Learning Physics Modeling with PhysX

Master the PhysX 3 Physics Engine and learn how to program your very own physics simulation

Krishna Kumar

[PACKT]
PUBLISHING

Learning Physics Modeling with PhysX

Master the PhysX 3 Physics Engine and learn how to program your very own physics simulation

Krishna Kumar

Learning Physics Modeling with PhysX

Credits

About the Author

Krishna Kumar is a Graphics and Game Programmer. He completed his Bachelor of Engineering in Computer Science in 2010. Since then, he has been working in the field of graphics, game programming, 3D interactive applications, and virtual reality. He feeds on the advancement of graphics and game technologies. In his free time he learns new things or plays FPS games such as Crysis, Far Cry, and COD. He also maintains a website at `www.gfxguru.org`, which is dedicated to graphics and game programming.

About the Reviewers

Devin Kelly-Collins is currently a student at Kansas State University, pursuing his undergraduate degree in Computer Science. He has mostly worked with Java and C#, developing multithreaded desktop applications and Web applications. He also has experience in developing games using XNA and Unity.

He is currently working with Surface Systems and Instruments, developing software that is used to process road profiling data in real-time. He has also worked with Kansas State University, developing web-based tools.

> I would like to thank my girlfriend, Kalen Wright, for providing me with a base of operations.

Rui Wang is the founder and CTO of Beijing iLyres Technology Co. Ltd. He is in charge of new media interactive applications development. He is one of the most active members of the official OpenSceneGraph community, and contributes to this open source 3D engine regularly. He wrote the books *OpenSceneGraph 3.0 Beginners' Guide*, *OpenSceneGraph 3 Cookbook*, and *Augment Reality with Kinect*, all of which are published by *Packt Publishing*. He is also a novel writer and a guitar lover in his spare time.

> Many thanks to the writer and the *Packt Publishing* team for making such a great book about PhysX, the world-famous Physics Engine. And my deep gratitude to my family, for their love and spiritual support.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?
- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Welcome to *Learning Physics Modeling with PhysX*. Video games are emerging as a new form of entertainment, and are developed for all kind of platforms, such as PCs, consoles, Tablet PC, mobile phones, and other hand-held devices. Current-generation games are much more sophisticated and complex than ever. Third- party physics engines are widely used in video games as middleware to achieve a physically-realistic world behavior such as gravity, acceleration, collision, force, friction, and so on. Nvidia PhysX is the state-of-the-art cross-platform physics engine that is widely used by top-notch game studios and developers. It contains virtually all of the physics-related components that a developer may want to integrate into their game. PhysX Physics Engine exploits the parallel-processing capability of a modern GPU as well as multi-core CPUs to make a game as physically realistic as possible.

PhysX Physics Engine is not only useful for game developers but also for developers who want to make an interactive walkthrough, training, or any other 3D application that requires real-time physics simulation.

What this book covers

Chapter 1, *Starting with PhysX 3 SDK*, covers a brief history, features, licence terms, system requirements, and what's new in PhysX SDK. We will also learn how to configure PhysX SDK with VC++ 2010 compiler.

Chapter 2, *Basic Concepts*, covers the basic concepts of PhysX SDK, including terminologies such as scenes, actors, materials, shapes, and how they are created, updated, and destroyed in PhysX SDK.

Chapter 3, Rigid Body Dynamics, covers rigid body properties such as mass, density, gravity, velocity, force, torque, and damping, and how we can modify these in PhysX SDK. We will also learn about kinematic actors, sleeping state, and the solver accuracy of a rigid body.

Chapter 4, Collision Detection, covers collision shapes and their types, trigger shapes, collision detection phases such as Broad-Phase Collision Detection, Narrow Phase Collision Detection, Enabling Continuous Collision Detection (CCD), and so on.

Chapter 5, Joints, explains exploring joints and their types, such as a fixed joint, revolute joint, spherical joint, distance joint, prismatic joint, and D6 joint.

Chapter 6, Scene Queries, covers types of scene queries such as raycast queries, sweep queries and overlap queries, and their mode operations.

Chapter 7, Character Controller, covers the basics of a character controller, including creating and moving a character controller, updating its size, and other related properties such as auto stepping and slope limit.

Chapter 8, Particles, covers the creation of particles, and particle systems, and their types. We will learn about particle system properties and particle creation, updating, and releasing. We will also cover particle drains and collision filtering.

Chapter 9, Cloth, covers creation of cloth and cloth fabric, tweaking cloth properties such as cloth collision, cloth particle motion constraint and separation constraint, cloth self-collision, intercollision, and GPU acceleration.

Chapter 10, PhysX Visual Debugger (PVD), covers the basics of PVD, connecting to PVD using TCP/IP network, saving a PVD datafile to a disk, and PVD connection flags.

What you need for this book

You need a Windows PC (preferably with Windows 7 OS or higher) with Microsoft Visual C++ 2010 Express compiler installed on it. You can download VC++ 2010 Express for free from `http://www.microsoft.com`. You also need to download Nvidia PhysX SDK 3.3.0 from `https://developer.nvidia.com/physx-downloads`, which requires you to register for the Nvidia Developer Program. You may also want to download the freeglut library for Windows, which is freely available at `http://freeglut.sourceforge.net`. This library is used in the example code to render the PhysX components.

Who this book is for

This book is for game developers, hobbyists, or anybody who wants to learn about the PhysX Physics Engine with minimal prior knowledge of it. You don't have to be a die-hard programmer to get started with this book. Basic knowledge of C++, 3D mathematics, and OpenGL will be fine.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can explicitly wake an actor by calling `PxRigidDynamic::wakeUp()`, which requires an optional real value that determines how long until the body is put to sleep."

A block of code is set as follows:

```
PxMaterial* mMaterial = gPhysicsSDK->createMaterial(0.5,0.5,0.5);
PxRigidDynamic* sphere = gPhysicsSDK->createRigidDynamic(spherePos);
sphere->createShape(PxSphereGeometry(0.5f), *mMaterial);
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes, for example, appear in the text like this: "We have to include the PhysX library files and header files in **VC++ Directories** that can be found at **View | Property Pages**."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title in the subject of your message.

If there is a topic that you have expertise in and on which you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Starting with PhysX 3 SDK

This chapter sheds some light on the history, features, license terms, and system requirements of PhysX. In it, we will learn how to download the PhysX SDK and configure it with MS Visual C++ 2010 for compiling PhysX programs.

Brief history

PhysX SDK is a mature physics engine, which has been under development since 2004. It was developed by **Ageia** with the purchase of **ETH Zurich spin-off NovodeX**. Ageia was a fabless semiconductor company and the first company that developed a dedicated co-processor capable of performing physics calculations , which was much faster than the general purpose CPUs available at that time.

The intention of Ageia was to sell **PPU** (**Physics Processing Unit**) cards much like the dedicated **GPU** (**Graphics Processing Unit**) cards that we buy today. It developed the PhysX software SDK (formerly NovodeX SDK) to harness the processing power of a PPU. The company also licensed out the PhysX SDK as a physics middleware library for game production. Unfortunately, the PPU cards didn't sell very well commercially in the market. On February 4, 2008, **Nvidia** announced that it would acquire Ageia. On February 13, 2008, the merger was finalized. The PhysX engine is now known as Nvidia PhysX. The potential reason of Ageia acquisition by Nvidia was to implement PhysX on top of their CUDA architecture enabled GPU(s), for hardware-accelerated physics processing. The PhysX GPU acceleration is exclusive to Nvidia GPU(s), which gives Nvidia an edge over its competitors; that is, GPU manufacturers such as ATI/AMD.

PhysX SDK 3.3.0 is the latest release at the time of writing this book. PhysX 3.x features a new modular architecture and a completely rewritten PhysX engine. It provides a significant boost in overall performance as well as efficiency. It is a heavily-modified version written to support multiple platforms but has a single base code. Supported platforms include Windows; Linux; Mac OS X; game consoles such as XBOX 360 and PS3; and even Android-powered handheld devices. PhysX 3.3.0 added support for new platforms such as Xbox One, PS 4, Nintendo Wii U, Apple iOS, PS Vita, and Windows RT. PhysX SDK 3.x has undergone architecture and API improvement, and the code is cleaned at many levels to get rid of obsolete and legacy features and to integrate new physics capabilities.

PhysX features

Nvidia PhysX is a state-of-the-art physics engine, which provides the following features:

- **Rigid body dynamics**: Rigid body dynamics is the most essential aspect of physics simulation, and makes use of physics concepts such as position, velocity, acceleration, forces, momentum, impulse, friction, collision, constraints, and gravity. These properties give us the power to simulate or mimic real-world physics scenarios.

- **Character controller**: Character controller is a special type of physics collider, which is mainly used for third-person or first-person player control, or any other kinematic body that may want to take advantage of the properties associated with the character controller.

- **Vehicle dynamics**: Vehicle dynamics gives you the capability to simulate vehicle physics by using spherical wheel shapes that can simulate sophisticated tire friction models. A joint-based suspension system is used for vehicle suspension.

- **Particles and fluid simulation**: Two of the most exciting features of PhysX are particles and fluid simulation. These features can be used to achieve a vast variety of cinematic effects. Particles can used for creating effects such as fire, spark, and debris, whereas fluid particles, also known as **SPH** (**Smoothed Particle Hydrodynamics**), are used to simulate liquid, gases, smoke, or any other SPH-based particle effect.

- **Cloth simulation**: This feature allows you to simulate realistic cloth, which can be used for cloth simulation of the characters in the game or any other cloth-based objects, such as flags and curtains. These cloth objects can also be stretched, torn, or attached to other physical bodies.
- **Softbody simulation**: This feature allows you to simulate volumetric deformable objects.

New in PhysX 3

Notable features in PhysX 3 are as follows:

- Automatic and efficient multithreading, and a unified code base for all supported platforms.
- Improved task manager and a managed-thread pool that is optimized to harness the processing capability of multi-core processors on all platforms.
- A new aggregate concept in which multiple PhysX actors can be combined into one entity having a common collision boundary, which simplifies processing when large numbers of objects are involved.
- A new binary in-place serialization by which we can efficiently insert the PhysX actors into a scene with minimal data copying and without extra memory allocation. Creation and destruction of actors is now decoupled from the insertion and removal of scenes, thus allowing flexible asset management strategies.
- A highly optimized physics runtime that has better a response time, with lower memory footprints.
- A new release of **PhysX Visual Debugger** (**PVD**) that allows for better performance profiling and in-depth memory analysis with enhanced visualization of all PhysX content across all major platforms.
- A full vehicle model that includes components such as engine, clutch, gears, autobox, differential, wheels, tires, suspension, and chassis.

Downloading PhysX SDK and tools

Downloading PhysX SDK requires you to register as an Nvidia developer, which you can do for free at `https://developer.nvidia.com`. Once you have successfully created a Nvidia developer account, you need to apply for the **APEX/PhysX Registered Developer Program**, which you can find by clicking on **My Account** on the top right of the Nvidia developer web page. The approval request may take one to three business days to process.

After the successful approval of your **APEX/PhysX Registered Developer Program** request, click on **PhysX/Tools Download**, select your platform, and then download the PhysX SDK. Please note that for this book, we will download the SDK for PC (Windows) platform. Configuration to include the PhysX SDK's `Include` files and `Library` files that are covered in this chapter is also for the Windows platform.
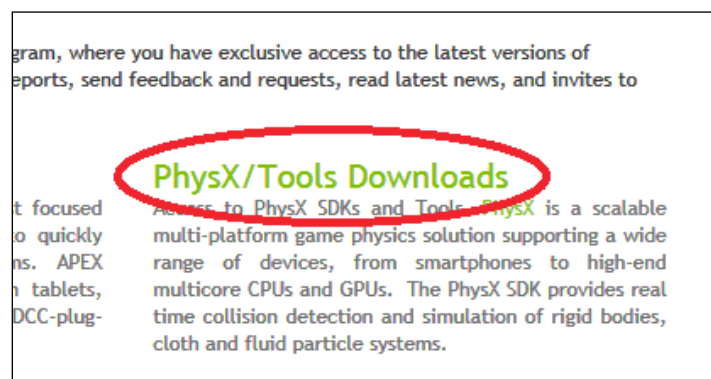
The PhysX SDK license

The Nvidia PhysX SDK is totally free for the Windows platform, both commercial and noncommercial use. For Linux, OS X, and Android platforms, the Nvidia binary PhysX SDK and tools are free for educational and noncommercial use. For commercial use, the binary SDK is free for developers who work on their respective PhysX applications and make less than $100K in gross revenue. More information on license agreements can be found at `https://developer.nvidia.com/content/physx-sdk-eula`.

System requirements for PhysX

The minimum requirements to support the hardware-accelerated PhysX is an Nvidia GeForce 8 series or later GPU with a minimum of 32 CUDA cores and a minimum of 256 MB of dedicated graphics memory. However, each PhysX application has its own GPU and memory recommendations. In general, 512 MB of graphics memory is recommended unless you have a GPU that is dedicated to PhysX. The Nvidia graphics drivers are made in such a way that they can also take advantage of multiple GPU(s) in a system. These can be configured to use one GPU for rendering graphics and the second GPU only for processing PhysX physics.

Configuring with VC++ Express 2010

We will use Microsoft Visual C++ 2010 Express for compiling the PhysX program. It is freely available at `www.microsoft.com`. We have to include the PhysX library files and header files in **VC++ Directories** that can be found at **View | Property Pages**. **Property Pages** can also be modified from **Property Manager**. A **Property Manager** window enables us to modify project settings that are defined in property sheets. A project property sheet is basically an `.xml` file that is used to save project configurations and can also be applied to multiple projects because it is inheritable.

Configuring VC++ 2010 Express requires the following steps:

1. After downloading the PhysX 3.x SDK for the Windows platform, which comes in a ZIP file, you need to extract it to any preferred location on your PC. For this book, we will extract the PhysX SDK's ZIP file to `C:\dev`. Finally, our PhysX SDK location will look like `C:\dev\PhysX-3.3.0_PC_SDK_Core`.

2. Before including the PhysX library files and header files in **Property Manager**, we first need to create a new Visual C++ Win32 Console application. To do this, open your MS VC++ compiler from the toolbar and navigate to **File | New | Project**. Then, a **New Project** window will pop up. Select **Win32 Console Application** and also provide **Name** and **Location** for the project. Finally, click on the **OK** button to proceed further as shown in the following screenshot:
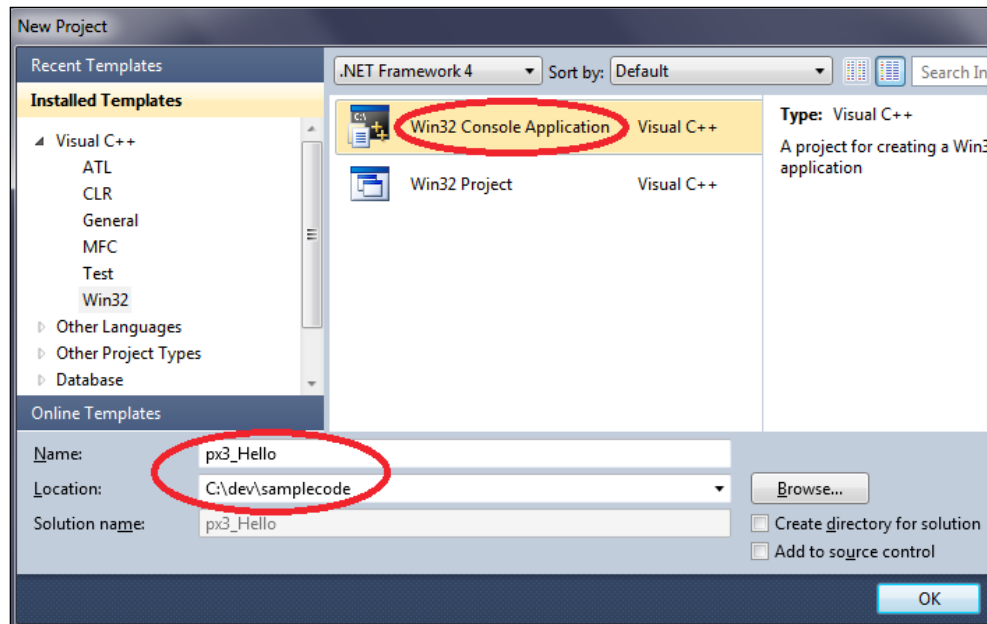
3. Soon after, a **Win32 Application Wizard** window will pop up. Here, click on the **Next** button to get the **Application Settings** screen, where you need to make sure that the **Empty project** option is checked under **Additional options**. Finally, click on the **Finish** button as shown in the following screenshot:
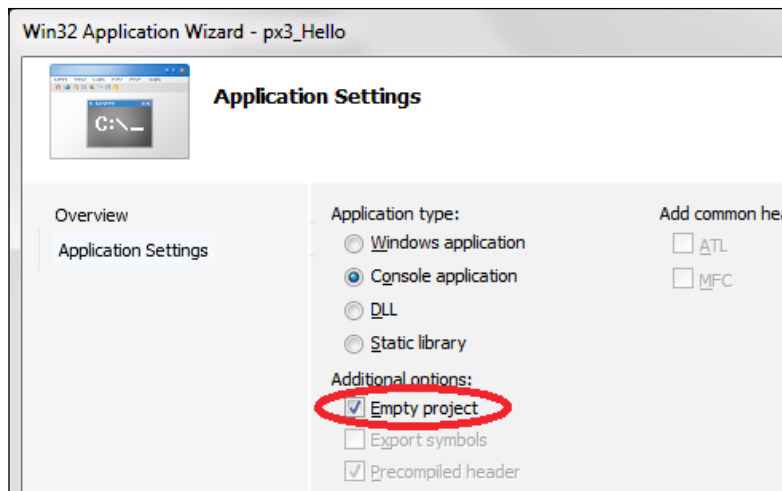
4. Next, we need to configure our project's VC++ directories so that it can find the PhysX SDK header files and libraries that are required for compiling the PhysX program. We will include the absolute path for PhysX SDK **Include Directories** and **Library Directories**. To do this in VC++ 2010 Express, navigate to **View | Property Manager**. If the **Property Manager** option is not visible there, navigate to **Tools | Settings** and select **Expert Settings**; this will enable the **Property Manager** option in **View**. In the **Property Manager** window, double-click on a configuration-and-platform node, for example, **Debug | Win32** or **Release | Win32,** as shown in the following screenshot:
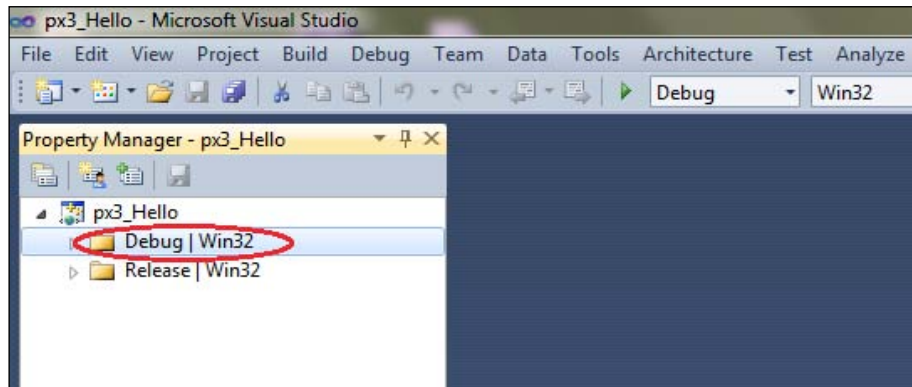
5. Double-clicking on a configuration-and-platform node, such as **Debug | Win32** or **Release | Win32**, will open **Property pages** for the respective node configuration, such as, **Debug Property Pages** or **Release Property Pages**. This can also be opened by navigating to **View | Property pages**.

6. When configuration-specific Property Pages (namely **Debug Property Pages** or **Release Property Pages**) will pop up, select **VC++ Directories** and add the following entries:

    1. Select **Include Directories** and then click on **<Edit...>** to add `C:\dev\ PhysX-3.3.0_PC_SDK_Core\Include`.

    2. Select **Library Directories** and then click on **<Edit...>** to add `C:\dev\ PhysX-3.3.0_PC_SDK_Core\Lib\win32` (for a 32-bit platform) or `C:\ dev\PhysX-3.3.0_PC_SDK_Core\Lib\win64` (for a 64-bit platform).
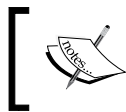
> For this book, we will include libraries for a 32-bit platform because it can run on either a 32-bit machine or a 64-bit machine.
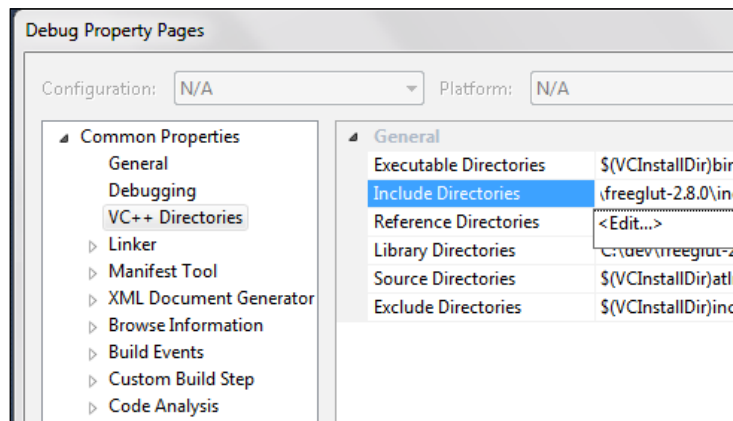
7. Finally, click on the **OK** button to save your changes and close the window.

These PhysX SDK directory settings are saved on a per user basis and not on per project basis. So whenever you create a new VC++ project in VC++ 2010 Express, PhysX directories will automatically be added to your `Include Directories` project. We are now finally done with the PhysX configuration in VC++ 2010 Express. In the next chapter, we will create our first PhysX program.

Summary

This chapter enlightened us with all of the basic PhysX-related information that is needed to proceed with the rest of this book. We learned how to register as a Nvidia developer and download the PhysX SDK from the Nvidia website. We also learned how to include the PhysX SDK files in Visual C++ 2010 for compiling PhysX programs.

2
Basic Concepts

This chapter provides an overview of the concepts that we use in PhysX. It will familiarize you with terms such as scene, actor, material, shape, and so on.

The topics covered in this chapter are as follows:

- Initializing PhysX and creating the scene and actors
- Creating shapes and materials and then assigning them to actors
- Simulating and then shutting down PhysX

Scene and Actors

You must have heard the quote written by William Shakespeare:

> "*All the world's a stage, and all the men and women merely players: they have their exits and their entrances; and one man in his time plays many parts, his acts being seven ages.*"

As per my interpretation, he wanted to say that this world is like a stage, and human beings are like players or actors who perform our role in it. Every actor may have his own discrete personality and influence, but there is only one stage, with a finite area, predefined props, and lighting conditions.

In the same way, a world in PhysX is known as *scene* and the players performing their role are known as *actors*. A scene defines the property of the world in which a simulation takes place, and its characteristics are shared by all of the actors created in the scene. A good example of a scene property is *gravity*, which affects all of the actors being simulated in a scene. Although different actors can have different properties, independent of the scene. An instance of a scene can be created using the `PxScene` class.

An actor is an object that can be simulated in a PhysX scene. It can have properties, such as shape, material, transform, and so on. An actor can be further classified as a static or dynamic actor; if it is a static one, think of it as a prop or stationary object on a stage that is always in a static position, immovable by simulation; if it is dynamic, think of it as a human or any other moveable object on the stage that can have its position updated by the simulation. Dynamic actors can have properties like mass, momentum, velocity, or any other rigid body related property. An instance of static actor can be created by calling PxPhysics::createRigidStatic()function, similarly an instance of dynamic actor can be created by calling PxPhysics::createRigidDynamic() function. Both functions require single parameter of PxTransform type, which define the position and orientation of the created actor.
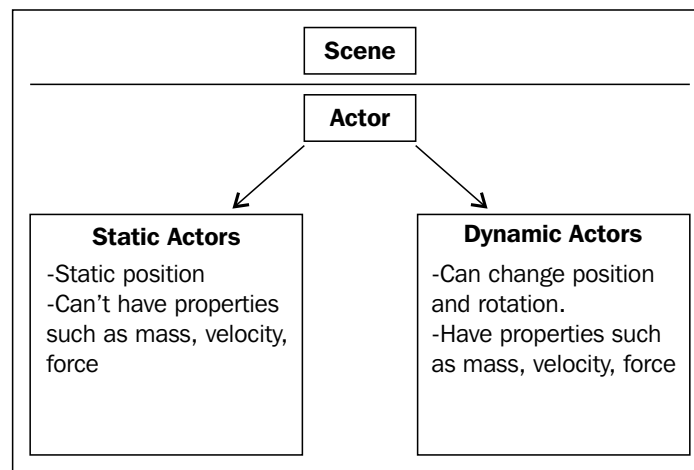
Materials

In PhysX, a material is the property of a physical object that defines the friction and restitution property of an actor, and is used to resolve the collision with other objects. To create a material, call `PxPhysics::createMaterial()`, which requires three arguments of type `PxReal`; these represent static friction, dynamic friction and restitution, respectively.

A typical example for creating a PhysX material is as follows:

