***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 a goal in which they must obtain the highest points possible.

***Interaction of the physical bodies***

**Collision Detection Utilised by the Engine**

**Sphere to Sphere Collision**

**Sphere to Box Collision**

**Collision Detection Implemented into the Engine**

**Sphere to Plane Collision**

**Box to Plane Collision**

**Box to Box Collision**

**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 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.

**Sphere to Sphere Collision**

This form of collision is detected by checking if either sphere is penetrating the set radius.

**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. This penetration is then used to resolve the collision on the box.

**Box to Plane Collision**

The box determines if the two collide by checking between the spaces around the corners. If the plane is penetrating the box’s local space it will locate each contact point. This allows the plane to resolve the collision and move the box based on its specific attributes.

**Box to Box Collision**

Each box checks their corners for the penetration of the other box in which it can calculate contact points from. This allows the boxes to resolve the collision between the two and generate an outcome.

**Collision Resolution**

**Plane**

As the object has an infinite distance, any object that traverses the plane’s normal will create a point on the plane in which the collision for the other object is determined from.

**Spring**

Springs are flexible connections between two objects to which they apply a restorative force and dampers to. This is calculated through their displacement and velocity and take on each other’s attributes such as velocity to create a bouncy effect.

**Sphere**

To detect if an object has collided with the sphere, it checks within a set radius in which if collided with a 'penetration' of the object is created. This is used to resolve and accurately simulate the collision.

**Box**

Collision detection loops through the gaps between each corner of each box and determines whether it has penetrated that space.

Collision detection implemented into the game

* What it is
* how it works

Collision detection the system can handle

* What it is
* how it works

Collision resolution,

how it works for dynamic and static physics objects

Physics Objects ALL

***Improvements to the custom physics system***

The following improvements could be made to the engine:

* **Object Pooling**

Its intent is 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 particular variables.

* **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 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 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 scene are added to. Once more objects are added, it ultimately splits into four sub-nodes where each object is placed inside of based off its position in the space. Any object that does not fit within a node’s boundary is placed into the parent node. Even with addition 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 each frame as the quadtree needs to subdivide each object quickly and efficiently into free child nodes. To properly implement into the system, it 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

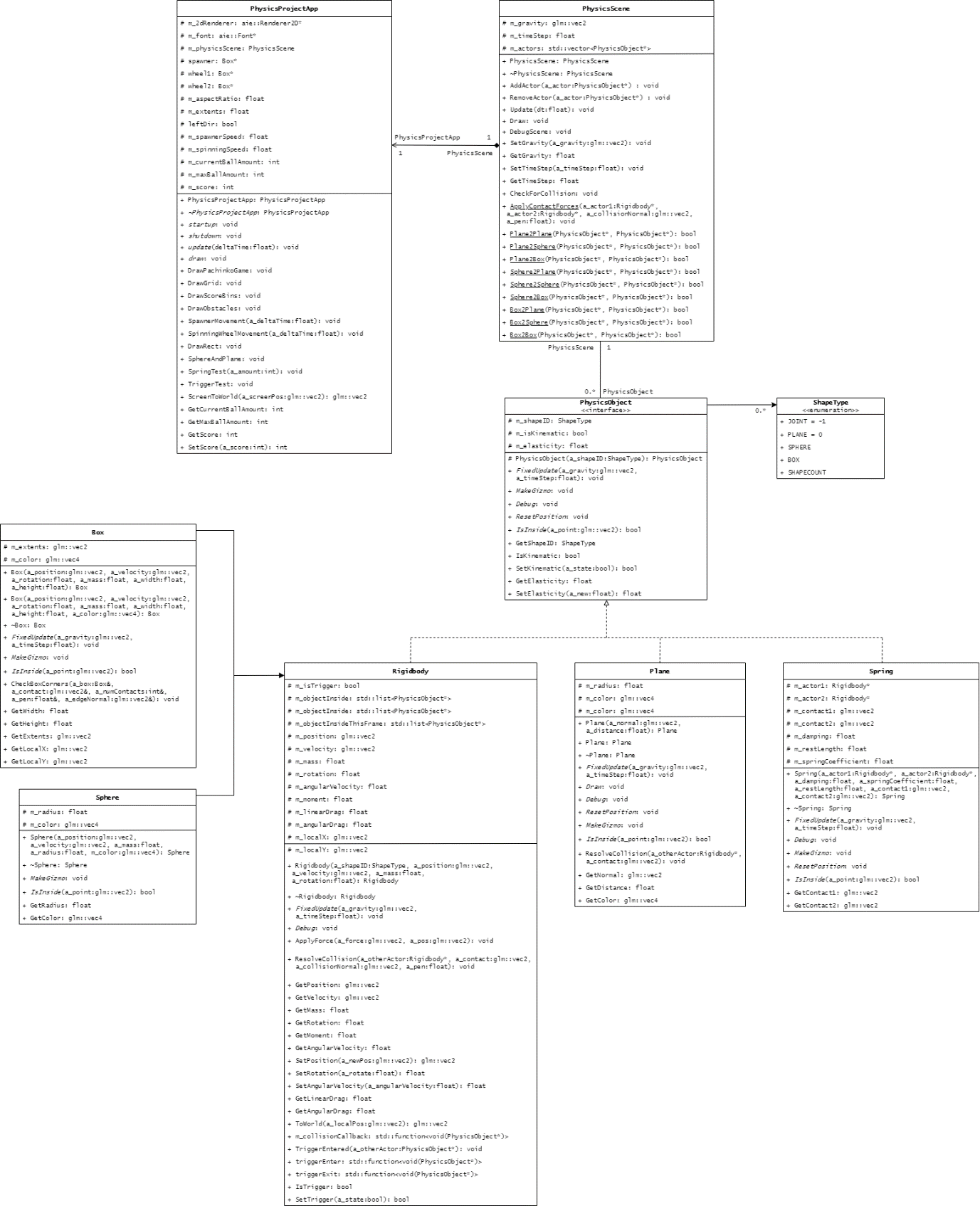
***References / Research Material Used***

*2D Collision Detection - Game Development*, MDN, [website], < [*https://developer.mozilla.org/en-US/docs/Games/Techniques/2D\_collision\_detection*](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*](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*](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*](https://gamedev.stackexchange.com/questions/6345/quad-tree-vs-grid-based-collision-detection) > accessed 6 March 2021.

***Class Diagram***