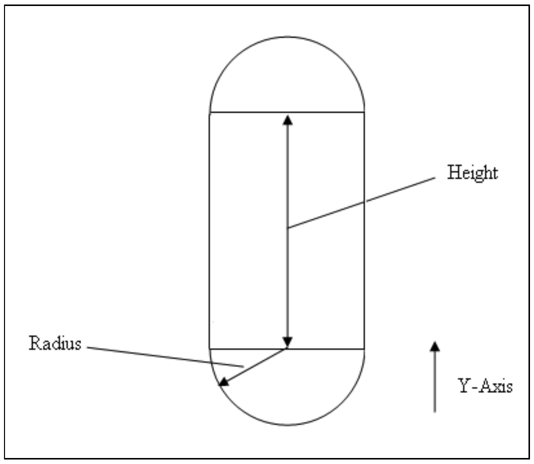

Figure 5. A Capsule Shape.

A code example to create a Capsule Shape is presented bellow:

3.4.3. *CONVEX SHAPE*

The Convex Shape is defined by a set of vertices. These vertices can be read from a file, and must be "cooked" with a memory buffer, in order to be used in a convex mesh afterwards. In this context, "cook" means to optimize internally, putting data in another format to allow the engine perform faster operations. To "cook" meshes, PhysX supplies the Cooking API, also used in the following sample. The shape must use the type NxConvexShape.

```
//Create a box convex mesh.
NxVec3 verts[8] = { NxVec3(-1,-1,-1),NxVec3(-1,-1,1),
                    NxVec3(-1,1,-1),NxVec3(-1,1,1),
                    NxVec3(1,-1,-1),NxVec3(1,-1,1),
                    NxVec3(1,1,-1),NxVec3(1,1,1) };

NxU32 vertCount = 8;

// Create descriptor for convex mesh
NxConvexMeshDesc convexDesc;
convexDesc.numVertices          = vertCount;
convexDesc.pointStrideBytes     = sizeof(NxVec3);
convexDesc.points               = verts;
convexDesc.flags                = NX_CF_COMPUTE_CONVEX;

// See SampleCommonCode\src\Stream.cpp for MemoryWriteBuffer and
MemoryReadBuffer.

MemoryWriteBuffer buf;
bool status = NxCookConvexMesh(convexDesc, buf);

NxConvexMesh *mesh = gPhysicsSDK-
>createConvexMesh(MemoryReadBuffer(buf.data));

// Now create an instance of the mesh.

NxConvexShapeDesc convexShapeDesc;
convexShapeDesc.meshData = mesh;

NxConvexShape *convexShape=actor->createShape(convexShapeDesc)->isConvex();
//Note isConvex() is a safe way to cast.
```

3.4.4. HEIGHT FIELD SHAPE

A Height Field Shape can use a Height Field Map to generate a collision terrain. The shape must use the type NxHeightFieldShape.

In the Height Field Descriptor class, the number of columns and rows determine the resolution of the field. The vertical extent defines how far below ground the height extents. The convex edge threshold parameter is used to determine if a height field edge is convex and can generate contact points. Then, the Height Field generated is used in the Height Field Shape Descriptor, which is different from the Height Field Descriptor itself, as a parameter. The scale parameter is used to resize the height field, and the type of material named hole designates holes in the height field.

```
NxHeightFieldDesc heightFieldDesc;

heightFieldDesc.nbColumns           = nbColumns;
heightFieldDesc.nbRows              = nbRows;
heightFieldDesc.verticalExtent      = -1000;
heightFieldDesc.convexEdgeThreshold = 0;

// allocate storage for samples
heightFieldDesc.samples             = new NxU32[nbColumns*nbRows];
heightFieldDesc.sampleStride        = sizeof(NxU32);

NxU8* currentByte = (NxU8*)heightFieldDesc.samples;

for (NxU32 row = 0; row < nbRows; row++)
    {
    for (NxU32 column = 0; column < nbColumns; column++)
        {
        NxHeightFieldSample* currentSample =
(NxHeightFieldSample*)currentByte;

        currentSample->height          = computeHeight(row,column); //desired
height value. Singed 16bit integer
        currentSample->materialIndex0 = gMaterial0;
        currentSample->materialIndex1 = gMaterial1;

        currentSample->tessFlag = 0;

        currentByte += heightFieldDesc.sampleStride;
        }
    }

NxHeightField* heightField = gScene-
>getPhysicsSDK().createHeightField(heightFieldDesc);


NxHeightFieldShapeDesc heightFieldShapeDesc;

heightFieldShapeDesc.heightField     = heightField;
heightFieldShapeDesc.heightScale     = gVerticalScale;
heightFieldShapeDesc.rowScale        = gHorizontalScale;
heightFieldShapeDesc.columnScale     = gHorizontalScale;
heightFieldShapeDesc. = 0;
heightFieldShapeDesc.holeMaterial = 2;

NxHeightFieldShape *heightFieldShape=actor->createShape(heightFieldShapeDesc)-
>isHeightField();
//Note isHeightField() is a safe way to cast.
```

3.4.5. PLANE SHAPE

A Plane Shape is a plane collision detection primitive. By default, it is configured to be the y == 0 plane. The developer can then set a normal and a d variable to specify an arbitrary plane. d is the distance of the plane from the origin along the normal, assuming the normal is normalized. The shape must use the type NxPlaneShape.

```
NxPlaneShapeDesc planeDesc;

//create a plane at y=1
planeDesc.normal = NxVec3(0.0f,1.0f,0.0f);
planeDesc.d = 10.0f;

NxPlaneShape *planeShape=actor->createShape(planeDesc)->isPlane();
//Note isPlane() is a safe way to cast.
```

3.4.6. SPERE SHAPE

A sphere shaped collision detection primitive is implemented by the NxSphereShape class. It is defined by the attribute "radius".

```
NxSphereShapeDesc sphereDesc;
sphereDesc.radius = 1.0f;

NxSphereShape *sphereShape=actor->createShape(sphereDesc)->isSphere();
//Note isSphere() is a safe way to cast.
```

3.4.7. TRIANGLE MESH SHAPE

The NxTriangleMeshShape class is a shape instance of a triangle mesh object.

```
//create triangle mesh instance
NxTriangleMeshShapeDesc meshShapeDesc;
meshShapeDesc.meshData   = triangleMesh; //See NxTriangleMesh

NxTriangleMeshShape *triangleMeshShape=actor->createShape(meshShapeDesc)-
>isTriangleMesh();
//Note isTriangleMesh() is a safe way to cast.
```

3.4.8. WHEEL SHAPE

This is a special shape used for simulating a car wheel, as illustrated in Figure 6. The shape must use the type NxWheelShape.

