Physics Simulation

CGP3012M

Workshop #1: Introduction to PhysX

Objectives

1. Navigating your way around PhysX
2. Installing it at home

Guide time:

4 hours (2 weeks)

Tools / Libraries:

1. PhysX, libraries and examples, installed.
    a. The sources are provided to you throughout this workshop.
2. Microsoft Visual Studio 2019 (Community or above)
3. The Visual Studio solution files that contains the skeleton files to build PhysX

Introduction

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.

Basic Program

Before you start, you **must** your own installation of the PhysX SDK at home. Please refer to the Installation Notes section. SDK is already installed on the Lab's PCs.

The sources for this tutorial are located on blackboard and/or in week 01 on this modules MS Teams Group. Please download and unzip the following files:

- PhysX-3.4.2.zip
- PhysXTutorial-master2021.zip

Installation Notes

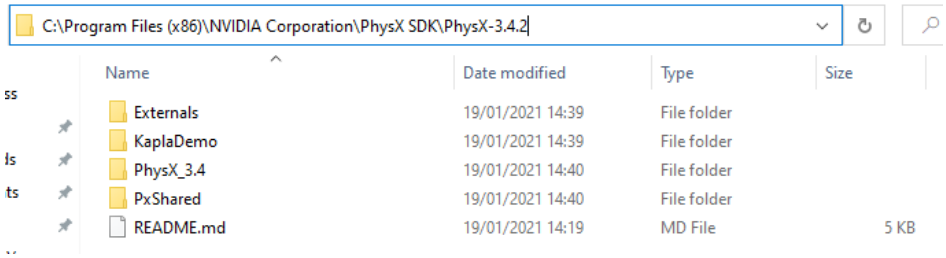
Visual Studio 2019 ONLY

You **must** install the PhysX SDK 3.4.2. correctly for the basic PhysX program code to successfully compile and run. To achieve this, follow these steps:

Download the binary release of PhysX SDK 3.4.2 and unpack it. Then head to following directory (create these directories manually if it does not already exist).

`C:\Program Files (x86)\NVIDIA Corporation\PhysX SDK\PhysX-3.4.2`

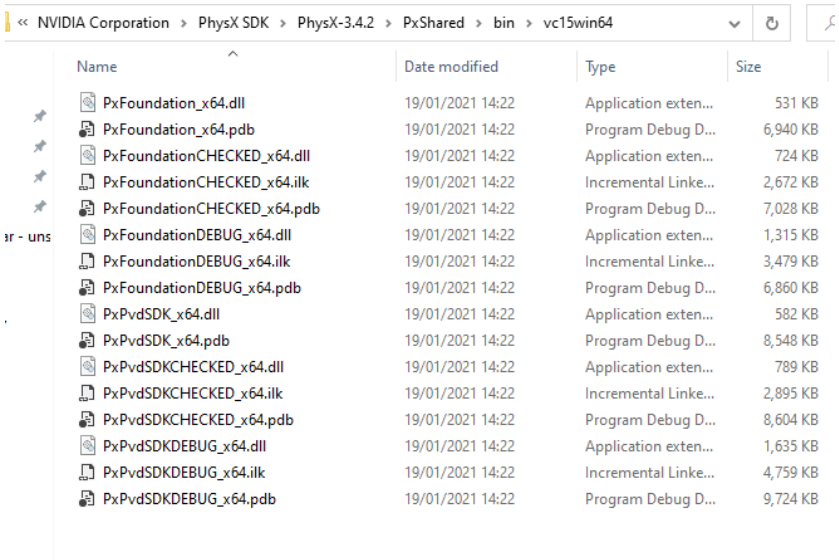
Copy all the files from the unpacked PhysX-3.4.2.zip file into the folder mentioned above. Your file structure in explorer should look like this:



To test if you successfully unpacked/installed the SDK, open up a run window (keyboard shortcut: windows key + r) and copy paste this path:

`C:\Program Files (x86)\NVIDIA Corporation\PhysX SDK\PhysX-3.4.2\PxShared\bin\vc15win64`

If you unpacked your PhysX correctly, then an explorer window should pop up:



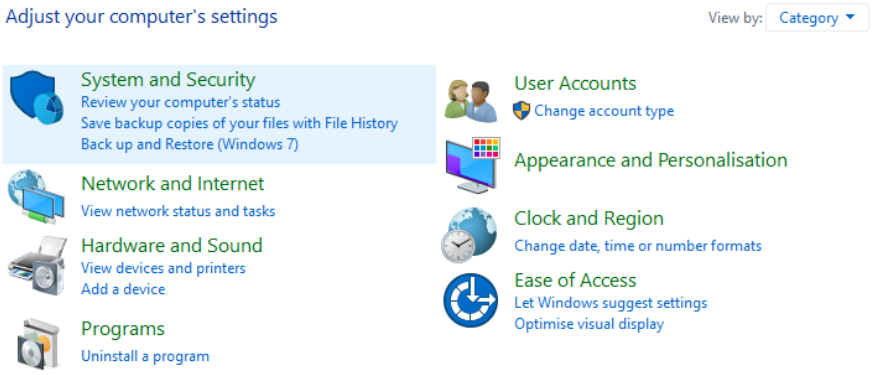
If an error message pops up, please double check your paths and folder structure.

