Physics for Games Assessment *Mara Dusevic*

Brief Description of system

The engine is used to perform multiple collision interactions in a 2D environment. It uses the object's position, velocity, angular velocity, rotation, mass, gravity, moment, and angular or linear drag, to calculate the outcomes of these interactions. This combined helps simulate the proper sequences of events between object collisions. With the physics engine, a remake of the game *Pachinko* was made to showcase all the custom engine's functions. In this game, players will obtain points for collecting balls in a variety of point bins with different score amounts. To challenge players, obstacles are placed between the balls and bins, directing the balls into lower point bins. The obstacles implemented consist of a spinning wheel, bouncy pads, and multiple rows of spheres to block and redirect the ball. The balls are also spawned from a moving platform by the player using the *Space* key and are limited to a select amount. This provides players with a goal in which they must obtain the highest points possible.

Interaction of the physical bodies

Collision Detection

Sphere to Sphere Collision

To determine if the two spheres are colliding, the length between both centre points of the spheres are compared with the sum of each of their radii. If this returns a penetration value greater than zero a collision has occurred.

Sphere to Box Collision

To determine if they have collided, the box will convert the sphere's position to its local space to locate the closest point on the box to the sphere. It then converts that coordinate to the global world coordinates to find the penetration of the sphere into the box. If the value it returns is greater than zero a collision has ensued.

Sphere to Plane Collision

To check for collision, the magnitude of the sphere's position and the planes normal are used to calculate the penetration of the sphere into the plane's normal. The penetration helps find the contact point of the collision and resolve the collision through the sphere's normal collision resolver.

Box to Box Collision

To detect a collision, each box will check if the other's edges overlap with their own. Each edge is extended in each direction of our box's space. If no contacts are found on one or neither axes, no collision has occurred. In the case both horizontal and vertical axes have contacted the other box, it is identified as a collision in which the penetration vector is calculated to be used in resolving the collision.

Box to Plane Collision

To check for collision, the origin of the plane is used to test whether the box's corners have passed the penetrated the plane. This is calculated via the plane's normal and length in which we can determine the plane's origin and whether it is penetrating the box's edges. If one or more points on the box are colliding with the plane are collision has occurred.

Collision Resolution

Plane

To resolve a collision between a plane and another object type, the contact point found when detecting the collision is used to find the local contact point on the plane. The relative velocity of the other object and the object's velocity into the plane is calculated. Afterwards, the torque is found, which applies a force to rotate the object as well as the force considering the object's mass. Then as the plane is a static object the force is only applied to the other object. For triggers as they are static as well, the plane will not collide with the object and not return any value for the trigger. This also applies to kinematic objects in scene.

Rigid body (Sphere & Box)

To resolve a collision between two objects of box and sphere types, the normalised vector between their centres or provided direction of force is calculated to define the normal. This value is then used to determine the velocity of the contact point on each object and check whether they are moving closer. In the case they are, both object's masses are used to calculate how much the objects will move when applied force. As well as the impact of the collision on both objects even with elasticity added. The resolution then checks whether either object is a trigger. If that returns true, no impulse is applied, and the trigger is labelled as 'entered'. Otherwise, the engine calculates an impulse and magnitude to apply to both objects, but in opposite directions. If the object is kinematic, it will apply force to the dynamic rigid-body, but the kinematic object will remain in its current state.

Improvements to the custom physics system

The following improvements could be made to the engine:

- **Object Pooling**

  It intends to improve performance and memory usage in the engine by reusing objects where necessary. It also allows for a faster way of creating large amounts of objects for a scene instead of individually creating and assigning objects with particular variables. However, using object pooling can affect the system's memory if not handled properly. The pool does not deallocate objects will they are no longer in use by the system and remain in its memory. Additionally, when creating pools that store various types of instances or subclasses, enough memory needs to be allocated to each slot to fit the largest possible sized object that could be created. This is done to safeguard memory so any large object added will not affect the next object stored and ruin the system's memory. Still, in doing so it creates another issue regarding memory usage as if no object is called to that size and each is given a larger slot than necessary wasted memory is created.

- **Awake and Sleep States**

  These states are used to divide dynamic objects into two groups to determine whether to ignore or run collision detection. An 'awake' state is placed on an object when moved or accelerated by another object or outside influence. A 'sleeping' state is an object at rest, where it is not being affected by anything. This means at least one awake dynamic object needs to be checked for collisions. This implementation will reduce the number of objects in the scene that need collision detection to run, allowing for a steady frame rate and memory usage. However, this method will only prove to be efficient in improving the system if multiple collisions are taking place in the scene. The finer details regarding collision detection between the two dynamic states would need to be considered on implementation making it a time-consuming task.

- **Quadtrees**

  The data structure is used to divide a given 2D region into multiple manageable parts. It acts as an extended binary tree where it utilised four child nodes rather than two. They start as a single node that objects in the scene are added to. Once more objects are added, it ultimately splits into four sub-nodes where each object is placed inside of based on its position in the space. Any object that does not fit within a node's boundary is placed into the parent node. Even with additional objects in the scene, each sub-node can be divided to accommodate

them. If implemented into the engine, it is no longer required to run expensive detection algorithms to determine whether objects have collided. For example, only objects in the fourth quadrant can collide with those in the same quadrant, ruling out all other objects outside the section.

However, quadtrees are complex to implement properly. Additionally, it becomes more complex with multiple moving objects in each frame as the quadtree needs to subdivide each object quickly and efficiently into free child nodes. To properly implement into the system, would be a time-intensive task.

Third-party libraries

The following libraries were used to create the engine:

- **GLFW** – OpenGL Library
- **GLM** – Maths Library
- **IMGUI** – Debug Overlay
- **STB** – Image Loading

References / Research Material Used

2D Collision Detection - Game Development, MDN, [website], < *https://developer.mozilla.org/en-US/docs/Games/Techniques/2D_collision_detection* > accessed 19 February 2021.

Object Pool – Game Programming Patterns / Optimization Patterns, Bob Nystrom, [website], < *https://gameprogrammingpatterns.com/object-pool.html* > accessed 19 February 2021.

Use Quadtrees to Detect Likely Collisions in 2D Space, Steven Lambert, [website], < *https://gamedevelopment.tutsplus.com/tutorials/quick-tip-use-quadtrees-to-detect-likely-collisions-in-2d-space--gamedev-374* > accessed 6 March 2021.

Quad Tree vs Grid-Based Collision Detection, Game Development Stack Exchange, [website], < *https://gamedev.stackexchange.com/questions/6345/quad-tree-vs-grid-based-collision-detection* > accessed 6 March 2021.

Class Diagram