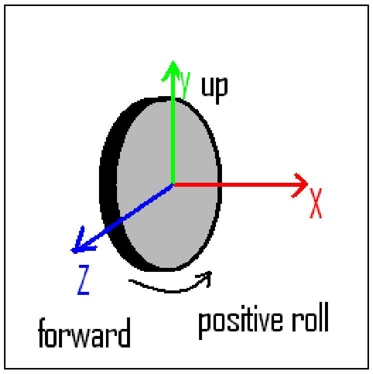

Figure 6. A Wheel Shape.

The Y-axis should be directed toward the ground, and a ray is cast from the shape's origin along this axis. When the ray strikes something, three contact situations may happen:

- If the distance from the shape origin is less than wheelRadius, then a hard contact is created;
- If the distance from the shape origin is between wheelRadius and (suspensionTravel + wheelRadius), then a soft suspension contact is created;
- If the distance from the shape origin is greater than (suspensionTravel + wheelRadius), then no contact is created.

A positive wheel steering angle corresponds to a positive rotation around the shape's Y-axis.

3.5. *Joints*

Joints are connections between rigid bodies. A joint is a point that connects two actors. In the description of a joint, it is defined how the actors can move with respect to each other about this point. An example of a typical joint can be seen in Figure 7.

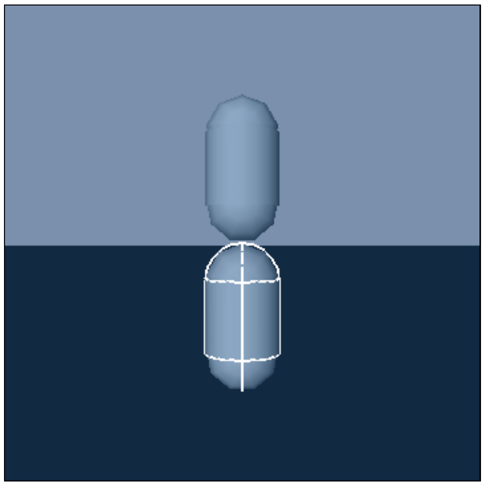

Figure 7. Joints.

A very simple example of a joint would be to connect two actors so that they move perfectly in unison over a point, keeping their orientations aligned and the local vector between their centres of mass constant.

A joint restricts motion between two actors over a point with respect to both, rotation and translation. The first joint, the revolute joint, allows two actors to rotate with respect to each other around a single axis over a point. The revolute joint allows a single degree of freedom of motion: one axis of rotation and no axes of translation. Future joints will unlock other axes of rotation and translation, allowing freedom of movement around different combinations of the three axes of rotation and translation.

Joints are a very useful and powerful feature of the PhysX SDK. This tutorial will aproach, on Chapter 8, many more types of joints and their properties, as well as how to constrain them, put springs and motors on them, etc. Some implementation examples will also be shown.

3.6. *Material*

Material is a class to describe the shape's surface properties. It can be created a default material, setting the values for friction and restitution. Otherwise, it can be created a material that has different friction coefficients depending on the direction that a body in contact is trying to move in. This is called anisotropic friction, and it's very useful for modelling things like sledges and skies.

An anisotropic material example is presented bellow:

```
// Create an anisotropic material
NxMaterialDesc  material;

//anisotropic friction material
material.restitution        = 0.0f;
material.staticFriction     = 0.1f;
material.dynamicFriction    = 0.1f;
material.dynamicFrictionV   = 0.8f;
material.staticFrictionV    = 1.0f;
material.dirOfAnisotropy.set(0,0,1);
material.flags              = NX_MF_ANISOTROPIC;

NxMaterial *anisoMaterial = gScene->createMaterial(material);
```

On the other hand, a default material example can be created as follows:

```
NxMaterial* defaultMaterial = gScene->getMaterialFromIndex(0);

defaultMaterial->setRestitution(0.5);
defaultMaterial->setStaticFriction(0.5);
defaultMaterial->setDynamicFriction(0.5);
```

4. MATHEMATICAL SUPPORT

Some classes are provided by PhysX to support math operations and aid programmers to pass information to the core of the physics engine. These classes include stateless math operations, like trigonometric relations and helper functions (i.e. sin, tangent, absolute value etc.), matrix operations (transposing, inverting, multiplying etc.), vector operations (dot product, cross product etc.) and quaternion's stuff.

To encapsule these operations, PhysX delivers some wrapper classes which enclose related functions: NxMath, NxMat33, NxMat34, NxVec3 and NxQuat. These functions are explained next.

4.1. NXMATH

The NxMath is a static class that contains methods to deal with floating point scalar math functions and other features that may help programmers shorten codes and make them more readable. NxMath also defines some mathematical constants that may be useful, like NxPiF64, NxHalfPiF64, NxTwoPiF64, NxInvPiF64 and its 32 bits correspondents. A sample of the methods available in NxMath follows: equals, floor, ceil, sign, max, min, clamp, mod, sqrt, pow, exp, degToRad, radToDeg, sin, cos, tan, rand and logarithm functions using base 10, 2, and "e".

The code branch bellow is a sample of utilization of the NxMath class. The method max() is invoked receiving as parameter two numeric vales, and returning the largest one.

```
NxRealvehicle::_calculateMotorRPM(NxReal rpm) {
      NxReal temp = _getGearRatio() *_differentialRatio;
      NxReal motorRpm = rpm * temp;

      if (motorRpm > mMotor->_maxRpm) {
            return mMotor->_maxRpm;
      }

      return NxMath::max(rpm * temp,mMotor->_minRpm);
}
```

4.2. *NxMat33*

NxMat33 is an abstraction to a 3x3 matrix. This type of matrix is often used to represent rotations or inertia tensors. Since NxMat33 is a simple rotation matrix, affine transformations cannot be applied to it. Therefore, operations such scaling are disabled through this class.

The NxMat33 class, like others mathematical abstractions in the Ageia PhysX (NxVec3, NxMat34, NxQuat), was made to be storage format independent. This feature sets the programmer free, which can use several types of external data structures to represent his/her rotations, in NxMat33 case. A common example of multiple data structure use is the interface between the physics and graphics engines. The most used graphics platforms, OpenGL and Direct3D, work differently concerning matrix formats. Therefore, PhysX can give these matrixes in both formats, row or column major, abstracting the final use of the matrix's values.

The main matrix's operators were implemented in NxMat33:

- direct access to matrix's elements, through the operator()(int, int) method;
- sum and subtraction (+/-);
- matrix and scalar product (*).

With these operators, the developer can perform a linear transformation easily, not concerning the internal implementation.

Beyond the operators, inverting, transposing and converting functions are also available. An important issue to look at is the conversion between NxMat33 and NxQuat (described in Section 4.5), which is a concise and complete manner to represent rotations, considering the lack of "gimbal lock" in the quaternion's math. The following example takes a vector in the local space of an actor, and transforms it into world space.

```
NxVec3 localVec, worldVec;
NxMat33 orient, invOrient;

worldVec = NxVec3(0,0,1);
orient = actor->getGlobalOrientation();
orient.getInverse(invOrient);
localVec = invOrient * worldVec;
```

4.3. *NxMat34*

The NxMat34 is a wrapper class composed by a NxMat33 and a NxVec3 (explained in Section 4.4). With this class it is possible to do affine transformations, since it groups rotation and translation in a single object. Its attributes are public and named M (NxMat33) and t (NxVec3), making possible any access to their internal methods.