```
PxMaterial* mMaterial = gPhysicsSDK->createMaterial(0.5,0.5,0.5);
```

Static friction represents the friction exerted on a rigid body when it is in a rest position, and its value can vary from 0 to infinity. On the other hand, dynamic friction is applicable to a rigid body only when it is moving, and its value should always be within 0 and 1. Restitution defines the bounciness of a rigid body and its value should always be between 0 and 1; the body will be more bouncy the closer its value is to 1. All of these values can be tweaked to make an object behave as bumpy as a Ping-Pong ball or as slippery as ice when it interacts with other objects.

Shapes

When we create an actor in PhysX, there are some other properties, like its shape and material, that need to be defined and used further as function parameters to create an actor. A shape in PhysX is a collision geometry that defines the collision boundaries for an actor. An actor can have more than one shape to define its collision boundary. Shapes can be created by calling `PxRigidActor::createShape()`, which needs at least one parameter each of type `PxGeometry` and `PxMaterial` respectively.

A typical example of creating a PhysX shape of an actor is as follows:

```
PxMaterial* mMaterial = gPhysicsSDK->createMaterial(0.5,0.5,0.5);
PxRigidDynamic* sphere = gPhysicsSDK->createRigidDynamic(spherePos);
sphere->createShape(PxSphereGeometry(0.5f), *mMaterial);
```

An actor of type `PxRigidStatic`, which represents static actors, can have shapes such as a sphere, capsule, box, convex mesh, triangular mesh, plane, or height field. Permitted shapes for actors of the `PxRigidDynamic` type that represents dynamic actors depends on whether the actor is flagged as *kinematic* (the kinematic actor is explained in the next chapter) or not. If the actor is flagged as *kinematic*, it can have all of the shapes of an actor of the `PxRigidStatic` type; otherwise it can have shapes such as a sphere, capsule, box, convex mesh, but not a triangle mesh, a plane, or a height field.

Creating the first PhysX 3 program

Now we have enough understanding to create our first PhysX program. In this program, we initialize PhysX SDK, create a scene, and then add two actors. The first actor will be a static plane that will act as a static ground, and the second will be a dynamic cube positioned a few units above the plane. Once the simulation starts, the cube should fall on to the plane under the effect of gravity.

Because this is our first PhysX code, to keep it simple, we will not draw any actor visually on the screen. We will just print the position of the falling cube on the console until it comes to rest.

We will start our code by including the required header files. `PxPhysicsAPI.h` is the main header file for PhysX, and includes the entire PhysX API in a single header. Later on, you may want to selectively include only the header files that you need, which will help to reduce the application size. We also load the three most frequently used precompiled PhysX libraries for both the **Debug** and **Release** platform configuration of VC++ 2010 Express compiler shown as follows:

In addition to the `std` namespace, which is a part of standard C++, we also need to add the `physx` namespace for PhysX, as follows:

```
#include <iostream>

#include <PxPhysicsAPI.h> //PhysX main header file

//-------Loading PhysX libraries----------]
#ifdef _DEBUG
#pragma comment(lib, "PhysX3DEBUG_x86.lib")
#pragma comment(lib, "PhysX3CommonDEBUG_x86.lib")
#pragma comment(lib, "PhysX3ExtensionsDEBUG.lib")
#else
#pragma comment(lib, "PhysX3_x86.lib")
#pragma comment(lib, "PhysX3Common_x86.lib")
#pragma comment(lib, "PhysX3Extensions.lib")
#endif
using namespace std;
using namespace physx;
```

Initializing PhysX

For initializing PhysX SDK, we first need to create an object of type `PxFoundation` by calling the `PxCreateFoundation()` function. This requires three parameters: the version ID, an allocator callback, and an error callback. The first parameter prevents a mismatch between the headers and the corresponding SDK DLL(s).

The allocator callback and error callback are specific to an application, but the SDK also provides a default implementation, which is used in our program. The foundation class is needed to initialize higher-level SDKs.

The code snippet for creating a foundation of PhysX SDK is as follows:

```
static PxDefaultErrorCallback  gDefaultErrorCallback;
static PxDefaultAllocator      gDefaultAllocatorCallback;
static PxFoundation*        gFoundation = NULL;


//Creating foundation for PhysX
gFoundation = PxCreateFoundation
  (PX_PHYSICS_VERSION, gDefaultAllocatorCallback,
   gDefaultErrorCallback);
```

After creating an instance of the foundation class, we finally create an instance of PhysX SDK by calling the `PxCreatePhysics()` function. This requires three parameters: the version ID, the reference of the `PxFoundation` object we created earlier, and `PxTolerancesScale`. The `PxTolerancesScale` parameter makes it easier to author content on different scales and still have PhysX work as expected; however, to get started, we simply pass a default object of this type. We make sure that the PhysX device is created correctly by comparing it with NULL. If the object is not equal to NULL, the device was created successfully.

The code snippet for creating an instance of PhysX SDK is as follows:

```
static PxPhysics*       gPhysicsSDK = NULL;

//Creating instance of PhysX SDK
gPhysicsSDK = PxCreatePhysics
  (PX_PHYSICS_VERSION, *gFoundation, PxTolerancesScale() );

if(gPhysicsSDK == NULL)
{
cerr<<"Error creating PhysX3 device, Exiting..."<<endl;
  exit(1);
}
```

Creating scene

Once the PhysX device is created, it's time to create a PhysX scene and then add the actors to it. You can create a scene by calling `PxPhysics::createScene()`, which requires an instance of the `PxSceneDesc` class as a parameter. The object of `PxSceneDesc` contains the description of the properties that are required to create a scene, such as gravity.

The code snippet for creating an instance of the PhysX scene is given as follows:

```
PxScene*    gScene = NULL;

//Creating scene
PxSceneDesc sceneDesc(gPhysicsSDK->getTolerancesScale());

sceneDesc.gravity = PxVec3(0.0f, -9.8f, 0.0f);
sceneDesc.cpuDispatcher = PxDefaultCpuDispatcherCreate(1);
sceneDesc.filterShader = PxDefaultSimulationFilterShader;

gScene = gPhysicsSDK->createScene(sceneDesc);
```

Then, one instance of `PxMaterial` is created, which will be used as a parameter for creating the actors.

```
//Creating material
PxMaterial* mMaterial =
//static friction, dynamic friction, restitution
gPhysicsSDK->createMaterial(0.5,0.5,0.5);
```

Creating actors

Now it's time to create actors; our first actor is a plane that will act as a ground. When we create a plane in PhysX, its default orientation is vertical, like a wall, but we want it to act like a ground. So, we have to rotate it by 90 degrees so that its normal will face upwards. This can be done using the `PxTransform` class to position and rotate the actor in 3D world space. Because we want to position the plane at the origin, we put the first parameter of `PxTransform` as `PxVec3(0.0f,0.0f,0.0f)`; this will position the plane at the origin. We also want to rotate the plane along the z-axis by 90 degrees, so we will use `PxQuat(PxHalfPi,PxVec3(0.0f,0.0f,1.0f))` as the second parameter.

Now we have created a rigid static actor, but we don't have any shape defined for it. So, we will do this by calling the `createShape()` function and putting `PxPlaneGeometry()` as the first parameter, which defines the plane shape and a reference to the `mMaterial` that we created before as the second parameter. Finally, we add the actor by calling `PxScene::addActor` and putting the reference of `plane`, as shown in the following code:

```
//1-Creating static plane
PxTransform planePos = PxTransform(PxVec3(0.0f, 0,
0.0f),PxQuat(PxHalfPi, PxVec3(0.0f, 0.0f, 1.0f)));
PxRigidStatic* plane =  gPhysicsSDK->createRigidStatic(planePos);
plane->createShape(PxPlaneGeometry(), *mMaterial);
gScene->addActor(*plane);
```

The next actor we want to create is a dynamic actor having box geometry, situated 10 units above our static plane. A rigid dynamic actor can be created by calling the `PxCreateDynamic()` function, which requires five parameters of type: `PxPhysics`, `PxTransform`, `PxGeometry`, `PxMaterial`, and `PxReal` respectively. Because we want to place it 10 units above the origin, the first parameter of `PxTransform` will be `PxVec3(0.0f,10.0f,0.0f)`. Notice that they component of the vector is 10, which will place it 10 units above the origin. Also, we want it at its default identity rotation, so we skipped the second parameter of the `PxTransform` class. An instance of `PxBoxGeometry` also needs to be created, which requires `PxVec3` as a parameter, which describes the dimension of a cube in half extent. We finally add the created actor to the PhysX scene by calling `PxScene::addActor()` and providing the reference of `gBox` as the function parameter.

