Physics Simulation
CGP3012M

Workshop #3: Joints and Triggers

Objectives

1. Joints -- Distance, Revolute & Motors
2. Handling collisions -- Triggers, filtering, CCD

Guide time:

4 hours (2 weeks)

Tools / Libraries:

1. PhysX, libraries and examples, installed & 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.

Joints and Triggers

Set Tutorial 3 as your start-up project. Additionally, you may wish to copy the code you have been working on in previous workshops into this project.

Joints

PhysX implements a number of different joints which can be used to connect two actors, in a way which restricts their relative movement. The objects still move under dynamics (e.g. gravity, friction, etc.) but additional restrictions are set on their relative positions and orientation.

Due to their nature, only non-static actors can be linked by joints (i.e. kinematic or dynamic actors). Furthermore, it is also possible to attach a single actor to a specific joint -- this is equivalent to linking this actor to a point in the world.

To create a joint which connects two actors, a function to create the joint (e.g. **`PxDistanceJointCreate`**) can be used, along with pointers to both actors and the joint's relative poses with respect to both actors (see for example, the **`DistanceJoint`** class).

Alternatively, a joint can be fixed to a position in the world instead of another actor. To do this, a null pointer must be provided instead of the first actor, and the first local pose parameter must be given a pose in global coordinates.

It is often useful to visualise the state and behaviour of joints when working with them. This is especially useful when debugging them. For example, you may wish to enable the following function in the debug renderer:

- **`PxVisualizationParameter::eJOINT_LOCAL_FRAMES`**: to visualise joint local axes;
- **`PxVisualizationParameter::eJOINT_LIMITS`**: to visualise the limits of joints.

You will also need to turn on the visualisation for each individual joint in your scene, with `PxJoint::setConstraintFlag(PxConstraintFlag::eVISUALIZATION, true)`. Note: visualising distance joints is currently not supported.

Distance Joints

Joints Task 1: Distance joints are particularly useful for creating spring-like objects. Your first task is to create a trampoline object, using the `Trampoline` class.

The trampoline consists of two thin rectangular boxes, and four springs holding the boxes separate. Since the `Trampoline` class consists of multiple actors, you will need to use the `Trampoline::AddToScene()` method to add all objects into the scene.

To test the behaviour of the trampoline, drop a rectangular box from a height onto the trampoline and observe the behaviour of the objects.

Joints Task 2: Try tweaking the values of spring stiffness and damping parameters, and observe their effect on the behaviour of the trampoline.

Revolute Joints

Joints Task 3: Create a scene with two boxes joined together with a revolute joint.

You can use the commented code for guidance, and a basic implementation of this. For revolute joints, it is possible to specify a custom axis of rotation, which is **x** by default. In the example seen in the commented code, the joint is oriented by 90 degrees along the **y** axis. This results in the axis of rotation pointing away from the screen (when seen in the debug visualisation mode).

Once this is done, apply a force to the second actor and see how this actor behaves.

Joints Task 4: Change the state of the first actor to dynamic, and see what happens.

Joints Task 5: Remove the first actor from the code (using a null pointer instead) to fix the first actor to a global point. Observe the changes in behaviour of the actors.

Keyboard handling

To enable user interaction within the simulated world, it is often useful to handle keyboard events. The keyboard events are already implemented as part of the camera and object handling functions in **`VisualDebugger.cpp`**.

Keyboard handling is done with GLUT, and the key press callbacks are implemented as `KeyPress()`, `KeyRelease()`, and `KeySpecial()`. Furthermore, the KeyHold() function is also called in every simulation step. Familiarise yourself with these functions, and see how keyboard handling can be implemented.

Additionally, there are also three functions which can be used for user-defined keyboard handling, including `UserKeyPress()`, `UserKeyRelease()` and `UserKeyHold()`. To understand these further, look at the implemented example to see how to interface the keyboard events with the simulation.

Motors

Joints Task 6: Some types of joints feature motors, which can animate the connected actors. An example usage of this may be a motor which drives the wheels on a car.

Use the scenario from **Joints Task 3** (the two boxes connected by a revolute joint). Switch on the motor by providing the drive velocity value (**`RevoluteJoint::DriveVelocity`**) and observe the behaviour of the actors. You can also switch off gravity for the actor with (**`PxActor::setActorFlag(PxActorFlag::eDISABLE_GRAVITY, true)`**) to see the action of the motor unaffected by external forces.

Joints Task 7: Implement code that will change the direction of the motor speed after pressing a key.

Joints Task 8: Implement an additional method in **`RevoluteJoint`** class which will set/get the value of the gear ratio. Tweak the values for this parameter, and see what happens.