Using the same concepts of NxMat33, NxMat34 allows the programmer to use simplified operators, like matrix product (operator *), and conversions to column and row major 4x4 rendering matrix formats. The next example takes a position in world space and transforms it into the local space.

```
NxActor* actor;
…
NxVec3 localPos, worldPos;
NxMat34 mat, invMat;

worldPos = NxVec3(0,0,1);
mat = actor->getGlobalPose();
mat.getInverse(invMat);
localPos = invMat * worldPos;
```

4.4. *NxVec3*

NxVec3 is a simple class that puts together x, y, and z coordinates. As NxMat34, NxVec3 gives free access to its members to provide code compatibility with other engines that assume this strategy.

This class implements many common functions used in vectors' operations, as well as access functions, that may assume the vector internally uses an array, or separated values to represent the 3D coordinates.

NxVec3 may reference either a point or a vector in 3D space, and encloses functions to handle both representations.

Some operations contained in NxVec3 are:

- magnitude (vector);
- distance (point);
- cross product (vector);
- dot product (vector);
- all vector's arithmetic.

The following example extracts the 3 local axes of an actor.

```
NxMat33 orient;
NxVec3 xaxis, yaxis, zaxis;

orient = actor->getGlobalOrientation();
orient.getRow(0, xaxis);
orient.getRow(1, yaxis);
orient.getRow(2, zaxis);
```

4.5. *NxQuat*

NxQuat is the quaternion representation used by PhysX. As well as most of the implementations of quaternions, its internal state is constructed upon four real numbers, which stand for the scalar part "w", and the vector part "x, y, z" that are coefficients to the complex numbers "i, j, k", respectively. According to Hamiltonian Arithmetic, quaternions are a vector space over the real numbers. In applied mathematics, quaternions are used to represent and simplify 3D rotations. This is the meaning of quaternions' presence in the world of PhysX. It is the most concise way to describe rotation around multiple axes. A single quaternion is equivalent to an angle around an axis, presented as a vector. To concatenate rotational operations, another quaternion with an arbitrary rotation must be multiplied with the first, which acts like an accumulator, as well as seen in affine transformations using matrixes.

Using the class NxQuat, a quaternion can be initialized by an axis and an angle, or a 3x3 rotation matrix. The NxQuat also has methods to get the axis and angle of a quaternion, as well as operators to multiply by other quaternion and other quaternion arithmetic functions. Another main feature of the NxQuat is the slerp method, which interpolates a rotation between two keyframes, represented by other quaternions. In slerp, a real number between 0 and 1 adjusts how close the resultant quaternion is to the final keyframe, as a common interpolation.

In the following example is created a quaternion from an angle and an axis, and an orientation matrix is obtained from a quaternion.

```
NxQuat q;
q.fromAngleAxis(90, NxVec3(0,1,0));

NxQuat q;
…
NxMat33 orient;
orient.fromQuat(q);
```

5. Scene

As introduced in Section 3.2, a scene is a collection of bodies, constraints and effectors, which can interact. In this chapter, it will be shown an example of a simple scene creation.

The first step is declaring the variables that are going to be used:

```
static NxPhysicsSDK*    gPhysicsSDK = NULL;
static NxScene*         gScene = NULL;
static NxVec3                 gDefaultGravity(0.0f, -98.1f, 0.0f);
static ErrorStream            gErrorStream;
```

Then, the world must be initialised:

```
// Initialize PhysicsSDK
gPhysicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION, NULL,
                                  &gErrorStream);
```

If wanted, the debug information can be visualised, by setting the world parameters following the template:

```
// setParameter(VISUALIZATION_TYPE, TRUE|FALSE);

gPhysicsSDK->setParameter(NX_VISUALIZE_COLLISION_SHAPES, 1);
```

Now, a scene must be createdand its parameters set, and the scene assigned to the created world:

```
// Create a Scene with the gravity force declared above
NxSceneDesc sceneDesc;
sceneDesc.gravity                  = gDefaultGravity;
sceneDesc.userTriggerReport        = &gMySensorReport;
gScene = gPhysicsSDK->createScene(sceneDesc);
```

Afterward, it should be created a material, in order to insert possible objects in the scene:

```
NxMaterial * defaultMaterial = gScene->getMaterialFromIndex(0);
defaultMaterial->setRestitution(0.0f);
defaultMaterial->setStaticFriction(0.0f);
defaultMaterial->setDynamicFriction(0.0f);
```

Finally, creating a floor for the scene allows that the objects, even being attracted, won't fall and disappear because the gravity force:

```
// Create ground plane
NxPlaneShapeDesc PlaneDesc;
NxActorDesc ActorDesc;
ActorDesc.shapes.pushBack(&PlaneDesc);
gScene->createActor(ActorDesc);
```

In this empty scene, developer will get something like presented in Figure 8, if he/she sets the `NX_VISUALIZE_COLLISION_SHAPES` parameter to true.

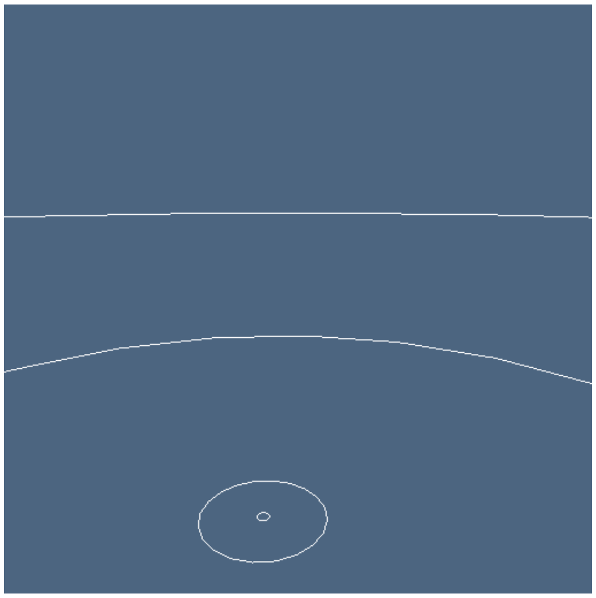

Figure 8. An empty scene without objects.

6. Actors

Actors are the basis for any simulation context. They represent the physical objects in an environment space. Many properties are bonded to actors, such as global position, orientation and interaction with the other actors that share the same scene. In the behavioural context, actors can be classified as static, dynamic or kinematic, depending on how they react to forces or contact interaction with other parts of the world.

