Sol Game Engine

Simon O'Neill

simon.oneill1@student.dit.ie

Supervisor: Bryan Duggan

2nd Reader: John Kenny

March 26, 2015

This Report is submitted in partial fulfillment of the requirements for the award of Bachelor of Science (Hons) in Computing of the School of Computing, College of Sciences and Health, Dublin Institute of Technology.

Abstract

This report describes the research, design and development that goes into the creation of a game engine. The development of such a piece of software is not simple and there are many possibilities and complexities involved. The system described in this report implements such features as an entity-component system, a complete rendering engine using OpenGL and GLSL shaders, a physics engines with simple colliders and rigid bodies and built in support for Lua scripting. All these features are written from scratch using C and C++ to ensure efficiency, speed and complete control of computer memory. Although this report features some select code snippets from the system, the entire project is available freely at

`https://github.com/oneillsimon/GameEngine`

Acknowledgements

I would like to thank the following people for all their help in the process of completing this project.

Damien Bourke
For all the organisational help with the project, interim report and this report.

Bryan Duggan
For being a helpful and insightful supervisor as well as a fellow game engine enthusiast.

Chris Gregan
For taking the time to talk to me about my project and the development industry itself.

Declaration

I **Simon O'Neill** hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed _____
Simon O'Neill

Witnessed _____
Bryan Duggan

Contents

List of Figures

Chapter 1

Introduction

I chose to develop a game engine for my final year project. I chose this topic because I have a great interest in the systems that systems are made of. Rather than making a game, I thought that making the software that games are made of would be more interesting and rewarding. Some students chose to use technologies that they had experience with and develop systems familiar to them. I chose to develop a game engine precisely because I knew next to nothing about them or the technologies used to create them. I saw this project as a great opportunity for an invaluable learning experience regardless of how functional the end result would be. Overall a curiosity of how it all works is why I chose to develop a game engine from scratch.

At the core of it all a game engine is just numbers. As such a big part of the system is the implementation of a custom maths library, consisting of classes that out line vectors, quaternions and matrices. This library is defined by the core engine. Since the majority of classes need access to the mathematical functions and constructs all the other engines connect to the core engine. Since this engine acts as a common ground it also contains the definitions for more contextually important classes such as the Game Object class and the Game Component class. These classes also illustrate that this system implements a custom ECS (albeit heavily modeled after Unity's implementation). Composition means that this system in compliant to contemporary models and therefore be picked up with relative ease by someone already familiar with the concepts of game engines. Since it also needs access to core definitions, the Lua scripting system is also written into the core engine. This scripting sub-engine is designed to treat Lua scripts as Game Components so that they tie into the ECS nicely. Lua scripting is supported in a "write and forget" fashion, that is to say all the hard work of converting it to a Game Component is done internally so that programmers familiar with Lua can just write the scripts, call the function to load them and the system will worry about attaching and executing them.

The rendering engine is the most complete, and in my opinion the most impressive engine in the system. Using OpenGL and GLSL shader programs, the rendering engine is capable of drawing meshes in 3D space, rendering textures, lighting and shadow mapping. Meshes

can either be specified by their vertices in code or loaded from '.obj' files. Textures can then be loaded alongside corresponding normal and displacement maps to give the impression of a higher polygon count. The rendering engine also uses custom lighting shaders, more specifically shaders that render directional lights, spot lights and point lights. Each of these lighting shaders have wrapper classes in C++ meaning that the user can develop and change the lighting properties such as colour, intensity and attenuation through C++ rather than mess around with GLSL.

The physics engine is, like the other components, implemented entirely in house. As such it might not be as fleshed out or indeed as functional as a more established library, however I think at the very least, it is a good proof of concept. The physics engine uses colliders and rigid bodies to simulate collision and motion in world space. The current implementation only supports 3 colliders: spheres, boxes and planes. While not an extensive list it serves the purpose of very basic collision testing. Rigid bodies are the components that specify an object's physical properties such as mass, inertia tensor, damping, friction and restitution. For optimisation a spatial partitioning algorithm is also used on world space. This system uses Octrees to reduce the number of checks the physics engine must complete each frame.

Unfortunately, due to time constraints tied with the massive scale of this project I never managed to create an engine to play audio. It will be something to look into in future iterations, beyond the scope of this project.

Chapter 2

Existing Systems

2.1 Introduction

There are many Game Engines is use today. Two of the most prolific are Unity3D and Unreal, each engine has their own strengths and weaknesses that this chapter will examine.

2.2 Unity3D

Unity3D, commonly referred to as just Unity, is a game engine developed by Unity technologies. First seen in 2005 at Apple's Worldwide Developer's conference, Unity is an engine that focuses on portability and simplicity. Although it was initially developed exclusively for Mac OS it has since been extended to target more than 17 different platforms. [2]

Developing in Unity is aimed to be simple. Unity employs an Entity-Component based System (ECS) in its purest form. Everything is either a Game Object or a Game Component. Each Game Object consists of a transform, a list of all child Game Objects and a list of all its Game Components. The transform member has 3 elements: a position vector, an orientation quaternion and a scale vector. The list of child Game Objects, often just called children, are simply Game Objects that move, scale and rotate relative to their parent. The list of Game Components are what defines the functionality of a Game Object. This functionality can be anything from how the object is drawn, how it moves or it behaves during run time.

2.3 Unreal

Unreal Engine is a game engine created by Epic Games, named after its first showcased game Unreal in 1998. Written in C++, Unreal Engine is often seen as more low level the Unity and has a more established presence in the industry. [19]

The design pattern of Unreal is similar to Unity in that is uses an ECS, however it is implemented slightly differently. There are 3 main object types in Unreal: Objects, Actors and Components. Objects are instances of classes that inherit from UObject, the base class of all classes. In truth both Actors and Components are instances of Objects but are more specialised. Actors can be thought of as game play objects with Components being the objects that further describes the Actor's and Object's behaviour. An example that Unreal give in their documentation is that of a car. The car as a whole is the Actor, whereas the parts such as the wheels and doors are components. [11]

2.4 Conclusion

This chapter looked at two of the biggest players in the industry. These systems will be used as the inspiration for my design and it is from these I will derive my system's requirements.

Chapter 3

Technologies Researched

3.1 Introduction

This system will make use of various technologies. Some of them will be more important than others but the end result will be a sum of all parts. Regardless of how big the role of the technology is, each one adds important functionality to the system, increasing its applications and robustness. This chapter will summarise the technologies researched during development. It will also examine the functional and non-functional requirements of the proposed end system as a whole.

3.2 Technologies

3.2.1 C++

The system will be written in the C++ Language. This is the language the vast majority of Game Engines are written in. The main reasons for choosing C++ are optimisation and OOP. With C++ you can write programs at a high level without sacrificing low level control. When it comes to Game Engines memory management is vital as many games are aimed at consoles, which have a finite amount of memory. With garbage collectors, as featured in Java and C# there is no way of knowing accurately how much memory you are consuming at any given moment, you also have no control over when the garbage collector is called potentially slowing down some of your threads. This can become most apparent on systems that use and require a fixed time step.

OOP is also critical to good Game Design. With OOP we can write hierarchical classes and use Polymorphism to make classes of similar types and behaviour. For example we can describe

many classes of type Game Component that share a similar interface of member data and functionality.

3.2.2 OpenGL

The system will use OpenGL (Open Graphics Library) as the API for rendering graphics to the screen. OpenGL was first released in 1992 by Silicon Graphics Inc. [16] and was and still is widely used in CAD, simulations and video game software applications. Today OpenGL is managed by Khronos Group, a non-profit technology organisation. As OpenGL is free and open source software it has wide support among many platforms meaning that a rendering engine that employs it will be more portable than one using DirectX. OpenGL is an open standard meaning that it's quick to adopt any new features that graphics cards use, OpenGL has better compatibility with NVIDIA's SLI for example [18]. OpenGL is also more lightweight than DirectX, which has modules for input and sound on top of rendering. Since OpenGL is also aimed at low level calls it fits well with the low level access C++ needs for allocating and freeing memory.

3.2.3 GLSL

OpenGL Shading Language or GLSL, is a high-level shading language created by the OpenGL Architecture Review Board (ARB) as an alternative to using ARB assembly language. It was developed with a C like syntax to remain familiar with developers and comes with built in functions for common math and shader operations. Shaders allow a programmer to have greater control and flexibility on operations performed on the graphics pipeline. Programming at this level is usually achieved by the use of Fragment Shaders and Vertex Shaders. As of GLSL version 3.3, GLSL has been developed alongside OpenGL with a new release to mirror OpenGL's new features [13]. There are many benefits to writing shaders in GLSL including:

- Cross platform compatibility – if a system can use OpenGL it can use GLSL.
- The ability to customise how rendering is done on any graphics card that supports GLSL.

Shader programs can be generated at run time or can be read in from an existing text file. Either way they are passed to the graphics card driver as a set of plain text strings, where they are compiled and executed.

3.2.4 Lua

The system will also support scripting via the Lua language. Lua is a fast, lightweight and embeddable scripting language, used in many games. Lua is a simple procedural language,

dynamically typed it runs by interpreting byte code for a register based virtual machine. There are several reasons to use Lua, including:

- Lua is fast, several benchmarks show Lua as the fastest interpreted scripting language [10].
- Portability, Lua will compile on anything that has a standard C compiler. It can be run on any OS, even on embedded microprocessors.
- Size, Lua is small, the uncompressed source code and documentation only comes in at 960KB, meaning very little overhead for a Game Engine [15].

3.2.5 LuaBridge

To expose my C++ code to the Lua run time I used a header only library called LuaBridge. Using this I can call C++ objects and functions in Lua.

3.2.6 OpenAL

Originally developed in 2000 by Loki Software, the Open Audio Library (OpenAL) is a cross platform audio API. It is designed for efficient playback of three dimensional positional Audio. OpenAL's programming interface follows the conventions used by OpenGL. The system will implement an Audio Engine using OpenAL to add three dimensional sound to a game.

3.2.7 Assimp

Open Asset Import Library (Assimp for short) is a header only library that reads in many 3D object files in a uniform manner. I simply use this library to read in vertices and texture indices, before mapping these values to my own internal vertex and index objects. In essence it is a powerful string parser and I felt that developing my own would be too time consuming.

3.2.8 SDL

Simple DirectMedia Layer, is a cross platform library that gives low level aspects to things like audio, keyboard and graphics. In this system SDL is used to create the OpenGL context and read input from the mouse and keyboard.

3.2.9 GLEW

The OpenGL Extension Wrangler. GLEW is simply a C based API for calling OpenGL functions.

3.3 Requirements

Based on the research in technologies and existing systems, the system I propose will mimic the design structure of Unity. The system will feature an ECS, having Game Components define the function of Game Objects. The system will also support Lua scripting to provide an pseudo high level coding environment.

The system itself will be modularised to prevent interdependency of components. The system will consist of 3 specialised engines, one for rendering, one for physics and one for audio. To tie this engines together the system will have a core engine. This core engine is the backbone of the system as a whole. It will contain all the implementations for the mathematics behind game engines as well as provide an interface through which the other engines can communicate, if needed.

Below I have listed the requirements for the core, rendering, physics and audio engines.

3.3.1 Functional Requirements

Core Engine

- Will implement a vector class for 2D and 3D vectors.
- Will implement common math functions for the vector classes.
- Will implement a quaternion class for handling orientation in 3D space.
- Will implement common math functions for the quaternion class.
- Will implement matrix classes and the common math functions associated with them.
- Will implement a transform class (position, orientation and scale).
- Will implement an Entity Component System.
- Will support the execution of Lua scripts.

Rendering Engine

- Will be able to import and render 3D mesh files.
- Will have support for customer GLSL shader programs.
- Will implement the Phong Lighting Model, using directional, point and spot lighting.

