Game Engine Programming

Introduction

“A game engine is the software that provides game creators with the necessary set of features to build games quickly and efficiently” (Unity, 2018). Game engines allow developers access to reusable components and a range of tools that are used commonly when producing a game. They provide a solid foundation to develop upon with an arsenal of pre-programmed tools ready to use.

The architecture of the game engine will be a Component Entity System. Entities will be added to the scene and act as a repository in which components can be added. Components allow behaviours and data to be added to entities. The game engine will support a 3D environment using the OpenGL framework. Components that the game engine will feature include:

Component	Function
Mesh Renderer **Material**	Load and display a triangulated 3D model from an .obj file. Load in a texture image from a file and apply the texture to the corresponding mesh.
Orthographic	Allows the creation of a graphical user interface, displaying mesh’s in from an orthographic viewpoint.
Transform	Assigned to the entity upon creation. Stores the entities position, rotation and scale.
Keyboard Handler	Monitors keyboard input, checking which keys are being pressed and released.
Mouse Handler	Stores the mouse position and keeps track of any mouse input.
Sound	Loads in sound files allowing them to be played upon request.
Camera	Calculates the view matrix which is used to create the viewpoint in the scene.
Box Collision	AABB bounding box collision used to check if two objects are colliding.
Mesh Collider	Using spatial portioning to divide up a mesh into a number of columns. Sorts and stores the individual faces of the mesh into the corresponding column depending on its

	position. Columns are then used to check if an object is colliding with a section of the mesh.
Button	GUI system that creates a button. The checks if it has been interacted with using the mouse.
Exception Handler	Outputs runtime errors allowing the game to continue running even if a minor error occurs.

Research

The game engine will be programmed using C and C++ as it is a cross-platform language allowing the program to be compiled on many platforms. Furthermore, CMake will also be used in order to control the compilation process allowing the program to be compiled on different operating systems.

One in-built component of the game engine is collision detection. An example is Axis-Aligned Bounding Box (AABB) collision detection. AABB collision occurs "when two AABB bounding box intersect, if and only if they are on the X axis and Y axis and Z axis are overlapping" (Ye, 2017). This is one of the simplest forms of collision detection as it creates a simple geometric shape around the object which is then used as a bounding box. Shapes that are commonly used include cuboid boxes or spheres. See figure 1 for an implementation algorithm. This algorithm is for a 2D environment and checks a rectangular bounding box around the object. To adapt this into a 3D environment an extra comparison will have to be made checking the coordinates of the z axis. This is elaborated by Elfizar (2014).

```
if(object1.x > object2.x
    && object1.x < object2.x + object2.width)
{
    // Collision in the x axis.
    if (object1.y > object2.y
        && object1.y < object2.y + object2.height)
    {
        // Collision in the y axis.
        return true;
    }
}
return false;
```

Figure 1: Algorithmic solution to AABB collision

An advantage of AABB collision is that it is simple to implement and efficient as it does not consider the geometric shape of the object and only uses the objects coordinate positions. Consequently, some issues arise. For example, if an object has a high velocity, it may pass through another without being detected. Another issue is due to the bounding box not considering the shape of the mesh, collision may be detected in an area with no mesh (see figure 2). A solution to this problem would be to use mesh collision.

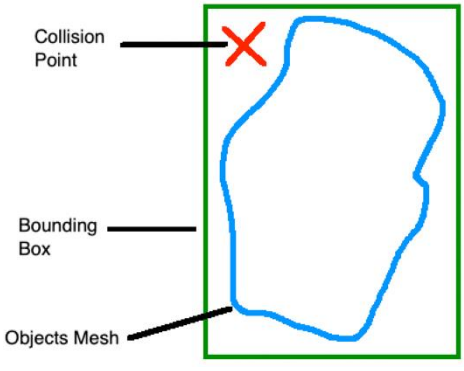

Figure 2: Example of a bounding box problem

3D models are imported into the engine as a triangulated mesh, with each triangle representing a face of the model. This enables the use of