```
PxRigidDynamic*gBox);

//2) Create cube
PxTransform     boxPos(PxVec3(0.0f, 10.0f, 0.0f));
PxBoxGeometry  boxGeometry(PxVec3(0.5f,0.5f,0.5f));

gBox = PxCreateDynamic(*gPhysicsSDK, boxPos, boxGeometry, *mMaterial,
1.0f);
gScene->addActor(*gBox);
```

Simulating PhysX

Simulating a PhysX program requires calculating the new position of all of the PhysX actors that are under the effect of Newton's law, for the next time frame. Simulating a PhysX program requires a time value, also known as time step, which forwards the time in the PhysX world. We use the `PxScene::simulate()` method to advance the time in the PhysX world. Its simplest form requires one parameter of type `PxReal`, which represents the time in seconds, and this should always be more than 0, of else the resulting behavior will be undefined. After this, you need to call `PxScene::fetchResults()`, which will allow the simulation to finish and return the result. The method requires an optional Boolean parameter, and setting this to true indicates that the simulation should wait until it is completed, so that on return the results are guaranteed to be available.

```
//Stepping PhysX
PxReal  myTimestep = 1.0f/60.0f;
void StepPhysX()
{
  gScene->simulate(myTimestep);
  gScene->fetchResults(true);
}
```

We will simulate our PhysX program in a loop until the dynamic actor (box) we created 10 units above the ground falls to the ground and comes to an idle state. The position of the box is printed on the console for each time step of the PhysX simulation. By observing the console, you can see that initially the position of the box is (0, 10, 0), but the y component, which represents the vertical position of the box, is decreasing under the effect of gravity during the simulation. At the end of loop, it can also be observed that the position of the box in each simulation loop is the same; this means the box has hit the ground and is now in an idle state.

```
//Simulate PhysX 300 times
for(int i=0; i<=300; i++)
{
//Step PhysX simulation
  if(gScene)
    StepPhysX();

  //Get current position of actor (box) and print it
  PxVec3 boxPos = gBox->getGlobalPose().p;
cout<<"Box current Position ("<<boxPos.x <<" "<<boxPos.y <<"
"<<boxPos.z<<")\n";
}
```

Shutting down PhysX

Now that our PhysX simulation is done, we need to destroy the PhysX related objects and release the memory.

Calling the `PxScene::release()` method will remove all actors, particle systems, and constraint shaders from the scene. Calling `PxPhysics::release()` will shut down the entire physics. Soon after, you may want to call `PxFoundation::release()` to release the foundation object, as follows:

```
void ShutdownPhysX()
{
  gScene->release();
  gPhysicsSDK->release();
  gFoundation->release();
}
```

Summary

We finally created our first PhysX program and learned its steps from start to finish. To keep our first PhysX program short and simple, we just used a console to display the actor's position during simulation, which is not very exciting; but it was the simplest way to start with PhysX. In subsequent chapters, we will also visualize simulation by using the OpenGL library.

3
Rigid Body Dynamics

The topics that are covered in this chapter are as follows:

- Basics of rigid body dynamics.
- Changing rigid body properties, namely mass, density, velocity, acceleration, and angular motion.
- Rigid body sleeping state and solver accuracy.

Exploring a rigid body

A rigid body is a physical body having definite mass and a fixed shape. One important property of a rigid body is that it always retains its original shape when we apply external force to it. This property greatly simplifies the physics-related calculations performed on a body (actor), assuming that it follows the properties of a perfect rigid body. A rigid body can also be made of multiple interconnected rigid bodies, and can have properties, such as velocity, force, torque, center of gravity, angular motion, and others. In this chapter we will learn methods and parameters for changing the properties of rigid body dynamics.

Mass

Mass is one of the essential properties of a rigid body; other properties, such as moment of inertia and momentum also depend on it. In PhysX the easiest way to set the mass of a rigid body is by calling `PxRigidBody::setMass()`, which requires a single parameter of type `PxReal`, which represents the mass. The function `PxRigidBody::getMass()` can also be used for getting the current mass of any rigid body.

Density

The density of a rigid dynamic actor is the mass per unit size of its colliding shape. The mass of a rigid dynamic actor is proportional to the size of its shape. This means that if you increase the size of its shape, the mass of the object will also increase. The density of a rigid body can be set by calling `PxRigidBodyExt::updateMassAndInertia()`, which takes two parameters. The first parameter is of type `PxRigidBody` and takes the reference of a rigid body whose density needs to be updated. The second parameter is of type `PxReal` and sets the density of the body.

Gravity

Gravity is an essential and simple-to-use property for realistic physics simulation. We have already used this property in our first PhysX program. To set the gravity for a scene, simply call the `PxScene::setGravity()` method, which requires a parameter of type `PxVec3` containing three real numbers. These three numbers represent acceleration due to gravity in the x, y, and z axes respectively. Here, distance is represented in meters, and time is represented in seconds.

In a typical physics simulation, gravity is generally set at `PxVec3(0.0f, -9.8f, 0.0f)`, which is the same as the Earth's gravity. PhysX has the ability to enable or disable gravity for each dynamic rigid body created. To do this, just set the following flag to `true`:

```
PxActor::setActorFlag(PxActorFlag::eDISABLE_GRAVITY, true);
```

Velocity

In PhysX we can set the linear velocity of a dynamic rigid body at any point of time by calling `PxRigidBody::setLinearVelocity()`. In its simplest form, this requires a parameter of type `'PxVec3'`, which contains three real numbers representing velocity in the x, y, and z axes respectively. We can also set the angular rotation of a rigid dynamic body by calling `PxRigidBody::setAngularVelocity()` and providing `PxVec3` as a parameter, which represents the magnitude of angular velocity in the x, y, and z axes respectively.

Force and Torque

In PhysX one way to move a rigid dynamic actor is by applying a force at its center of mass, which causes it to move in a linear motion. We can also rotate a dynamic actor about its axis by applying a torque to it.

This can be done in PhysX using the following two functions:

```
void PxRigidBody::addForce
   (const PxVec3& force, PxForceMode::Enum mode, bool autowake);

void PxRigidBody::addTorque
   (const PxVec3& torque, PxForceMode::Enum mode, bool autowake);
```

The simplest form of `PxRigidBody::addForce()` requires a 3D vector, which represents the magnitude of force and its direction in the x, y, and z axes of the 3D world. The second and third parameters are optional. The second parameter is an `enum` of type `PxForceMode`, which defaults to `PxForceMode::eFORCE`. The other possibilities are `PxForceMode::eIMPULSE`, which applies an impulsive force, and `PxForceMode::eVELOCITY_CHANGE`, which applies force on a rigid body without taking its mass into consideration. The third parameter is a Boolean value, which defaults to `true`, which wakes the actor up, if it is sleeping.

There are some more member functions of class `PxRigidBodyExt`, which provide a more fine-grained way to apply force on dynamic rigid bodies, as follows:

```
void PxRigidBodyExt::addForceAtPos
   (PxRigidBody& body, const PxVec3& force,
   const PxVec3& pos, PxForceMode::Enum mode, bool wakeup);

void PxRigidBodyExt::addForceAtLocalPos
   (PxRigidBody& body, const PxVec3& force,
   const PxVec3& pos, PxForceMode::Enum mode, bool wakeup);

void PxRigidBodyExt::addLocalForceAtPos
   (PxRigidBody& body, const PxVec3& force,
   const PxVec3& pos, PxForceMode::Enum mode, bool wakeup);

void PxRigidBodyExt::addLocalForceAtLocalPos
   (PxRigidBody& body, const PxVec3& force,
   const PxVec3& pos, PxForceMode::Enum mode, bool wakeup);
```

`PxRigidBodyExt::addForceAtPos()` applies a force defined in the *global coordinate* frame, acting at a particular point in *global coordinates* to the actor.

`PxRigidBodyExt::addForceAtLocalPos()` applies a force defined in the *local coordinate* frame, acting at a particular point in *global coordinates* to the actor.

`PxRigidBodyExt::addLocalForceAtPos()` applies a force defined in the *actor local coordinate* frame, acting at a particular point in *global coordinates* to the actor.

`PxRigidBodyExt::addLocalForceAtLocalPos()` applies a force defined in the *actor local coordinate* frame, acting at a particular point in *local coordinates* to the actor.

Please note that the global coordinate (world coordinate) here is the PhysX scene coordinate. On the other hand, the local coordinate will always be referred to some PhysX actor's local coordinate system. This is explained in the following figure:
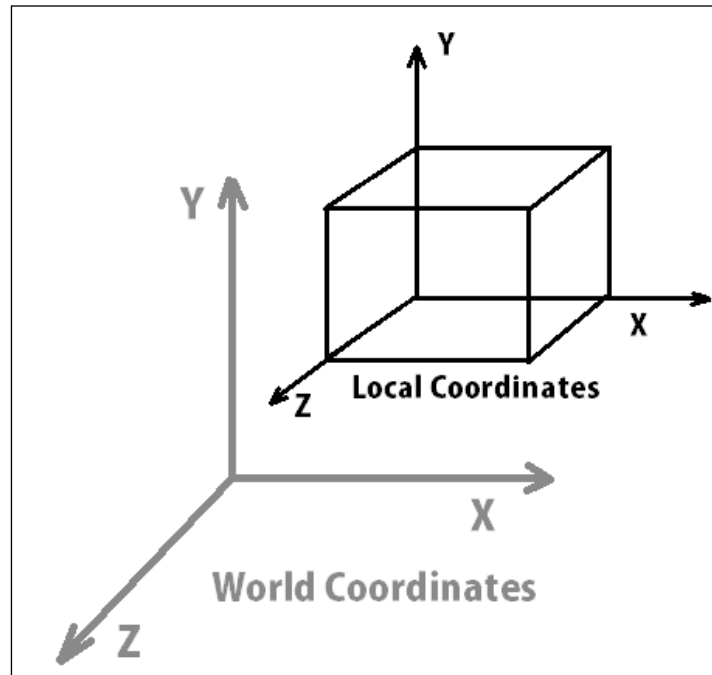
Damping

Damping is a property that reduces the linear and angular momentum of a rigid dynamic actor until it comes to rest, assuming that no external force is applied. The functions that are used to set the linear and angular momentum of a rigid actor are in shown the following lines of code:

```
void PxRigidDynamic::setLinearDamping(PxReal linDamp);

void PxRigidDynamic::setAngularDamping(PxReal angDamp);
```

`PxRigidDynamic::setLinearDamping()` sets linear damping for a rigid body, and requires a parameter of type `PxReal`, which represents the magnitude of the damping force, which is equal to `linDamp` times the velocity. In the same way, `Px RigidDynamic::setAngularDamping()` sets angular damping, and requires a parameter of type `PxReal`, which represents a damping torque equal to `angDamp` times the angular velocity.

Kinematic actors

Kinematic actors are rigid dynamic actors that move in the physics world by calling an explicit update function, and that don't follow Newton's law of motion directly. A kinematic actor can influence any other rigid dynamic body in the scene but the opposite is not true. Kinematic actors are mostly used for character controllers in a physics engine, which update the game's character position and collision boundaries. To make a rigid dynamic actor kinematic, the following function is called:

```
PxRigidDynamic::setRigidDynamicFlag(PxRigidDynamicFlag::
  eKINEMATIC, true
```

This function basically sets the `PxRigidDynamicFlag::eKINEMATIC` flag to `true`.

Once an actor is made kinematic, its position is updated by calling `PxRigidDyna mic::setKinematicTarget()`, which requires a parameter of type `PxTransform` and contains the next destination to move to. The `setKinematicTarget()` function is called in every simulation step, which will move the kinematic body in the physics world regardless of external force, gravity, or collision acting on the body. A kinematic body will always be treated as if it has infinite mass, and will push any dynamic actor that gets in its way.

Sleeping state

In PhysX, a rigid dynamic actor can have a sleeping state, which decides whether it will be simulated or not. This is done to enhance the performance of PhysX simulation by avoiding unnecessary calculations. An actor comes to a sleeping state when its kinetic energy is below some threshold value for a certain period of time. Once a rigid body is in a sleeping state, it can wake up only if it comes in contact with another awake object, or if the application of the rigid body changes its position or velocity. We can explicitly wake an actor up by calling `PxRigidDynamic::wakeUp()`, which requires an optional real value that determines how long until the body is put to sleep. We can also put an actor in the sleep state by explicitly calling `PxRigidDynamic::putToSleep()`. To check whether an actor is sleeping or not in the PhysX scene, we can call `PxRigidDynamic::isSleeping()`, which returns `true` if the actor is sleeping.

Solver accuracy

When a collision is detected in the physics engine, a proper response has to be calculated, depending on the properties of the colliding objects. Solvers come into play to resolve the collision response. Solvers try to satisfy the constraints applied to a body and restrict the body motion by iterating through all of the constraints of the body. Increasing the number of iterations will result in a more accurate simulation. By default, the PhysX solver iteration is set to 4 for position iteration and 1 for velocity iteration. These iterations can be individually set for each actor by using the following function:

```
void PxRigidDynamic::setSolverIterationCounts(PxU32 minPositionIters,
PxU32 minVelocityIters);
```

In general, increasing the iteration count is only required for actors with a large number of joints and a lower tolerance for joint inaccuracy.

Summary

In this chapter we learnt about many properties that are related to rigid dynamic bodies. These properties include sleeping state and solver accuracy. We learnt how to set the mass, density, and gravity for a rigid dynamic body, and how to apply force, torque, and damping on it. We also learnt about kinematic bodies and where they can be used.

4

Collision Detection

The topics covered in this chapter are as follows:

- Collision shapes and types of geometries available in PhysX.
- Trigger shapes, simulation events, and filter shader.
- Broad-Phase collision detection and Narrow-Phase collision detection.
- Continuous collision detection.

Collision shapes

This chapter covers collision detection and some other important topics related to it. A collision can only occur when two or more bodies having definite shape collide with (hit) each other. Therefore, for a PhysX rigid body actor, a shape is always required to define its spatial volume.

We have already learnt about shapes and materials in *Chapter 2*, *Basic Concepts*, where we saw that creating a shape in PhysX requires an object of `PxGeometry` and a reference to `PxMaterial`. Here, the spatial volume of a PhysX actor (geometry) is defined by the `PxGeometry` class. PhysX SDK provides some commonly-used geometries for defining the shape of an actor, as explained in the following sections.

Geometry

As we know, an instance of geometry is required for creating a shape in PhysX, and this defines the collision boundary (spatial volume) of a PhysX actor. The types of geometries available in PhysX are given in the following sections.

Sphere

The function `PxSphereGeometry()` defines a sphere geometry for an actor's shape, and requires a single parameter of type `PxReal`, which represents the radius of the sphere.

The code snippet for creating a sphere shape out of sphere geometry is as follows:

```
PxReal radius = 2.0f;
gPhysicsSDK->createShape(PxSphereGeometry(radius),gMaterial);
```

Here, `gPhysicsSDK` is the instance of the created PhysX SDK and `gMaterial` the instance of the created PhysX material.

Box

The function `PxBoxGeometry()` defines a box geometry for an actor's shape, and requires three parameters of type `PxReal`, which represent side lengths of a box in half extent. That means if you want a side of length *a*, you have to give its value in the parameter as *a/2*.

The code snippet for creating a box shape out of a box geometry is as follows:

```
PxReal hx = 0.5f; //half-extent x
PxReal hy = 0.5f; //half-extent y
PxReal hz = 0.5f; //half-extent z

gPhysicsSDK->createShape(PxBoxGeometry(hx,hy,hz),gMaterial);
```

Capsule

The function `PxCapsuleGeometry ()` defines a capsule geometry, and requires two parameters of type `PxReal`. The first parameter represents the radius of the capsule, and the second parameter represents the height in half extent.

The code snippet for creating a capsule shape out of a capsule geometry is as follows:

```
PxReal r = 0.5f; // capsule radius
PxReal h = 1.0f; // capsule half-extent height

gPhysicsSDK->createShape(PxCapsuleGeometry(r,h),gMaterial);
```

Plane

The function `PxPlaneGeometry()` divides the PhysX space into *above* and *below*. Everything *below* the plane will collide with it. The function doesn't require any parameters, and the plane's collision volume totally depends on its position.

The code snippet for creating a plane shape out of plane geometry is as follows:

```
gPhysicsSDK->createShape(PxPlaneGeometry(),gMaterial);
```

Trigger shapes

All shapes that can be created in PhysX can be flagged as a trigger shape. A trigger shape doesn't participate in the simulation, but can be configured to participate in scene queries. Its shape doesn't involve any physical collision. A trigger shape is mainly used to detect if any rigid body overlapped its spatial volume or not. If yes, it triggers a callback function. This feature is frequently used in games. For example, let's say you are making a sci-fi game in which whenever a player comes in front of any door, it is automatically opened for the player. To do this, simply place an actor with a cube shape in front of the door and flag it as `triggerShape`. Then write a game logic that plays the door opening animation whenever the cube actor detects an overlap between the player's shape and its own shape.

The code snippet to flag the shape of an actor as a trigger is as follows:

```
//Setting trigger flag true for 'triggerShape'
PxShape* triggerShape;
triggerActor->getShapes(&triggerShape, 1);
triggerShape->setFlag(PxShapeFlag::eSIMULATION_SHAPE, false);
triggerShape->setFlag(PxShapeFlag::eTRIGGER_SHAPE, true);
```

The important thing in the code snippet is that we are setting `PxShapeFlag::eSIMULATION_SHAPE` to `false`, which will disable its participation in physical simulation, and are setting its `PxShapeFlag::eTRIGGER_SHAPE` flag to `true`, which will make it act as a trigger shape.

The overlapping of the trigger shape with other rigid dynamic bodies is reported through the implementation of `PxSimulationEventCallback::onTrigger`. Thus, you have to inherit your program class from the `PxSimulationEventCallback` class.

Simulation event

Whenever a rigid body overlaps with a *trigger shape,* or when two rigid bodies collide with each other, a function callback is called, providing the required filter shader. These simulation events can be received by inheriting the `PxSimulationEventCallback` class. This is basically an interface class that helps us listen to simulation events. Once you create a derived class from the `PxSimulationEventCallback` class, an instance of it is used to register the simulation callback in PhysX.

The code snippet for registering the simulation event callback is as follows:

```
// sceneDesc is an instance of 'PxSceneDesc'
sceneDesc.simulationEventCallback = &gContactReportCallback;
```

In this example, `sceneDesc` is the instance of the `PxSceneDesc` class, and the function `gContactReportCallback` is the instance of a class derived from `PxSimulationEventCallback`.

The `PxSimulationEventCallback` class provides two collision query-related function callbacks, which are explained in the following sections.

Trigger event

The function `PxSimulationEventCallback::onTrigger()` is called whenever a *trigger shape* is overlapped by any rigid body shape that provides some filter shader configuration. The callback function has two parameters. The first parameter is of type `PxTriggerPair`, and provides an array of all of the trigger pairs of the PhysX simulation. The second parameter is of type `PxU32`, and is the total number of trigger pairs in the PhysX simulation. The `PxTriggerPair` class contains information for two actors that form trigger-pairs, for example, the shape, which is marked as a trigger, the actor attached with the trigger shape, and the other shape, which caused the trigger event.

By using the `PxSimulationEventCallback::onTrigger()` callback function we can iterate through all of the trigger pairs of the PhysX scene and perform any trigger event dependent task.

The code snippet for getting information about the trigger actor and overlapped an actor from the trigger-pair is given as follows:

```
void onTrigger(PxTriggerPair* pairs, PxU32 nbPairs)
{

//loop through all trigger-pairs of PhysX simulation
  for(PxU32 i=0; i < nbPairs; i++)
```

```
    {
      //get current trigger actor & other actor info
      //from current trigger-pair
      const PxTriggerPair& curTriggerPair = pairs[i];

      PxRigidActor* triggerActor = curTriggerPair.triggerActor;
      PxRigidActor* otherActor = curTriggerPair.otherActor;

    }
  }
```

Contact event

The function `PxSimulationEventCallback::onContact()` is called whenever two rigid bodies collide with each other. It has three parameters. The first parameter is of type `PxContactPairHeader`, and contains information about two actors forming a contact-pair, and related flag information. The second parameter is of type `PxContactPair`, and provides all of the contact-pairs of the PhysX simulation. The third parameter is of type `PxU32`, and is the total count of contact-pairs in the PhysX simulation.

The code snippet for printing contact positions of all contact-pairs is given as follows:

```
void ContactReportCallback::onContact
  (const PxContactPairHeader& pairHeader,
   const PxContactPair* pairs, PxU32 nbPairs)
{
  const PxU32 buff = 64; //buffer size
  PxContactPairPoint contacts[buff];

  //loop through all contact pairs of PhysX simulation
  for(PxU32 i=0; i < nbPairs; i++)
  {
    //extract contant info from current contact-pair
    const PxContactPair& curContactPair = pairs[i];
    PxU32 nbContacts = curContactPair.extractContacts
      (contacts, buff);

    for(PxU32 j=0; j < nbContacts; j++)
    {
      //print all positions of contact.
      PxVec3 point = contacts[j].position;
      cout<<"Contact point
        ("<<point.x <<" "<< point.y<<" "<<point.x<<")\n";
    }
  }

}
```

Filter shader

The filter shader in PhysX is used for collision filtering, and to customize the collection of flags describing the actions to take on a collision pair, such as whether to report the collision of rigid bodies or not. PhysX 3 also provides a default implementation of the filter shader using the `PxDefaultSimulationFilterShader` class, which basically emulates the PhysX 2.8.x collision filtering. The PhysX filter shader basically customizes the collection of flags defined in the `PxPairFlag` structure. We can use it to customize the callback events of the PhysX simulation that we are interested in.

The code snippet of a user-defined filter shader is as follows:

```
PxFilterFlags contactReportFilterShader(PxFilterObjectAttributes
attributes0, PxFilterData filterData0,
PxFilterObjectAttributes attributes1, PxFilterData filterData1,

PxPairFlags& pairFlags, const void* constantBlock, PxU32
constantBlockSize)
{
  // all initial and persisting reports for
//everything, with per-point data

  pairFlags = PxPairFlag::eCONTACT_DEFAULT
        | PxPairFlag::eTRIGGER_DEFAULT
        | PxPairFlag::eNOTIFY_TOUCH_PERSISTS
        | PxPairFlag::eNOTIFY_CONTACT_POINTS;

return PxFilterFlag::eDEFAULT;

}
```

Once you have defined a custom filter shader, you need to assign it to the PhysX scene descriptor class, as follows:

```
sceneDesc.filterShader  = customFilterShader;
```

In this code, `sceneDesc` is the instance of the `PxSceneDesc` class and `customFilterShader` is the user-defined filter shader.
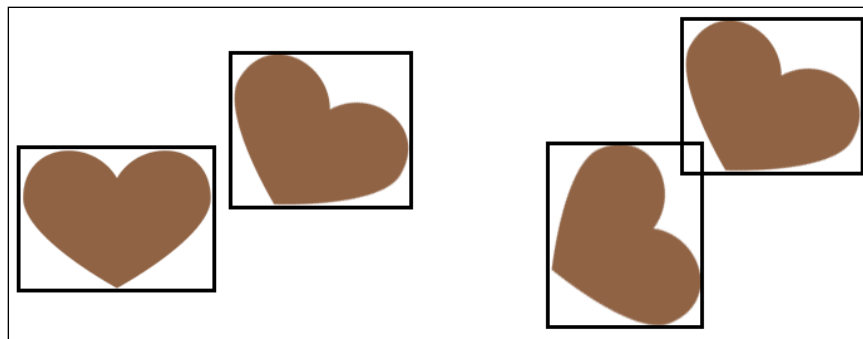
Broad-Phase collision detection

An **axis aligned bounding box** (**AABB**) for a PhysX object is the smallest box that can enclose that object provided the box edges are always parallel to the coordinate axes. The AABB of an object never rotates, although the object itself can rotate in any axis.
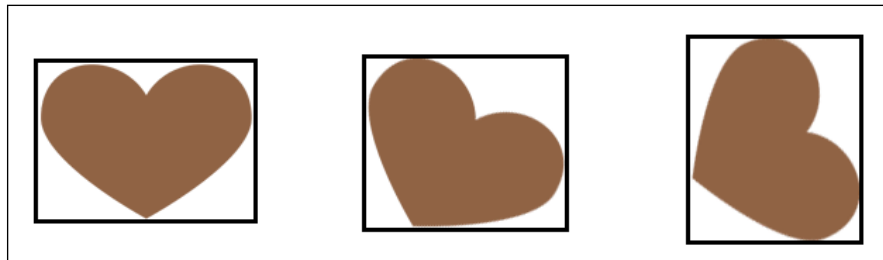
AABB (axis aligned bounding box) of a heart shaped object

If the AABB of two objects are not overlapping or colliding with each other, under no circumstances objects of AABB can collide with each other. On the other hand if AABB of two objects are overlapping/colliding, collision between actual objects can happen but it's not guaranteed. Checking collision using AABB is much cheaper than checking collision of actual objects because it's just a box, but the object itself may be made of a large numbers of polygons, which is expensive in terms of CPU processing.

AABB of two objects without overlapping and with overlapping

In broad phase collision detection, we can avoid the unnecessary calculation of *object to object* collision check for the objects that are found to be negative to the AABB collision test. Objects pairs that are found positive to the test are further tested for Narrow-Phase collision detection, which is more expensive but is essential for determining whether the objects are actually colliding with each other or not.

The Broad-Phase algorithms supported in PhysX 3.3.0 are:

- **Sweep-and-prune** (**SAP**)
- **Multi box pruning** (**MBP**)

Sweep-and-prune (SAP)

This is also known as the sort-and-sweep algorithm, and is a very popular Broad-Phase collision detection algorithm. Typically, its performance is very good when a large numbers of objects are in a sleeping state. Performance is degraded when the majority of the objects in the scene are either moving, or are being added to or removed from the scene. This algorithm doesn't require any world bound definition to work.

Multi box pruning (MBP)

The Multi-box-pruning Narrow-Phase collision detection algorithm is new to PhysX SDK and was added in PhysX version (3.3.0). It can perform better than the sweep-and-prune algorithm when the majority of objects in the scene are either moving or being removed from or added to the scene. However, its performance can be worse than SAP when the majority of the objects in the scene are in a sleeping state. This algorithm requires a world bound definition to work.

We can select the broad phase algorithm as per our requirements, by using `PxBroadPhaseType enum`, within the `PxSceneDesc` structure.

Narrow-Phase collision detection

Object pairs that are found to be positive in the Broad-Phase collision detection algorithm are further tested for Narrow-Phase collision detection. In this phase, the actual object pairs are tested for collision instead of their AABB, which can be expensive because the object's polygon count can vary from 3 to n vertices. Other information that is calculated in this phase are point of intersection, collision normal, and penetration depth. These properties are important for the collision response phase.

Continuous collision detection

There are two collision detection techniques that are typically used in the physics engine. The first one is **Discrete Collision Detection** (**DCD**), and the second one is **Continuous Collision Detection** (**CCD**). DCD gives better performance because it checks for collision between two time steps. Collision is only detected after some level of intersection of colliding objects. However, it is also prone to missing the collision of fast moving objects because of its non-continuous collision technique. On the other hand, CDD considers a continuous motion of moving objects and does not miss any collision during motion between two time steps. Because of its continuous nature, CCD is more performance hungry than DCD.

In PhysX, by default, DCD is enabled because it is more performance friendly than CCD. However, there may be situations where we need to enable CCD, in order to take advantage of its collision detection accuracy and reliability.

To enable CCD in PhysX, the following things are done:

1. Enable CCD in the scene descriptor, as follows:

```
PxPhysics* physx;
...
PxSceneDesc desc;
desc.flags |= PxSceneFlag::eENABLE_CCD;
```

2. Enable CCD in pair filter:

```
static PxFilterFlags testCCDFilterShader(
  PxFilterObjectAttributes attributes0,
  PxFilterData filterData0,
  PxFilterObjectAttributes attributes1,
  PxFilterData filterData1,
  PxPairFlags& pairFlags,
  const void* constantBlock,
  PxU32 constantBlockSize)
{
  pairFlags = PxPairFlag::eRESOLVE_CONTACTS;
  pairFlags |= PxPairFlag::eCCD_LINEAR;
  return PxFilterFlags();
}

desc.filterShader = testCCDFilterShader;
physx->createScene(desc);
```

3. Lastly, CCD needs to be enabled for the dynamic rigid body, as follows:

```
PxRigidBody* body;

body->setRigidBodyFlag(PxRigidBodyFlag::eENABLE_CCD, true);
```

CDD is only activated between objects whose relative speeds are greater than the sum of their respective CCD velocity thresholds. These velocity thresholds are automatically calculated based on the shape's properties, and supports non-uniform scales.

Summary

In this chapter we learnt about collision detection and available collision shapes in PhysX. We learnt about trigger shapes, simulation events, and the filter shader. We also went through Broad-Phase collision detection, Narrow-Phase collision detection, DCD, and CCD.

5
Joints

In this chapter, we will learn about the various joints available in PhysX and their properties and configurations. Topics that are covered in this chapter are as follows:

- Fixed joints
- Revolute joints
- Spherical joints
- Distance joints
- Prismatic joints
- D6 joints

Joints in PhysX

Joints in PhysX are constraints that set the properties of two interconnected actors and their interaction with each other. Joints can be used in many ways in physics simulation, for example, making a ragdoll using joints for a game character, connecting doors and windows with a hinge joint, and creating chains or ropes by connecting a series of joints. Joints are always created between two actors, out of which at least one must be movable. This means that it should be either of type `PxRigidDynamic` or `PxArticulationLink`, and the other actor of that joint may be any of those as well as `PxRigidStatic`. The `PxArticulationLink` class is used for creating articulated joints and currently is in the experimental stage. Articulated joints are specifically used for creating joints of actuated characters. The `PxRigidStatic` class is used for creating static actors having spatial volume, but they remain fixed in the PhysX scene. Static actors can be used as a fixed support in world space and can be further connected to a rigid dynamic actor by using joints.

In PhysX 3, there are six different types of joints available; they are as follows:

- **A fixed joint**: A fixed joint has fixed orientation and movement along the connected bodies. It doesn't allow any linear movement or rotation along any axis of the joint. The PxFixedJoint class is used for creating a fixed joint.

- **A revolute joint**: This is also called a hinge joint. It doesn't allow linear motion along it, but allows free rotation around one common axis. A swinging door connected with a wall is a good example of a hinge joint. The PxRevoluteJoint class is used for creating revolute joints.

- **A spherical joint**: This is also known as a ball-socket joint. It doesn't allow linear movement along the joint, but allows the orientation to vary freely. An adjustable mirror connected to a vehicle is a good example of a spherical joint. The PxSphericalJoint class is used for creating spherical joints.

- **A distance joint**: A distance joint always keeps the origins of connected bodies within a certain distance range. The PxDistanceJoint class is used for creating distance joints.

- **A prismatic joint**: This is also known as a slider joint. It always has fixed orientation, but allows linear movement along the common x axis. The PxPrismaticJoint class is used for creating prismatic joints.

- **A D6 joint**: A D6 joint is a highly flexible and highly configurable joint that provides all six degrees of freedom for customizing the joint. Many types of joints can be derived from a D6 joint as per your requirements and configuration. The PxD6Joint class is used for creating D6 joints.

Fixed joints

A fixed joint in PhysX connects two actors so that they can't move or change their orientation in relation to each other.
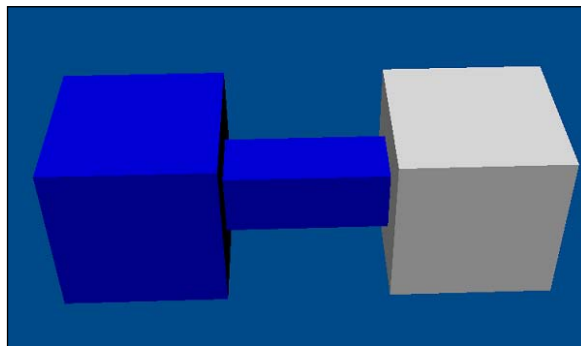
In an ideal condition, the connected bodies will maintain their spatial relationship; however, there may be circumstances that can disrupt the stability of a fixed joint, thus causing a drift. A fixed joint is useful where we want to show the breakage of two connected actors by some external force during simulation.

The `PxFixedJointCreate()` function is used for creating a fixed joint between two actors. It requires five parameters. The first parameter is of the `PxPhysics` type, and requires the instance of the created PhysX SDK. The second parameter is of type `PxRigidActor`, and requires the reference of the first actor to which the joint will be attached. The third parameter is of type `PxTransform`, and represents the position and orientation (transform) of a joint relative to the first actor. Similarly, the fourth and fifth parameters represent the reference of the second actor and the transform of the joint relative to the second actor. On success, the function returns the instance of the joint that was created. Same signature of function parameters are followed for creating other PhysX joints.

The code snippet for creating a fixed joint is given as follows:

```
PxVec3 pos = PxVec3(0,10,3); //position of static actor
PxVec3 offset = PxVec3(0,2,0); // offset of connected actor from joint

//creating actors
PxRigidActor* staticActor = PxCreateStatic(*gPhysicsSDK,
  PxTransform(pos), PxSphereGeometry(0.5f), *mMaterial);

PxRigidDynamic* connectedActor = PxCreateDynamic(*gPhysicsSDK,
  PxTransform(PxVec3(0)), PxBoxGeometry(0.5,0.5,2), *mMaterial,
  1.0f);

//creating fixed joint  between actors
PxFixedJoint* fixedJoint =
  PxFixedJointCreate (*gPhysicsSDK, staticActor,
  PxTransform(offset), connectedActor, PxTransform(-offset));

//adding actors to scene
gScene->addActor(*staticActor);
gScene->addActor(*connectedActor);
```
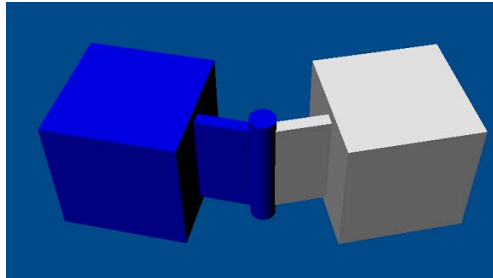
Here, `gPhysicsSDK` is the instance of the created PhysX SDK and `mMaterial` is an instance of the PhysX material.

Revolute joints

A revolute joint prevents all movements except rotational, which is common to the connected bodies.

The code snippet for creating a revolute joint is given as follows:

```
//creating actors
PxRigidActor* staticActor = ...
PxRigidDynamic* connectedActor = ...
PxVec3 offset = PxVec3(0,2,0); // offset of connected actor from joint

//creating Revolute joint between actors
PxRevoluteJoint* revoluteJoint =
PxRevoluteJointCreate (*gPhysicsSDK, staticActor,
  PxTransform(offset), connectedActor, PxTransform(-offset));
```

We can also set the upper and lower rotational limits for a revolute joint. The `setLimit()` function is used to set the limit of a revolute joint. This function requires a parameter of type `PxJointLimitPair`. The `PxJointLimitPair` parameter itself requires three parameters of type `PxReal`, where `lowerLimit` specifies the lower value of the limit, `upperLimit` specifies the upper value of the limit, and `limitContactDistance` specifies the distance from the upper or lower limit at which the limit constraint becomes active. The lower limit value must always be less than the upper limit value. We enable the defined limits of a revolute joint by calling `setRevoluteJointFlag()` and setting the `PxRevoluteJointFlag::eLIMIT_ENABLED` flag to `true`.

The code snippet for setting the upper and lower rotational limits for a revolute joint is given as follows:

```
revoluteJoint->setLimit(PxJointLimitPair(lowerLimit, upperLimit,
  limitContactDistance));

revoluteJoint ->setRevoluteJointFlag(PxRevoluteJointFlag::

eLIMIT_ENABLED, true);
```

Spherical joints

A spherical joint restrains linear movement along the connected bodies, but allows swinging and twisting of the joint.
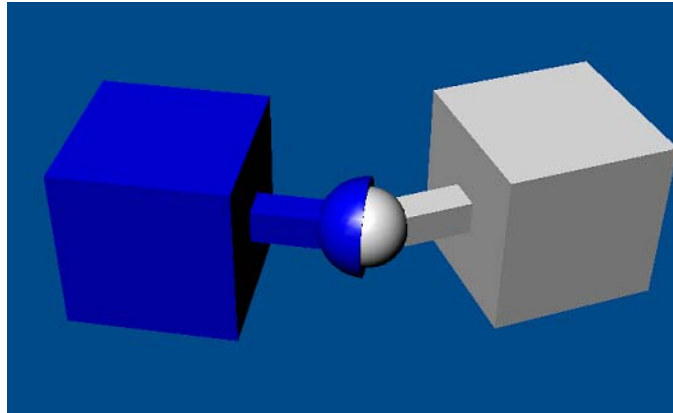
The code snippet for creating a spherical joint is given as follows:

```
//creating actors
PxRigidActor* actor1 = ...
PxRigidDynamic* actor2 = ...
PxVec3 offset = PxVec3(0,2,0); // offset of connected actor from joint

//creating Spherical joint between actors
PxSphericalJoint* sphericalJoint =
PxSphericalJointCreate (*gPhysicsSDK, actor1, PxTransform(offset),
  actor2,PxTransform(-offset));
```

The spherical joint supports a cone limit, which specifies how far the connected body can swing from a given axis, and a twist limit, which limits the twisting of the connected body around its own axis.

The code snippet for setting the limits of a spherical joint is given as follows:

```
sphericalJoint->setLimitCone(PxJointLimitCone(yLimitAngle,
  zLimitAngle, limitContactDistance));

sphericalJoint->
setSphericalJointFlag(PxSphericalJointFlag::eLIMIT_ENABLED, true);
```

The `SetLimitCone()` function requires a parameter of type `PxJointLimitCone`, whose The `PxJointLimitCone` constructor itself requires three parameters of type `PxReal`, where `yLimitAngle` represents the angle limit from the y axis of the constraint frame, `zLimitAngle` represents the angle limit from the z axis of the constraint frame, and `limitContactDistance` represents the distance from the upper or lower limit at which the limit constraint becomes active. We enable the defined limits of the spherical joint by calling `setSphericalJointFlag()` and setting the `PxSphericalJointFlag::eLIMIT_ENABLED` flag to `true`.

Distance joints

A distance joint keeps the origin of the connected actors within a certain range of distance.
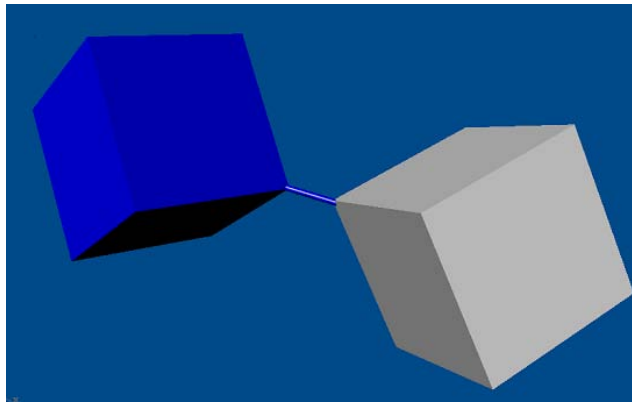
The code snippet for creating a distance joint is given as follows:

```
//creating actors
PxRigidActor* actor1 = ...
PxRigidDynamic* actor2 = ...
PxVec3 offset = PxVec3(0,2,0); // offset of connected actor from joint

//creating Spherical joint between actors
PxDistanceJoint* distanceJoint =
PxDistanceJointCreate (*gPhysicsSDK, actor1, PxTransform(offset),
  actor2,PxTransform(-offset));
```

We can also apply upper and lower bounds for distance joints as follows:

```
distanceJoint->setMinDistance(2.0f);
distanceJoint->setMaxDistance(10.0f);
distanceJoint->setDistanceJointFlag(eMAX_DISTANCE_ENABLED, true);
distanceJoint->setDistanceJointFlag(eMIN_DISTANCE_ENABLED, true);
```

The `setMinDistance()` and `setMaxDistance()` functions set the minimum and maximum allowed distance for the joint, respectively.

Prismatic joints
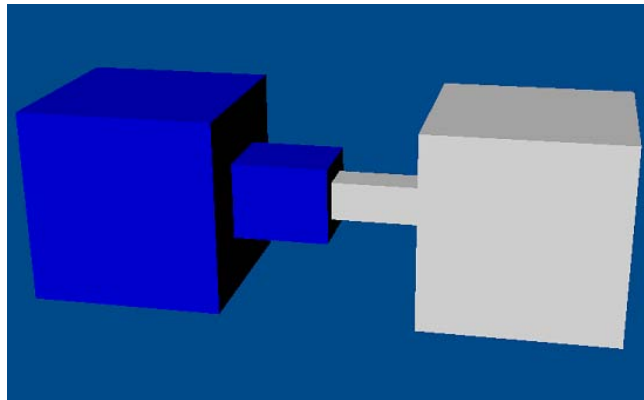
A prismatic joint always has a fixed relative orientation between two connected bodies, but allows linear movement along a common axis.

The code snippet for creating a prismatic joint is given as follows:

```
//creating actors
PxRigidActor* actor1 = ...
PxRigidDynamic* actor2 = ...
PxVec3 offset = PxVec3(0,2,0); // offset of connected actor from joint

//creating Prismatic joint between actors
PxPrismaticJoint* prismaticJoint =
PxPrismaticJointCreate (*gPhysicsSDK, actor1, PxTransform(offset),
  actor2,PxTransform(-offset));
```

Linear movement can be limited by setting upper and lower bounds on the distance between the origins of connected bodies, as follows:

```
prismaticJoint->setLimit(PxJointLimitPair(-10.0f, 20.0f, 0.01f);
prismaticJoint ->

setPrismaticJointFlag(PxPrismaticJointFlag::eLIMIT_ENABLED, true);
```

The `setLimit()` function is used to set the limit of a prismatic joint and requires a parameter of type `PxJointLimitPair`,which itself requires three parameters of type `PxReal`, where `lowerLimit` specifies the lower value of the limit, `upperLimit` specifies the upper value of the limit and `limitContactDistance` specifies the distance from the upper or lower limit at which the limit constraint becomes active. The lower limit value must always be less than the upper limit value. We enable the defined limits of a prismatic joint by calling `setPrismaticJointFlag()` and setting the `PxPrismaticJointFlag::eLIMIT_ENABLED` flag to `true`.

D6 joints

As we have already mentioned, the D6 joint is a highly flexible and highly configurable joint, which provides all six degrees of freedom to customize; that is, translational freedom in all x, y, and z axes as well as rotational freedom in all three axes. Many types of joints can be made by configuring the D6 joint.

The code snippet for creating a D6 joint is given as follows:

```
//creating actors
PxRigidActor* actor1 = ...
PxRigidDynamic* actor2 = ...
PxVec3 offset = PxVec3(0,2,0); // offset of connected actor from joint

//creating D6 joint between actors
PxD6Joint* d6Joint =
PxD6JointCreate (*gPhysicsSDK, actor1, PxTransform(offset),
  actor2,PxTransform(-offset));
```

By default, all degrees of freedom are locked in the D6 joint and it behaves similar to a fixed joint. The `setMotion()` function is required for setting the degrees of freedom of the D6 joint, and requires two enumerable types, `PxD6Axis` and `PxD6Motion`. The `PxD6Axis` parameter sets the axis around which a motion is specified, and the possible values are as follows:

- `PxD6Axis::eX`: This is motion along the x axis
- `PxD6Axis::eY`: This is motion along the y axis
- `PxD6Axis::eZ`: This is motion along the z axis
- `PxD6Axis::eTWIST`: This is motion around the x axis
- `PxD6Axis::eSWING1`: This is motion around the y axis
- `PxD6Axis::eSWING2`: This is motion around the z axis

`PxD6Motion` defines the motion type around a specified axis. The possible values are as follows:

- `PxD6Motion::eLOCKED`: The DOF is locked; it does not allow relative motion
- `PxD6Motion::eLIMITED`: The DOF is limited; it only allows motion within a specific range
- `PxD6Motion::eFREE`: The DOF is free and has its full range of motion

A number of joints can be created by configuring the D6 joint; for example, the revolute joint, spherical joint, prismatic joint, and others that are not available in PhysX 3, such as the cylindrical joint and pint-to-plane joint.

Summary

In this chapter, we learned about all of the types of joints that are available in PhysX SDK and their configurations. We learned about six types of joints available in PhysX, which are the fixed joint, revolute joint, spherical joint, distance joint, prismatic joint, and D6 joint. We also went through their specific behavior and properties.

6
Scene Queries

In this chapter we will learn about the various types of scene queries and their mode of operations.

Topics that are covered in this chapter are as follows:

- Raycast queries and its mode of operation
- Sweep queries and its mode of operation
- Overlap queries and its mode of operation

Raycast queries

The raycast query is a way to detect collisions by using rays, just like how we use the laser pointing device during any slide show presentation, which creates a red dot on the first object its ray collides with. Raycast queries have many uses, such as detecting bullet collision point in an FPS game, checking line of sight for the enemy characters, and selecting in-game objects by using a mouse.

In PhysX, raycasting is done by calling the `PxScene::raycast()` function and it can be used in different modes depending on the arguments you are passing to the function; they are described as follows:

- `raycast()` with `PxRaycastBuffer buf` and `PxQueryFlag::eANY_HIT`
- `raycast()` with `PxRaycastBuffer buf` and a default constructor
- `raycast()` with `PxRaycastBuffer buf(PxRaycastBuffer, PxU32)`

When using the `raycast()` function with `PxQueryFlag::eANY_HIT`, it just returns `true` on colliding with any PhysX object. It is the least expensive to call, because it doesn't perform any calculation related to a hit shape and an impact point. It requires six parameters. The first two parameters are of the `PxVec3` type, and specify the origin and direction of the ray. The third parameter is of the `PxReal` type and it represents the maximum allowed distance of the ray. The fourth parameter is of the `PxRaycastBuffer` type and contains the result of the hit.
The fifth parameter is of the `PxHitFlag` type and it determines which optional fields are needed to be filled in the `PxQueryHit` structure. For default configuration, we can use `PxHitFlags(PxHitFlag::eDEFAULT)` and the last parameter is of the `PxRaycastBuffer` type, and it must be set to `PxQueryFlag::eANY_HIT` to use this mode.

A typical example for `raycast()` with `PxRaycastBuffer buf` and `PxQueryFlag::eANY_HIT` is as follows:

```
PxVec3 origin = PxVec3(0,3,0);    //[in] Ray origin
PxVec3 unitDir = PxVec3(0,1,0);   //[in] Normalized ray direction
PxReal maxDistance = 1000.0f;     //[in] Raycast max distance
PxRaycastBuffer hit;              //[out] Raycast results

PxQueryFilterData fd; fd.flags |= PxQueryFlag::eANY_HIT;

bool status = gScene->raycast(origin, unitDir, maxDistance, hit,
  PxHitFlags(PxHitFlag::eDEFAULT),fd);
```

The second way of using `raycast()` is with `PxRaycastBuffer buf` and a default constructor. This not only returns if the ray hits any shape, but it also gives the exact information about the first collided shape and its hit information. For this method, only the first four parameters are needed, which specify the origin, direction, ray max distance, and raycast-buffer, which contains the raycast hit information. The function returns `true` if the ray collides with a PhysX actor having a shape, otherwise it returns `false`.

A typical example is as follows:

```
PxVec3 origin = PxVec3(0,3,0);    //[in] Ray origin
PxVec3 unitDir = PxVec3(0,1,0);   //[in] Normalized ray direction
PxReal maxDistance = 1000.0f;     //[in] Raycast max distance
PxRaycastBuffer hit;              //[out] Raycast results

bool status = gScene->raycast(origin, unitDir, maxDistance, hit);
```

In the third mode, we can use `raycast()` so that it will cast a ray, which can penetrate all shapes that come in its way, and carries information about all the penetrated shapes and its collision points in a buffer. For this, the first three parameters are the same, which specify ray origin, ray direction, and ray max distance respectively. The fourth and the last parameter will be replaced with `PxRaycastBuffer buf(hitBuffer, bufferSize)`, which will keep all the objects touched by the ray along with the hit information.

The function returns a boolean value that indicates whether the `hitBuffer` parameter contains a blocking hit or not.

A typical example is as follows:

```
const PxU32 bufferSize = 256;         // [in] size of 'hitBuffer'
PxRaycastHit hitBuffer[bufferSize];  // [out] User provided buffer for
results
PxRaycastBuffer buf(hitBuffer, bufferSize); // [out] Blocking and
touching hits will be stored here

// Raycast against all static & dynamic objects (no filtering)
// The main result from this call are all hits along the ray, //stored
in 'hitBuffer'
bool hadBlockingHit =
gScene->raycast(origin, unitDir, maxDistance, buf);
```

Sweep queries

Sweep queries are just like the raycast queries, but the only difference is that sweep queries cast a shape or you can say sweep a shape instead of a ray. As with raycast, in PhysX the `PxScene::sweep()` function is used to perform sweep queries and can be used in many different modes; they are as follows:

- `sweep()` with `PxSweepBuffer buf` and `PxQueryFlag::eANY_HIT`
- `sweep()` with `PxSweepBuffer buf` and a default constructor
- `sweep()` with `PxSweepBuffer buf(PxSweepBuffer, PxU32)`

The geometry required by the sweep function can be a box, sphere, capsule, or convex.

The `sweep` function sweeps all of the specified geometry objects through space, and finds all of the rigid actors that get hit during the sweep. Each successful collision between a sweep shape and the scene actors generates hit information specified by the `PxSweepBuffer` field.

The `sweep()` function with `PxSweepBuffer buf` and `PxQueryFlag::eANY_HIT` is the cheapest to call among all variants of the `sweep()` function, because it doesn't perform any calculation to find the hit shape and impact point. It just returns `true` on successful collision of the sweep shape with other bodies in the PhysX scene. The function requires seven parameters. The first parameter is of the `PxGeometry` type, and represents the geometry, which will be used for sweeping. The next three consecutive parameters specify the origin, direction, and max distance of the sweep geometry from the origin respectively. The fifth parameter is of `PxSweepBuffer` type, and it stores the result of hit. The sixth parameter is of the `PxHitFlag` type, and it determines which optional fields are needed to be filled in the `PxQueryHit` structure; for default configuration we can use `PxHitFlags(PxHitFlag::eDEFAULT)`. And the last parameter is of the `PxQueryFilterData` type, and must be set to `PxQueryFlag::eANY_HIT` to use this mode.

A typical example for `sweep()` with `PxSweepBuffer buf` and `PxQueryFlag::eANY_HIT` is as follows:

```
//[in] Sweep geometry
PxGeometry sphereGeometry =  PxSphereGeometry(0.5f);
//[in] Geomery transform
PxTransform initialPos   =  PxTransform(PxVec3(0,5,0));
PxVec3 unitDir = PxVec3(0,-1,0); //[in] Normalized sweep direction
PxReal maxDistance = 1000.0f;    //[in] Sweep max distance
PxSweepBuffer sweepBuff;            //[out] Sweep result

PxQueryFilterData fd;
fd.flags |= PxQueryFlag::eANY_HIT;

bool status = gScene->sweep(sphereGeometry,initialPos,unitDir
  ,maxDistance, sweepBuff,PxHitFlags(PxHitFlag::eDEFAULT),fd);
```

Using `sweep ()` with `PxSweepBuffer buf` and a default constructor, not only detects the collision with the sweep geometry, but it also carries the hit information about the first object it collided with. The function requires five parameters similar to the `sweep()` function mentioned previously:

A typical example for `sweep()` with `PxSweepBuffer buf` and a default constructor is as follows:

```
//[in] Sweep geometry
PxGeometry sphereGeometry =  PxSphereGeometry(0.5f);
//[in] geomery transform
PxTransform initialPos   =   PxTransform(PxVec3(0,5,0));
PxVec3 unitDir = PxVec3(0,-1,0); //[in] Normalized sweep direction
PxReal maxDistance = 1000.0f;     //[in] Sweep max distance
PxSweepBuffer sweepBuff;                 //[out] Sweep result

bool status = gScene->sweep(sphereGeometry,initialPos,
  unitDir,maxDistance,sweepBuff);
```

In the third mode of the `sweep()` function, with `PxSweepBuffer buf(PxSweepBuffer, PxU32)`, it can be used in such a way that it will penetrate all the PhysX actors that come in its way and carry information about all the actors it collided with until the sweep geometry reaches the max allowed distance.

A typical example for the `sweep()` function with `PxSweepBuffer buf(PxSweepBuffer, PxU32)` is as follows:

```
PxGeometry sphereGeometry =  PxSphereGeometry(0.5f); //[in] Sweep
geometry
PxTransform initialPos =  PxTransform(PxVec3(0,5,0)); //[in] geomery
transform
PxVec3 unitDir = PxVec3(0,-1,0); //[in] Normalized sweep direction
PxReal maxDistance = 1000.0f;     //[in] Sweep max distance
PxSweepBuffer buff;      //[out] Sweep result

const PxU32 bufferSize = 256;          // [in] size of 'sweepBuffer'
PxSweepHit sweepBuffer[bufferSize];//[out] User provided buffer for
results
PxSweepBuffer buf(sweepBuffer, bufferSize); //[out] Hits stored here

bool status = gScene->sweep(sphereGeometry,
  initialPos,unitDir,maxDistance,buf);
```

Currently supported input shapes are boxes, spheres, capsules, and convex for all types of sweep queries.

Overlap queries

In an overlap query, we determine whether geometry has collided or overlapped against any other body in a PhysX scene; if an overlap is reported, we can also find the information about the overlapping object. The methods are similar to `raycast()` and `sweep()` except that it doesn't support `PxHitFlags`, because there's no specific first point from where the overlap occurs. The supported shapes are box, sphere, capsule, and convex.

In PhysX, overlap queries are done by calling the `PxScene::overlap()` function, and it can be used in different modes depending on the arguments you are passing to the function; they are described as follows:

- `overlap()` with `PxOverlapBuffer buf` and `PxQueryFlag::eANY_HIT`
- `overlap()` with `PxOverlapBuffer buf(PxOverlapBuffer, PxU32)`

The `overlap()` function with `PxOverlapBuffer buf` and `PxQueryFlag::eANY_HIT` just returns `true` if the shape volume is overlapped by any other PhysX actor. The function has four parameters. The first one is of `PxGeometry` type, and it represents the geometry of an object to check for an overlap. The second parameter is the transformation of an object, and the third one is of `PxOverlapBuffer` type and it contains the result of the overlap. The last parameter is of `PxQueryFilterData` type, and it must be set to `PxQueryFlag::eANY_HIT` to use this mode.

A typical example for `overlap()` with `PxOverlapBuffer buf` and `PxQueryFlag::eANY_HIT` is as follows:

```
//[in] Sweep geometry
PxGeometry sphereGeometry =  PxSphereGeometry(0.5f);
//[in] geomery transform

PxTransform initialPos   =  PxTransform(PxVec3(0,5,0));
PxOverlapBuffer buf; //[out] Buffer for overlap results

PxQueryFilterData fd;
fd.flags |= PxQueryFlag::eANY_HIT;

bool status = gScene->overlap(sphereGeometry,initialPos,buf,fd);
```

The `overlap()` function with `PxOverlapBuffer buf(PxOverlapBuffer, PxU32)` returns the number of objects it overlapped with its shape and has three parameters. The first two parameters define the geometry and transformation of the object to check for overlap, and the third one is of `PxOverlapBuffer` type, and it contains information about all of the overlapped objects.

A typical example for the `overlap()` function with `PxOverlapBuffer buf(PxOverlapBuffer, PxU32)` is as follows:

```
PxGeometry sphereGeometry =  PxSphereGeometry(0.5f);
PxTransform initialPos =  PxTransform(PxVec3(0,5,0));

const PxU32 bufferSize = 256
PxOverlapHit overlapBuffer[bufferSize];
PxOverlapBuffer buf(overlapBuffer, bufferSize);

bool status = gScene->overlap(sphereGeometry,initialPos,buf);
```

Summary

In this chapter we learned about raycast queries, sweep queries, and overlap queries. We saw that these queries can perform different modes of operations depending on the number and types of arguments fed to their functions. We also learned that each mode has a different performance cost, and we can customize the queries as per our requirements.

7
Character Controller

In this chapter, we will learn the concept of a character controller and how to use it in PhysX.

The topics that will be covered in this chapter are as follows:

- Basics and need of a character controller
- Creating and moving a character controller
- Updating the position, shape, and size of a character controller
- Auto-stepping and slope limit

Character controller basics

A character controller is a special kinematic actor with a collider shape, which we use in games for creating the collider of a game's characters. It has dedicated properties and methods, with fine grained control over its movement and interaction with the surrounding environment.

A character controller can be used for simulating the movement of any character or AI in games or simulation applications. It can be used to create playable characters in FPS (First Person Shooter) or TPS (Third Person Shooter) games, and even for making NPC in a strategic game.

The need of a character controller

In computer games, a character controller is intended to represent the collision boundary of a player character or enemy AI. Although this can be done using a kinematic rigid body, sooner or later you may get into many problems that can be avoided by using a character controller.

Typical problems that you may face when using physics engine default rigid body instead of a character controller are as follows:

- Typically, physics engine bodies use the DCD (Discrete Collision Detection) algorithm for detecting the collision between objects, which may sometimes fail to detect the collision of fast moving objects, thus causing a character to pass through a wall.
- A rigid dynamic body can be moved by using force or impulse, but its final position can't be controlled precisely. Thus, using it as a replacement of character controller is not a good idea.
- A rigid dynamic body will slide down a sloping surface, but any body that represents a character should be able to stand on an uneven and sloping surface.
- Typical bodies in physics engine have restitution properties, which make the bodies bounce off when it hits a surface; this is not a desirable property for character simulation.

Creating a character controller

Before creating a character controller, you need to create a controller manager, which will manage all character controllers and their interaction with each other in the PhysX scene.

The code snippet for creating a controller manager is as follows:

```
PxScene* gScene; //Instance of PhysX scene
PxControllerManager* manager = PxCreateControllerManager(*gScene);
```

After this, we can create a character controller for each character in our game, which is given as follows:

```
PxCapsuleControllerDesc  desc;

//initial position
desc.position  = PxExtendedVec3(0.0f,0.0f,0.0f);
//controller skin within which contacts generated
desc.contactOffset = 0.05f;
//max obstacle height the character can climb
desc.stepOffset  = 0.01;
```

```
desc.slopeLimit  = 0.5f; // max slope the character can walk
desc.radius    = 0.5f; //radius of the capsule
desc.height    = 2; //height of the controller
desc.upDirection = PxVec3(0, 1, 0); // Specifies the 'up' direction
desc.material  = NULL;

PxController* c = manager->createController(desc);
```

When we create a character controller, its bounding volume should always be on the top of the surface or static mesh on which it will move. Overlapping of the character controller shape with the surface on which it will move can result in an undefined behavior such as falling of the character controller by penetrating through the surface.

Moving a character controller

Since a character controller is a kinematic body (kinematic bodies are explained in *Chapter 3*, *Rigid Body Dynamics*), its position can only be updated (moved) through the explicit move() call and not by an external force exerted through other dynamic actors in the PhysX scene. Even the effect of gravity will be created programmatically, because PhysX gravity can't influence kinematic bodies.

To move a character controller by updating its position in each frame, the following function is called:

```
collisionFlags =
PxController::move(disp, minDist, elapsedTime, filters, obstacles);
```

The PxController::move() function requires five parameters:

- The first parameter is of the PxVec3 type, and it represents the magnitude of displacement in x, y, and z axis for the current frame. Its *y* component is updated to simulate the effect of gravity, as well as jumping of the character controller, and it is done programmatically. The *x* and *z* components are updated for lateral movement of the character controller.

- The second parameter is of the Px32 type, and it represents the minimal length used to stop the recursive displacement algorithm early when the remaining distance to travel goes below this limit.

- The third parameter is of the PxF32 type, and it represents the elapsed time since the last call to the move function.

- The fourth parameter is of the PxControllerFilters type, and it can be used to filter out actors the character controller is colliding with.

- The last parameter is of the `PxObstacleContext` type, and is optional; these are the additional obstacles that the character controller should collide with.

The `PxController::move()` function returns `PxControllerFlag`, which can be used to detect whether the character controller is touching the ground, side walls, or something else.

Useful methods and properties

A character controller has many properties and methods that are very useful for precisely controlling the movement and interaction with surrounding environments. Some essential properties are discussed next.

Position update

It is important to know the current position of a character controller so that we can sync the visuals of our game character with it. We can find the position of a character controller as follows:

```
const PxExtendedVec3& PxController::getPosition() const;
```

The `PxController::getPosition()` function returns the current center position of the character controller shape. It should be noted that a character controller never rotates, therefore, we don't have any function to get the current rotation of the character controller.

We can also get or set the bottom position, that is, the foot position of a character controller by using the following function:

```
//get foot position of character controller
const PxExtendedVec3& PxController::getFootPosition() const;

//set foot position of character controller
bool PxController::setFootPosition(const PxExtendedVec3& position);
```

Shapes of a character controller

PhysX currently supports two types of shapes for defining the collision boundary of a character controller, which is explained as follows:

- AABB (Axis Aligned Bounding Box) is a box shape defined by a position and an extents vector. The AABB doesn't rotate even if the player graphic is rotating. This avoids getting stuck in places too tight for the AABB to rotate.

- A capsule shape defined by its height and radius represents the collision volume of a character in a better way than AABB, but it is slightly more expensive in terms of CPU use.

To make the character controller collision boundary tolerable against other colliding shapes, which may cause some numerical issues, a skin volume is maintained around the character controller volume. The skin width can be set through the `PxControllerDesc::contactOffset` function and can be read through the `PxController::getContactOffset()` function. When visualizing the character controller using **Debug Renderer**, its skin width should also be taken into consideration for determining the rendered volume of the controller.

Size update

The size of a character controller's shape can be changed dynamically at runtime to simulate the crouch behavior of a game's character. For example, if your game's character height is 1.5 meters, you may want to reduce it to 1 meter on the crouch mode, which will avoid it from colliding with low-level objects placed in a game.

The functions that can be used for changing the size of a character controller are as follows:

- For the AABB shape, you can use the following function:

```
bool PxBoxController::setHalfSideExtent(PxF32 halfSideExtent)
  = 0;
bool PxBoxController::setHalfForwardExtent(PxF32halfForwardExtent)
  = 0;
```

- For the capsule shape, you can use the following function:

```
bool PxCapsuleController::setRadius(PxF32 radius) = 0;
bool PxCapsuleController::setHeight(PxF32 height) = 0;
```

When we decrease the height of a character controller at run-time, the character controller may levitate for some time above the ground until it touches the ground again. This happens because its height is reduced from the center, which affects both the upper-half and lower-half extent of the character controller. Therefore, it's necessary to reposition it appropriately so that it touches the ground after resizing its height. This can be done automatically by using the following function:

```
void PxController::resize(PxF32 height) = 0;
```

Auto-stepping

In PhysX, auto-stepping is a feature for a character controller, that allows the character controller to sweep across an uneven surface without the intervention of a user. Just like in the real world when a person walks he/she crosses small obstacles without even thinking about them. This allows a character controller to have some tolerance against the uniform surface or terrain when it moves over the terrain or a static triangle mesh. This tolerance can be set by defining the `PxControllerDesc::stepOffset()` function. Implementing the auto-stepping feature requires the SDK to know about your `up` vector, which can be defined in the `PxController::upDirection()` function.

Slope limit

When a character controller moves on a surface that is inclined by some angle, or moves on an uneven terrain consisting of slopes and narrow surface, the movement of the character controller can be limited by setting the `PxControllerDesc::SlopeLimit()` function, which requires the maximum allowed slope limit for a character controller in radians. It is expressed as the cosine of desired limit angle.

The following code snippet allows a character controller to walk on a surface inclined not more than 30 degrees.