- Will implement shadow mapping for dynamic shadows.
- Will implement rendering textures to 3D meshes.
- Will implement displacement mapping.
- Will implement normal mapping.
- Will implement Multi-Sample Anti-Aliasing (MSAA).
- Will implement Fast Approximate Anti-Aliasing (FXAA).

Physics Engine

- Will implement basic colliders (AABBS, Spheres).
- Will implement rigid body physics.
- Will implement a quad tree for faster integration.
- Will implement Hamiltonian physics for realistic movement and rotation of objects.

Audio Engine

- Will be able import common audio files (.mp3, .wav).
- Will be able to simulate positional sound in 3D space.

3.3.2 Non-Functional Requirements

Core Engine

- Create Vector, Quaternion and Matrix objects.
- Create Game Objects and add them to the world.
- Create custom Game Components.
- Attach Game Components to Game Objects.
- Parent Game Objects with child Objects.
- Load and run arbitrary Lua scripts.
- Interact with system via keyboard and mouse input.

Rendering Engine

- Load custom shader programs.
- Load customer 3D mesh files.
- Load customer textures, along with normal maps and displacement maps.

- Pass custom / predefined colours to shader programs.
- Define material objects, with specular properties.
- Create light objects and define attenuation properties.

Physics Engine

- Create rigid body objects.
- Attach colliders to rigid body objects.
- Apply arbitrary forces to rigid bodies.

Audio Engine

- Playback sound files.
- Simulate sound occurring in 3D space.

3.4 Conclusion

This chapter focuses on laying out the specifics of what the system will attempt to accomplish and the technologies used to achieve them. The requirements for each engine has been laid out separately and the next chapter will focus on how these engines will be designed and how they will meet their requirements.

Chapter 4

Design

4.1 Introduction

This chapter looks at the design of the rendering, physics, audio and core engines, it also examines the methodology used and why it was used. However before we discuss those there are some important aspects about the design philosophy behind the proposed system, and more critically about games using ECSs in general.

4.2 Inheritance vs Composition

4.2.1 Inheritance

Prior to the ECS becoming an established norm the most common design implemented was the Inheritance Model (IM). The IM was a pattern that consisted of a base Game Object from which every other object extended. While this is a logical approach, particularly for programmers, it does have some limitations.

Let's say for example you have a class `Game Object` , extending this you have classes called `Player` , `Car` and `Enemy` .

What happens if we want objects of type `Car` to be playable sometimes? We could make car extend `Player` .

But suppose we want objects of type `Car` to be enemies as well. We could make `Car` extend both `Player` and `Enemy` , if the development language supports it. Another approach is to make another two classes `FriendlyCar` and `EnemyCar` which both extend `Car` . Then have `Player` and `Enemy` extend `FriendlyCar` and `EnemyCar` respectively.

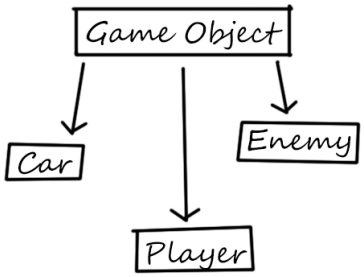

Figure 4.1: Simple Inheritance Model.

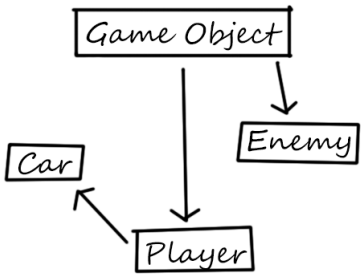

Figure 4.2: Alternative Inheritance Model.

This isn't a very neat solution and it's extremely clear if a system as small as the suggested one becomes so messy which just a few classes, a system with many classes will be practically unreadable.

An IM can also lead to unintended consequences and behaviour. When a virtual function is called recursively through the class hierarchy, it's common to lose track of what action is being performed for what classes. This can lead to strange and unwanted behaviour. For example, a developer for Age of Empires II, Herb Marselas, describes that because an IM was used *"functionality can be added or changed in a single place in the code to affect many different*

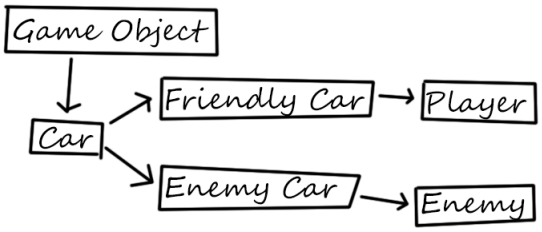

Figure 4.3: A more complex Inheritance Model.

game units. One such change inadvertently added line of sight checking for trees" [20] which caused massive performance problems.

The more and more inheritance is used, the more and more difficult it becomes to comprehend the system. To understand one class you will need to look at its base class, and its base class's base class etc until you finally get to the absolute base class. This is inelegant, unintuitive and can simply lead to human errors occurring during development.

4.2.2 Composition

It is uncommon for games nowadays to use this deep IM. Instead the new standard is fast becoming ECSs, which describes Game Objects as aggregations of Game Components. The ECS approach describes the separation of functionality into single independent components. An Entity (Game Object) is then simply a collection of these components, a sum of all its parts [4]. With this approach components are completely independent, which means portability and reuse. Every Game Object also contains a list of Game Objects known as children objects. Every game has one Game Object (usually known as the `rootObject`, to which every other Game Object is attached. This means that when the `rootObject` updates, its children update and their children update etc. When we discussed IMs, a simple system of a `GameObject`, `Player`, `Enemy`, `Car`, `FriendlyCar` and `EnemyCar` was suggested. The equivalent system using an ECS might look like:

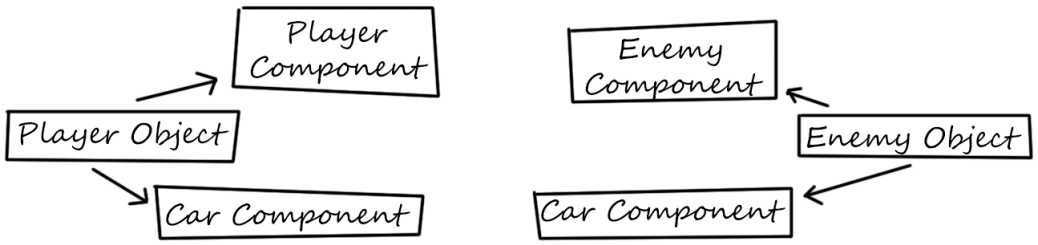

Figure 4.4: Composition.

4.3 The Engines

4.3.1 The Rendering Engine

The rendering engine is in charge of everything relating to graphics and drawing. The primary function of the rendering engine is to draw to the screen. This is done by making calls to

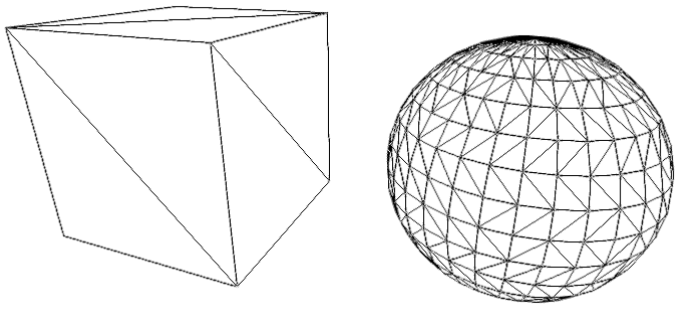

Figure 4.5: A cube and sphere mesh rendered in the system.

the graphics card via OpenGL functions. OpenGL is managed by the GLEW library, which wraps OpenGL calls into C functions. OpenGL itself executes small programs called shaders to determine pixel colour at any point on the screen. The rendering engine tells OpenGL what shaders to use by calling them in a shader pair. A shader pair is a vertex shader (a shader which can manipulate the vertices of polygons) and a fragment shader (sometimes called pixel shaders, fragment shaders determine the colour of individual pixels). The rendering engine loads these shaders, assigns values to any variables, where applicable and then passes these to the OpenGL graphics pipeline [17].

The rendering engine also takes care of loading meshes. Meshes are a collection of vertices (indexed points in 3D space). Meshes can either be created during run time or loaded in from files (currently only ".obj" files are supported). To draw a mesh, OpenGL splits it into polygons (geometric primitives). OpenGL supports a variety of geometric primitives, e.g. Quads, Triangles and Stripes being the most basic. Triangles are the most common polygon used, simply because they are the 2D poylgon with the lowest amount of vertices. More complex polygons can also be broken down into a series of triangles [9].

Textures are also a key point for the rendering engine. Textures, much like meshes, can be loaded in from a file or created at run time. Textures are used in a variety of rendering functions e.g. lighting and shadow mapping. We will look at these in more detail in Chapter 5, along with other, more complex rendering tasks like Multi-Sampling Anti-Aliasing (MSAA) and Fast Approximate Anti-Aliasing (FXAA).

4.3.2 The Physics Engine

The Physics Engine contains references to any Game Object with a Physics Component attached. In this system, Physics Components consist of a collider and a rigid body. Rigid bodies cause the object to become subject to the laws of motion. This includes linear and angular acceleration and damping, friction and restitution [12].

Colliders are objects that detect collision between other colliders and create contact data based on the collider's properties and rigid body data.

When a collision is detected the physics engine will generate this contact data and attempt to resolve it in a minimum number of passes. This is done by moving the object appropriately until it is no longer collision. How it is moved is determined by the rigid body and collider.

The physics engine will detect collisions using a combination of broad phase collision detection and narrow phase collision detection. Broad phase collision detection in the process of finding objects that are likely to be in collision. When these objects are found narrow phase collision detection is performed which is where the real collision resolution is done [12]. This system will use Octrees for broad phase collision detection. For more detail on what these are and how they work see their section in Chapter 5.

Unlike its rendering counterpart, the physics engine will use no 3rd party library and will be implemented entirely in the system.

4.3.3 The Audio Engine

The audio engine is the simplest engine in the system by far. Due to its relatively low complexity it is low priority. The engine will make use of OpenAL to read in audio files and play them during run time. OpenAL is OpenGL's audio counterpart, as such I would be familiar with the API and structure of the library. The audio engine is in charge of playing audio normally and mimicking it in 3D space using panning in stereo sound. Other features of the engine include amplification and playing sounds generated by the system.

Unfortunately due to time constraints I have decided that the audio engine is beyond the scope of this project. Since it is so much more simpler than the rendering and physics engine I have decided it is not worth the time to develop right now, although in the future it is something I will endeavour to implement.

4.3.4 The Core Engine

The core engine is the spine of the whole system. It contains the definitions for Game Objects and Game Components, as well as the common math constructs the other engines need; vectors, quaternions and matrices for example. Lua script generation is also done by the core engine, since Lua scripts are treated as Game Components there was no need to write a separate scripting engine.

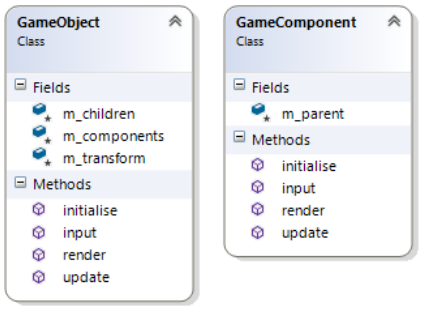

Figure 4.6: The Game Object and Game Component class diagrams.

4.3.4.1 The Game Object and Game Component

Game Objects are comprised of 3 notable member variables: a transform object (position, rotation and scale), a list of children Game Objects and a list of Game Components. Game Objects also have 4 important functions, initialise, input, update and render.

Comparatively, Game components have one key member variable: a pointer back to the Game Object they are attached to called their parent object. Similarly, they have the same 4 functions as Game Objects: initialise, input, update and render.

The structure of these two classes is not coincidental. This is the implementation of an ECS. For example, when the Game Object's update function is called,it calls update in its children and its components. This means that the update behaviour of a Game Object can be customised by creating a Game Component and writing the desired update behaviour.

