Physics Simulation
CGP3012M

Workshop #2: Complex Shapes and Materials

Objectives

1. Rendering, Compound Shapes and Materials

Guide time:

- 4 hours (2 weeks)

Tools / Libraries

1. PhysX, Libraries and examples, installed and compiled.
2. Microsoft Visual Studio 2019
3. The Visual Studio Solution files

Introduction

The aim of these workshops is to introduce you to the practical aspects of real-time physics programming using the PhysX middleware. Each session is meant to serve as a tutorial for a topic in physics simulation with explanations and additional practical tasks which should help you to understand the theory covered during lectures.

This workshop will give you some guidance to help you to complete this week's tasks. These tasks are not assessed but are designed in a way which should lead you to the solutions required in the practical assessment for this module. To make the most out of these sessions, discuss your answers with your colleagues and/or ask demonstrators/favourite lecturer for feedback.

Rendering and User Interface

The remote visual debugger is very useful for inspecting different object properties. However, it would be great to have more control over the looks and feels of the implementation whilst being able to handle keyboard and mouse input. Therefore, these tutorials will use a custom made visual debugger based on the OpenGL/GLUT library that will provide such functionalities. In your solution you will find the code for the second workshop in your solution explorer. It has a similar structure to the previous work found in Tutorial 1. Set the relevant workshop project as startup project.

The core physics simulation functionalities are implemented in "PhysicsEngine.h/.cpp", "BasicActors.h" and "MyPhysicsEngine.h". You may notice some changes and additions that were implemented to support visualisation and interaction with the user. The rendering, camera view and movement, and handling of keyboard/mouse input are implemented in "VisualDebugger.h/.cpp" files. It uses some of the OpenGL functions that you may already know. Additional files supporting object and HUD rendering are located in the "Extras" directory.

Rendering and User Interface Task 1: *Familiarise yourself with the new functionality and then compile and run the project. To build the code, you need to switch the active project to Tutorial 2 (right-click on the project in the Solution Explorer/Set as Start-up Project).*

The simulated scene contains a static plane and a box which can be manipulated by applying forces through keyboard. Play with the new functionalities that are provided in the project and refer to the help screen for details.

Rendering and User Interface Task 2: *Create a scene with various actors and change their colour property (Actor::Color) by specifying normalised RGB values (from .0 to 1.) as PxVec3. You may use an external tool to obtain a nice a nice consistent look from various colour schemes. E.g. https://kuler.adobe.com/explore/most-popular/?time=all*

PhysX supports rendering of additional debugging information along the visualisation of object shapes (similar to the remote visual debugger). This extra visualisation can be enabled by pressing the 'F7' key during the simulation. By default, the code in MyScene::SetVisualisation method enables visualisation of wireframe collision shapes only. Read about different debugging parameters in the PhysX documentation and note their default values.

Rendering and User Interface Task 3: *Enable other debug information: for example visualise actor's axes and their linear velocity. Create a series of visually different Actors (e.g.*

box, spheres, etc) using the code that you created last week. For larger objects you might need to increase the default scale factor to correctly render the axes.

Compound Shapes

Actors can be represented by several collision shapes of different type. Implement a class called CompoundObject inheriting from DynamicActor class that will represent an actor consisting of two adjacent cubes. Both cubes are of 1m each.

To add additional shapes to your actor, you have to call Actor::CreateShape() method multiple times with the desired object geometry. You can access each shape by calling Actor::GetShape(PxU32 index) method and change its local position by calling PxShape::setLocalPose() method. For example, to move the second box by 1m along the X axis you would need to write the following line:

```
GetShape(1)->setLocalPose(PxTransform(PxVec3(1.f,.0f,.0f)));
```

You can also change the colour of each individual shape. Check the appearance and behaviour of the created object.

Compound Shapes Task 1: *Alter the position of both shapes such that the actor's axes are placed in the middle of the object. experiment with different values of displacement between two boxes and observe the appearance and behaviour of the compound object. Finally, parameterise your implementation such that the class works with two adjacent boxes of arbitrary size specified in the constructor.*

At this point of the tutorial, we have only considered the position of actors and shapes. However, PxTransform represents the entire pose, so it is also possible to specify and alter orientation of objects and shapes. PxTransform accepts two parameters:

- p (PxVec3)
    - Representing position
- q (PxQuat)
    - Representing orientation.

The second parameter is called a quaternion, a 4d vector of complex numbers, which allows for a compact and efficient representation of orientations. The intricacies of quaternion algebra are not important at the moment, therefore we will only learn their practical uses. One of the overloaded PxQuat constructors allows for specifying the orientation as an angle (in radians) about the desired axis of rotation:

- PxQuat::PxQuat(PxReal angle, PxVec3 axis).