Then, you **must** update the system path, so your program knows where to find the PhysX DLLs. You **must** set these paths before you start Visual Studio.

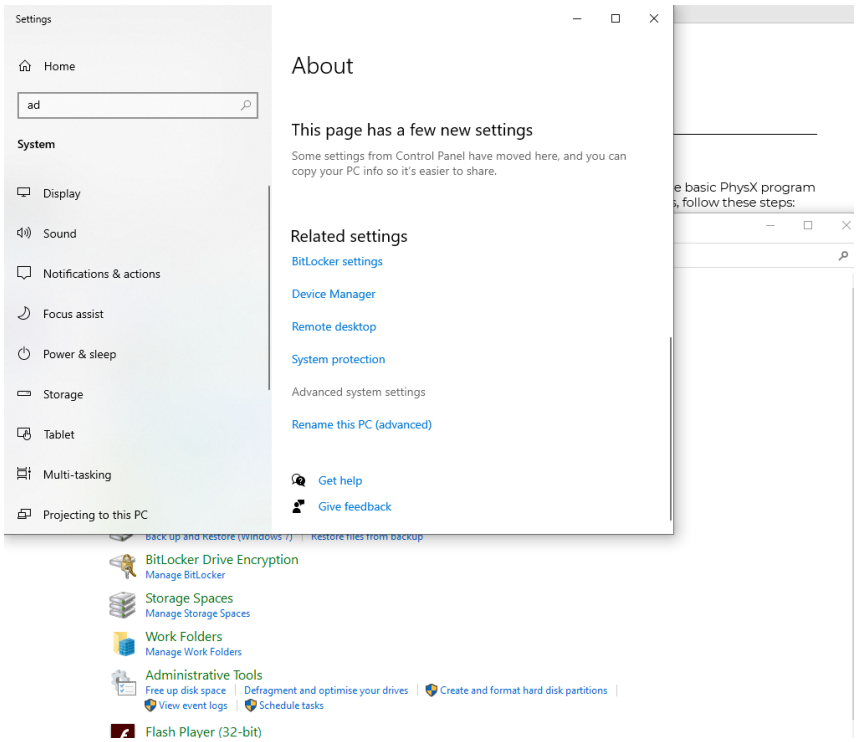
Open the Windows Control Panel, choose **System and Security -> System -> Advanced System Settings -> Environment Variables -> Path -> New**

The screenshots below show where to find the environment system path window. Note: These screenshots may differ from your machine, please use them to navigate your way around the environment tables path table.

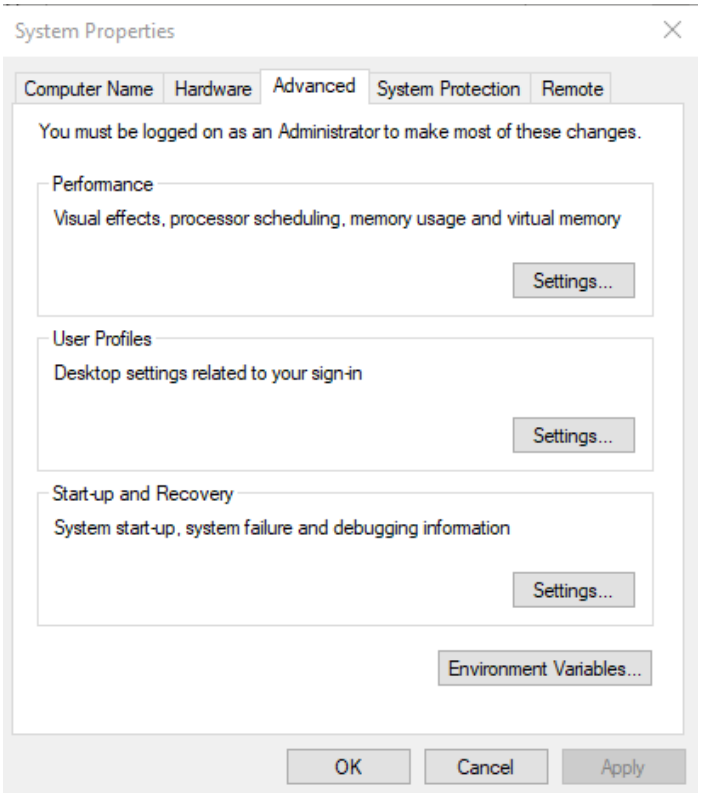
Select System and Security (inside Control panel, View by Category)