```
slopeLimit = cosf(PxMath::degToRad(30.0f));
```

Setting the slope limit to zero will disable any slope-related constraint on the character controller. It should be noted that the slope limit is ignored if the touched shape is attached to a dynamic or kinematic rigid body, or if the touched shape is a sphere or capsule attached to a static body. Getting a contact normal from heightfields, triangle meshes, convex meshes, and boxes is more readily supported than with spheres and capsules, so these shape types are all involved in the slope limit calculations, provided they are attached to a static body.

Summary

In this chapter we learned the concept of the character controller and how we use it in PhysX. We learned how to create, update, define a shape for a character controller, and other related properties such as auto-stepping and slope limit.

8
Particles

In this chapter, we will learn about particle simulation in PhysX. The topics that will be covered in this chapter are as follows:

- Types of particle system and their creation
- Particle system properties
- Creating, updating, and releasing a particle
- Particle drains and collision filtering

Exploring particles

Particles are widely used in games and other 3D applications requiring particle effects. They are used for simulating particle-based effects such as explosions, smoke, fluid, debris, dust, snow, and many others. They greatly increase the originality of a simulating world by creating photo-realistic visuals.

Creating a particle system

In PhysX, before creating actual particles, we need to create a particle system. A particle system class manages a set of particles and the phases of their life cycle such as creating, updating, and releasing. The created particles can interact with other objects in the PhysX scene, whether the objects are static or dynamic. Particles can also be influenced by gravity and force.

PhysX mainly supports two types of particle system, which are as follows:

- Particles without intercollision
- Particles with intercollision (Smoothed Particle Hydrodynamics)

Particles without intercollision

The main characteristic of these particles is that they may collide with the surrounding environment or with other PhysX actors (static or dynamic), but they can't collide with a particle of their own type. These particles are suitable for creating particle effects such as debris, sparks, dust, snow, flurry, and so on.

To create a particle system that doesn't support intercollision, the `PxPhysics::creat eParticleSystem()` function is called. Its simplest form requires a single parameter of the `PxReal` type, which specifies maximum number of particles that can be placed in the created particle system. On success, it returns the reference of the newly created particle system of the `PxParticleSystem` type, otherwise, null is returned.

The code snippet for creating a particle system of the `PxParticleSystem` type is as follows:

```
// set immutable properties.
PxU32 maxParticles = 100;

// create particle system in PhysX SDKs
PxParticleSystem* ps =
mPhysics->createParticleSystem(maxParticles);

// add particle system to scene, in case creation was successful
if(ps)
mScene->addActor(*ps);
```

Particles with intercollision

The particles with intercollision support may collide with PhysX actors (static or dynamic) as well as with particles of their own type. Technically, this is is SPH (Smoothed Particle Hydrodynamics) and it can precisely simulate the behavior of dynamic (moving) fluids. More information about SPH is available at `http://en.wikipedia.org/wiki/Smoothed_particle_hydrodynamics`. These particles are suitable for creating fluid-based effects such as fluid flowing, oil spilling, water fountains, volumetric smoke, and gases.

To create a particle system that supports intercollision, the `PxPhysics:: createFluid()` function is called. Its simplest form requires a single parameter of the `PxReal` type, which specifies the maximum number of particles that may be placed in the created particle system. On success, it returns the reference of the newly created particle system of the `PxParticleFluid` type, otherwise, `null` is returned.

The code snippet for creating a particle system of the `PxParticleFluid` type is as follows:

```
// set immutable properties.
PxU32 maxParticles = 100;

// create particle system in PhysX SDK
PxParticleFluid* pf = mPhysics->createParticleFluid(maxParticles);

// add particle fluid to scene, in case creation was successful
if (pf)
mScene->addActor(*ps);
```

Both particle classes, the `PxParticleSystem` class and the `PxParticleFluid` class are derived from the `PxParticleBase` abstract base class, which contains abstract methods common to both particle systems.

Particle system properties

A particle system in PhysX contains lots of properties that are used to define the simulation behavior of a particle and its interaction with the surrounding objects. These properties can be classified on the basis of their mutability (changeability) when they are defined for a particle system.

The classification of properties on the basis of mutability is as follows:

- The properties that are immutable (can't be changed) after particle creation are explained as follows:
    - `maxParticles`: It specifies the maximum number of particles allowed for a particle system
    - `particleBaseFlags, PxParticleBaseFlag::ePER_PARTICLE_REST_OFFSET`: It is a flag for enabling/disabling per-particle rest offset

- The properties that are immutable when the particle system is a part of the PhysX scene are explained as follows:
    - `maxMotionDistance`: This property specifies the maximum distance a particle can move in a simulation time step. Increasing this value too much may affect the performance.
    - `gridSize`: This property sets a grid size to subdivide particles into special groups for optimizing performance and parallelization.

- ° `restOffset`: This property sets the minimum rest offset distance between particles and the surface of rigid actors that is maintained by the collision system.

- ° `contactOffset`: This property sets the minimum distance at which contacts between particles and rigid actors are created. It is internally used to avoid jitter and penetration. It should always be greater than `restOffset`.

- ° `particleReadDataFlags`: This property allows the application/user to read some particle-related property after simulation.

- ° `particleBaseFlags, PxParticleBaseFlag::eGPU`: It is a flag for enabling/disabling GPU acceleration.

- ° `particleBaseFlags, PxParticleBaseFlag::eCOLLISION_TWOWAY`: It is a flag for enabling/disabling two-way interaction between rigid bodies and particles.

- ° `restParticleDistance`: This property sets the resolution of the fluid particle (only `PxParticleFluid`)

- The properties that are mutable (that can be changed at any time) are explained as follows:

  - ° `restitution`: This property specifies the restitution of particle collision

  - ° `dynamicFriction`: It specifies the dynamic friction of particle collision

  - ° `staticFriction`: This property specifies the static friction of particle collision

  - ° `damping`: This property specifies the velocity damping applied to particles

  - ° `externalAcceleration`: This property specifies the acceleration applied to particles at each time-step of PhysX simulation

  - ° `particleBaseFlags, PxParticleBaseFlag::eENABLED`: It is a flag for enabling/disabling particle simulation

  - ° `particleBaseFlags, PxParticleBaseFlag::ePROJECT_TO_PLANE`: It is a flag for enabling/disabling projection mode, which confines the particles to a plane

  - ° `projectionPlaneNormal, projectionPlaneDistance`: This property defines a plane for the projection mode

  - ° `particleMass`: This property sets the mass of particles

  - ° `simulationFilterData`: This property sets filter data used to filter collisions between particles and rigid bodies

- Mutable properties only applicable to `PxParticleFluid` are explained as follows:

  ° `stiffness`: This property denotes the compressibility of fluid particles. Reducing this value will increase the compressibility of the particle that means, it will behave like a sponge, while increasing this value increases the rigidness of the particle. Setting this property to a high value may cause simulation instability. Its optimal value varies from 1 to 200.

  ° `viscosity`: This is the measure of a fluid particle's resistance to gradual deformation by any external force. Increasing the value of this property will increase the thickness of a liquid, for example, honey, while reducing its value, will decrease the thickness of a liquid, for example, water. Its optimal value varies from 5 to 300.

Creating particles

Once we are done with creating a particle system, it's time to create particles. If the particle system is of the `PxParticleSystem` type, we call the `PxParticleSystem::createParticle()` function to create a generic particle. On the other hand, if the particle system is of the `PxParticleFluid` type, we call the `PxParticleFluid::createParticle()` function to create an SPH particle. Both the functions require at least one parameter of the `PxParticleCreationData` type, which is basically a descriptor-like user-side class describing buffers for particle creation. It contains a particle-related description such as `numParticles`, `indexBuffer`, `positionBuffer`, `velocityBuffer`, `restOffsetBuffer`, and `flagBuffer`. Specifying the particle indices and the position is mandatory, although other information can be skipped. Once the particles are created, they can be accessed through constant array indices until they are not destroyed.
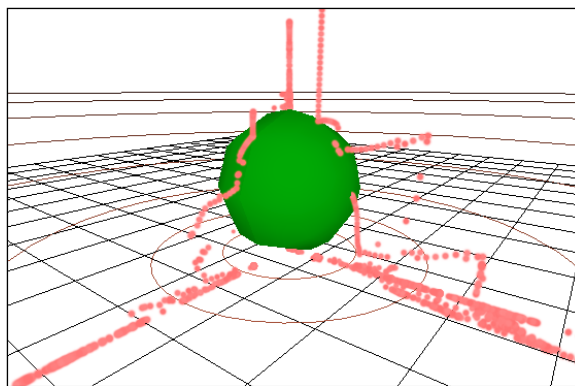
Fluid particles falling on to sphere actor

A typical example of creating a few particles is as follows:

```
//declare particle descriptor for creating new particles
PxParticleCreationData particleCreationData;
particleCreationData.numParticles = 3;

PxU32 myIndexBuffer[] = {0, 1, 2};
PxVec3 myPositionBuffer[] =  { PxVec3(0,0.2,0), PxVec3(0.5,1,0),
PxVec3(0,2,.7) };
PxVec3 myVelocityBuffer[] = { PxVec3(0.1,0,0), PxVec3(0,0.1,0),
PxVec3(0,0,0.1) };

particleCreationData.indexBuffer =
PxStrideIterator<const PxU32>(myIndexBuffer);

particleCreationData.positionBuffer =
PxStrideIterator<const PxVec3>(myPositionBuffer);

particleCreationData.velocityBuffer =
PxStrideIterator<const PxVec3>(myVelocityBuffer);

// create particles in *PxParticleSystem* ps
bool success = ps->createParticles(particleCreationData);
```

It should be noted that access to particles such as creating, updating, releasing, and reading particle-related properties can only be done when the PhysX scene is not being simulated. The indices of the particles should not exceed `PxParticleBase::getMaxParticles()`. When creating particles of the `PxParticleFluid` type, the spawning distance between the particles should be close to `PxParticleFluid::getRestParticleDistance()`, otherwise, it may spread instantly in all directions.

Updating particles

Each particle can be explicitly accessed for updating its position and velocity immediately.

The code snippet for updating the positions of particles is given as follows:

```
PxVec3 particlePositions[] =  {...};
PxU32 particleIndices[] = {...};

PxU32 numParticles = ...;
PxStrideIterator<const PxVec3> newPositionBuff (particlePositions);
PxStrideIterator<const PxU32> indexBuff (particleIndices);
```

```
//ps is the instance of PxParticleSystem or PxParticleFluid
ps->setPositions(numParticles, indexBuff, newPositionBuff);
```

We can also update the forces on the particles by using `addForces()`, as follows:

```
PxU32 numParticles = ...;
PxStrideIterator<const PxVec3> forceBuffer = ...;
PxStrideIterator<const PxU32> indexBuffer = ...;

//ps is the instance of PxParticleSystem or PxParticleFluid
ps->addForces(numParticles, indexBuffer,forceBuffer,
  PxForceMode::eFORCE);
```

The particle rest-offset can be updated if the `PxParticleBaseFlag::ePER_PARTICLE_REST_OFFSET` function is set to `true` on creating the particle system.

The code snippet for setting `PxParticleBaseFlag` is as follows:

```
//ps is the instance of PxParticleSystem or PxParticleFluid
ps->setParticleBaseFlag(PxParticleBaseFlag::
ePER_PARTICLE_REST_OFFSET, true);
```

Releasing particles

Particles can be released by using the `PxParticleBase::releaseParticles()` function, which require two parameters. The first parameter is of the `Px32` type and holds the count of particles that have to be released. The second parameter is of the `PxStrideIterator<PxU32>` type and describes the indices of particles that should be deleted. The indices always have to be consistent with the particle count. We can also use the same function without any parameter, which will release all particles in a PhysX scene.

```
//ps is the instance of PxParticleSystem or PxParticleFluid
ps->releaseParticles(numParticles,indexBuffer);
```

Particle drains

Particle drains are the PhysX objects or shapes, which act as a particle cleaner and whenever PhysX particles come in touch with that object, particles are released from the scene. To mark a shape as a particle drain, simply call the `PxShape::setFlag(PxShapeFlag::ePARTICLE_DRAIN, true)` function.

Collision filtering

Particles can be selectively configured to collide with any PhysX object. Depending on our requirement, we can ignore the collision of particles with a particular type of actor, whether they are static or dynamic. This can be done to avoid unnecessary performance overhead or to avoid undesired collisions.

For enabling particle-based collision filtering, the `PxParticleBase::setSimulationFilterData()` function is called. This requires a parameter of the `PxFilterData` type, which is a user-definable data that gets passed into the collision filtering shader and/or call-back.

Summary

In this chapter, we learned about the types of particle system, the difference between them, and how each can be used. We learned about particle properties and how to create, update, and destroy particles. We also learned about the use of particle drain and how we can customize the particle collision with other PhysX actors.

9
Cloth

In this chapter we will learn about cloth simulation in PhysX. Topics that are covered in this chapter are as follows:

- Exploring a cloth
- Creating a cloth fabric
- Creating a cloth
- Tweaking the cloth properties, such as cloth collision, cloth particle motion constraint, cloth particle separation constraint, cloth self-collision, cloth intercollision, and cloth GPU acceleration

Exploring a cloth

The PhysX SDK also provides cloth simulation. It can be used for a number of purposes such as simulating cloth behavior of a game character or for simulating realistic behavior of a flag and curtain in a game environment.

Creating a cloth fabric

For creating a cloth in PhysX, we first require an instance of cloth fabric to be created, which basically stores the cloth particle constraints and rest values. A cloth fabric in PhysX is a type of `PxClothFabric,` and to create an instance of it, we need to call the `PxClothFabricCreate()` function that requires three parameters. The first parameter is a type of `PxPhysics`, it is the reference of the created PhysX SDK. The second parameter is a type of `PxClothMeshDesc`, which basically is a descriptor class for cloth mesh. The third parameter is a type of `PxVec3` and it represents the normalized vector specifying the direction of gravity. Here, we need to fill all the description of `PxClothMeshDesc` that contains the information of the cloth mesh such as cloth particles, its inverse weight, and the primitive formed by the cloth particles, which finally make the cloth mesh.

A typical example of creating a cloth fabric is given in the following code:

```
//Array of cloth particles containing position and inverse masses.