Triangle–Triangle (mesh) collision detection “by testing each triangle of on object against every triangle of the other object. If two triangles, one from each object, intersect we know that the two objects are in collision.” (Bäckman, 2010). The main advantage of this type of collision if that the collision box is wrapped to the model, making sure the object has collided with it. Furthermore, this can also be used to check where the collision point has occurred. If in a shooter game, double damage can be assigned to the critical areas. One implementation type of mesh collision would be to brute force a solution by comparing collision between each face of the objects in question. A problem with this is that it can lead to a large number of comparisons causing an extensive number of calculations. This is processor intensive and can be slow especially if the models are large and are highly complex. A resolution to this problem would be to implement spatial partitioning. As discussed by Madera (2011), “spatial portioning approach divides the scene into regions and tests if objects overlap the same region of space.” In reference to the game engine, the implementation process for this to work will be to split the size of the mesh into a grid consisting of a number of columns. When loading in the mesh, store a list of all the faces of the model into the corresponding column depending on its location. Therefore, when checking if an object has collided with the mesh, the column in question can be located and iterated through. This drastically decreases processing time as the object is only being compared to the faces inside the specific column. One issue that arises with this solution is that the object must be static in order to keep track of the grid’s location. Nonetheless, if this was applied to a map level this would be static anyhow.

Program Design

When developing games many objects require the same behaviours. Instead of duplicating code, inheritance can be used. However, objects that are nothing alike can cause inheritance hierarchies to become increasing complex and not practical for use. A solution to this is a Component Entity System. Entities are added to the scene acting as repositories for the components which hold all the data. Components can be assigned to any entity that requires the specific behaviour. This is greatly advantageous as it removes inheritance from objects out of the equation whilst also removing repeated and redundant code. Instead behaviours all inherit from the component class (see figure 3). Having a large collection of components allows ideas to be quickly prototyped as many complex behaviours are preprogramed and ready for use.

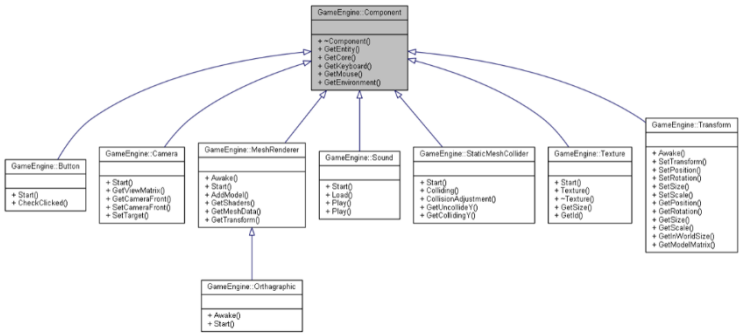

Figure 3:Component Inheritance diagram

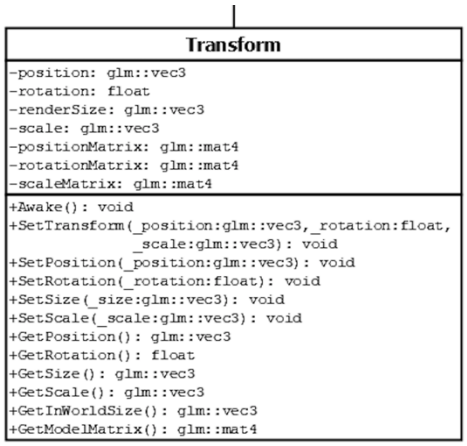

Figure 4: Transform Component

When an entity is created in the scene, a transform component is automatically assigned (see figure 4). This stores and handles the position, rotation and scale of the object. For ease of use the users can set all three of these variables from a single function call, or individually depending on the users need. If a mesh renderer is also attached to the entity, an extra feature provided by the transform component is the capability to get the render size of the object. This works by iterating through all the vertices of the model comparing the values to find the maximum and minimum coordinates. This is extremely useful for the collision components as it provides the correct dimensions for the objects bounding box.

Another design choice added to the game engine is a resource class. This is called when a new resource (mesh, texture or sound) is added to the game. The resource class keeps a list of all the assets loaded into the game. If a new resource is added to an object, the program will check to see if the asset has previously been used. If so, the previously loaded assets data is located and returned (see figure 5). The benefits of this is that it prevents the storage of duplicated data and increases the efficiency of the program as an already used asset doesn't have to be reloaded from its file.