6.1. *Static Versus Dynamic Actors*

Static actors are actors that take part of the landscape, such as non-moving characters, buildings or anything stuck in the scene. The main purpose of static actors is to provide only collision detection. They don't need to have mass, momentum, inertia or any physical attributes specified, because the position of these objects doesn't change across the simulation time.

At the opposite side, stand the dynamic actors, which may also be called "bodies". This type of actor must contain a body properties set, which is enclosed in the NxBodyDesc class.

Dynamic or static, actors may have a set of shapes, which may be empty, to represent the collision control. These shapes may be used as triggers, as well, as seen in Chapter 9.

The creation of actors in PhysX consists in some steps:

- At first, to create and populate the actor's description, the NxActorDesc class has to be filled. An example on how to fill a NxActorDesc follows:

```
// Create a dynamic 'box' actor. A dynamic actor has a non NULL
// body.
NxActorDesc actorDesc;
NxBodyDesc bodyDesc;

actorDesc.setToDefault();//reset to default values.
actorDesc.body = &bodyDesc;
actorDesc.density = 10;
// set initial position.
actorDesc.globalPose.t = NxVec3(0.0f,10.0f,0.0f);
```

- Then, the shapes that will be the collision model for the actor have to be created.

```
NxBoxShapeDesc boxDesc;

// The actor has one shape, a 1m cube.
boxDesc.dimensions.set(0.5f,0.5f,0.5f);
actorDesc.shapes.pushBack(&boxDesc);
```

Actors may have a lot of shapes attached to it. To add more shapes, new shape descriptions have to be added in the shapes attribute of NxActorDesc class. More examples of shapes are given in Section 3.4.

An optional step is to set flags in the body description. Flags can modify the behaviour of the actor in some aspects, disabling gravity on it, or changing its interaction with the world. To set an actor immune to the gravity force, the NX_BF_DISABLE_GRAVITY must be raised. An important example is setting the NX_BF_KINEMATIC flag to an actor. The kinematic flag is responsible to disable the force or torque interaction in actors, including even the gravity. To change the position of a kinematic actor, the programmer has to set its global position. The kinematic actor is an intermediate type of actor between static and dynamic, which can become a full dynamic actor at runtime, at user's will. The function set to move a kinematic actor is located at NxActor class and described bellow:

```
NxBodyDesc body;
body.setToDefault();
body.flags |= NX_BF_KINEMATIC;

NxActorDesc actDesc;
actDesc.body = &body;
```

- The final step is related to the actual creation of the actor, through the NxScene instance.

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

Some parameters are needed for a realistic simulation, and are classified as rigid body properties. These settings hold parameters like mass (total mass of the actor), position of the center of mass (in local space), linear velocity (of the center of mass), resultant force exerted in the actor, inertia tensor (describes the mass distribution of the body), orientation, angular velocity and torque. All these parameters are found in the NxBodyDesc class.

```
//moves and/or rotates a kinematic actor
void  moveGlobalPose (const NxMat34 &mat);
//moves an kinematic actor
void  moveGlobalPosition (const NxVec3 &vec);
//rotates a kinematic actor using a rotation matrix
void  moveGlobalOrientation (const NxMat33 &mat);
//rotates a kinematic actor using a quaternion
void  moveGlobalOrientationQuat (const NxQuat &quat);
```

6.2. *Materials*

Materials are concepts assigned with the collision reaction. As well as all about collision, materials are bonded to the shape of an actor. Through the material description, the engine will decide if two surfaces will bounce when crashing, how much force will be applied to move these surfaces when they are bonded together, or how strong has to be an external impulse to break the inertia of an instantly static body.

All shapes have a material assigned to it. When the programmer doesn't set a specific pre-created material, the engine applies the default material. Materials are stored internally by the PhysX engine, and every material created has an index that is uniquely mapped to each of them. Therefore, the creation process of a shape that reacts using different material properties is done using the index of an already created material, as seen in the following example:

In this code snippet, three important properties, which need to be discussed, are mentioned:

- Restitution - is related to how bouncy will be the contact between the owner shape and the other contact part. This value must stay between "0" and "1", being more bouncy when getting close to "1";
- Static friction - the coefficient of static friction. This value should be in range "0" to "+infinity". This property is used when the body is resting;

- Dynamic friction - the coefficient of dynamic friction. This value should stay in range “0” to “1” and is used when the body is moving.

7. Elements Interaction

In this chapter will be given some examples on how to interact with elements (actors) in PhysX. This can be made applying forces and/or torque across several force modes, which will be discussed bellow.

7.1. Force and Torque

The most natural way of interacting with a body is applying an external force that directly controls its acceleration and indirectly controls its velocity and position. Usually, when applying force the simulation will be able to keep all the defined constraints (joints and contacts) satisfied, but it will not achieve immediate response.

The forces applied to a body are accumulated before each simulation frame, used in the simulation and then it is canceled in preparation for the next frame.

7.2. Applying Force and Torque

It is necessary to make use of NxActor’s methods to apply force and torque. The following methods are the basic ones:

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

The addForce() method applies a force defined in the global coordinate frame to the actor whereas the addLocalForce() method uses the local coordinate frame. The same is true for addTorque() and addLocalTorque(). These methods apply force or torque at the center of mass.

The first parameter of these methods is a 3D vector that defines the magnitude and orientation of the force or torque. The second one is the force mode, which is detailed in the next section.

The following methods are used to apply forces or torques at different positions on an actor:

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

The same considerations about global and local coordinate frames can be used in these methods, i.e., the force vector can be referenced by a global or a local coordinate, as well as the position where the force will be applied can be in a global or a local coordinate.

7.3. Force Modes

The SDK divides the time step into a number of substeps that are integrated by the physics, and normally when a force is applied to an actor, it will be applied all at once on the first substep.

The enumeration NxForceMode defines the force mode that will be applied using addForce()/addTorque() methods and its derivative; these modes are detailed below:

- NX_FORCE – to apply a force (mass * distance / time^2) to an object at the first substep of the time step;
- NX_IMPULSE – to apply an impulse, which is a momentum change, (mass * distance / time) to the actor at the first substep of the time step;
- NX_VELOCITY_CHANGE – to adjust the velocity (distance / time) of an object directly at the first substep of the time step;
- NX_SMOOTH_IMPULSE – to apply an impulse to the object over every substep of the time step;
- NX_SMOOTH_VELOCITY_CHANGE – to make a velocity change to an object over every substep of the time step;
- NX_ACCELERATION – to apply an acceleration (distance / time^2) to an object at the first substep of the time step.

8. Joints

As it was introduced in Section 3.5, joints are connections between rigid bodies. In sequence, many types of joints and its parameters will be explained.

8.1. Types

Joints are classified in diverse types, namely spherical, revolute, prismatic, cylindrical, fixed, distance, point in plane, point on line and pulley. These types of joints are described in the following subsections.

8.1.1. Spherical

Spherical joints allow three degrees of freedom, to rotate around every joint axis while remaining in a fixed position.

