Report:

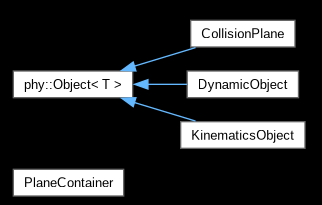
For the Physics for games Assignment 2, we were tasked with creating a small scale physics simulation program using c++. This was to assess my understanding of physics simulation methods and programming skills.

The program uses an OOP design by creating multiple classes, each with their own properties and methods while all inheriting from one parent class, which handles the 3D graphics. The classes are as follows: DynamicObject, KinematicsObject and CollisionPlane.

The DynamicObject class handles the physically simulated objects, objects that can receive impulses, and be moved, blocked and effected by other objects. The class works by using its update function, which checks for collisions and responds to them, uses Euler method to solve the physics step, then sets the physical position and rotation for the 3D graphics to render. Some of these steps are broken down into separate functions to keep the code simpler to read and to follow DRY principles.

The KinematicsObject class handles objects that are physical, and can effect objects from the DynamicObject, but that do not respond to other objects.

The CollisionPlane class works similarly to the KinematicsObject, but handles specifically plane collisions, and span an infinite distance, this is mostly useful for creating a floor to the simulation.



There is also a PlaneContainer struct. This struct is designed to hold a list of all CollisionPlane objects, which the DynamicObject can then access through a pointer to the PlaneContainer.

I keep track of each object in a vector for each class, when I want to operate on these objects I must iterate through the vector, the benefit of having the objects stored in a vector is that I can loop through the vector, operating on each object. I can also create theoretically infinite objects, which can each be operated on, without the need to hard code each individually.

To compute the velocity and position I use Euler method, which is an explicit first-order intergration method which means that local error is proportional to the square of the step size, the global error is proportional to the step size and each step is calculated from the previous step. Euler method does not give the exact solution to the differential equations, rather an approximation. <https://x-engineer.org/euler-integration/>

The impulse responses for sphere to sphere collisions work by first testing if the spheres are colliding, simply by testing their distance from one another, then if the spheres collide, an impulse force will be applied to the sphere in the direction away from other sphere. The force varies depending on factors like elasticity, relative velocity and mass.

For sphere to plane collisions, a similar process occurs. The sphere checks for a collision with the plane. Then if the collision check is successful, the sphere has an impulse force applied to it. This force is different from the sphere to sphere collision however, as the balls direction is mirrored across the planes normal, and if the ball has a low enough velocity compared to its elasticity, the ball will roll across the surface of the plane.

I have also created a system to load in parameters from a text file. The code will go through each line of the config file, adding it to a 2d vector of strings, with each column being a different line, and each row being a different parameter. After getting this 2d vector, I can loop through each line and match the contents of each row to a parameter to initialize an object. The format for the config looks something like this:

DynamicObject, Shape, ColorR, ColorG, ColorB, PositionX, PositionY, PositionZ, VelocityX, VelocityY, VelocityZ, AccelerationX, AccelerationY, AccelerationZ, ScaleX, ScaleY, ScaleZ, Radius, Mass, Elasticity,

KinematicsObject, Shape, ColorR, ColorG, ColorB, PositionX, PositionY, PositionZ, ScaleX, ScaleY, ScaleZ,

CollisionPlane, PositionX, PositionY, PositionZ, NormalX, NormalY, NormalZ,

Some of the strengths of my program are its simplicity of parameters. It’s very easy to customise the simulation and it has many parameters that can be changed which makes it very adaptable. Another benefit is the compartmentalised source-code, meaning future modifications to the engine are much easier to implement, and due to the separation of the graphics and physics, it makes the code much more modular, and I could easily port this to another project and use it. Some weaknesses however are the lack of more complex features such as spatial partitioning. If I were to implement spatial partitioning it would mean that there would be a greater potential performance increases by separating the world into different chunks, limiting the number of collision checks that must be conducted. <https://gameprogrammingpatterns.com/spatial-partition.html>

Another downside would be the lack of any sort of physical constraints. Physical constraints like springs or joints can be very useful when creating more advanced physics simulations that implement things like cloth physics or rope physics, which can be made up of a bunch of joint constraints. <https://web.archive.org/web/20070610223835/http://www.teknikus.dk/tj/gdc2001.htm>

I think in the future I would like to add the ability to load in FBX files as collision meshes, with the ability to reduce their geometry. To implement this feature I would have to find a way to build meshes from individual vertices, then a way to make the object interact with the environment. A simpler compromise would be to create a kinematics body which doesn’t simulate physics from CollisionPlane objects, this would create a mesh that could effect the DynamicObjects still but not require physical simulation.