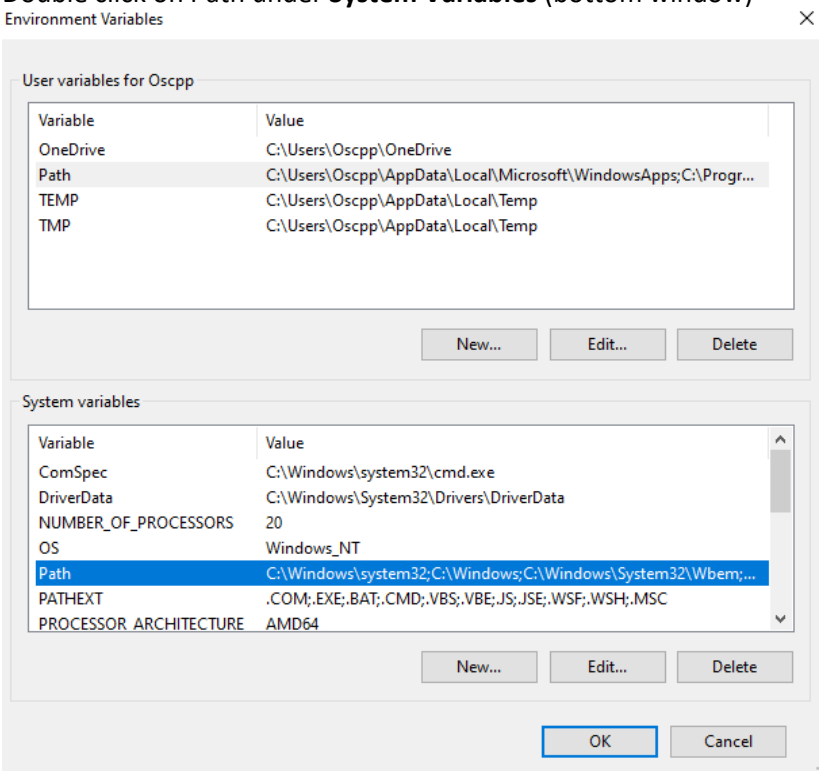
Chose Advanced System Settings.



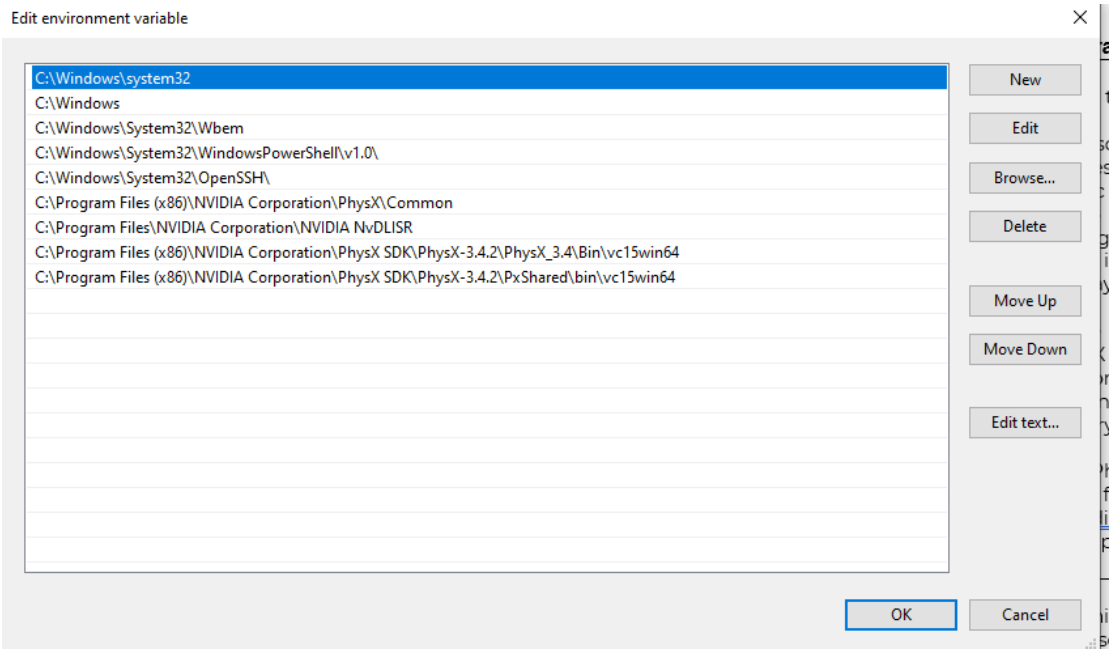
Select Environment Variables



Double click on Path under **System Variables** (bottom window)



Select New and add the paths as detailed below.