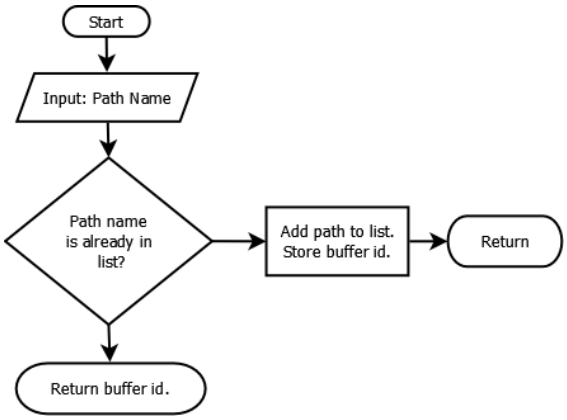

Figure 5: A flowchart showing the resource class process

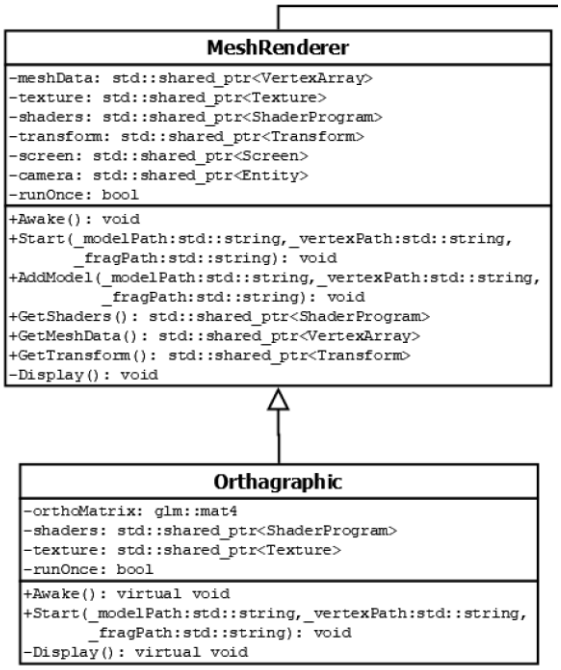

Figure 7: Mesh Renderer and Orthographic Component Classes

Multiple components have been programmed to allow entities to perform a variety of behaviours. An example of this is the mesh renderer (see figure 6). this allows a 3-Dimensional model to be allocated to an entity. When the mesh renderer component is called it takes in parameters for a model file path location, vertex shader and fragment shader. This allows the user the flexibility to add different shaders to their models.

If the user requires a graphical user interface the orthographic component can be added to the entity. This simply provides an orthographic matrix to its own prewritten shaders, in order to display the image in an orthographic viewpoint in front of the camera.

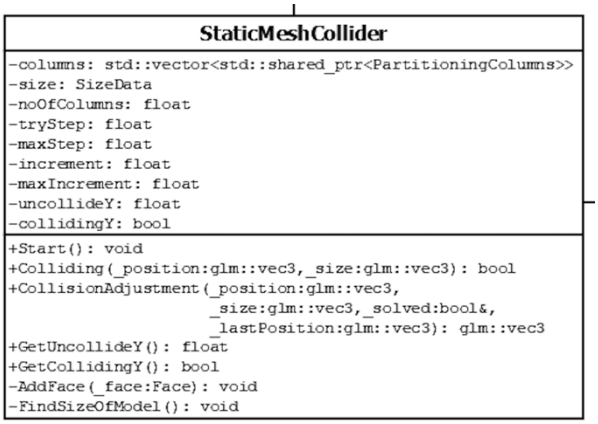

Figure 6: Static Mesh Collider Component Class

A static mesh collider is also provided by the game engine (see figure 7). This component has numerous different features. Firstly, the mesh will be divided up into a number of equal columns. The individual faces of the model are then stored into a corresponding column depending upon their positioning, creating spatial partitioning. This drastically improves efficiency when performing collision checks. If collision with the mesh is detected, the collision adjustment function very slightly adjusts the coordinates of the colliding object until it is no longer in contact with the mesh. The GetCollidingY function is useful for the user as it returns a boolean variable if the object is touching the mesh in the y coordinate i.e. the ground. This can be used to implement gravity into the game as the it can detect if an object is grounded or not.

Below is a test table to determine if the collision detection is working as it should.

Test	Input	Expected Output	Actual Output	Resolution
Check if the box collision is working.	Move the player into a star.	Star to disappear upon collision.	Star disappears upon collision.	n/a
Check if the player collides with the level mesh.	Walk towards a wall of the house.	To be pushed back slightly instead of walking through the wall.	Player can't keep moving through the wall.	n/a
Check is the player stop falling when	Make the player jump.	The player to stop moving	Player stops moving downward	n/a

colliding with the ground.		downwards when the floor is hit.	when the ground is reached.	

See Figure 8 and 9 for a class diagram showing the entirety of the game engine as well as the game demonstration.