This system is also designed with a parent - child system. This means the transform member of any Game Object is updated relative to its parent's transform [4]. It is important to note that a Game Object has no limit to the number of children it has, but a Game Object can only ever have 1 parent.

Indeed, the parent - child system is an important part of the simplicity. The system is designed in such a way that every game has an implicit Game Object member known as the root object. This is technically the only Game Object the core engine cares about. When a new Game Object is added to the game world, it is actually added as a child of the root object. So when the core engine calls the root object's member functions, the root object calls the respective function on its children recursively down the parent - child structure.

4.3.4.2 The Mathematics

At the end of the day a game engine is all about the maths. In this system I have implemented by own maths library. This section will look at the different mathematical elements used in this

system.

4.3.4.2.1 Vectors

A vector is simply an array of numbers of N length [7]. When referring to vectors in this system the notation vectorN will be used where N is the number of elements in the vector. A vector2 has 2 elements, a vector3 has 3 etc.

Vector3s are the most common type of vector used in this system. Most prominently they are used to store an objects position in 3D space since we can map an object's coordinates to the X, Y and Z values. In a 2D system we could simply use a vector2. Vector3s are also used to store an objects scale along each axis and are used in numerous parts throughout the entire system; for calculating distances, directions, velocities and forces for example.

Vector2s are rarely used in the system outside the rendering engine. The rendering engine uses vector2s to map texture coordinates to meshes and to calculate aliasing on shadows and edges.

The system also supports vector4s, however these are used exclusively as a base class for the colour class since we can map RGBA to XYZW. This spares us the time of rewriting a separate colour class that will inevitably share common functionality with the vector class. The only difference is that a colour class vector has a length of 255. This is to avoid issues with normalising the vector since OpenGL shaders take RGBA values between 0 and 1. For instance, suppose we take the colour yellow RGB = (255, 255, 0). If we calculate the length using the formula

$$L = \sqrt{x * x + y * y + z * z}$$

We get $L = 360.62$. So the normalised vector Vn, determined by

$$Vn = (255/L, 255/L, 0/L)$$
$$Vn = (0.707, 0.707, 0)$$

will not represent true yellow in OpenGL. However if we say $L = 255$ then Vn becomes

$$Vn = (1, 1, 0)$$

which does represent yellow in OpenGL.

4.3.4.2.2 Quaternions

Quaternions provide a mathematical structure for storing orientation in 3D space. They are preferred to storing the orientation as a vector3 since they don't suffer from the problems of Gimbal lock and axis order [7].

The theory behind quaternions is very complex and as such won't be discussed in great detail here.

Quaternions can be defined as

$$Q = (xi + yi + ji + w)$$

where i is such that $i^2 = -1$,, x, y and z are therefore imaginary numbers that represent the Cartesian axes of rotation. w represents $cos(\theta/2)$ where θ is equal to the rotation amount in radians [7].

The implementation in this system supports important quaternion functionality such as calculating the conjugate and linear and spherical interpolation.

4.3.4.2.3 Matrices

This system implements two types of matrices a 4 by 4 matrix (matrix4) and a 3 by 3 matrix (matrix3). Matrix4s are used to encapsulate a Game Objects transform and represent it in world space. Every transform can be represented by a matrix4 known as the transformation matrix [12].

Matrix3s are used to store an Game Object's inertia tensor in the context of physics. We'll talk more about what the inertia tensor is in Chapter 5. For now, a simple explanation is that it describes the rotational shape of a Game Object.

4.3.4.3 Lua Scripting

This system is designed to treat Lua scripts as Game Components. Therefore, each Lua script created for this system follows the same internal structure as a Game Component, meaning each script has access to its parent's transform object and has the appropriate initialise, input, update and render functions.

The system uses LuaBridge, a header only library, to expose C++ code to the Lua run time.

Since this system treats Lua scripts like Game Components, the update and render function is called every frame. It turns out that Lua scripts weren't really designed for this. There was also some issues with mapping one transform object to multiple scripts. I addressed these problems by designing an algorithm that generates Lua code from multiple sources. For a more detailed look refer to the Lua section in Chapter 5.

4.4 Methodology

Due to the vast amount of research I need to complete and all the technologies I need to investigate it isn't feasible to use too structured a methodology for development. Development methodologies such as the Waterfall Model require too much prior knowledge of how the system will be implemented. As a result I have chosen to follow an Iterative Development Model.

This model will mean I can begin development by specifying and implementing just part of the software, and when that part is implemented and tested it can be used to identify further parts. This process will be repeated and at the end of every cycle I will have a system that becomes closer and closer to the completed system.

An Iterative Development Model is very applicable to this project. The proposed system is big, with many complex components and it would be difficult to fully account for everything in planning. However the overall requirements of the complete system are known and can be compared to existing similar systems. While the requirements are clear, many of the steps necessary to complete them are not. Many technologies will have to be learned and tested, but an Iterative Development Model allows for this, whereas it might interfere and hinder development following a more strict development model. An iterative model is also useful as design problems, bugs or defects can be determined very early in development as the system is implemented step by step. This means that the system can adapt to problems as it grows and often won't require any major overhaul of design that a system following a stricter plan might.

4.5 Conclusion

This chapter took a closer look at how the engines in the system are designed. An overview of the rendering engine and physics engine is given, and the aspects of the core engine; the ECS, maths and Lua, are examined. The next chapter will talk about how these aspects are implemented in each engine in the system.

Chapter 5

Implementation

5.1 Introduction

This chapter focuses on how I implemented the project features. It is split into 3 overall sections: the core engine, the rendering engine and the physics engine. I did not include a section on the audio engine as it was never implemented.

5.2 The Core Engine

As stated earlier, the core engine is the heart of it all. It is the engine that implements all the mathematical functionality, the engine that defines the Game, Game Object and Game Component classes and the engine which runs all the other engines.

5.2.1 The Main Loop

Every Game runs in some kind of loop, which executes the code that runs the Game. Typically the structure of a Game loop looks something like

```
while(!done)
{
    for(int i = 0; i < numGameObjects; i++)
    {
        gameObjects[i]->update(deltaTime);
        gameObjects[i]->simulate(deltaTime);
        gameObjects[i]->render();
    }
```

```
    if(getKeyPressed() == KEY_ESC)
    {
        done = false;
    }
}
```

We can see here that every Game Object is updated, simulated (physics) and rendered every pass of the loop. Every pass is usually referred to as a frame, so if somebody says that a game is running at 60 frames per second (fps), it means that the main loop goes through 60 iterations in a second. A critical aspect of the main loop is calculating the $\Delta time$ (delta time), also known as the time step. The $\Delta time$ is the time difference between frames and is used to make sure that variable frame rates don't affect the Game. For instance if we move a Game Object we should multiply the amount moved by the $\Delta time$. The $\Delta time$ of a Game running a 60 fps is 0.01666.... ms, and a Game running at 30 fps has a $\Delta time$ of 0.0333.... ms. Suppose we move a character 15 units in a direction, without multiplying the movement units by the $\Delta time$ the Game running at 60 fps will move the unit twice as far as the Game running at 30 fps. This is because for every pass completed at 30 fps, 2 passes will be completed at 60 fps. By multiplying the movement units by the $\Delta time$ we address this problem [8].

When $\Delta time = 0.0333$ (30 fps), the movement units (15 in this example) when multiplied will be equal to

$$M = 15 * 0.0333$$
$$M = 0.4995$$

When the $\Delta time = 0.0166$ (60 fps)

$$M = 15 * 0.0166$$
$$M = 0.249$$

But since we know that 60 fps will run twice as fast as 30 fps, we can multiply this number by 2, so

$$M = 0.249 * 2$$
$$M = 0.498$$

Therefore, by multiplying the movement by the $\Delta time$, the resulting movement distance was functionally identical.

5.2.2 The Game

The Game class is designed as a base class for user defined Games. Similar to XNA, if a user was creating a Game with this system they would first create a class that iherited from Game. The structure for a Game is defined as

```
class Game
{
protected:
   GameObject m_rootObject;
   void addToScene(GameObject* object);

public:
   Game() {};
   virtual ~Game() {};

   virtual void initialise();
   virtual void input(const Input& input, float delta);
   virtual void update(float delta);
   virtual void integrate(PhysicsEngine* physicsEngine, float delta);
   virtual void render(RenderingEngine* renderingEngine, const Camera& camera);

   GameObject& getRoot();

   void setEngine(CoreEngine* engine);
};
```

The `initialise`, `input`, `update` and `render` functions all call the respective function on the protected member `m_rootObject`. The `integrate` function is a bit different in that it only calls the `physicsEngine->simulate` function passing in the `delta` as a parameter.

It's important to note that although all these function are virtual, if any function is overridden the base function must be called or the functions on `m_rootObject` won't be called.

Another important function is the protected member `addToScene`. This is simply syntactic sugar, a wrapper function for `m_rootObject.addChild`. From an end user point of view, `addToScene` is a better descriptor for the process of adding Game Objects to the Game. Calling `addToScene` is identical in function to calling `getRoot().addChild`.

5.2.2.1 The Game Object

The definition of a simple Game Object class is

```
class GameObject
{
private:
   std::vector<GameObject*> m_children;
   std::vector<GameComponent*> m_components;

protected:
   Transform m_transform;
   virtual void initialise();
```

```
    virtual void input(const Input& input, float delta);
    virtual void update(float delta);
    virtual void render(const Shader& shader,
                        const RenderingEngine& renderingEngine,
                        const Camera& camera) const;

public:
    GameObject(const Vector3& position, const Quaternion& rotation, float scale);
    ~GameObject();

    GameObject* addChild(GameObject* child);
    GameObject* addComponent(GameComponent* component);
};
```

Here we can see that every Game Object has a list of Game Objects (`m_children`) and a list of Game Components (`m_components`). We also see the protected Transform member `m_transform` which stores the position, rotation and scale values. The `initialise` , `input` , `update` and `render` functions call the respective function on each Game Object in `m_children` and each Game Component in `m_components` .

So whenever a member function is called, it is called recursively through `m_children` and `m_components` . This is the premise of the `Game::m_rootObject` . Since every Game Object added to the game is a child of `Game::m_rootObject` , we only need to call `Game::m_rootObject.initialise` to initialise every Game Object and Game Component in the scene.

The public members of a Game Object are very straight forward. The constructor will construct the `m_transform` member, passing in the position, rotation and scale. We also have a destructor which cycles through `m_children` and `m_components` calling their destructors in a recursive fashion. We then have 2 functions `addChild` and `addComponent` which simply add Game Objects and Game Components to their respective lists.

It should be noted that the only purpose of `initialise` , `input` , `update` and `render` in a Game Object is to call the corresponding functions in `m_components` . No other behaviour is defined in the Game Object class, since an ECS states that the Game Components describe how the Game Object behaves.

5.2.2.2 The Game Component

The definition of the Game Component class is similar to the Game and Game Object classes, albeit much simpler

```
class GameComponent
{
```

```
protected:
   GameObject* m_parent;

public:
   virtual void initialise() {}
   virtual void input(const Input& input, float delta) {}
   virtual void update(float delta) {}
   virtual void render(const Shader& shader,
                       const RenderingEngine& renderingEngine,
                       const Camera& camera) const {}

   void setParent(GameObject* parent);
   GameObject* getParent();
};
```

Much like the Game class, the Game Component is a designed as a base class to be inherited from. We see that the `initialise` , `input` , `update` and `render` functions are all virtual with empty bodies. I decided against making them purely virtual as some Game Components might not have a need to implement every function. A component that is solely dedicated to describing a rendering behaviour might not necessarily need to implement an input function for example.

We also see that the only member variable of the base Game Component class is a pointer to the Game Object it is attached to, referred to as the parent. The only additional functionality is a getter and setter for `m_parent` .

5.2.3 The Mathematics

At the end of the day, a game engine is just performing mathematical operations. The physics, shaders, cameras, everything is just a bunch of numbers. This next section focuses on the 3 main mathematical structures that this system implements and the functionality they perform. I won't be going into detail on the theory and logic as it isn't really important for the scope of this project or paper.