For example, PxQuat(1.f,PxVec3(.0f,1.f,.0f)) represents the orientation of 1 radian angle about the Y axis (in XZ plane). When working with radians it is also useful to use a built-in definition of a constant π – PxPi

Compound Shapes Task 2: *Create a scene with a resting box oriented by 45 degrees about the Y axis and visually inspect the results. You can also orient an actor along multiple axes using the above method. For that you need to multiply all the individual quaternions to get the desired outcome.*

Compound Shapes Task 3: *Create a set of twenty boxes that are stored within an array. Set the initial orientation of each of box to 45+(current instance of the box times two) degrees on the Y axis first and then to 45 degrees about the Z axis.*

A similar approach can be applied to add a specific angle to the existing orientation of an object. To achieve that, you have to first obtain the actor's global pose (PxRigidActor::getGlobalPose), CGP3012M Physics Simulation, Tutorial 2 multiply the pose's q component by the desired orientation (PxQuat) and set the updated pose back (PxRigidActor::setGlobalPose). To get the PxActor pointer, you can use Actor::Get() function which you then need to convert to PxRigidActor pointer to be able to use the methods for setting and getting the global pose. For example, to get the actor's global pose your code may look as follows:

```
PxTransform pose = ((PxRigidBody*)box->Get())->getGlobalPose();
```

Compound Shapes Task 4: *Implement code that will consistently update the orientation of a box in each simulation step about a fixed angle around Y axis. Experiment with different angle values.*

You may extend the existing implementation of the Box class, by implementing additional constructors which accept the initial position as PxVec3 and initial orientation as PxVec3 parameter specifying three angle values (specified in degrees) around X, Y and Z axis.

All above manipulations can also be applied to local shapes within actors. For example, to set the orientation of a local shape, specify the desired pose using PxShape::setLocalPose().

Compound Shapes Task 5: *Create a compound shape lying on XZ plane consisting of four capsules pointing into four different directions, and rotating with a constant speed of 1 rotation per second (2π rad/s).*

Compound Shapes Task 6: *Implement a class representing a rectangular enclosure using compound shapes. Parameterise the class such it accepts dimensions and thickness of the enclosure*

Compound Shapes Task 7: *Place a rectangular box inside the enclosure (e.g a separate object) and then change the orientation of the enclosure around X axis. Observe behaviour of the box.*

Mesh Shapes

It is sometimes necessary or simply more convenient to use user defined shapes instead of built-in types. The creation of such shapes requires 'cooking' – a process which converts a provided set of vertices/triangles into a binary form. The input data can also be created in some other program (e.g. Blender) and loaded from the disk although this functionality is not supported directly by our framework. The user defined shapes can be specified as convex (PxConvexMesh) or concave (PxTriangleMesh). Only the convex shapes can be used for dynamic actors whilst triangle meshes are typically used for modelling static obstacles.

The provided source code contains two wrapper classes: ConvexMesh and TraingleMesh which implement all steps of mesh creation and cooking. The wrapper classes can be used to create customised mesh shapes: see the Pyramid class for example. You can adapt this code to create your own shapes. For that, you need only to alter a single line of code which specifies the vertex coordinates. Similarly the PyramidStatic class demonstrates the use of triangle meshes. This class requires an additional list of triangle indices, grouped in three and following the counter-clockwise order. Each triangle can start from any vertex but the order must be preserved to assure the correct rendering.

Mesh Shapes Task 1: *Create a scene with a pyramid convex object and observe its behaviour. Then, Implement a class that will represent a triangular dynamic wedge shape.*

Mesh Shapes Task 2: *Implement a class representing an extruded hexagon using the convex shape. Parameterise the implementation so it only requires the hexagon's side length and its height.*

Mesh Shapes Task 3: *Implement a class representing a static rectangular enclosure with an open top (similar to the compound object developed in a previous task). Parameterise the implementation so it requires only the height, width, depth and wall thickness parameters.*

Materials : Static and Dynamic Friction

You may have to refer to the material presented in week 5 to complete the next set of tasks.

Static and Dynamic Friction Task 1: *Create a rectangular box resting on the static plane. Modify the default material parameters (static and dynamic friction coefficients) to model the following types of materials: wood-on-wood, teflon-on-teflon and iron-on-iron. Inspect behaviour of the box after applying each of the materials. Several examples of different coefficient values are listed on the following website:*
http://www.engineeringtoolbox.com/friction-coefficients-d_778.html

Then, select one of the specified materials and then double the mass of the box. Observe how this change affects the simulation.

Use the rectangular enclosure class developed in the previous task to create a wide box that will act as a support base. Then place a rectangular box on top of the box. Set the orientation of the base to 20 degrees around the Z axis. Use one of the materials defined in the previous task.

Static and Dynamic Friction Task 2: *Find the minimum value of the coefficient of static friction that will keep the box still.*

Materials : Coefficient of Restitution

The third parameter describing different materials is a coefficient of restitution. This parameter specifies the elasticity (bounciness) of different objects and therefore their behaviour after collisions. The value of the parameter is between 0.0 and 1.0. Lower values of the coefficient corresponding to low elasticity (high energy loss during the impact) while the value of 1.0 defines completely elastic collisions where no energy is lost during the impact. Several examples of the coefficient values are listed on the following website: https://hypertextbook.com/facts/2006/restitution.shtml

Coefficient of Restitution Task 1: *Find the minimum value of the coefficient of static friction that will keep the box still.*

Materials: Multiple Instances of Materials

It is also possible to define multiple materials for different objects on the scene. The helper function PhysicsEngine::CreateMaterial() creates a new material and adds them into an internal list of materials. Individual materials can then be accessed by PhysicsEngine::GetMaterial(PxU32 index) and assigned to specific shapes by using Actor::Material().

Multiple Materials Task 1: *Create two different materials: wood-on-wood and teflon-on-teflon. Then assign them to two different spheres. Drop the spheres from some height and observe their behaviour.*