Lincoln School of Computer Science
University of Lincoln
CGP3012M Physics Simulation

Complex Shapes and Materials

Before you start, download and unpack the source code for Tutorial 2 from Blackboard.

1 Custom Visual Debugger

1.1 Rendering and User Interface

While the remote visual debugger is very useful for inspecting different object properties, we would like to have more control over the visualisation and also be able to handle keyboard and mouse input. We will therefore use a custom made visual debugger based on the OpenGL/GLUT library that will provide such functionality.

The new project called Tutorial 2 has a similar structure to the project from Tutorial 1. The core physics simulation functionality is implemented in "PhysicsEngine.h/.cpp", "BasicActors.h" and "MyPhysicsEngine.h" files. 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 - you should recognise some of the OpenGL functions. Additional files supporting object and HUD rendering are located in the "Extras" directory. Spend some time to study the code and 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. Check out the new functionality provided in the project – refer to the help screen for details.

Task4U: Create a scene with various actors and change their colour property (`Actor::Color`) by specifying normalised RGB values (from .0 to 1.) as `PxVec3`.To get a nice consistent look check out one of the colour schemes (https://kuler.adobe.com/explore/most-popular/?time=all).

1.2 Additional Debugging Support

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.

Task4U: Switch on some other debug information: for example visualise actor's axes and their linear velocity. For larger objects you might need to increase the default scale factor to actually see the axes.

2 Compound Shapes

2.1 Positioning

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 (of size 1 m 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.

Task4U: Alter the position of both shapes such that the actor's axes are placed in the middle of the object.

Now 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.

2.2 Orientation

So far we have considered the position of actors and shapes only. 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 and `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 so important at the moment and 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`.

Task4U: 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.

Task4U: Set the initial orientation of the box to 45 degrees about 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 (`PxActor::getGlobalPose`), multiply

the pose's q component by the desired orientation (`PxQuat`) and set the updated pose back (`PxActor::setGlobalPose`).

Task4U: 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.

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()`.

Task4U: 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).

Task4U: Implement a class representing a rectangular enclosure using compound shapes. Parameterise the class such it accepts dimensions and thickness of the enclosure.

Task4U: Place a rectangular box inside the enclosure (a separate object!) and then change the orientation of the enclosure around X axis. Observe behaviour of the box.

3 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 adopt 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.

Task4U: Create a scene with a pyramid convex object and observe its behaviour.

Task4U: Implement a class that will represent a triangular dynamic wedge shape.

Task4U: 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.

Implement a class representing a static rectangular enclosure with an open top (similar to the compound object developed in Task 2.2). Parameterise the implementation so it requires only the height, width, depth and wall thickness parameters.

4 Materials

4.1 Static and Dynamic Friction

Task4U: 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

Task4U: 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.

Task4U: Find the minimum value of the coefficient of static friction that will keep the box still.

4.2 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 correspond 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: http://hypertextbook.com/facts/2006/restitution.shtml.

Task4U: Create a sphere falling onto a static plane. Test the behaviour of the sphere for different values of the coefficient of restitution.

4.3 Multiple 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()`.

Task4U: 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.