5.2.3.1 Vectors

Vectors are by far the most commonly used object in this system. Every engine uses them in one way of another. In Chapter 4, a vector is defined as an array of numbers of N length. A vector3 can simply be defined as

```
class Vector3
{
private:
   float m_x;
```

```
    float m_y;
    float m_z;

public:
    Vector3(x, y, z);

    float getX() { return m_x };
    float getY() { return m_y };
    float getZ() { return m_z };

    void setX(float f) { m_x = f };
    void setY(float f) { m_y = f };
    void setZ(float f) { m_z = f };
}
```

This system also implements vector2 and vector4 classes, however, since the math is largely the same I will only discuss Vector3s. For any Vector3 functionality described the functionality is the same; for vector2s only without the Z component and for vector4s with an additional W component.

The most important functions implemented by the vector class are

```
float length() const;
float squareLength() const;
Vector3 normalised() const;
float dot(const Vector3& v) const;
Vector3 cross(const Vector3& v) const;
Vector3 inversed() const;
Vector3 absolute() const;
Vector3 reflect(const Vector3& normal) const;
```

`length` returns the length of a vector (also known as the magnitude) [7]. It is given by the formula

$$|V| = \sqrt{x * x + y * y + z * z}$$

Since the squared length is also needed throughout the system the function `squareLength` returns

$$x * x + y * y + z * z$$

with `length` returning the square root of `squareLength` . Length is used for various reasons such as checking the distance between two objects, describing the attenuation of lights, checking collision depth and more.

The `normalised` function simply returns a vector that is identical in angle, but only 1 unit in length [7]. This is achieved by dividing the vector elements by the length. To the normalised vector3 N of vector3 V is calculated as

$$N = (V_x/|V|, V_y/|V|, V_x/|V|)$$

Normal vectors are most commonly used as direction vectors. They are used in areas like normal mapping, lighting, collision detection and the rotation of vectors. Often times, a Game Object will have a vector called the look vector or direction vector. This vector is always normalised, in fact any orientation vector, even those derived from a quaternion are always normalised.

The `reflect` method also takes a normal vector as a parameter. This function returns a normalised vector that describes the direction something, such as an object or light, should reflect off of a surface. A reflection vector3 R of a vector3 V moving in a direction vector3 N is determined by the formula

$$R = 2 * (V \cdot N) * N - V$$

The `reflect` function makes use of the `dot` function, denoted by $\cdot$. This is an extremely important function. The dot product, somethings called the scalar product is used to determine the cosine of the angle between to vectors [7]. This is useful is many regards. The dot product D of a vector3 V^1 and vector3 V^2 is calculated as

$$D = V_x^1 * V_x^2 + V_y^1 * V_y^2 + V_z^1 * V_z^2$$

Another important function of the vector class is the `cross` function, often denoted by $\times$. `cross` is used to calculate the cross product (also known as the vector product) returns a vector that is mutually perpendicular to the 2 input vectors [7]. This can be useful in calculating movement and normal surface vectors. For instance if we want a player to be able to strafe right and left, we can work out the right vector regardless of the orientation since we can just cross the forward vector and the up vector. Since the formula is pretty long winded I won't show it here, but the code to calculate the cross product is

```
Vector3 Vector3::cross(const Vector3& v) const
{
   float x_ = y * v.getZ() - z * v.getY();
   float y_ = z * v.getX() - x * v.getZ();
   float z_ = x * v.getY() - y * v.getX();

   return Vector3(x_, y_, z_);
}
```

In addition there are some smaller functions that are useful to have such as `inversed` and `absolute` . `inversed` returns a copy of the vector but every element is inversed, it is the same as multiplying the vector by -1. `absolute` simply returns an identical vector except that every element is an absolute value.

The vector classes implement many more functions as can be seen in Math3D.h (located at 'Engine/Core/Math3D.h'). The system also overloads the '[]' operators to access elements quickly (so v[0] will access the 'x' element of vector 'v') and all the relevant operators for adding, subtracting, multiplying and dividing vectors by vectors and vectors by scalars.

Scalars allude to single element values (such as floats) that modify the length of the vector, without changing the direction [7]. Scalars get their name because they scale vectors up or down. For example if we have a vector3 V such that

$$V = (0, 2, 3)$$

If we multiply V by a scalar, let's say 3, V becomes

$$V = (0 * 3, 2 * 3, 3 * 3)$$
$$V = (0, 6, 9)$$

V still has the same direction, only the length has been scaled up.

5.2.3.2 Quaternions

As stated in Chapter 4, quaternions are mathematical constructs that in this system are used to store rotation in 3D space. In terms of code the implentation of a quaternion is straight forward. A simple outline of the class is

```
class Quaternion
{
private:
   float m_x;
   float m_y;
   float m_z;
   float m_w;

public:
   Quaternion(float x, float y, float z, float w);
   Quaternion(const Vector3& axis, float angle);

   float length();
   Quaternion normalised();
   Quaternion conjugate() const;
   float dot(const Quaternion& q) const;
   Quaternion lerp(Quaternion destination, float lerpFactor);
   Quaternion slerp(Quaternion destination, float lerpFactor);
};
```

Similar to vectors in structure but much different in nature, a quaternion is composite of 4 elements. We can see from the second constructor that the `m_x` , `m_y` and `m_z` represent the axis of rotation and the `m_w` represents the angle of rotation (in radians). Like the vector it has a `length` function and a `normalised` function. These are implemented exactly like the respective vector functions with the length of a quaternion Q being calculated as

$$|Q| = \sqrt{x * x + y * y + z * z + w * w}$$

and the normalised quaternion N being calculated by

$$N = (x/|Q|, y/|Q|, z/|Q|, w/|Q|)$$

Also like vectors the `dot` function returns the angle between 2 quaternions. The dot product D of quaternion Q^1 and quaternion Q^2 is given by

$$D = (Q_x^1 * Q_x^2 + Q_y^1 * Q_y^2 + Q_z^1 * Q_z^2 + Q_w^1 * Q_w^2)$$

The `conjugate` function is used to calculate an identical rotation quaternion that is pointing the opposite direction. That is to say it represents the multiplicative inverse of a quaternion [7] and can be determined by simply negating the `m_x` , `m_y` and `m_z` elements of a quaternion. So the conjugate C of quaternion Q is given as

$$C = (-Q_x, -Q_y, -Q_z, Q_w)$$

The functions `lerp` and `slerp` mean linear interpolation and spherical interpolation respectively. They are methods of producing natural looking rotations. Rather than just setting a rotation to the desired quaternion, the process of interpolation involves calculating intermediate rotations to produce a gradual rate of turning. Linear interpolation is the faster of the two in terms of computation, however its results are not as natural looking. The rate of rotation is constant and can be plotted on a line, hence the name. Spherical interpolation however, is more natural looking but has a heavier run time cost [14].

This is just a small snippet of the quaternion class in this system. Like the vector classes its full definition is outlined in Math3D.h.

5.2.3.3 Matrices

This system implements 2 types of matrices as stated in Chapter 4. In this system the matrices are simply wrapper classes for 2 dimensional arrays. One of the most important matrix4s is the projection matrix. Which is used to calculate the Model View Projection (MVP) in the shader programs.

Matrix4s are commonly used to store the transform of a Game Object, as shown in Chapter 4. This is the transformation matrix that it multiplied by the projection matrix to calculate the MVP. A typical transformation matrix would look like

$$\begin{bmatrix} S_x & 0 & 0 & P_x \\ 0 & S_y & 0 & P_y \\ 0 & 0 & S_z & P_z \\ R_x & R_y & R_z & R_w \end{bmatrix}$$

Where P, R and S represent the transform's position, rotation and scale respectively.

Matrix3s are also used in this system. The only use of matrix3s is to calculate the inertia tensor of a rigid body. An inertia tensor I can be calculated as

$$I = \begin{bmatrix} C * M * (D_y + D_z) & 0 & 0 \\ 0 & C * M * (D_x + D_z) & 0 \\ 0 & 0 & C * M * (D_x + D_y) \end{bmatrix}$$

where M is the mass of the object, D represents half the dimensions of the object squared and C is a constant value. There is no right or wrong value for C, though 0.3 is used commonly [12].

Like vectors and quaternions this system also overloads all the mathematical operators to easily add, subtract, multiply and divide matrices. It should be noted however that since this system only implements 2 types of matrices the math operators only work on matrices of the same dimensions. That is the matrix3 operators only work with other matrix3s, the same stands for the matrix4 operators.

5.2.4 Lua Scripting

This system supports Lua files that are structured like Game Components, in fact that's exactly how the system treats them. A typical layout of a Lua file used in this system would be

```
function initialise()

end

function input(delta)

end

function update(delta)

end

function render()

end
```

When a user wants to load a Lua file they must first create a `Scripter` object. These are special Game Components that are designed to execute the Lua code and work to combine multiple Lua files into one file.

```
class Scripter : public GameComponent
{
private:
   lua_State* m_L;
   std::string m_scriptName;

   std::vector<std::string> m_otherCode;
```

```
    std::vector<std::string> m_localCode;
    std::vector<std::string> m_initCode;
    std::vector<std::string> m_inputCode;
    std::vector<std::string> m_updateCode;
    std::vector<std::string> m_renderCode;

    void loadScript(const std::string& fileName);

public:
    Scripter();
    ~Scripter();

    virtual void initialise();
    virtual void input(const Input& input, float delta);
    virtual void update(float delta);
    virtual void render(const Shader& shader,
                        const RenderingEngine& renderingEngine,
                        const Camera& camera) const;

    void addScript(const std::string& script);
};
```

When a Lua file is loaded with `Scripter::loadScript` keywords such as "function" and "local" are identified and stored in the relevant list, so if an `update` function is found `Scripter` puts the body in `Scripter::m_updateCode`. When it does this for the rest of the file a random number is generated and this is prepended to the function or variable name. For example if a Lua file contains a variable called `someVar`, the `Scripter` will rename it to `s123someVar`, where "123" is a randomly generated number and "s" is needed since variable names can't start with a number.

When the declarations of the functions and variables have been assigned their random prefixes, a custom find and replace algorithm is run through the file to update their references in the code. The algorithm has been designed with robustness and speed in mind, it is intelligent enough to know when things should be renamed and when they shouldn't. For example if we have a variable `position` which calls on a position member of some object like `position = obj:position` and the algorithm renames `position` to `s2585position` and `obj` to `s2585obj`. The resulting code will become `s2585position = s2585obj:position`. This is because the algorithm is smart enough to know the difference between the local variable `position` and the member variable `obj:position` which is defined elsewhere.

Key functions, namely `initialise`, `input`, `update` and `render` don't have their function names renamed. Instead their bodies are stored and aggregated, to be written out later under one function header each. Since Lua scripts are treated as Game Components, each script must implement at least one of these functions to have an affect of the scene. By aggregating the bodies we call one larger function instead of multiple smaller ones.