Spherical joints are commonly used to simulate shoulder joints and other ball-and-socket joints. The joints are free to rotate around any axis, but may have limits placed on the amount they can rotate around each axis, as seen in Figure 9, which shows two objects bonded by an spherical joint limited by the red area.

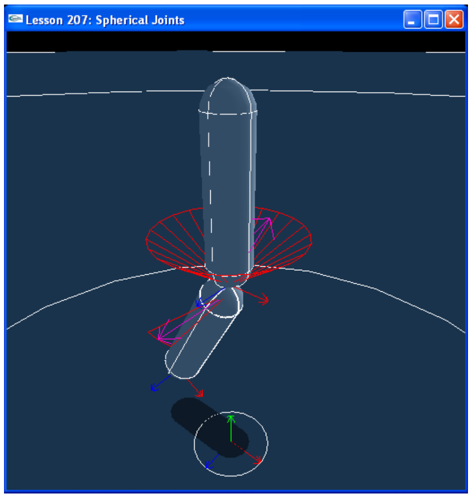

Figure 9. A spherical joint.

In order to create a spherical joint developer will need the reference of the two actors, which will interact with this joint. Furthermore, the developer will need to define an anchor point, which is where the joint will be positioned, and a global axis, which will be used to set the limits for that axis.

Then, it's time to configure the swing and twist limits. The swing limit will not let the defined global axis trespass the determined angle, and the twist limit will not allow the determined axis spin on itself freely.

Bellow follows an example of code creating a spherical joint:

```
NxSphericalJoint* CreateSphericalJoint(NxActor* a0, NxActor* a1)
{
    NxVec3 globalAnchor = NxVec3(0,5,0);
    NxVec3 globalAxis = NxVec3(0,1,0);

    NxSphericalJointDesc sphericalDesc;
    sphericalDesc.actor[0] = a0;
    sphericalDesc.actor[1] = a1;
    sphericalDesc.setGlobalAnchor(globalAnchor);
    sphericalDesc.setGlobalAxis(globalAxis);

    sphericalDesc.flags |= NX_SJF_SWING_LIMIT_ENABLED;
    sphericalDesc.swingLimit.value = 0.3*NxPi;

    sphericalDesc.flags |= NX_SJF_TWIST_LIMIT_ENABLED;
    sphericalDesc.twistLimit.low.value = -0.05*NxPi;
    sphericalDesc.twistLimit.high.value = 0.05*NxPi;

    return (NxSphericalJoint*)gScene->createJoint(sphericalDesc);
}
```

8.1.2. Revolute

This joint is often called the "hinge" joint, as it can be used to represent the hinge on a door (see Figure 10).

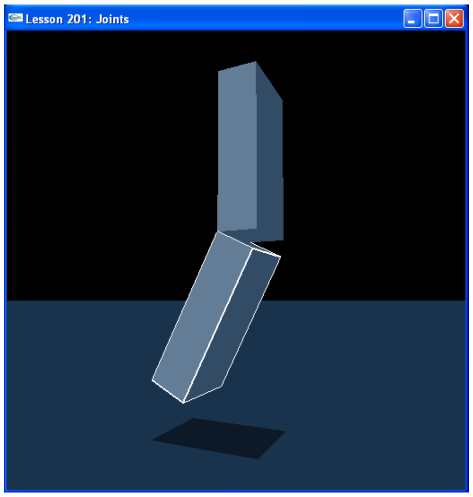

Figure 10. A revolute joint.

The following code branch is very similar to the previous one. Developer needs to set the global anchor and the global axis the same way as before. This example shows the particularity of a collision enabled flag, which will say that the actors will collide between themselves.

```
NxRevoluteJoint* CreateRevoluteJoint(NxActor* a0, NxActor* a1, NxVec3
globalAnchor, NxVec3 globalAxis)
{
    NxRevoluteJointDesc revDesc;

    revDesc.actor[0] = a0;
    revDesc.actor[1] = a1;
    revDesc.setGlobalAnchor(globalAnchor);
    revDesc.setGlobalAxis(globalAxis);

    revDesc.jointFlags |= NX_JF_COLLISION_ENABLED;

    return (NxRevoluteJoint*)gScene->createJoint(revDesc);
}
```

8.1.3. PRISMATIC

Prismatic joints are purely translational and only free to move along its primary axis. A good example of a prismatic joint is a shock absorber, as seen in Figure 11.

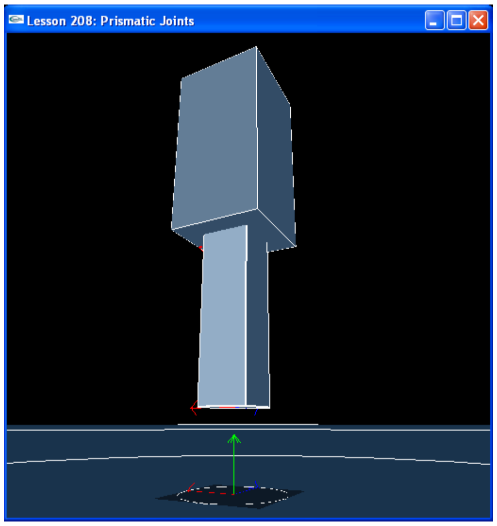

Figure 11. A prismatic joint.

For the prismatic joint, the global anchor will be set and the top of the lower actor and two limit planes will be set at the boundaries of the upper actor. This way, the joint will be confined within the planes.

8.1.4. CYLINDRICAL

Cylindrical joints are free to rotate around the primary axis like a revolute joint, as well as translate around the primary axis like a prismatic joint. Figure 12 shows an example.

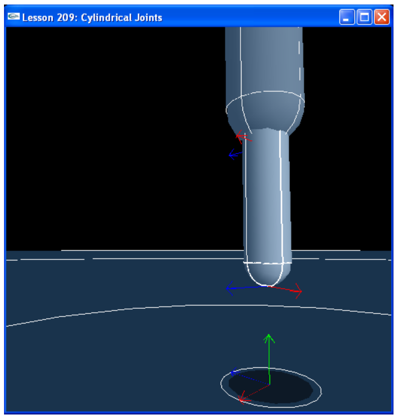

Figure 12. A cylindrical joint.

The setup for the cylindrical joint is identical to the prismatic joint, as well as the creation function.

8.1.5. FIXED

Fixed joints are useful to unite two actors to work as one, without any degree of freedom.

Its implementation is identical to the other joints, with a global anchor and a global axis.

8.1.6. DISTANCE

The distance joint connects two actors by a rod. Each end of the rod is attached to an anchor point on each actor (see Figure 13).

