RESEARCH INTO THE SIMULATION OF SHOCK WAVES

Technical Evaluation:

Unity GP Physics Engine

Authors:
Liam Ireland, Michael Jones, Tomasz Mackow

Implemented Features

Detecting intersections

One of the first problems we had to solve when trying to implement our own physics engine was how to detect if two objects are intersecting. After doing some research with the Unity documentation, we discovered that unity (and most physics engines) uses something called a bounds class. This class would create a box which completely surrounds every point of the object. E.g. if you had a sphere with a radius of 3, the bounds would be a cube with a side length of 6 as to completely encapsulate the sphere. We knew this would be used in the first stage of detecting a collision and instead of rewriting this class we decided to use the pre existing unity bounds class.

We decided it would be simplest to try to first try detect a collision between two cubes and use the bounds class to do this. We discovered that in the unity Physics class there were two functions that would help us achieve this which we had to mimic: CheckBox and OverlapBox. CheckBox simply returns a boolean which tells us if there are any colliders within the input box parameters. OverlapBox works similarly to CheckBox except instead of returning a boolean, it returns an array containing all of the colliders inside the given area. With both of these functions implemented in our own Physics class, we were able to detect intersections between cuboids.

The next, more difficult challenge in detecting intersections was to find out the exact point of intersection. This would be necessary in order to have accurate collisions. We never managed to solve this problem and it was one of the reasons we decided building our own, accurate physics engine was not a realistic goal within our given project time frame.

Forces and Force Modes

Forces were implemented in the API as a method on rigid bodies. The method exerts a force on the rigid body, which accelerates the body with an immediate change to the body’s velocity.

The force’s value is given in one of several units or “Force Modes”, which specify whether or not the rigid body’s mass and the duration of the force are factors in the resultant acceleration. For example, the Force Mode “Impulse” is given in the units Newton-seconds (Ns). As duration (seconds) is a factor in a given Impulse value, duration is not a factor in the resultant acceleration. In contrast, a force with Force

Mode “Force” is given in Newtons (N), so duration is a factor in the resultant acceleration. (Duration of a force in this model is the time between updates).

This model was sufficient in that it allowed various types of force/acceleration to be applied to rigid bodies in a scene, and simulated the effects accurately.

Bounciness

Bounciness is a physical property of a rigidbody. The higher the value the more energy is conserved following a collision; ie: more speed is retained following a collision. We were successful in implementing bounciness to a certain extent. We managed to have spheres bouncing off of each other in different directions based on their value of bounciness. We were able to implement a scale for bounciness where 0 was not bouncy and 1 was most bouncy; all momentum was conserved. With the help of the normal of the collision we were able to calculate the correct angle of reflection.

If time allowed, we would have continued to develop bounciness further, as in we would have considered scenarios such as spheres colliding with boxes or walls.

Dynamic colliders colliding with static colliders

An example of a Dynamic collider colliding with a static collider would be a ball bouncing off a wall. In our scene, the ball would have a rigidbody attached as it would need to be affected by forces whereas the wall would not have a rigidbody as it would be a stationary object in the scene. We did this by checking if the collider each object is intersecting with has a rigidbody. If it did, it was a movable object in the scene and a force should be applied to it.

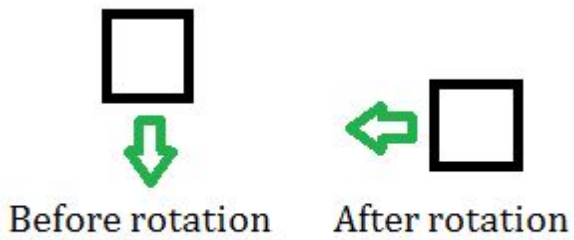
Partially Implemented Features

Torque

Implementing torque was difficult as it relied heavily on other sections of the engine which had not been built yet. As a result of this we hardcoded the rotations which ensured that the scripts were running and were actively changing the position of the object. However although the object was successfully rotated, there was no rotation animation; the object instantly updated its new position. We were not able to complete the implementation of torque due to not being able to resolve the issues encountered within the time constraints of this project.

Constant forces (such as gravity)

We implemented constant forces (such as gravity) initially as a constant negative acceleration in the y axis. This was applied to each object separately. However, there were issues with this such as when an object was rotated, the force of gravity acting upon the object would not resolve correctly. This was because gravity was a vector based on the orientation of the object. After applying a rotation to the object the force of gravity would rotate with the object rather than constantly pulling down. The diagram below shows this effect where the black box is the object and the green arrow is the direction of gravity. The object in the diagram has been rotated 270 degrees clockwise and is in 2d for simplicity.



Resolving collisions

Collisions were resolved for any rigid bodies found to be intersecting during an update. Collisions were resolved by applying forces to the colliding bodies as a function of their momentum and bounciness.

Collisions should be resolved by exerting forces over the course of several updates. However, our implementation did not achieve this properly. If two objects collided

and moved apart, but were still intersecting after the first update, they would effectively collide again during the second update and move towards each other (and so on).

We implemented a workaround for this by only resolving collisions for the first update (and not subsequent updates) that objects were found to be intersecting. However, this meant that collisions were resolved as instantaneous impulses, rather than forces exerted over the course of several updates. This also meant that objects that were not bouncy would stop and remain intersecting forever.

We could not find a collision model that worked correctly for all cases.

Features Not Implemented

If we decided to keep going with implementing our own physics engine, we would have had to consider the following:

- Friction
- Other colliders colliding with each other (mesh, capsule)
- Centre of mass of an object
- Hinges
- Resolving collisions
- Dynamic colliders colliding with static colliders