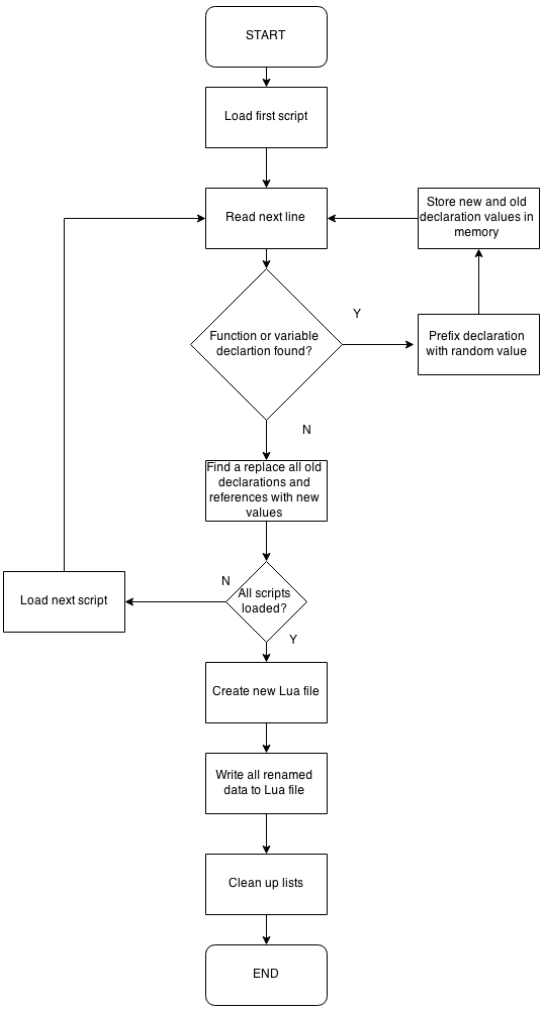

Figure 5.1: Lua file generation

When all the code has been aggregated and renamed, a Lua script is generated and all the renamed code is written to it. To ensure uniqueness this generated file is given the memory address of the parent object as its name. It's this file that the system uses and when the `Scripter` has its destructor called this file is deleted. This extensive back end has been developed so that it is invisible to the user. From their point of view Lua files can be written and loaded as normal and the system takes care of all the hard work.

5.3 The Rendering Engine

The rendering engine is the engine that represents everything on the screen. It calls OpenGL functions and GLSL shaders to decide what colour what pixel should be at anytime.

5.3.1 The Render Function

The Game has a function, `Game::render` which calls the render function on `Game::m_rootObject`. In the core engine this render function is invoked by `m_game->render(m_renderingEngine, *m_mainCamera)` . This function invokes the render function on the rendering engine like so

```
void Game::render(RenderingEngine* renderingEngine, const Camera& camera)
{
    renderingEngine->render(m_rootObject, camera);
}
```

Here we see the `Game::m_rootObject` being passed into the rendering engine. This invokes a larger function that draws the object, calculating things like shadows, lighting and colour. A simple overview of the `RenderingEngine::render` function looks like

```
void RenderingEngine::render(const GameObject& object, const Camera& camera)
{
    glClearColor(0, 0, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    object.render(m_defaultShader, *this, camera);
}
```

Above we can see the calls for OpenGL to clear the screen to black and clear the drawing buffers, before calling render on the Game Object. The following sections will examine the rendering in more detail, explaining what the camera is, what `m_defaultShader` is and how more complicated shaders are called.

5.3.2 The Camera

Simply put the camera is a matrix4 often reffered to as the projection matrix. There are many different projections that a system can use: orthographic, isometric, axonometric [9] but this system implements the perspective projection. To create a perspective projection we can use the function `Matrix4::initPerspective` which is implemented as

```
Matrix4 Matrix4::initPerspective(float fov, float a, float zn, float zf)
{
   float h = tanf(fov / 2);
   float zr = zn - zf;

   m[0][0] = 1.0f / (h * a);   m[1][0] = 0;
    m[0][1] = 0;               m[1][1] = 1.0f / h;
    m[0][2] = 0;               m[1][2] = 0;
    m[0][3] = 0;               m[1][3] = 0;

    m[2][0] = 0;               m[3][0] = 0;
    m[2][1] = 0;               m[3][1] = 0;
    m[2][2] = (-zn - zf) / zr; m[3][2] = 2 * zf * zn / zr;
    m[2][3] = 1;               m[3][3] = 0;
}
```

The above code defines the frustum of the camera. The parameter `fov` describes the field of view, that is how wide the angle of the frustum is (in degrees). The `a` parameter is the aspect ratio of the window and the `zn` and `zf` parameters are the Z near and Z far respectively. The Z near is the minimum distance from the camera an object needs to be on the Z axis for it to be drawn, any lower and it won't be shown. The Z far is maximum distance an object can be on the Z axis before it is no longer drawn. The MVP is then calculated by multiplying an Game Object's transformation matrix by the projection matrix. When the 2 matrices are multiplied objects with a small Z coordinate in their transform will be divided by a small Z from the projection matrix's transformation matrix, which means they get blown up, objects with a large Z coordinate will be shrunk down since they are divided by a large number. This results in objects with a large Z value i.e., objects that are far away, appearing smaller than those with a lower Z value [9].

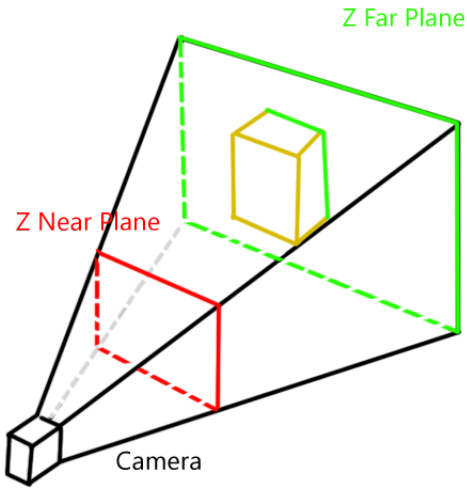

Figure 5.2: An illustration of the frustum.

5.3.3 Meshes

OpenGL draws shapes between vertices. That is a collection of points in 3D space. Meshes are simply a collection of vertices and can be created in many 3D modeling programs. This system only supports ".obj" files at this time. There can also be additional information stored in mesh files such as texture indices and surface normals.

When a mesh is loaded in its data (vertices, normals, etc) are mapped to the file name. If the same mesh is loaded again I check to see if that file name is in the map and if it is the system returns the corresponding mesh data. This is an optimisation technique that potentially saves on unnecessary loading. A mesh might contain hundreds of vertices, if that mesh is loaded in once, it is kept in memory to save it from being loaded twice.

5.3.4 Colours

As stated in the section on vectors, the Colour class is an extension on the vector4 class with the X, Y, Z, and W elements representing the R, G, B and A values. Colours can also be defined by a hex value. For convenience there are a total of 142 predefined colours in the system. All these colours are taken from 142 colours XNA has predefined so that anyone familiar to developing in XNA will already know the existing colour set.

The fact that colours are represented as vector4s also makes converting them to GLSL values easy since GLSL already stores them as vector4s (called `vec4` in GLSL)

5.3.5 Shaders

Shaders in this system are loaded in pairs. Every vertex shader has a fragment shader associated with it. Vertex shaders, as the name would suggest, determine where the given vertex is located in 3D space and calculates the respective pixel location on the screen. Fragment shaders, also known as pixel shaders, are the shaders that decide what colour the pixel determined by the vertex shader will be [9]. A simple example of a shader pair is shown below. First the the vertex shader.

```
#version 120
attribute vec3 position;
attribute vec2 texCoord;

varying vec2 texCoord0;

uniform mat4 T_MVP;

void main()
{
```

```
    gl_Position = T_MVP * vec4(position, 1.0);
    texCoord0 = texCoord;
}
```

This vertex shader shows us the `gl_Position` (the pixel) calculation being done. `position` and `texCoord` are values being sent in from the mesh data and `T_MVP` is the transform model view projections that's calculated by multiplying the Game Object's transform by the world transform. We also see the variable `texCoord0` , this will be sent to the fragment shader and is used to calculate what part of the mesh should be what colour. Shown below is a simple fragment shader that might be paired with this vertex shader

```
#version 120

varying vec2 texCoord0;
uniform sampler2D diffuse;

void main()
{
   gl_FragColor = texture2D(diffuse, texCoord0);
}
```

Here `gl_FragColor` , (the pixel colour) is being assigned using `texCoord0` from the vertex shader and a variable called `diffuse` , which will determine the colour. An example is shown below.

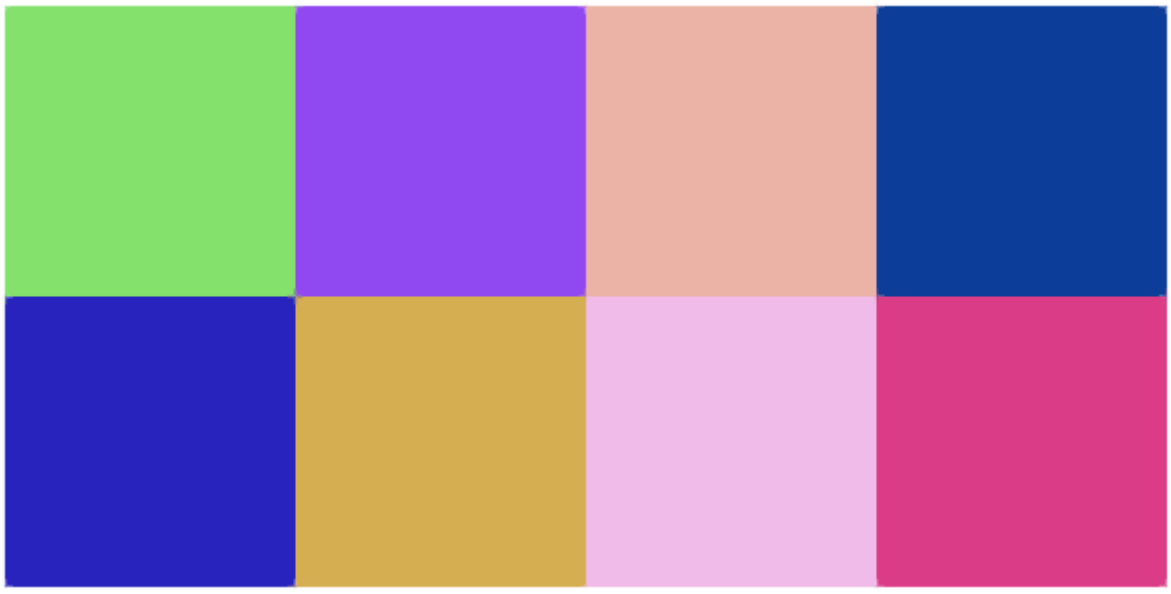

Figure 5.3: Some simple colours rendered to squares.

Simple changes to these files can have drastic consequences on the result rendered. For instance, the following fragment shader is identical to the one above except for one small difference.

```
#version 120

varying vec2 texCoord0;
uniform sampler2D diffuse;

void main()
{
    vec4 ambient = vec4(0.2, 0.2, 0.2, 1);
   gl_FragColor = texture2D(diffuse, texCoord0) * ambient;
}
```

The introduction of `ambient` as a multiplier of 0.2 will darken the colours on the screen. This is a simple yet important change that we can make to the shader code, especially in the calculations of lighting.

To make the management of shaders easier the system also features a class to wrap the data in C++. Like the way Lua is treated GLSL shaders run through a process of parsing. Variable names are found and are mapped to their data types. Then these variables (called uniforms) in GLSL can be assigned and updated at run time. Below is a snippet from the `Shader` class and shows the methods used to update shader variables

```
void setUniform(const std::string& fileName, int value) const;
void setUniform(const std::string& fileName, float value) const;
void setUniform(const std::string& fileName, const Matrix4& value) const;
void setUniform(const std::string& fileName, const Vector3& value) const;
void setUniform(const std::string& fileName, const Vector4& value) const;
```

5.3.6 Lighting

Like everything graphical lighting is calculated in GLSL on the graphics card. This system supports 3 different types of light. Directional lights, spot lights and point lights are all distinct yet we can generalise a base class called `BaseLight`

```
struct BaseLight
{
    vec3 colour;
    float intensity;
};
```

From this we can derive that every light has a colour and an intensity. The way the light is drawn is a different calculation for each light, however the logic to caluclate the amount of light needed can be done with the following function.

```
vec4 calcLight(BaseLight base, vec3 direction, vec3 normal, vec3 worldPos)
{
    float diffuseFactor = dot(normal, -direction);

    vec4 diffuseColour = vec4(0,0,0,0);
    vec4 specularColour = vec4(0,0,0,0);

    if(diffuseFactor > 0)
    {
        diffuseColour = vec4(base.colour, 1.0) * base.intensity * diffuseFactor;

        vec3 directionToEye = normalize(C_eyePos - worldPos);
        vec3 halfDirection = normalize(directionToEye - direction);

        float specularFactor = dot(halfDirection, normal);
        specularFactor = pow(specularFactor, specularExponent);

        if(specularFactor > 0)
        {
            specularColour = vec4(base.colour, 1.0) *
                             specularIntensity *
                             specularFactor;
        }
    }

    return diffuseColour + specularColour;
}
```

When used in conjunction with the ambient mulitplier as shown in the previous section, lighting can have very realistic and pleasing results. The ambient multiplier darkens the entire scene and the lighting brings certain aspects back up to their true colour. In fact, this is the technique that the Phong Shading Model employs.