Joints Task 9: Implement a simple clock consisting of two capsules (that represent hour and minute hands. The hands should rotate 60 times faster than a normal clock. (You can use motors here to move the arms!)

Joints task 10: Implement a simple planetary system consisting of three planets (spheres) rotating around the centre. Use revolute joints to achieve that behaviour. You do not need to consider modelling gravity between the planets.

Limits

It is also possible to specify joint limits that will further restrict the motion of the joined actors.

Joints task 11: Switch on the joint limits through the use of the `RevoluteJoint::SetLimits` method. Change values of the limit parameters and observe the behaviour of the actors in the scene.

Collisions

Triggers

Triggers are shapes used in games to cue events when particular objects intersect them. An example of this might be an automatic door which objects when the player is nearby (by intersecting a trigger close to the door). Triggers cannot be used for simulation, and therefore you have to switch the simulation flag off (see `Actor::SetTrigger()`).

Every time an object intersects with the trigger object, a special event callback is generated which calls a user-defined "trigger report" function which can be used to implement the desired response. In our case, this callback is `MySimulationEventCallback::onTrigger`.

Collision Task 1: Create a box that will act as a trigger object in your scene. Following this, create a sphere and drag it around the scene through the trigger box. Observe the output generated by the trigger report function, and see what is outputted when the two objects collide.

Collision Task 2: Create a counter variable, and increment this every time the trigger object is hit by the sphere. Log this counter to the console to show how many times the two objects have collided.

Collision Task 3: Modify the trigger report function such that the sphere will

disappear at the first contact with the plane.

Collision Task 4: Modify the trigger report function so that the sphere will bounce quickly off when it first collides with the plane. This could be done by applying a large instantaneous force to the sphere.

Following these tasks, create three different dynamic actors which are simple shapes: a box, a sphere and a capsule.

Collision Task 5: Modify the trigger report function to display the type of the object that just collided with the trigger object. You can check the geometry type with `PxShape::getGeometryType()`.

Collision Task 6: Modify the trigger report function so it only responds to collisions with a specific actor. One method to achieve this is by setting a unique actor name (`Actor::Name()`) in the init function, and then checking this name in the trigger report. (`PxTriggerPair::otherActor->getName()`).

Dynamic Triggers

To use triggers with dynamic actors, an additional shape should be added to the existing actor, which will follow its behaviour.

Collision Task 7: Use your previous implementation of a `CompoundObject` but without changing the local pose of the second shape -- so that both shapes are aligned. Then, switch on the trigger flag for the second shape.

Drag the object around a populated scene and notice all the trigger events which are caused.

Collision Filtering

Collision filtering enables customised collision handling which does not require triggers. This is realised by so-called "customised collision filter shaders". So far, the simulations we have considered have used the default shader `PxDefaultSimulationFilterShader` which was provided as an input argument for the Scene class.

An example of a custom filter shader is included with the provided code (see **CustomFilterShader**) which enables all necessary collision flags for a given pair of matching objects. It is the developer's choice which flags need to be raised -- they will all trigger a different behaviour.

A matching pair can be defined by using the **Actor**::SetupFiltering() function, e.g. box->SetupFiltering(**FilterGroup**::ACTOR0, **FilterGroup**::ACTOR1). The first argument here specifies the ID of the actor itself (in this case, the box), and the second argument specifies the ID of a matching actor (or many actors, these can be combined with the | operator).

The ID is defined as an enum type which can be customised to the needs of the developer (for example, different names, more IDs, etc.). To enable collision generation, the filtering flags for the matched actor should also be specified as well.

Collision Task 8: To try the new functionality, first create two dynamic actors with custom names (these should be unique). To switch on the custom shader, three steps need to be completed:

- Replace the default filter shader with the custom one defined in the **MyScene** constructor;
- Set the filtering flags for a matching pair of actors;
- Implement the appropriate response by overriding **PxSimulationEventCallback**::OnContact().

After that, you should see the names of two matched actors displayed together with the associated event whenever they collide.

Collision Task 9: Custom shaders also allow the use of "Continuous Collision Detection" (CCD), which helps with missing detections for fast and small objects (e.g. bullets).

CCD can be enabled for a specific actor by settings the corresponding CCD flags in the following places:

- **Scene**::Init();
- CustomFilterShader();
- The selected actor: **PxRigidBody**::setRigidBodyFlag(**PxRigidBodyFlag**::**eENABLE_CCD**, true).

Collision Task 10: Populate your scene with two resting dynamic actors. Then, set

the initial velocity for the first actor such that it collides with the other object. Once this is the case, find the initial velocity in that direction high enough to cause the first actor to “miss” the second one.

Following this, turn on CCD and observe the behaviour of the actors.