Lincoln School of Computer Science
University of Lincoln
CGP3012M Physics Simulation

Joints and Triggers

Download and unpack the source code for Tutorial 3. You may also wish to copy the code that you have developed in the previous sessions.

1 Joints

PhysX implements a number of different joints which are used to connect two actors, in a way which restricts their relative movement. The objects still move under dynamics (gravity, friction, etc.) but additional restrictions are set on their relative positions and orientation. Only non-static actors can be linked by joints (i.e. dynamic or kinematic). 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".

The joints usually connect two actors so to create a joint one has to use the respective create function (e.g. `PxDistanceJointCreate`) and provide 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 global point rather than the first actor instead; in this case, you need to provide a null pointer instead of the first actor and specify the first local pose parameter as a pose in global coordinates.

When working with joints it might be useful to visualise some of the aspects related to their state and behaviour. 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 joint limits.

In addition you will need to switch on the visualisation for each individual joint: `PxJoint::setConstraintFlag(PxConstraintFlag::eVISUALIZATION, true)`.

1.1 Distance Joints

Distance joints are useful for creating spring objects. Create a trampoline object using the Trampoline class. The trampoline is build out 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 to the scene. Drop a rectangular box on the trampoline and observe behaviour of the objects.

Task4U: Try out different values of spring stiffness and damping parameters.

1.2 Revolute Joints

Create a scene with two boxes joined together by a revolute joint; you can check out the commented code for a basic implementation. For revolute joints, it is possible to specify a custom axis of rotation, which is X by default. In the provided example (see the commented code) the joint is

oriented by 90 degrees along the Y axis, resulting in the axis of rotation pointing away from the screen (check it in the debug visualisation mode). Now apply the force to the second actor and see how it behaves.

Task4U: Change the status of the first actor to dynamic and see what happens.

Task4U: Remove the first actor from the code (specify the null pointer instead) to implement an actor fixed to a global point and observe changes in behaviour of the actors.

1.3 Keyboard handling

To enable a sensible user interaction with the simulated world, it is useful to handle keyboard events. The keyboard events are already implemented as part of camera and object handling functions in "VisualDebugger.cpp". GLUT keyboard callbacks are implemented as `KeyPress()`, `KeyRelease()` and `KeySpecial()` whilst `KeyHold()` function is being called in every simulation step. Familiarise yourself with these functions and see how the keyboard handling can be implemented.

In the code, there are also three functions dedicated to user defined keyboard handling including `UserKeyPress()`, `UserKeyRelease()` and `UserKeyHold()` functions. Have a look at the implemented example to see how to interface the keyboard events with the simulation.

1.4 Motors

Some types of joints feature motors which can animate the connected actors. Use the scenario from Task 1.2, switch on the motor by providing the drive velocity value (`RevoluteJoint::DriveVelocity`) and observe behaviour of the actors. You can also switch off the gravity for the actor (`PxActor::setActorFlag(PxActorFlag::eDISABLE_GRAVITY, true)`) to see the action of the motor unaffected by external forces.

Task4U: Implement code that will change direction of the motor speed after pressing a specified key.

Task4U: Implement an additional method in the `RevoluteJoint` class which will set/get value of the gear ratio. Experiment with different values of this parameter.

Task4U: Implement a simple clock consisting of two capsules that represent hour and minute hands. The hands should rotate 60 times faster than in a normal clock.

Task4U: Implement a simple planetary system consisting of three planets (spheres) rotating around the centre. Use revolute joints to achieve that behaviour. Do not consider modelling of gravity forces between the planets.

1.5 Limits

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

Task4U: Switch on the limits by using the `RevoluteJoint::SetLimits` method. Change values of the limit parameters and observe behaviour of the actors.

2 Collisions

2.1 Triggers

Triggers are shapes used in games to cue events when particular objects intersect them. They 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 method (in our case `MySimulationEventCallback::onTrigger`) which can be used to implement the desired response.

Create a box that will act as a trigger object. Then create a sphere and drag it around the scene through the trigger box. Observe the output generated by the trigger report function.

Task4U: Count and display the total number of times the trigger object is hit by the sphere.

Task4U: Modify the trigger report function such the sphere will disappear at the first contact with the plane.

Task4U: Modify the trigger report function such the sphere will bounce quickly off at the first contact with the plane for example by applying a large instantaneous force to the sphere.

Create three different dynamic actors consisting of simple shapes: box, sphere and capsule.

Task4U: Modify the trigger report function to display the type of the object that just got into contact with the trigger object. You can check the type of a particular shape by using `PxShape::getGeometryType()`.

Task4U: Modify the trigger report function so it only responds to collisions with a specific actor. One way of doing this is by setting a unique actor name (`Actor::Name()`) in the init function and then checking against that name in the trigger report (`PxTriggerPair::otherActor->getName()`).

2.2 Dynamic Triggers

To use triggers with dynamic actors, one should add an additional trigger shape to the existing actor which will follow its behaviour. To try out this functionality, use your previous implementation of the `CompundObject` 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.

Task4U: Drag the object around a populated scene and note all the trigger events caused.

2.3 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 our simulations used a 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 pair of matching objects. It is up to the developer to decide which flags need to be raised – they will all trigger a different behaviour.

To define a matching pair, one has to use `Actor::SetupFiltering()`, e.g.: `box->SetupFiltering (FilterGroup::ACTOR0, FilterGroup::ACTOR1)`. The first argument to this function specifies the ID of the actor itself and the second argument the ID of a matching actor (or actors; if there are more, use | operator to combine them). The ID is wrapped up as an enum type which can be

customised to your needs (e.g. different names, more IDs, etc.). To enable collision generation we have to specify the filtering flags for the matched actor as well.

Let us try the new functionality on a scene with two dynamic actors with customised names. To switch on the custom shader do the following:

- replace the default filter shader by the custom one 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.

Task4U: Populate your scene with four dynamic actors. Set the filtering flags such the first actor interacts with the second and fourth, and the third actor interacts only with the third.

2.4 Continuous Collision Detection

Custom shaders also enable the use of so called Continuous Collision Detection (CCD) which helps with missing detections for fast and small objects (e.g. bullets). To enable CCD for a specific actor you have to set the corresponding CCD flags in the following places:

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

You can enable CCD for all other actors by setting their respective flags.

Task4U: Populate your scene with two resting dynamic actors. Set the initial velocity of the first actor such it hits the other object. Next, find out the initial velocity in that direction high enough to cause the first actor to "miss" the second one. Now, switch on CCD and observe the behaviour of the actors.