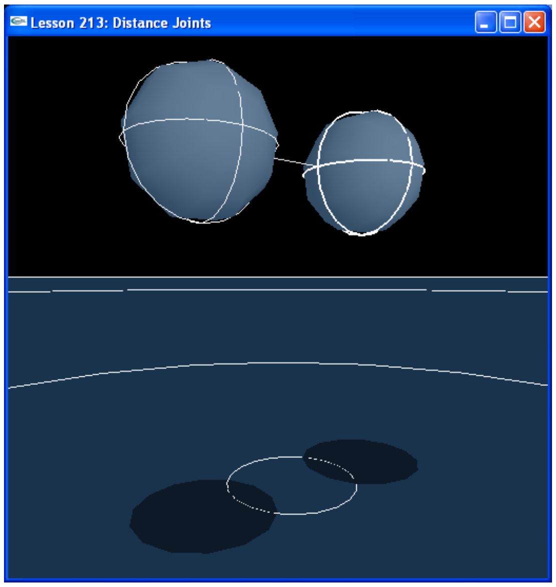

Figure 13. A distance joint.

So, differently from the other joints studied by now, this particular joint has two anchor points. These anchors are set locally, so if developer wants to fix them at the center of two spheres of radius 1, the developer must create the points as follows:

```
NxVec3 anchor1 = NxVec3(0,1,0);
```

In order to set the maximum and minimum distances between the objects connected by the joint, developer can calculate the initial distance between the objects and multiply it by a parameter. It may also be added a spring factor to the joint, if desired.

8.1.7. POINT IN PLANE

The point in plane (PIP) joint allows its actors to rotate about all axes with respect to each other and translate along two axes. An example of a PIP joint would be a refrigerator magnet, in other words something that moves along a planar surface and is free to rotate around any axis while attached to the plane. Figure 14 shows an example.

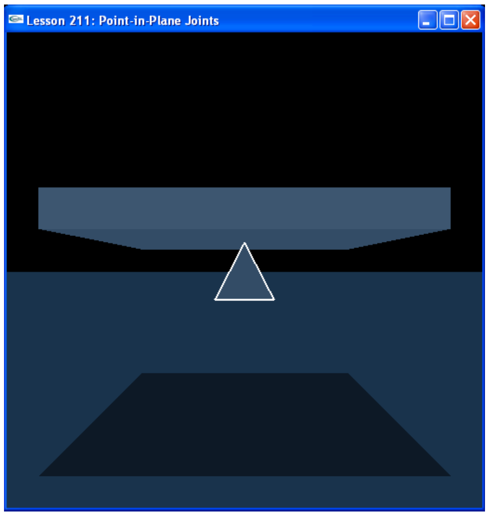

Figure 14. A point in plane joint.

Creating a PIP joint is no different from the previous joints explained. But, in the example bellow it can also be determined the area in which the joint will be free to mov

```
 NxPointInPlaneJoint* CreatePointInPlaneJoint(NxActor* a0, NxActor*
a1, NxVec3 globalAnchor, NxVec3 globalAxis)
{
    NxPointInPlaneJointDesc pipDesc;
    pipDesc.actor[0] = a0;
    pipDesc.actor[1] = a1;
    pipDesc.setGlobalAnchor(globalAnchor);
    pipDesc.setGlobalAxis(globalAxis);
    pipDesc.jointFlags |= NX_JF_COLLISION_ENABLED;

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

    return joint->isPointInPlaneJoint();
}
```

This can be done creating four limit planes for the joint, as shown on the next code branch:

```
void InitNx()
{
    NxJoint* jointPtr = &pipJoint->getJoint();
    jointPtr->setLimitPoint(globalAnchor);
    jointPtr->addLimitPlane(NxVec3(1,0,0), globalAnchor -
                             5*NxVec3(1,0,0));
    jointPtr->addLimitPlane(NxVec3(-1,0,0), globalAnchor +
                             5*NxVec3(1,0,0));
    jointPtr->addLimitPlane(NxVec3(0,0,1), globalAnchor -
                             5*NxVec3(0,0,1));
    jointPtr->addLimitPlane(NxVec3(0,0,-1), globalAnchor +
                             5*NxVec3(0,0,1));
}
```

8.1.8. POINT ON LINE

It can be thought of a point on line (POL) joint as a spherical joint that can translate along one axis. It is free to rotate around all three joint axes and can translate along one axis. An example of a POL joint would be something like a hanger from a shower curtain rod (that is free, as well, to twist along its axis perpendicular to the rod). Figure 15 illustrates this type of joints.

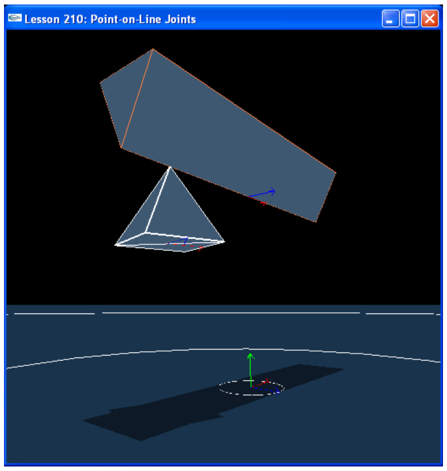

Figure 15. A point on line joint.

Creating the POL joint is identical to creating the PIP joint. The only exception is when defining the limit planes. In this case, the developer needs only to create two limit planes because the joint will translate along its axis.

```
void InitNx()
{
    NxJoint* jointPtr = &polJoint->getJoint();
    jointPtr->setLimitPoint(globalAnchor);
    // Add left-right limiting planes
    jointPtr->addLimitPlane(-globalAxis, globalAnchor +
                            5*globalAxis);
    jointPtr->addLimitPlane(globalAxis, globalAnchor -5*globalAxis);
…
}
```

8.1.9. PULLEY

Pulley joints are joints that simulate pulleys. Normally, this is a device, which is a wheel that has a rope hanging over it that holds up two objects.

In this simulation, this joint consists of two points through which the actors are connected by a "rope", as showed in Figure 16.

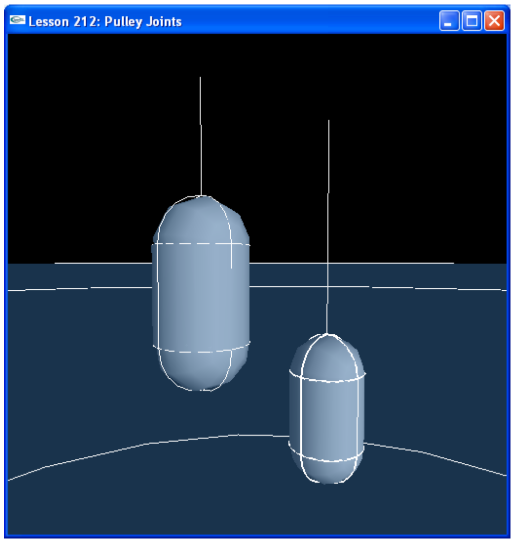

Figure 16. A pulley joint.