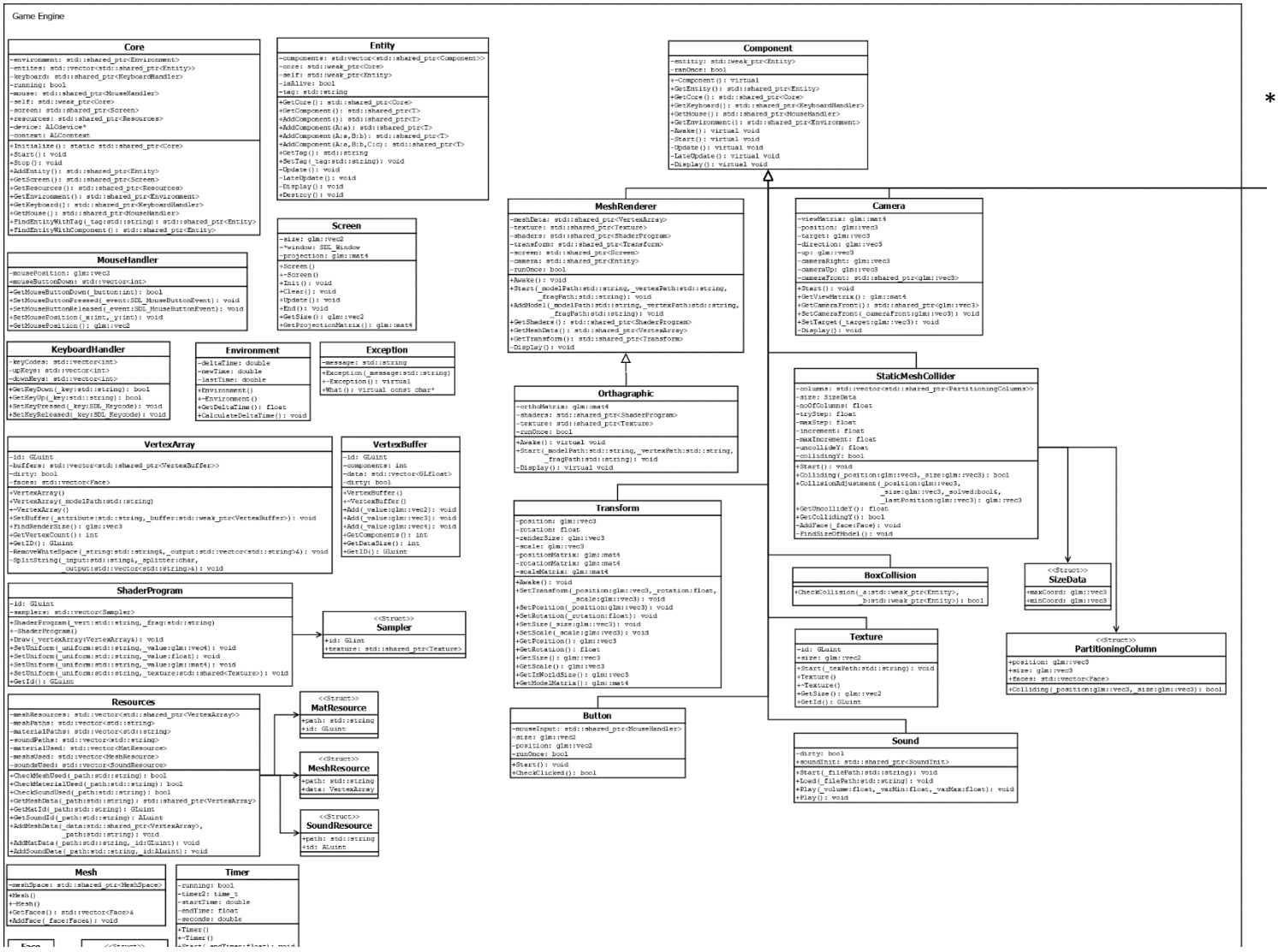

Figure 8: A Class Diagram of the Game Engine - Part 1

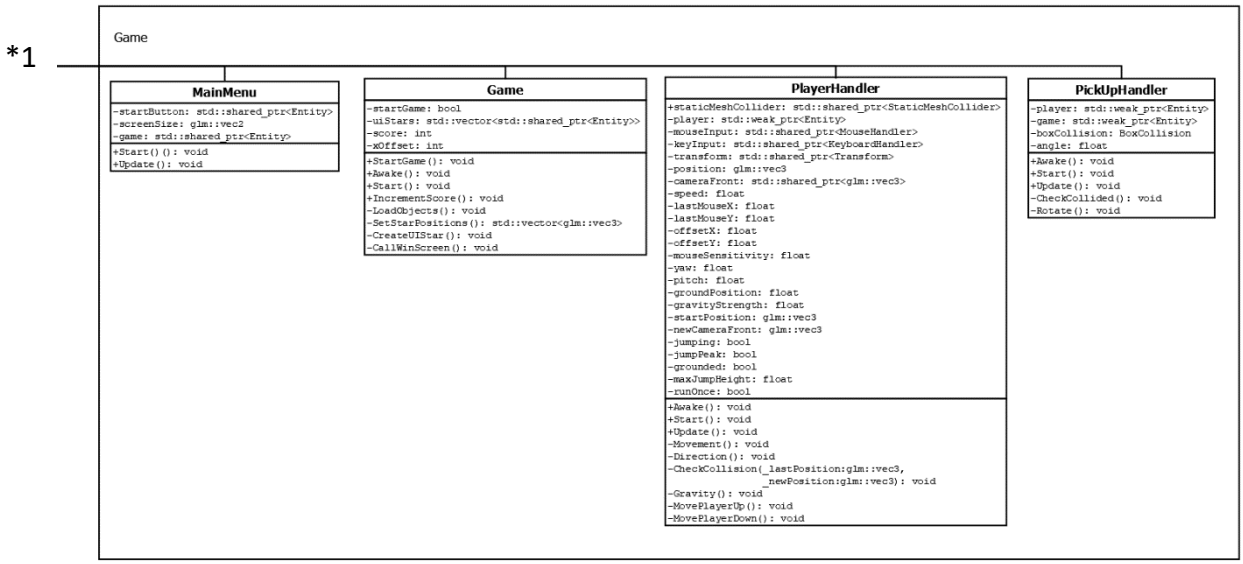

Figure 9: A Class Diagram of the Game Engine - Part 2

Analysis and conclusion

One strength of the game engine is the in-built mesh collision detection. Not only does this provide collision between the player and the level model but also adds spatial partitioning making this feature efficient. It effortlessly provides the user a quick and easy way to start prototyping and testing their level designs with full collision detection.

An additional strength of the engine is how runtime errors are handled with the exception handler. Instead of the program crashing if an error occurs, such as a missing model, the error is simply outputted to the console allowing the game to continue running without the player or game being disrupted.

A weakness of the project is the absence of lighting features built into the engine. Lighting is a commonly used feature in many games. Currently, the engine does not include any lighting features. However, in the future ambient, diffuse and specular lighting should all be considered and allowed to be modified by the user. The lighting calculations would be implemented into the fragment shader modifying the colour of the fragment before it is outputted to the screen. This would then enable the user to add different types of lights to the scene, such as directional light, spotlight and point light.

