Lincoln School of Computer Science
University of Lincoln
CGP3012M Physics Simulation

Introduction to PhysX

1 Introduction to Workshop Sessions

The aim of these workshops is to introduce you to practical aspects of real-time physics programming using the PhysX middleware. There are 4 tutorial sessions for this module in total, each designed to take 2 weeks or 4 hours. Each session is meant to serve as a tutorial for a particular topic in physics simulation with explanations and additional practical tasks which should help you to understand the theory covered during lectures. These practical tasks are indicated by *Task4U* label and are highlighted in italics. 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 for feedback.

2 Tutorial

2.1 What is PhysX?

PhysX is a real-time physics simulation middleware developed by NVidia. It is written in C++ and can be deployed on different operating systems and hardware platforms. A good source of information relevant to these tutorials can be found in the official PhysX SDK documentation, so when you see a new PhysX class/function remember to look it up in the documentation first. The documentation can be found online (http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/apireference/files/index.html) and also as a separate file "Documentation\PhysXAPI.chm" located in the PhysX SDK folder (e.g. "C:\Program Files (x86)\NVIDIA Corporation\NVIDIA PhysX SDK\3.3.4").

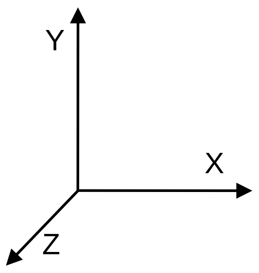
If you are intending to use the PhysX SDK on some other computer (e.g. at home) you will need to install the software yourself; you will find the detailed instructions at the end of this document.

2.2 Basic Program

Download the sources for Tutorial 1 from Blackboard ("Study Materials/Week B2"), unpack the archive into a directory and open the solution file "PhysX Tutorials.sln". The solution contains two projects: "Basic Program" and "Tutorial 1". "Basic Program" consists of a single file only and demonstrates a minimal setup needed for running the simulation with PhysX, without any visual debugging support. The code implements a simple scene with two actors: a static infinite plane and a box. The functionality of the program is limited to displaying the location and velocity of the box in global coordinates. There are three distinctive parts of the program: initialisation (where the PhysX SDK and the scene with actors are initialised), the main loop (performing one step of simulation, processing and display functions) and the final part that releases all resources. Familiarise yourself with the code and try to understand the main program loop.

The PhysX SDK makes use of several maths classes. The most important right now are `PxReal` (real data structure, like float or double) and `PxVec3` – a 3 dimensional vector. These classes come together with a set of functions and operators that will be very useful later on.

At this point, let us assume that the world/global coordinates are represented as follows:



For example if you want to set the position of an actor/object in global coordinates, specify x, y and z coordinates as a 3d vector: `PxVec3(x,y,z)`.

Now, try to compile and run the code. Inspect the global position and velocity of the box. Can you tell when the box hits the ground plane?

2.3 Simulation

The time interval `delta_time` between simulation steps is fixed and set to 1/60th s.

Task4U: How long does it take (in seconds) from the start of the simulation until the impact?

To calculate that value, implement a global counter that will count the number of simulation steps. Display the counter's value in the `Display` function and inspect its value visually during the impact.

Now, set the simulation step to 1/10th of a second.

Task4U: How many steps does it take from the start until the impact after the change?

Revert back to the original settings. Remove/comment out a line of code that adds a plane to the scene and see what happens to the box trajectory.

2.4 Properties

Revert back to the original settings. Read about the `PxRigidActor::setGlobalPose` function in the documentation to see how to set the pose of an actor. This function accepts a universal transformation `PxTransform` as input which can be any combination of translation and rotation. To specify translation only, provide a desired `PxVec3` vector as a parameter to `PxTransform`.

Task4U: Change the position of the box to 10m along the X axis. This change should happen after 10 steps of simulation. See what happens to the velocity and position of the box after the change is made.

Now, inspect the mass property of a box: `PxRigidBody::getMass`. You can, for example, display the mass value before the main loop starts. The mass of an actor is automatically calculated from its shape and density (mass per volume, specified in kg/m^3). Now, double the size of the box (see the `PxBoxGeometry` class) and inspect the mass value of the box again.

Task4U: Set the box dimensions to an approximate size of your body (width, depth and height) and find the density of the box that would result in approximate mass of your body.

2.5 Forces

Revert back to the original settings. Set an initial height of the box to its rest position (0, 0.5, 0). Apply an instant force of 100N (`PxRigidBody::addForce`) along the X axis just before the main loop starts and see what happens.

Task4U: Set the velocity of the box to zero when it moves 10 meters away from the initial position and observe its behaviour.

Revert back to the original settings. Set the initial height of the box to its rest position (0, 0.5, 0). Change the value of dynamic friction coefficient (we will learn about different material properties later on) for the default material in the `InitScene` function:

```
PxMaterial* default_material = physics->createMaterial(.0f, .2f, .0f);
```

Apply the same force again and observe what happens to the actor.

Task4U: Set the value of dynamic friction coefficient such that the box stops after approx. one meter, due to friction.

If the object stops to move, its state automatically changes to “sleeping” which allows for excluding the object from simulation calculations and optimise the overall performance.

Task4U: Repeat the previous task, but this time display also the sleeping property of the box (see PxRigidDynamic::isSleeping()). Find out when exactly the object changes its state.