This way, developer can make a pulley system more complicated than just a wheel. The developer can change the mechanics of the device so that the pulley wheel feeds rope to one actor or the other at different rates. The rest length and the stiffness of the rope are specified by the developer.

```
NxPulleyJoint* CreatePulleyJoint(NxActor* a0, NxActor* a1, const
NxVec3& pulley0, const NxVec3& pulley1, const NxVec3& globalAxis,
NxReal distance, NxReal ratio)
{
    NxPulleyJointDesc pulleyDesc;
    pulleyDesc.actor[0] = a0;
    pulleyDesc.actor[1] = a1;
    pulleyDesc.localAnchor[0] = NxVec3(0,2,0);
    pulleyDesc.localAnchor[1] = NxVec3(0,2,0);
    pulleyDesc.setGlobalAxis(globalAxis);

    pulleyDesc.pulley[0] = pulley0;
    pulleyDesc.pulley[1] = pulley1;
    // The distance is computed as ||(pulley0 - anchor0)|| +
                                   ||(pulley1 - anchor1)|| * ratio.
    pulleyDesc.distance = distance;
    pulleyDesc.ratio = ratio;
    pulleyDesc.flags = NX_PJF_IS_RIGID;

    // how stiff the constraint is, between 0 and 1 (stiffest)
    pulleyDesc.stiffness = 1;

    pulleyDesc.jointFlags |= NX_JF_COLLISION_ENABLED;

    return (NxPulleyJoint*)gScene->createJoint(pulleyDesc);
}
```

8.2. PARAMETERS

To get simulations closer to reality, when talking about joints, the developer may set some parameters to fine tuning the joint behavior. Each one of these parameters will be described in this section.

8.2.1. *Joint Limits*

Movement restrictions can be applied to joints simulating some aspects of the actual object. These restrictions are called limits, in PhysX's context. Limits can be applied in two manners: giving a range to the relative angle between the bodies that compose the joint, and creating bounding planes that will restrict the joint. These types of limit may also be used together, to achieve a desired effect. The engine doesn't limit the number of bounding planes. Therefore, in the next code branch, an example on how to define angular limits to a joint is given:

First is set the flag on the revolute joint descriptor, enabling the use of joint limits on this joint. Then, it are set joint limit values, one high and one low value. These are the angles through which the lower body is allowed to rotate with respect to the upper body. The low angle is 0 and the high angle is 0.25*NxPi (NxPi is a constant that defines the pi number), or 45 degrees. The restitution parameter, in each limit, is a value between 0 and 1, which defines the bounciness of the limit. In other words, this parameter is responsible for modeling the reaction of the bodies in case of a limit is reached.

8.2.2. *Breakable Joints*

There are situations where joints that could be broke apart are needed. To break a joint, a force or torque has to be applied into the set of bodies. An example of breakable joints is a door, which can be knocked down depending on the strength of the kick.

To implement this breakable feature with PhysX, the developer must inform the maximum force and torque that the joint can hold. On applying higher values, the joint will break, as desired. In sequence, an example follows:

After the joint creation, two max values are defined to limit the forces and torques that may be applied in the set, without cause a joint break. Next, the setBreakable() method is called, receiving as parameters the defined values.

8.2.3. *Joint Motor*

Joint motors are used to supply a relative torque between two bodies connected by a revolute or spherical joint. A simple example is showed in the code branch bellow:

```
{
    NxRevoluteJointDesc revDesc;
    ...

    revDesc.flags |= NX_RJF_MOTOR_ENABLED;

    NxMotorDesc motorDesc;
    motorDesc.velTarget = 1000;
    motorDesc.maxForce = 500;
    motorDesc.freeSpin = true;

    revDesc.motor = motorDesc;

    return (NxRevoluteJoint*)gScene->createJoint(revDesc);
}
```

At first, the motor joint is enabled setting the NX_RJF_MOTOR_ENABLED flag in the joint description. Then, a NxMotorDesc is created to set the motor parameters. The parameters that may be adjusted are:

- velTarget - the maximum velocity value;
- maxForce - the maximum force that will be applied to achieve the velTarget;
- freeSpin - in case of turning off the motor, will the body spin freely (inertia)? If the answer is true, it will be identified when the motor is turned off. This occurs when setting the maxForce to 0.

8.2.4. Joint Spring

Springs can be added in joints to create effects along its acting axis. The effect generated by a spring is quite similar to an actual spring, with stretching and compressing forces, pushing away the parts of the relative joint. Bellow an example is presented:

```
{
    NxRevoluteJointDesc revDesc;
    …
    revDesc.flags |= NX_RJF_SPRING_ENABLED;

    NxSpringDesc springDesc;
    springDesc.spring = 5000;
    springDesc.damper = 50;
    springDesc.targetValue = 0.5*NxPi;

    revDesc.spring = springDesc;

    return (NxRevoluteJoint*)gScene->createJoint(revDesc);
}
```

The target value is the angle 0.5*NxPi, so the second body rotates around z so that it is perpendicular to the ground once the simulation begins. The force that pushes the first body toward its target value is 5000. The spring is damped to 50, so the first body doesn't keep swinging back and forth around the target angle. It eventually loses energy and comes to rest.

9. Collision Detection

Collision detection is an expected feature when talking about physics engines. Specifically in PhysX, collisions are processed by the engine obeying attributes associated to the actors involved, taking into account bounciness, inertia tensors, momentum and others properties. Developers may ignore collision handling, leaving it to the engine. In some cases, events and information about collisions would be useful to create effects, change simulation or game states (contacts between flammable objects and particles of fire), or detect aspects that are important in simulation's context.

The collision concept may be used to create "warning zones", which will trigger events to be handled by specialized modules called User Reports. Within these reports, the actor inside the simulated collision event simply says: "Hey, I'm an actor and I'm passing through this zone! Do something with me or change the environment, please!".

To inspect or handle collision events, PhysX releases some techniques that can be used through "reports" supplied by the engine. These techniques are explained in sequence.

9.1. RAYCASTING

To explain this technique, a simple phrase can be said: "Raycasting is collision detection with rays". This procedure can be applied when doing mouse picking, checking the existence of a line of sight between two objects, calculating distances, simulating shooting actions, or even making ray casting for extra purposes.

As the actual name tells, raycasting consists in casting rays and inspect whatever they hit. A typical "point-and-shoot" technique recovers the objects intercepted by the ray and delivers to the programmer, according to some parameters adjusted before, the method execution.

In PhysX, raycasting can be executed from NxScene through the following methods:

- raycastAnyBounds(), raycastAnyShape() - returns a boolean value indicating if the specified ray hit anything's AABB (Axis Aligned Bounding Box, which is a bounding box that always has its axes aligned with the world axes) or anything's shape, respectively;
- raycastClosestBounds(), raycastClosestShape() - returns the closest shape stabbed by the ray and the distance between the ray emitter and the shape returned. Again, the first method will compute the intersection with the shapes' AABB, against the hard calculation used in the second. The use of an AABB takes advantage of skipping a lot of math operations, being faster than the usual approach, but it lacks on accuracy and may produce mistakes, hitting the wind and signaling that hit some shape that was accidentally close;
- raycastAllBounds(), raycastAllShapes() - returns the number of shapes stabbed by the ray. To enumerate and access all shapes, the developer must provide a NxUserRaycastReport, which will be signaled every time the ray hit any shape.