5.3.6.1 Directional Lights

Directional lights are often used in games to represent global lighting, for example if you wanted to simulate sunlight in a game you'd use a directional light. However, directional lights have no real world counterpart (the sun technically being a point light). Directional lights can be thought of as spot lights, but instead of emitting light in a conical shape, they emit light in a parallel fashion over an infinite area [9].

Directional lights can simply defined as

```
struct DirectionalLight
{
    BaseLight base;
    vec3 direction;
};
```

While conceptually the most abstract, the implementation of directional lights is quite simple using the `calcLight` function from above

```
vec4 calcDirectionalLight(DirectionalLight directionalLight,
                          vec3 normal,
                          vec3 worldPos)
{
    return calcLight(directionalLight.base,
                     -directionalLight.direction,
                     normal,
                     worldPos);
}
```

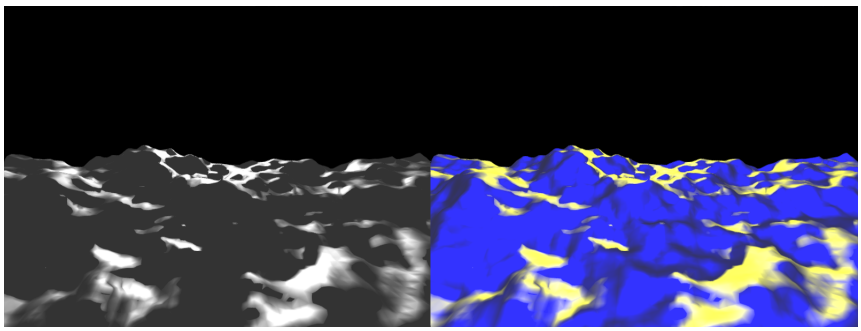

Figure 5.4: Left: a white directional light. Right: blue and yellow directional lights, shining in opposite directions.

5.3.6.2 Point Lights

A point light is simply a light mapped to a point in 3D space that emits light in all directions [9]. Since they calculate light in all directions, point lights are considerably more expensive than directional lights, despite conceptually being more intuitive. A spot light can be defined as

```
struct PointLight
{
    BaseLight base;
    Attenuation atten;
```

```
    vec3 position;
    float range;
};
```

Point lights also introduce the concept of attenuation. Attenuation is simply the opposite of amplification and describes the gradual loss of information (in this case light) over time. This makes point lights different to directional lights because directional lights don't fade over distance. As point lights illuminate in all directions and have the additional property of attenuation, the code to calculate a point light is noticeably more complicated than a directional light.

```
vec4 calcPointLight(PointLight pointLight, vec3 normal, vec3 worldPos)
{
    vec3 lightDirection = worldPos - pointLight.position;
    float distanceToPoint = length(lightDirection);

    if(distanceToPoint > pointLight.range)
    {
        return vec4(0,0,0,0);
    }

    lightDirection = normalize(lightDirection);

    vec4 color = calcLight(pointLight.base, lightDirection, normal, worldPos);

    float attenuation = pointLight.atten.constant +
                        pointLight.atten.linear * distanceToPoint +
                        pointLight.atten.exponent * distanceToPoint * distanceToPoint +
                        0.0001;

    return color / attenuation;
}
```

Figure 5.5: A selection of point lights.

5.3.6.3 Spot Lights

In games, spot lights are commonly used to represent torches or lights on ceilings. Unlike directional lights, spotlights have a definitive beginning and end. Spot lights are very versatile lights as you can define almost all aspects, such as length, hardness and field of view (the angle of the cone of light). Spot lights, rather than extend `BaseLight` directly, extend `PointLight` , this way can inherit the range, attenuation and position properties of `PointLight` . The only real difference between spot lights and point lights is that spot lights have a direction [9]. A spot light can be defined as

```
struct SpotLight
{
    PointLight pointLight;
    vec3 direction;
    float cutoff;
};
```

The calculation of a spot light is shown

```
vec4 calcSpotLight(SpotLight spotLight, vec3 normal, vec3 worldPos)
{
    vec3 lightDirection = normalize(worldPos - spotLight.pointLight.position);
    float spotFactor = dot(lightDirection, spotLight.direction);

    vec4 color = vec4(0,0,0,0);

    if(spotFactor > spotLight.cutoff)
    {
        color = calcPointLight(spotLight.pointLight, normal, worldPos) *
                (1.0 - (1.0 - spotFactor)/(1.0 - spotLight.cutoff));
    }

    return color;
}
```

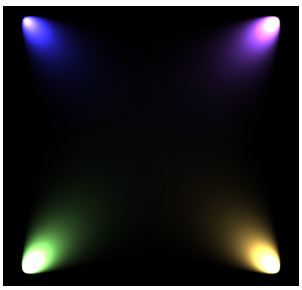

Figure 5.6: A selection of spot lights.

5.3.7 Phong Shading & Reflection Model

In the `calcLight` function there are 3 important variables: `specularExponent` , `specularFactor` and `specularIntensity` . These variables illustrate how the lighting functions are an implementation of the Phong Shading & Reflection Model. This model describes an algorithm of rendering illumination of the object and amplifying it to a desired result. To put it simply, this algorithm controls how shiny an object is rendered. There are 3 layers to this model. The ambient layer is the layer that renders the object in its true colour, multiplied by a fraction. In the aforementioned shading section I showed an example of this. This multiplier brings the colour down to a darker shade, which fakes areas of shadow. The diffuse layer is next, at this stage the light is reflected at normal angles to the mesh surface,this brings out the edges giving the illusion of 3D. Any area not hit by light will remain dark from the ambient stage. The third layer is known as the specular layer. The light hitting objects is amplified up making it even brighter. This gives the impression of shininess [21]. An illustration of the 3 stages is shown below.

Figure 5.7: The 3 stages of Phong shading. Ambient, diffuse and specular.

5.3.8 Normal & Displacement Mapping

Normal mapping is a rendering technique that is used to give the impression that a mesh has more vertices than reality. This results in the appearance of a high polygon count without the overhead of one [5]. We see in the shader `calcLight` function that a `normal` argument is taken in which affects the value of the light on a given pixel. Normal maps are a collection of vector normals stored as a texture. The RGB values of a pixel in the normal map are actually the XYZ values of the direction normal. The pixel at [0, 0] on the normal map is used as the normal direction for pixel [0, 0] on the texture to be rendered. The RGB values of the normal map are passed in as the `normal` argument in `calcLight` and over layed onto the rendered texture. The results are effective and cheap to calculate although it is important to note that the effect breaks down as the camera forward becomes more parallel with the texture. An example of normal mapping is shown below.

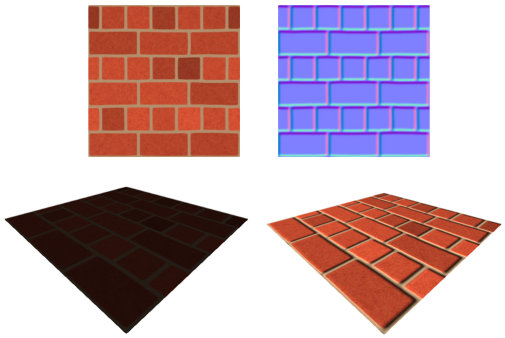

Figure 5.8: A normal map example and effect.

As an extension to normal mapping this system also supports displacement mapping. Displacement mapping is a rendering technique that offsets or displaces vertices based on a height map. Similar in function to normal mapping, displacement maps are stored as textures with the displacement values derived from the pixel RGB values. Displacement mapping can produce more realistic results than normal mapping at the cost of being more expensive to calculate [1].

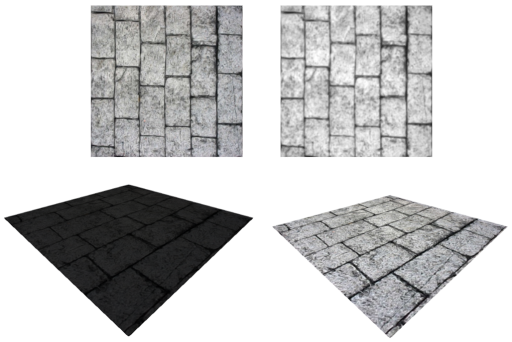

Figure 5.9: A displacement map example and effect.

5.3.9 Shadow Mapping

Shadow mapping is a simple way of rendering shadows. The logic behind the process is very simple. To begin we render the scene from the lights point of view generating a texture. This is the shadow map.

Then for every point rendered from the camera we find the corresponding point on the shadow map by tracing back to the lights location. If the point of the shadow map is closer to the light being tested then the tested point is in shadow.

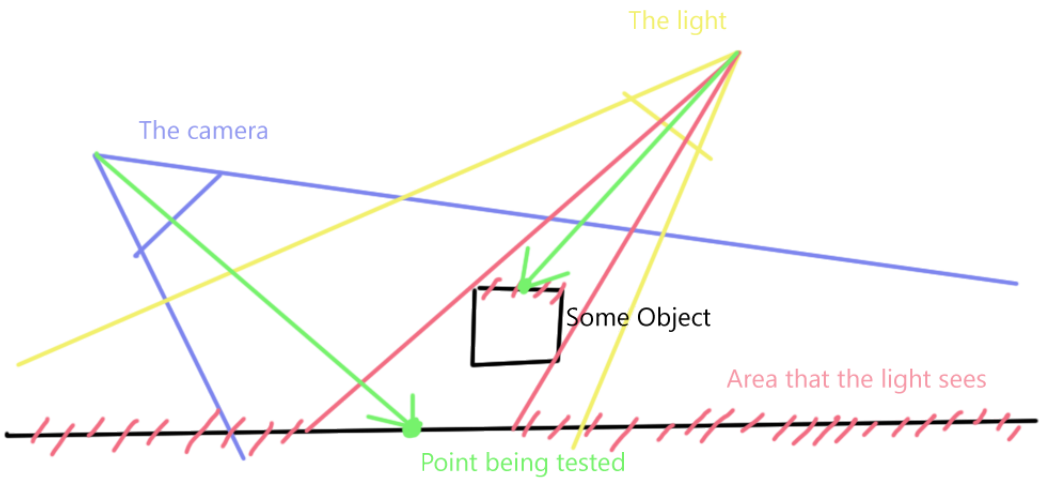

Figure 5.10: Shadow mapping diagram.

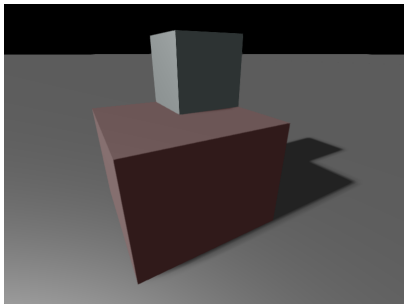

Figure 5.11: Some rendered shadows.

It should be noted that the current implementation only supports shadow mapping for directional lights and spot lights. Point lights could be supported but generating shadows for every direction is relatively expensive [6].

5.3.10 Anti-Aliasing

This system implements Fast Approximate Anti-Aliasing (FXAA) to remove stair-stepping artifacts (aliasing) associated with low resolution rendering.