The required DLLs are stored in two PhysX SDK binary folders. You **must** add both directories into the system path variable (PATH). Please add them to the environment variable using the "new" button for each path. You can copy paste the path:

```
C:\Program Files (x86)\NVIDIA Corporation\PhysX SDK\PhysX-3.4.2\PhysX_3.4\Bin\vc15win64

C:\Program Files (x86)\NVIDIA Corporation\PhysX SDK\PhysX-3.4.2\PxShared\bin\vc15win64
```

Beware: Double check your paths when copy pasting from the workshops. On some machines, the *dash* – is not copy pasted correctly.

Press OK, and close every control panel windows. The SDK is set up now.

Other Environments and Compilers

If you are using any other version of VS, or a different compiler/operating system, you must build the SDK from sources from NVidia themselves: https://developer.nvidia.com/physx-sdk. From version 3.4.2 PhysX is open source and available on GitHub: https://github.com/NVIDIAGameWorks/PhysX/tree/3.4.2-bsd. You will also need to adjust the project for your specific environment and be aware that some functionality such as cloth simulation is not available in newer SDKs. Please be aware that no support will be given for other OS, Development Environment and Compilers.

Programming tasks

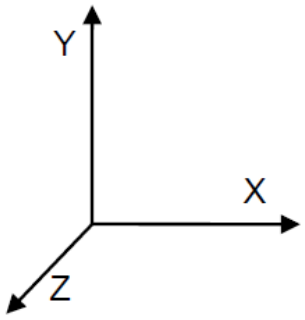
Open the solution file "PhysX Tutorials.sln" in visual studio.

The solution contains several projects but for this workshop we are interested only in the following two: **"Basic Program"** and **"Tutorial 1"**. "Basic Program" consists of a single file only and demonstrates the minimal setup required 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; It displays 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 thing for 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 are most useful.

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)`.

Compile and run the code. Inspect the global position and velocity of the box. When does the box hit the ground plane? Can you tell why?

Simulation

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

Simulation Task 1: 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.

Simulation Task 2: Set the simulation step to 1/10th of a second. How many steps does it take from the start until the impact after the change?

Remove/comment out a line of code that adds a plane to the scene and observe the box trajectory.

Properties

Revert back to the original settings (interval delta_time) before attempting this task.

Read about the `PxRigidActor::setGlobalPose` function in the documentation to understand 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`.

Properties Task 1: Change the position of the box to 10 m along the X axis. This change should happen after 10 steps of simulation. Observe what happens to the velocity and position of the box after the change is made.

Inspect the mass property of a box: `PxRigidBody::getMass()`. You can, for example, display the mass value in the console 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.

Properties Task 2: 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.

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.

Forces Task 1: Set the velocity of the box to zero when it moves 10 meters away from the initial position and observe its behaviour.

Forces Task 2: 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 (these properties will be shown later in the module) for the default material in the `InitScene` function:

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

```
.0f);
```

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

Forces Task 3: Set the value of dynamic friction coefficient such that the box stops after approximately one meter. If the object stops moving, PhysX will change its state automatically to "sleeping". This allows object to be excluded from simulation calculations and to optimise the overall performance

Forces Task 4: 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

Forces Task 5: Revert back to the original settings. Change the restitution/bounciness of the default material and observe any change of behaviour. Explore other restitution values!

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

Remote Visual Debugger

It is most 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. It is distributed as a separate application available through https://developer.nvidia.com/physx-visual-debugger.

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.

Remote Visual Debugger Task 1: Explore the remote debugger interface and inspect the parameters of all objects in the scene. Repeat previous tasks with the visualisation support.

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 may be 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, 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.

Actor Types Task 1: Drop a box from a significant height (e.g. 100 m) onto another box resting on the ground. Change the type of the resting box to dynamic, kinematic and static and observe the behaviour of all objects.

Actor Types Task 2: 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. Let a few boxes fall on these resting boxes.

Actor Types Task 3: 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.

Basic Shapes

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

Basic Shapes Task 1: Create similar functions for spheres and capsules. Populate your world with countless actors.

Structuring the Code

So far we have used a fairly simple simulation scenario in which we were not worried about the code structure. However, our simulations will be getting marginally more complex.

Therefore, it is a good idea to arrange the program into meaningful functions 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.

Structuring the Code Task 1: Restructure the code using an object-oriented approach. Repeat previous tasks to find out if your re-implementation still works as intended.