All methods cited above have as parameters a NxRay, which represents the ray. It has two attributes, being orig the ray origin, and dir the ray normalized direction. It is important to note that NxRay::dir() must always be normalized. Another common parameter is the NxShapesType enum, which may assume the values NX_STATIC_SHAPES, NX_DYNAMIC_SHAPES and NX_ALL_SHAPES.

In raycastClosest*, there is another parameter that will catch all the hitting information, saved in the output parameter hit that is of type NxRaycastHit. NxRaycastHit groups information like shape, world impact point, world impact normal, barycentric coordinates of the hit face, material index and distance to the ray origin.

Some code snippets that illustrate the raycasting technique are presented next. In the following piece of code is showed how to check if a ray can hit anything, by invoking the raycastAnyShape() method:

```
NxRay worldRay;
worldRay.dir = NxVec3(0.0f, 0.0f, 1.0f);
worldRay.orig = NxVec3(0.0f, 0.0f, 0.0f); //along z axis from origin.

if(gScene->raycastAnyShape(worldRay, NX_ALL_SHAPES)) {
      //the ray hit something ...
}
```

In the next code branch, an example presents how to use a NxUserRaycastReport to inspect all shapes hit by the specified ray:

```
class MyRaycastReport : public NxUserRaycastReport
{
    virtual bool onHit(const NxRaycastHit& hit)
    {
        NxActor &hitActor = hit.shape->getActor();

        //The ray hit actor, do something...
        return true;
    }
} gMyReport;

...

NxRay worldRay;
worldRay.dir = NxVec3(0.0f, 0.0f, 1.0f);
worldRay.orig = NxVec3(0.0f, 0.0f, 0.0f); //along z axis from origin.
gScene->raycastAllShapes(worldRay, gMyReport, NX_ALL_SHAPES;)
```

At last, follows a sample of how to retrieve the closest shape stabbed by a ray:

```
NxRay worldRay;
worldRay.dir = NxVec3(0.0f, 0.0f, 1.0f);
worldRay.orig = NxVec3(0.0f, 0.0f, 0.0f); //along z axis from origin.

NxRaycastHit hit;
gScene->raycastClosestShapes(worldRay, NX_ALL_SHAPES, hit);
NxActor &s = hit.shape->getActor();
//do something to the actor
```

It is important to remember that the engine state must remain untouched inside the User Report. In other words, objects cannot be created, modified, or be at influence of forces or torque in the scene. This will be true for all User Reports.

9.2. TRIGGERS

A trigger is a mechanism to detect shape's presence in specific zones. This is implemented with a shape (may be any type of shape) that permits other shapes to pass through it and warns the developerr about the intersection occurred. It is important to notice that Triangle mesh shapes are interpreted as hollow surfaces, not volumes, by the event generator. The act of passing through a trigger can generate a lot of events, including entering, leaving, or simply staying at trigger zone. The trigger event handling may be done by the programmer using an User Report provided by PhysX.

The trigger concept can be used to simulate presence sensors, surveillance areas, automatic doors etc.

The implementation consists in creating a shape, which may be attached to an actor, or not, and setting a NX_TRIGGER_* flag in shapeFlags, within the shape descriptor:

```
// This trigger is a cube
NxBoxShapeDesc boxDesc;
boxDesc.dimensions = NxVec3(10.0f, 10.0f, 10.0f);
boxDesc.shapeFlags |= NX_TRIGGER_ENABLE;
NxActorDesc actorDesc;
actorDesc.shapes.pushBack(&boxDesc);
NxActor * triggerActor = gScene->createActor(actorDesc);
```

In this example the NX_TRIGGER_ENABLE is set in shapeFlags, configuring the system to pass all events to the developer. Passing just specific events is also possible, changing NX_TRIGGER_ENABLE by NX_TRIGGER_ON_ENTER, NX_TRIGGER_ON_LEAVE, NX_TRIGGER_ON_STAY, or a combination of them.

Just setting this parameter does not make the application ready for treating trigger events. It is essential to create a class that will receive and process these events, and tell the engine the application is able to do anything when an event occurs. This is done in two steps:

- Create a NxUserTriggerReport and implement the onTrigger() method.

```
class SensorReport : public NxUserTriggerReport
{
     virtual void onTrigger(NxShape& triggerShape,
   NxShape& otherShape,
   NxTriggerFlag status) {

        if(status & NX_TRIGGER_ON_ENTER)
        {
            NxActor& triggerActor = triggerShape.getActor();
            NxActor& actor = otherShape.getActor();

// Actor entered the trigger region defined by
// triggerActor. Do something(eg explode, shoot at
// etc)...
        }
        if(status & NX_TRIGGER_ON_LEAVE)
         {
            NxActor& triggerActor = triggerShape.getActor();
            NxActor& actor = otherShape.getActor();

            // Actor left the trigger region defined by triggerActor.
            //Do something(eg stop shooting at them)...
        }
    }
} gMySensorReport;
```

The shapes involved in the trigger event are passed to onTrigger(), as well as the NxTriggerFlag, which says what event has been occurred. Possible values in the status parameter are: NX_TRIGGER_ON_ENTER, NX_TRIGGER_ON_LEAVE and NX_TRIGGER_ON_STAY.

- Set the created user trigger report in the scene manager, through the method setUserTriggerReport().

```
gScene->setUserTriggerReport(&gMySensorReport);
```

As seen in the last code line, the scene only permits one user trigger report. Therefore, all shapes marked as triggers will be managed inside the onTrigger() method.

10. Final Considerations

Delivering physics in games and graphics applications is no easy task. It's an extremely compute-intensive environment based on a unique set of physics algorithms that require tremendous amounts of mathematical and logical calculations, supported by massive memory bandwidth.

This way, PhysX SDK helps solving this matter by providing a set of functions, which encapsulate all the hard calculations and algorithms accelerating the development process.

10.1. Wrappers

PhysX SDK has already some wrappers implemented, such as NxOgre. This wrapper is briefly introduced next.

10.1.1. NxOgre

NxOgre is a physics wrapper which allows developer to bridge the gap between the PhysX physics engine and the OGRE 3D rendering system.

With PhysX being very fast, NxOgre does not slow down the physics simulation. It is quite common to have an acceptable frame rate between 30-40 FPS (Frames Per Second) with an excessive amount of rigid bodies on screen, and a far better frame rate with an acceptable amount of bodies.

For further information, the NxOgre webpage at http://www.nxogre.org may be visited, as well as the OGRE webpage at http://www.ogre3d.org.