Now, revert back to the original settings. Change the restitution/bounciness of the default material:

```
PxMaterial* default_material = physics->createMaterial(0.f, 0.f, .5f);
```

Note the resulting behaviour and explore other restitution values.

2.6 Remote Visual Debugger

As you have probably noticed, it might be quite difficult to realise what is going on in the simulated scene without actually seeing the objects. The easiest way to visualise your simulation is by using the PhysX Remote Visual Debugger, distributed as a separate application already installed on the lab computers. Before you run your program again, you have to start up the Remote Visual Debugger application – your program checks if the debugger is present only once at the beginning (see the `PxInit` function).

Now, revert back to the original settings and run the program again. You should now be able to see the scene in the debugger window. You can also remove delays in the main loop for smoother animation.

The debugger application automatically records all the information about the scene in so called ‘clips’ from the very beginning of the simulation until you press the stop button. After that you are able to inspect every frame of the simulation and all important object parameters; although before doing that, you first need to close your application or press the disconnect button in the debugger.

Task4U: Explore the remote debugger interface and inspect the parameters of all objects in the scene. Repeat tasks from Section 2.3 but this time with the visualisation support.

2.7 Actor Types

PhysX has a hierarchy of different actors which inherit from the base `PxActor` class. The main two types are dynamic and static actors. Dynamic are all standard objects with body and shape description. They react to forces, collisions etc. A rectangular box from the previous tasks is an example of a dynamic actor. It is a good practice not to move dynamic objects manually (although this is exactly what we have tried to do in our initial task!) and only use forces to affect their behaviour.

However, in some situations it is necessary to move actors manually (e.g. when implementing platforms, mechanisms, etc.). This should be done via the use of so called kinematic actors which are almost the same as dynamic objects but do not react to forces. To change a dynamic actor to a kinematic one, simply set its kinematic flag:

```
PxRigidBody::setRigidBodyFlag(PxRigidBodyFlag::eKINEMATIC, true);
```

Static actors do not have any dynamic properties (i.e. body description). Once created, they should not be manipulated in any way (although it is still possible to do so). Such objects can define for example static obstacles in the environment. A plane from the previous tasks is an example of a static actor.

To create a static actor, use the `PxPhysics::createRigidStatic()` method together with a line of code that creates its shape, similarly to the dynamic actor example. This time you should not provide any mass/density parameters as all static objects have infinite mass by default.

Drop a dynamic box from a significant height (e.g. 100 m) onto another box resting on the ground.

Task4U: Change the type of the resting box to dynamic, kinematic and static and observe the behaviour of all objects.

Now, create a single dynamic box resting on a plane as before. Place three additional resting boxes of different type (dynamic, kinematic and static) 5 meters apart along the X axis.

Task4U: In each simulation step, increment the global position of the first box along the X axis by 10 cm and observe its behaviour. Switch on the kinematic flag of the first box, re-run the simulation and observe the changes in behaviour. Instead of using `PxRigidActor::setGlobalPose()` use `PxRigidDynamic::setKinematicTarget()` to adjust the pose of the box. Observe what happens and note the differences.

2.8 Basic Shapes

Create a function called `PxRigidDynamic* CreateBox()` which will accept the three following parameters: initial position (`PxVec3`), dimensions (`PxVec3`) and density of the box (`PxReal`). Create multiple boxes with different parameters.

Task4U: Create similar functions for capsules and spheres. Populate your world with new actors!

2.9 Structuring the Code

So far we have used a fairly simple simulation scenario in which we were not so worried about the code structure. However, our simulations will be getting more complex and therefore it is a good idea to arrange the program into meaningful blocks and classes. Switch the active project in the solution to “Tutorial 1”, which implements the same simulation scenario but this time with core functionality implemented as classes. First explore the code a little and compare it to the previous version.

The core functionality is implemented in “PhysicsEngine.h/.cpp” files which contain the namespace PhysicsEngine and methods for PhysX initialisation and two new custom classes: Actor and Scene. The basic classes are extended in “BasicActors.h” (the Box and Plane actors) and “MyPhysicsEngine.h” implementing the `MyScene` class with customised initialisation and update functions. We will be adding more classes and functionality mainly to these two files.

Task4U: Repeat tasks in Section 2.8 but this time using class implementations.

3 Additional Information

3.1 Setting-up PhysX SDK

If you would like to set up the PhysX SDK on some other computer you have two choices:

- If you have Visual Studio 2015, you can download a binary release for PhysX SDK 3.3.4 from Blackboard (Study Materials/Week B2). You need to unpack and copy the folder into a directory “C:\Program Files (x86)\NVIDIA Corporation\NVIDIA PhysX SDK\3.3.4”.
- If you use other version of VS or a different compiler, you need to build the SDK from sources which you can access from https://developer.nvidia.com/physx-sdk. You will be asked to register first and then you can clone the PhysX folder to your local machine. Follow the building instructions for your target system and development environment. You can then copy the entire directory to “C:\Program Files (x86)\NVIDIA Corporation\NVIDIA PhysX SDK\3.3.4” to avoid having to make further changes to the project.

You will also need to update the system path, so your program knows where to find the PhysX related dlls. Dll files are stored in the PhysX SDK binary folder (e.g. “C:\Program Files (x86)\NVIDIA Corporation\NVIDIA PhysX SDK\3.3.4\Bin\win64”). You should add that directory to the system path variable (i.e., PATH) which can be set in “Start/Computer/Properties/Advanced System Settings/Environment Variables/”. You have to set that path before you start Visual Studio.