The algorithm is more complex in theory than Multi-Sample Anti-Aliasing (MSAA) but much cheaper to perform. It works in two steps, the first is edge detection. To detect edges the system samples an area of pixels in a cross shape pattern and compares the luminosity of each pixel with pixel adjacent. This isn't the only way to do this, but using diagonals cuts down

on checking every pixel and can help in deducing the direction of the edge.

The second step is to blur the detected edge. When we find the center of the sample section is on an edge we can then average the RGB value from weighing the luminosity of the surrounding pixels. This method of anti-aliasing is extremely fast as it is done in image space rather than object space.

5.4 The Physics Engine

The physics engine is the engine that applies the laws of motion to Game Objects and detects and resolves any collisions that occur.

5.4.1 The Physics Engine & Physics Component

Physics Components are a type of Game Component. When a Physics Component is attached to a Game Object a reference to that component is sent to the physics engine. All the Physics Components are then simulated by the physics engine using the function `PysicsEngine::simulate`

```
void PhysicsEngine::simulate(float delta)
{
   if(m_components.empty())
   {
      return;
   }

   m_contactData.reset(MAX_CONTACTS);

   updateComponentReferences(m_components, m_tree, delta);
   m_tree->getPotentialCollisions(&m_contactData);
   m_resolver.resolveContacts(m_contactData.getContactArray(),
                              m_contactData.getContactCount(),
                              delta);
}
```

The following sections will discuss the different members mentioned in this function.

5.4.2 Rigid Bodies

Rigid bodies are the part of the Physics Component that make the Game Object move. These are the members that have mass, velocity, friction and restitution. The process of updating these properties is called integration. `PhysicsComponent::update` is overridden so that when

it is called by the parent Game Object `RigidBody::integrate` is called. This integration method is an implementation of the standard Hamiltonian physics [12].

```
void RigidBody::integrate(float delta)
{
    if(!m_isAwake)
    {
        return;
    }

    m_lastFrameAcceleration = m_acceleration;
    m_lastFrameAcceleration += (m_forceAccum * m_inverseMass);

    Vector3 angularAccel = m_inverseInertiaTensorWorld.transform(m_torqueAccum);

    m_velocity += (m_lastFrameAcceleration * delta);
    m_rotation += (angularAccel * delta);

    m_velocity *= powf(m_linearDamping, delta);
    m_rotation *= powf(m_angularDamping, delta);

    m_parent->getTransform()->setPosition(m_parent->getTransform()->getPosition() +
                                          (m_velocity * delta));
    m_parent->getTransform()->setRotation(m_parent->getTransform()->getRotation() +
                                          (m_rotation * delta));

    clearAccumulators();
}
```

We can see here that the parent Game Object `m_parent` has its position and rotation changed depending on the forces that the rigid body is subject to. We see that `m_velocity` and `m_rotation` get their values increased by `m_forceAccum` and `m_torqueAccum` respectively. Every pass these accumulators are reset as to avoid constant stacking of values.

If we wanted Physics Components to be affected by gravity we could access the rigid body member and call `RigidBody::addForce(Vector3(0.0, -9.8f, 0.0f)` . `RigidBody::addForce` and `RigidBody::addTorque` are the 2 most important functions in regards to moving an object and the 2 functions that are used in collision resolution.

It's important to note `m_linearDamping` and `m_angularDamping` . These are multipliers, usually between 0 and 1, that multiply the velocity and rotation down every frame. This makes an objects motion slow down in a believable way. Changing these values can also have some interesting effects. If you wanted to prevent an object from moving, you could set the damping values to 0. If you wanted to simulate an object moving through space you could set the damping to 1.

5.4.3 Colliders

The system implements 3 types of colliders: spheres, boxes and planes. Each can be defined as

```
class ColliderSphere : public Collider
{
protected:
   float m_radius;

public:
   ColliderSphere(float radius);

   virtual void collide(Collider& collider, CollisionData& data);
};
```

```
class ColliderBox : public Collider
{
protected:
   Vector3 m_halfSize;

public:
   ColliderBox(const Vector3& dimensions);

   virtual void collide(Collider& collider, CollisionData& data);
};
```

```
class ColliderPlane : public Collider
{
protected:
   Vector3 m_normal;
   float m_distance;
   Vector3 m_extents;

public:
   ColliderPlane(const Vector3& normal, float distance);

   virtual void collide(Collider& collider, CollisionData& data);
};
```

Each collider extends a common abstract class `Collider` . The function `Collider::collider` is a pure virtual function which means that each collider must implement it. This generalised collider system makes collision detection simpler. To check if one collider is in collision with another we can simply call the `Collider::collide` function without worrying about the type.

Each collider's implementation consists of a switch case statement to help decide what collision algorithm to use. For example the `ColliderSphere::collide` looks like

```
void ColliderSphere::collide(Collider& collider, CollisionData& data)
{
   switch(collider.getType())
   {
   case SPHERE:
      CollisionDetector::sphereAndSphere(*this, (ColliderSphere&)collider, &data);
      break;
   case BOX:
      CollisionDetector::boxAndSphere((ColliderBox&)collider, *this, &data);
      break;
   case PLANE:
      CollisionDetector::sphereAndPane(*this, (ColliderPlane&)collider, &data);
      break;
   }
}
```

5.4.4 Collision Detection

Collision detection in this system takes place in 2 phases. Broad phase collision detection, finds component pairs that are likely to be in collision before going to the more computational intense narrow phase collision which check component pairs that are in collision [3].

5.4.4.1 Broad Phase Collision Detection

The idea of broad phase collision is to find component pairs that are likely to be in collision. These are the pairs that are checked using more computational expensive collision checks. There are many different methods for broad phase collision detection, the method this system implements is octrees. The following section will use quadtrees to illustrate the logic of broad phase collision. Quadtrees are the 2D counter part of octrees. The implementation is the same, the only difference being the exclusion of the Z axis. Using quadtrees will simplify the explanation and diagrams.

Octrees and quadtrees are a form of spatial partitioning. An area is divided down into smaller areas based on the number of components contained. Using quadtrees as an example, if a sample area contains more than N amount of objects, the sample area is divided into 4 [3]. Each area is called a quad, and if the resulting number of components in each quad is still greater than N it is subdivided again. This continues recursively until no quad has more than N amount of objects. Logically this creates a tree like structure of quads as shown.

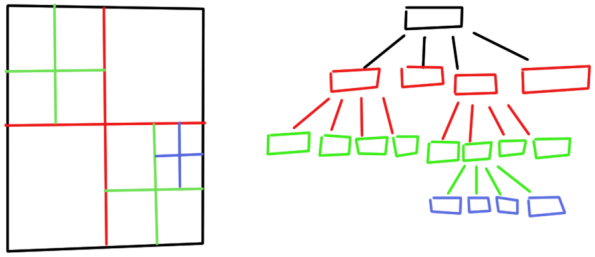

Figure 5.12: A visual representation of a quadtree and its logical structure.

Only the leaf nodes (quads containing components) have their components checked against other components in the same quad [3]. The idea being that components that are in different quads aren't near each other and therefore checking if they're colliding would be a waste of time. As an example of how effective this system can be let's take a sample scene containing 16 components are distributed more or less uniformly. If we specify that the quad should divide down if there are more than 4 contained components, the result looks like this

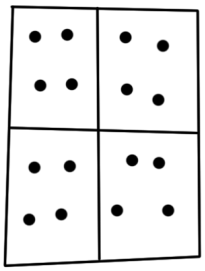

Figure 5.13: The 16 components are split into 4 groups of 4.

Without a quadtree the cost of checking each component against each component would be $O(N^2)$, in this case 16^2 equals 256 checks. However the same operation in a system using a quadtree would be $O(N^2 * M)$, where N is the number of components in each quad and M is the number of quads in the tree. The result in this example is $4^2 * 4$ which equates to just 64 checks. So we go from 256 checks without a quadtree to 64 with one. The decrease in checks is even more drastic in a similar octree scenario. Assuming 32 objects evenly distributed, without an octree the number of checks would amount to 32^2 which is 1024. Using an octree the check count becomes $4^2 * 8$ which is just 128 checks.

A simplified look at the octree class implemented in the system is shown as

```
class Octree
{
```

```
private:
   Vector3 m_min;
   Vector3 m_max;
   Vector3 m_centre;

   Octree* m_children[2][2][2];
   std::set<PhysicsComponent*> m_components;

public:
   Octree(const Vector3& min, const Vector3& max, int depth);
   ~Octree();

   void getPotentialCollisions(CollisionData* data);
   void generateContacts(PhysicsComponent& one,
                         PhysicsComponent& two,
                         CollisionData* data);
};
```

Here we see the `getPotentialCollisions` function that we saw in `PhysicsEngine::simulate` . This is the function that checks the finer collision on objects within the same octant. This process of checking for actual collisions is called narrow phase collision.

5.4.4.2 Narrow Phase Collision Detection

All narrow phase collision has the same logic. Find the penetration depth and collision normal for each collision pair. Since this system supports 3 colliders: spheres, boxes and planes, there are a total of 5 fine collision check methods.

- Sphere and sphere
- Sphere and box
- Sphere and plane
- Box and box
- Box and plane

Due to the fact that some of these functions are quite long, and the ideas behind them are similar I won't go into them here. To see the full implementation see "Collision.cpp" (Engine/Physics/Collision.cpp).

These functions are also responsible for setting the values in the `data` , which is of type `CollisionData` .

5.4.5 Collision Resolution

In the `PhysicsEngine::simulate` function we see a call to `m_resolver.resolveContacts` . `m_resolver` is of type `ContactResolver` . This is the class that is responsible for moving colliding components until they are no longer is collision. An overview ofthe resolution process can be seen in the `ContactResolver::resolveContacts` function

```
void ContactResolver::resolveContacts(Contact* contacts,
                                      unsigned numContacts,
                                      float duration)
{
   if(numContacts == 0)
   {
      return;
   }

   prepareContacts(contacts, numContacts, duration);
   adjustPositions(contacts, numContacts, duration);
   adjustVelocities(contacts, numContacts, duration);
}
```

We can see that there are 3 stages in the resolution process [3].

`ContactResolver::prepareContacts` calculates some important data for the rest of the process. In this function aspects of the collision such as contact speed and position are determined to be used later.

The two most important functions are next. `ContactResolver::adjustPositions` moves the colliding components until they are no longer intersecting and `ContactResolver::adjustVelocities` applies force and torque to give the objects the appropriate velocities and rotations.

5.5 Conclusion

In this chapter we discussed at length the code and functionality inside the core, rendering and physics engines. This is still only a fraction of the entire system and the full code base is available publicly at `https://github.com/oneillsimon/GameEngine`. The next chapter will talk about the testing and evaluation of the system.

Chapter 6

Testing & Evaluation

6.1 Introduction

This section looks at how the system was tested at each iteration. It also contains an evaluation section with includes an discussion with Chris Gregan and an evaluation from my own point if view.

6.2 Testing

The mathematics in the system were the simplest to test. The easiest way was to perform operations in my system and compare the results from identical operations performed in Unity, a system I trust to work.

The rendering was more difficult to formally test. Beyond making sure the GLSL shader files loaded and had no compilation errors, evaluation was done by seeing if it "looked right". Due to the visual nature of the engine and system as a whole there was no feasable way to test the rendering beyond this.

Scripting required intensive testing to get working, mostly due to my own lack of experience and knowledge of Lua. Testing involved using my algorithm on plain text files, testing the find and replace algorithms and generating text files based on arbitrary content. Testing was also done on the prefix method to make sure it only prefixed the correct items and the correct references to those items. Further testing was done when reading and sending data to the Lua run time. This started by pushing and pulling primitive data types such as ints and floats before moving on to more complex objects.

Since the physics engine is mostly maths a lot of the testing was done by testing similiar scenarios in Unity. In "Collision.cpp" there is also a class called `IntersectionTests` .

This is a static class that contains functions which will perform a quick test to see if two colliders are intersecting.