In the future a component that would be implemented is AI pathfinding. This is a common feature used for objects such as enemies allowing them to follow the player or wander the level to a certain destination. To implement this feature, a NavMesh system would have to be put into place determining which faces are walkable at ground level. A pathfinding algorithm such as A* can then be applied allowing the agent to navigate its way around the level or to the player.

Features built into the game engine include:

- Component Entity System
- Screen Handler
- Environment Handler
- Exception Handler
- Resource Handler
- Keyboard Handler
- Mouse Handler
- Mesh Renderer
- Orthographic Renderer
- Texture Renderer
- Shader Program
- Timer
- Camera component
- Transform component
- Button component
- Box Collision
- Static Mesh Collider
- Sound component

References

Bäckman, N., 2010. Collision Detection of Triangle Meshes using GPU [online]. Umea: Umea University.

Elfizar., Sukamto., 2014. Analysis of Axis Aligned Bounding Box in Distributed Virtual Environment. International Journal of Computer Applications [online], 105.

Madera., F., 2011. An Introduction to the Collision Detection Algorithms [online]. Merida: UADY University.

Unity., 2018. Unity. Available from: https://unity3d.com/what-is-a-game-engine [Accessed 20 November 2018].

Ye, L., 2017. Research on Collision Detection in 3D Games. Advances in Computer Science research, 73.

Assets:

Modernoise., 2007. HOUSE [3D Model]. Available from: https://www.turbosquid.com/FullPreview/Index.cfm/ID/352024 [Accessed 15 November 2018].

Butler., B. 2012. Star [3D Model]. Available from: https://www.turbosquid.com/FullPreview/Index.cfm/ID/664920 [Accessed 17 November 2018].

Tissot., B. ca. 2018. Ukulele [Music]. Available from: https://www.bensound.com/royalty-free-music/track/ukulele [Accessed 21 November 2018].