PxClothParticle vertices[] =
{
    PxClothParticle(PxVec3(0.0f, 0.0f, 0.0f), 0.0f),
    PxClothParticle(PxVec3(0.0f, 1.0f, 0.0f), 1.0f),
    PxClothParticle(PxVec3(1.0f, 0.0f, 0.0f), 1.0f),
    PxClothParticle(PxVec3(1.0f, 1.0f, 0.0f), 1.0f)
};

PxU32 primitives[] = { 0, 1, 3, 2 };

PxClothMeshDesc meshDesc;
meshDesc.points.data = vertices;
meshDesc.points.count = 4;
meshDesc.points.stride = sizeof(PxClothParticle);

meshDesc.invMasses.data = &vertices->invWeight;
meshDesc.invMasses.count = 4;
meshDesc.invMasses.stride = sizeof(PxClothParticle);

meshDesc.quads.data = primitives;
meshDesc.quads.count = 1;
meshDesc.quads.stride = sizeof(PxU32) * 4;

PxClothFabric* fabric = PxClothFabricCreate(physics, meshDesc,
PxVec3(0, -1, 0));
```

The cloth mesh mentioned in the previous code snippet is made of simple quad but practically it can be anything from a simple polygon primitive to a full-fledged character wardrobe. The mesh descriptor `PxClothMeshDesc` class requires cloth particle information that further contains the position in local coordinates and its inverse masses. Setting the inverse mass of a cloth particle to zero will make it kinematic, which means that it will not be simulated and its position can only be updated by making an explicit update call. After filling the cloth mesh descriptor `PxClothMeshDesc` class, we will create the cloth fabric by calling the `PxClothFabricCreate()` function.

Creating a cloth

Once we are done with the creation of the cloth fabric, we will finally create a cloth by calling the `PxPhysics::createCloth()` function that requires four parameters. The first parameter is a type of `PxTransform`, and it represents the global position of the cloth. The second parameter is a type of `PxClothFabric`, and here we will put the reference of the cloth fabric we created before. The third parameter takes the first element of the `PxClothParticle` array, which is also used in the mesh description class. The last parameter is a type of `PxClothFlags`, and it is used to set the flag to turn the cloth related features on/off such as GPU acceleration and **continuous collision detection** (**CCD**).

A typical example for creating a cloth is given in the following code:

```
PxTransform pose = PxTransform(PxVec3(1.0f));
PxCloth* cloth = gPhysicsSDK->createCloth(pose, fabric, vertices,
PxClothFlags());
gScene->addActor(cloth);
```

Tweaking the cloth properties

In PhysX, a cloth is made of a number of interconnected cloth particles, and each particle can be affected by other PhysX objects as well, as it has to be in constraint with other connected cloth particles. This makes cloth simulation very sensitive to numerical error. Therefore, the cloth solver iterates multiple times in a single time step to achieve physically accurate cloth simulation. We can set the solver iteration count by calling the `PxCloth::setSolverFrequency()` function that requires a parameter of the `PxReal` type, and it represents the solver frequency, that is, the number of solver iterations per second. For instance, setting the solver frequency to 120 corresponds to two iterations per frame if your application is running at 60 frames per second. Increasing this value will increase the cloth simulation accuracy, but the computational requirements will increase as well. The optimal value can be from 120 to 300.

Cloth collision

PhysX cloth can collide with other actors such as spheres, capsules, planes, convexes, and triangles. These colliding actors have to be added by using the `PxClothCollision*` classes, such as `PxClothCollisionSphere.Collision`, with other scene actors. These can also be enabled by using the `PxClothFlag::eSCENE_COLLISION` flag. Cloth collision with sphere and capsule are very cheap compared to convexes and triangles. Thus, if you are integrating cloth on your game character, it is recommended to use a combination of sphere and capsule for defining the shape of your game character.

To add a cloth colliding sphere, the `addCollisionSphere()` function is called. This requires a single parameter of the `PxClothCollisionSphere` type, which defines properties of the added sphere. A maximum of 32 spheres can be added to a PhysX scene.

The code snippet for adding a new cloth colliding sphere is as follows:

```
PxVec3 pos = PxVec3(0,0,0);
PxReal radius = 1.0f;
cloth->addCollisionSphere(
PxClothCollisionSphere(pos,radius));
```

To add a cloth colliding capsule, the `addCollisionCapsule()` function is called that requires two parameters of the `PxU32` type. A collision capsule is defined as the bounding volume of two spheres. Here, we need two spheres of the `PxClothCollisionSphere` type, for defining the upper extent and lower extent of the capsule. The first parameter is the first index of first sphere, and the second parameter is the second index of the second sphere. A maximum of 32 capsules can be added to a PhysX scene.

The code snippet for adding a new cloth colliding capsule is as follows:

```
PxClothCollisionSphere spheres[2] =
{
PxClothCollisionSphere( PxVec3(1.0f, 0.0f, 0.0f), 0.5f  ),
PxClothCollisionSphere( PxVec3(2.0f, 0.0f, 0.0f), 0.25f )
};

cloth->setCollisionSpheres(spheres, 2);
cloth->addCollisionCapsule(0, 1);
```

Cloth particle motion constraint

As we know that a PhysX cloth is made of many interconnected cloth particles, there may be situations where we want to limit the movement (not rotation) of cloth particles. In this case, although the cloth can simulate, its particles' movement will always be within the user-defined constraint.

We can constrain the cloth particle movement by using the `PxClothParticleMotionConstraints` structure, which holds the particle constraint information. It basically constrains the motion of the particle within an imaginary sphere whose local position and radius can be defined by the user. Setting the constraint radius to zero will lock the movement of the cloth particle to the center of the sphere. The `PxClothParticleMotionConstraints` array must either be null, to disable motion constraints, or be the same length as the number of particles in a cloth.

The code snippet to constrain the movement of the cloth particle is as follows:

```
PxClothParticleMotionConstraints motionConstraints[] =

{
PxClothParticleMotionConstraints(PxVec3(0.0f, 0.0f, 0.0f), 0.0f),
PxClothParticleMotionConstraints(PxVec3(0.0f, 1.0f, 0.0f), 1.0f), PxCl
othParticleMotionConstraints(PxVec3(1.0f, 0.0f, 0.0f), 1.0f),
PxClothParticleMotionConstraints(PxVec3(1.0f, 1.0f, 0.0f), FLT_MAX)

};

cloth->setMotionConstraints(motionConstraints);
```

Cloth particle separation constraint

The cloth particle separation constraint is exactly opposite of the cloth particle motion constraint. It will force a cloth particle to stay outside of an imaginary sphere. It can be used for defining any region where the cloth intersection is not required. The `PxCloth::setSeparationConstraints()` function is used to define any separation constraints for the cloth. It requires a single parameter of the `PxClothParticleSeperationConstraint` type, and its array must either be null to disable motion constraints, or be the same length as the number of particles in a cloth.

Cloth self-collision

We can set the property of a cloth so that its cloth particles can collide with each other. This behavior can be enabled by setting a positive value to the `PxCloth::setSelfCollisionDistance()` and `PxCloth::setSelfCollisionStiffness()` functions. The `PxCloth::setSelfCollisionDistance()` function requires a single parameter of the `PxReal` type, which represents the diameter of an imaginary sphere around each cloth particle.

During simulation, it is made sure by the solver that these spheres will not collide with each other. The self-collision distance should always be less than the inter-distance of cloth particles on the rest position. The larger value may create simulation instability or jittering in cloth particles. The `PxCloth::setSelfCollisionStiffness()` function requires a single parameter of the `PxReal` type, and it represents how strong the separating impulse should be when the imaginary sphere of the constraint collides with each other.

Cloth intercollision

If there are two or more cloth actors in a PhysX scene, we can enable cloth intercollision much like cloth self-collision, by calling the `PxScene::setInterCollisionDistance()` and `PxScene::setInterCollisionStiffness()` functions. Both functions require a single parameter of the `PxReal` type, which represents the diameter of an imaginary sphere around each cloth particle and the separating impulse, respectively.

Cloth GPU acceleration

PhysX cloth can be accelerated by a CUDA-enabled GPU such as NVIDIA GeForce and Quadro series GPUs. To enable GPU acceleration, we need to call the `PxCloth::setClothFlag(PxClothFlag::eGPU,true)` function, which will set the `PxClothFlag::eGPU` flag to `true`.

Summary

In this chapter we explained how to create a cloth, which also requires cloth fabric and cloth particles to be created. We learned about cloth properties, such as cloth collision, cloth particle constraint, cloth self-collision and intercollision, and cloth GPU acceleration. We also described various functions and parameters that are required to create a cloth in PhysX.

10

PhysX Visual Debugger (PVD)

In this chapter we will learn about the PhysX Visual Debugger, which is a software for visualization, debugging, and profiling a PhysX application. Topics that are covered in this chapter are as follows:

- Basic concepts of **PhysX Visual Debugger** (**PVD**)
- Connecting a PhysX application with PVD
- Saving PVD streamed data to a PC
- Customizing PVD flags

PhysX Visual Debugger (PVD) basics

PVD is a software that can be installed on your PhysX development platform, and it can be used for real-time visualization and profiling of PhysX simulation. It can visually represent the current simulating PhysX scene and its actors in a separate PVD window, which is independent of your PhysX application rendering implementations. PVD can also be used to record the PhysX simulation of your application, which you can use later on for visualization, analysis, and to find out potential simulation-related problems.

PVD can be downloaded from the same page where we downloaded the PhysX SDK (you may need to register as a developer at `https://developer.nvidia.com`). More information about the PVD and the download link can be found at `http://developer.nvidia.com/physx-visual-debugger`. Nvidia's official PVD video tutorial can be found at `https://developer.nvidia.com/pvd-tutorials`. A PVD user interface guide can be found by clicking on **Help** in the menu bar of the PVD window. At the time of writing, PVD is only available for the PC (Windows) platform.
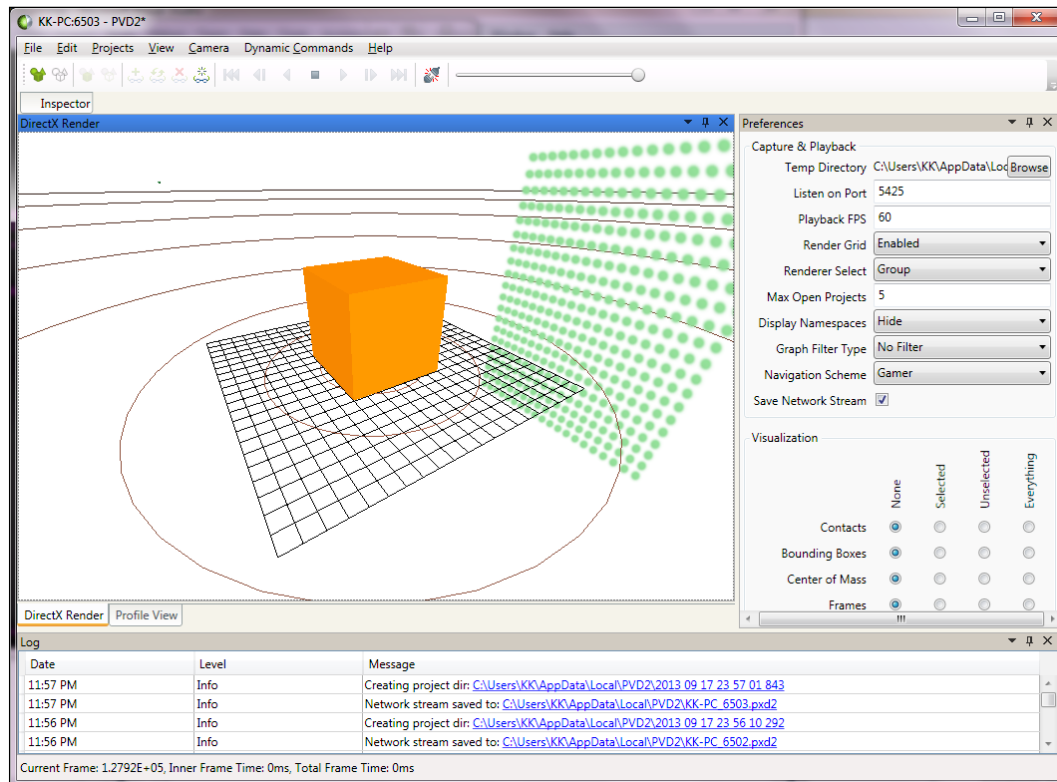
Connecting PVD using a network

Runtime visualization and analysis of a PhysX application by using PVD can be done by streaming the simulation-related data over the TCP/IP network on your local machine. Here, a PVD program works as a TCP/IP server and it must be launched before running your PhysX application. The default port used for listening is 5425.

The code snippet for connecting to PVD by using the TCP/IP network is given as follows:

```
// check if PvdConnection manager is available on this platform
if(gPhysicsSDK->getPvdConnectionManager() == NULL)
return;

// setup connection parameters
const char* pvd_host_ip = "127.0.0.1";  // IP of local PC machine
  PVD
int port = 5425; // TCP port to connect to, where PVD is listening
unsigned int timeout = 100; //time in milliseconds to wait for PVD
  to respond.
PxVisualDebuggerConnectionFlags connectionFlags =
  PxVisualDebuggerExt::getAllConnectionFlags();

// and now try to connect
debugger::comm::PvdConnection* theConnection =
  PxVisualDebuggerExt::createConnection(gPhysicsSDK
  ->getPvdConnectionManager(), pvd_host_ip, port, timeout,
  connectionFlags);

if(theConnection)
cout<<"PVD TCP/IP Connection Successful!\n";
```

Saving PVD data as a file

The PhysX simulation-related data can be streamed to a file, which can be saved to your PC for later analysis of a PhysX application. When PVD is connected through a network, there may be a situation of slow runtime visualization caused by network limitations or a large PhysX scene. In this scenario, streaming the PVD data to a file will be a better alternative. While saving the PVD file to a PC, it should be saved with the file extension `.pxd2`, which is recognized by the PVD software, and can be opened directly by double-clicking on it. You can save the file on your disk partition such as `D:\` but not in `C:\` because of restricted writing permission.

The code snippet for saving the streamed PVD datafile is given as follows:

```
// check if PvdConnection manager is available on this platform
if(gPhysicsSDK->getPvdConnectionManager() == NULL)
return;
// setup connection parameters
const char*     filename = "D:\\PvdCapture.pxd2";  // filename
  where the stream will be written to
```

```
PxVisualDebuggerConnectionFlags connectionFlags =
  PxVisualDebuggerExt::getAllConnectionFlags();

// and now try to connect
debugger::comm::PvdConnection* theConnection =
  PxVisualDebuggerExt::createConnection(gPhysicsSDK
  ->getPvdConnectionManager(), filename, connectionFlags);
```

Connection flags

We can filter out the PVD data that we want during the PVD connection-streaming.
This can be done by customizing `PxVisualDebuggerConnectionFlag`. It also helps
to reduce the size of streaming by ignoring data that is not required.

- `PxVisualDebuggerConnectionFlag::eDEBUG`: This mode transfers all
  possible debug data of rigid bodies, shapes, articulations, and so on. It is
  the most demanding mode in terms of streaming bandwidth.

- `PxVisualDebuggerConnectionFlag::ePROFILE`: This mode populates
  the PVD's profile view, and has very less streaming bandwidth
  requirements when compared to DEBUG. This flag works together with a
  `PxCreatePhysics` parameter and `profileZoneManager`, and it allows you
  to send profile events to PVD.

- `PxVisualDebuggerConnectionFlag::eMEMORY`: This mode transfers the
  memory-usage data, and it allows the users to have an accurate view of the
  overall memory usage of the simulation.

The code snippet for profiling PVD data is given as follows:

```
debugger::comm::PvdConnection * theConnection =
  PxVisualDebuggerExt::createConnection(mPhysics
  ->getPvdConnectionManager(), pvd_host_ip, port, timeout,
  PxVisualDebuggerConnectionFlag::ePROFILE);
```

Summary

In this chapter, we explained what a PVD is and how it can be used for visualization
and debugging of a PhysX application. We learned how to connect PVD with your
PhysX application by using the TCP/IP network. We also described how to save
a PVD-streamed datafile to your PC and customizing the PVD flags.

Index

P

Thank you for buying
Learning Physics Modeling with PhysX

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
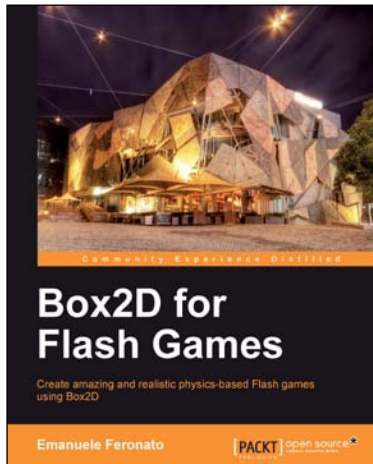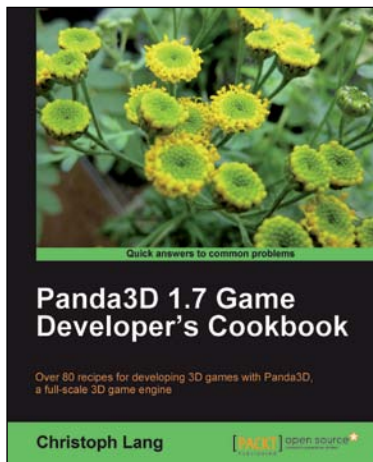
Box2D for Flash Games

ISBN: 978-1-849519-62-5   Paperback: 166 pages

Create amazing and realistic physics-based Flash games using Box2D

1. Design blockbuster physics game and handle every kind of collision

2. Build and destroy levels piece by piece

3. Create vehicles and bring them to life with motors

Panda3D 1.7 Game Developer's Cookbook

ISBN: 978-1-849512-92-3   Paperback: 336 pages

Over 80 recipes for developing 3D games with Panda3D, a full-scale 3D game engine

1. Dive into the advanced features of the Panda3D engine

2. Take control of the renderer and use shaders to create stunning graphics

3. Give your games a professional look using special effects and post-processing filters

4. Extend the core engine libraries using C++

Please check **www.PacktPub.com** for information on our titles

Build a Game with UDK

ISBN: 978-1-849695-80-0   Paperback: 156 pages

Get started with the Unreal Development Kit and make your very first game!

1. Make games using the Unreal Development Kit

2. Design and create worlds

3. Learn to use powerful tools that are currently being used in the industry

HTML5 Game Development with GameMaker

ISBN: 978-1-849694-10-0   Paperback: 364 pages

Experience a captivating journey that will take you from creating a full-on shoot 'em up to your first social web browser game

1. Build browser-based games and share them with the world

2. Master the GameMaker Language with easy to follow examples

3. Every game comes with original art and audio, including additional assets to build upon each lesson.