6.3 Evaluation

6.3.1 Meeting with Chris Gregan

Chris Gregan from Snozbot was kind enough to take the time to look at my system and give me feedback. Overall, Chris seemed impressed with the system and said that the engine would make a very impressive portfolio piece.

Some advice he gave me regarding the physics engine, which was having some issues was to use a fixed time step. Something I never took into account. Since physics simulation requires extreme precision, a varying time step could contribute to errors in the behaviour.

Chris gave me some excellent advice about future work as well. He told me that if I wanted to make myself stand out, it would be important to make something that looked impressive. It would be good to get a website and host a collection of my work, with some downloadable demos. At the very least host some screen shots of the system in action. Chris also told me that the fact I could write C++ was important skill to advertise, the experience with Lua was a bonus too. Although I can write in C++ it's vital to know how to write it in the most optimised fashion, and that an in depth knowledge of C++ was what employers look for. So I will bear that in mind for any future iterations of this system and I am definitely going to advance my knowledge of C++ in the future.

6.3.2 Self Evaluation

In my opinion, this project was a success. At the very least I have a good jumping off point for a new version of the entire system. In regards to each engine I will now go over what I feel I did right and more importantly, what I did wrong.

6.3.2.1 The Core Engine

Strengths

- The implementation of the ECS is simple and efficient.
- Implementing custom Game Components is easy.
- The Lua scripting system is robust and means that the cost of running multiple Lua scripts is always $O(1)$ per Game Object.

- Profiling for optimisation can be done using built in classes.
- Extensive functions for mathematical classes.

Issues

- Maths library should use template. Having separate classes define matrix3 and matrix4 leads to repeated code and a rigid system. The same applies for having separate classes for vector2s, vector3s and vector4s.
- System is modularised to the point of a fault, where it can often be difficult to send information across systems.
- Needs a way of generating a fixed time step. Current implementation always uses a varying time step.
- Lua needs LuaBridge to expose C++ to the Lua run time, SDL is used for input and to create OpenGL context. I'd rather not use any 3rd party systems.
- Exposing classes to Lua, even with LuaBridge, is tedious. An automated system would be better.
- Needs to have the functionality for changing certain properties at run time, for example screen resolution and a full screen toggle.

6.3.2.2 The Rendering Engine

Strengths

- Lights work extremely well.
- Implementing custom Game Components is easy.
- Extensive colour library.
- Loading GLSL shaders is simple, all the OpenGL calls are wrapped up in a Shader class.

Issues

- Should use deferred rendering instead of forward rendering. Forward rendering doesn't scale well with many lights.
- A system should be set up for arbitrary shaders, right now it only supports the shaders it's using.
- The system needs the ability to render font to the screen.
- Debugging graphics would be useful, for example simple shapes to indicate lights, cameras and orientation.

- No easy way to draw simple primitives such as lines and quads without using meshes.
- Most shader values are set in C++ using hard coded strings, while this works a more dynamic system would be better.

6.3.2.3 The Physics Engine

Strengths

- Octrees build automatically and work recursively.
- Simple collisions, such as sphere and sphere, sphere and plane work well.
- An object's inertia tensor is calculated automatically, but doesn't have to be.
- Colliders are generalised so the system is simple.
- Game Objects are implicitly added to the physics engine when a physics component is attached, this keeps the top level API simple.

Issues

- While building the octree works well, keeping it up to date during run time can be extremely expensive.
- A more extensive list of supported colliders is needed.
- Some collisions, such as box and box, are extremely expensive. Up to 16 checks a frame.
- Box colliders are still buggy, sometimes causing undefined behaviour when interacting with other colliders.
- The physics engine needs to use a fixed time step.
- Most shader values are set in C++ using hard coded strings, while this works a more dynamic system would be better.
- Modifying a physics component's properties during run time doesn't always work.
- No simple collider types such as 2D squares and circles.

I feel that a lot of the problems in this system, particularly the optimisation issues, are a result of my lack of experience in C++. The original implementation of this system was in Java, however porting it to C++ had to be done for memory management's sake. Compared to the earliest iterations the current system is very stable and is the sum of my knowledge and experience in C++. I'm sure someone more experienced with C++ could pick a million holes in my code and I only look forward to learning from my mistakes. There are some other obvious issues, such as

the complete lack of audio engine. Above I have mentioned a lot of the issues I feel exist in the system and fixing these issues will be the focus in future iterations.

6.4 Conclusion

In this chapter the testing of the system was discussed. It also features an evaluation by Chris Gregan and my own personal critique of the system. In the next and final chapter I give my thoughts on the system and project as a whole. I give my concluding comments on what I have achieved and mention how I would like to move forward with the system.

Chapter 7

Conclusion

I feel that I took on a big undertaking in developing this system and while it is far from perfect I'm happy to have it as a portfolio piece. While I regret not getting some aspects working, such as the audio engine, I think the amount I did get working is noteworthy. From a development point of view I feel that the system was rewarding and successful, The fact that it was modularised helped isolate different areas to study and I think that understanding the concept of modularisation will carry positive connotations for any future I have in software development.

The core engine is the heart and sole of this engine. Before I began development I didn't understand what an ECS was or how it worked. After having implemented an ECS myself I can see why it is so advantageous over an inheritance model. The idea of treating behaviour defining features as components also helped model the system as a whole. The idea of keeping the rendering engine, physics engine and audio engine separate was inspired by the ECS. While this helped keep the software compartmentalised, I did notice some difficulties in situations where I might want the separate engines to talk to each other. I can only conclude that the way these engines are defined in the core engine needs to change. During development I thought of deeper integration to the core engine, but I felt that losing that modularisation was antithetical to the design.

Lua scripting wasn't something I had initially planned to implement, being such a late addition I feel it is a portion of the engine I would like to go back and give the attention required. One of the most important requirements was for Lua to support my C++ classes. I had to use a 3rd party library, LuaBridge, in order to achieve this. I feel that this doesn't violate my idea of building an engine from scratch as I see the exposing of C++ classes as something Lua requires, rather than my engine. That being said I would have rather not used LuaBridge. The process of exposing C++ classes is rather tedious and as a future endeavour I plan to write my own system that would make the exposing of C++ classes much simpler. Despite all this I think the algorithm I wrote to execute and generate Lua files is one of the most robust algorithms in the engine. While it not be the perfect solution to the problem of having multiple Lua files mapped to the a single Game Object, I think it is an elegant one, at least from the top

level.

The rendering engine is the engine that I'm most proud of. Developing this engine was definitely more rewarding and fun than developing the other engines. The ability to see results straight away in a purely graphical way made me more invested into getting this system right. This is arguably the most important engine, assuming that if the physics and core engines worked perfectly, without the rendering engine nobody would be able to see them working perfectly. Obviously, the engine is critical for presenting the project as well. I know the odds of someone caring about what's going on beyond what they can see is low, so to have an engine capable of drawing an observer in is important. The rendering engine allows me to make something that looks impressive, which sometimes is just as good as making something that is impressive. Although the rendering engine is the most complete out of all the engines, it still has some problems. As I stated in prior chapter, this engine works well with the shaders its currently using, however the introduction of a new shader, especially if it's complex could potentially involve changing parts of the engine.

That being said the rendering engine does some of the most impressive work out of the entire system. The lighting calculations were tough to understand but the end result is very pleasing. The shadow mapping is also a simple touch that adds a lot to a scene. I think if I were to rewrite the rendering engine it would only be changed slightly compared to the physics and core engines.

The physics engine was the last engine I implemented and by far the most troublesome. One of the biggest tasks was efficient collision detection. The implementation of the octree was a big help in that regard however I feel that it's still not as fast as it could be. On top of that not all the colliders work properly and if I had had more time perhaps I could have fixed them, or perhaps there is a flaw in my implementation that I don't see. In the next iteration I feel it would be wiser to keep things simple. In this system I tried to account for too many types of collider before I even got 1 type working. The next iteration of the physics engine would have to start with basic 2D colliders such as squares and circles before moving on to their 3D counter parts. If in the future I managed this, I could perhaps move on to more complex convex shapes. The collision resolution is also another part that took a while to get right. It's been difficult to debug properly as I feel that any issues with it are down to the colliders not working rather than the resolution not working.

The rigid bodies were simple enough to implement however a future version might see the Runge Kutta integration used instead of the Hamiltonian, just to gain experience with other approaches.

Unfortunately I can't give my conclusion on the audio engine as I never got the chance to code it.

In terms of development skill I can only say I've doubtlessly improved. Before this project I didn't know any C++ and I feel that if you're going to get familiar with a language then writing 1 big piece of software from scratch and getting it to work is a good way to learn. The same

can be said for Lua, prior to this undertaking I knew nothing about this language. Now that I've implemented support for it and learned more about it I can see it being a major focus in the next version of this system. I've also become much more familiar with OpenGL and GLSL and how they all work together. I have become more comfortable with the maths a game engine requires: vectors, quaternions and matrices, which will only lead to a more elegant implementation in future versions. All in all this entire project has been nothing but a learning experience for me. I do regret not getting everything that was planned finished but I don't think that this is a case of me having "bitten off more than I can chew", rather a case of not having enough time to "chew it".

I can only conclude that despite the short comings, I'm glad that I chose a system that was complex, unfamiliar and notably different to all the other projects.

Bibliography

[1] Ben Cloward. *Offset Mapping Shader*. URL: http://www.bencloward.com/shaders_offset.shtml.

[2] Jon Brodkin. *How Unity3D Became a Game-Development Beast*. Dice News. URL: http://news.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/ (visited on 12/01/2014).

[3] Christer Ericson. *Real-Time Collision Detection*.

[4] *Component . Decoupling Patterns*. URL: http://gameprogrammingpatterns.com/component.html.

[5] Eric Lengyel. *Computing Tangent Space Basis Vectors for an Arbitrary Mesh (Lengyel's Method)*. URL: http://www.terathon.com/code/tangent.html.

[6] Etay Meiri. *Shadow Mapping*. URL: http://ogldev.atspace.co.uk/www/tutorial23/tutorial23.html.

[7] Fletcher Dunn. *3D Math Primer For Graphics And Game Development (Wordware Game Math Library)*.

[8] Glenn Fiedler. *Fix Your Timestep*. URL: http://gafferongames.com/game-physics/fix-your-timestep/.

[9] Graham Sellers. *OpenGL SuperBible: Comprehensive Tutorial and Reference (6th Edition)*. ISBN: 978-0321902948.

[10] *http://benchmarksgame.alioth.debian.org/u64/lua.php*.

[11] *https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/index.html*.

[12] Ian Millington. *Game Physics Engine Development*.

[13] *John Kessenich, LunarG. The OpenGL Shading Language Specification (20-Jul-2014)*.

[14] Keith Maggio. *Math Magician – Lerp, Slerp, and Nlerp*. URL: https://keithmaggio.wordpress.com/2011/02/15/math-magician-lerp-slerp-and-nlerp/.

[15] *lua.org/about.html*. URL: http://www.lua.org/about.html.

[16] *Mark Segal, Kurt Akeley. The OpenGL Graphics System. A Specification. (Version 4.0 (Core Profile) – March 11, 2010)*.

[17] *OpenGL Pipline*. URL: https://www.opengl.org/wiki/Rendering_Pipeline_Overview.

[18] *Simon Green. NVIDIA OpenGL Update GDC 2006*.

[19] *The Unreal Engine's Rise to Video Game Fame*. Popular Mechanics. URL: http://www.popularmechanics.com/technology/gadgets/video-games/how-the-unreal-engine-became-a-real-gaming-powerhouse-15625586 (visited on 12/01/2014).

[20] Kyle Wilson. *Game Object Structure: Inheritance vs. Aggregation*.

[21] Xichun Jennifer Guo. *Phong Shading and Gouraud Shading*. URL: http://www.nbb.cornell.edu/neurobio/land/OldStudentProjects/cs490-95to96/guo/report.html.