BSc in Computer Science

School of Computing, Science and Engineering

Final Year Project Report

Comparison of OpenGL and Metal Graphical APIs

Author: [redacted]

Supervisor: Norman Murray

2018-2019

Abstract

In 2014, Apple introduced their own proprietary graphical API called Metal. In 2018, Apple deprecated OpenGL making Metal the only graphical API supported on Apple devices. This caused lots of criticism from developers. Therefore, this work sets out to compare OpenGL and Metal on Apple devices in order to find out how these two APIs compare with each other and whether the switch from OpenGL to Metal is beneficial. In order to answer this question each API was used to create a 3D scene with an identical set of graphical techniques. A performance comparison was then carried out by profiling both scenes and analysing their CPU, GPU usage and their frame rates. Secondly, both APIs were compared in terms of the difficulty of coding and time-efficiency through code comparisons. It was found out that Metal performs better than OpenGL especially when it comes to CPU utilisation where Metal tends to be 60% to 120% better than OpenGL. Frame rate of the Metal scene was higher as well than framerate of the OpenGL scene. GPU utilisation is inconclusive as there is need for more tests to be done to provide any conclusions. In terms of their coding styles both APIs are similar. Metal code tends to be more structured however longer offering more explicit control than OpenGL code. As both APIs are similar learning one API makes it considerably easier to learn the other as concepts are the same. The similarity of both APIs makes it easy to port one API code to the other.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The aim of this project is to compare the Metal and OpenGL graphical APIs for MacOS and iOS. This would be carried out through creating two exactly the same graphical scenes showcasing a set of rendering concepts in both APIs and then comparing the efficiency, user-friendliness, complexity, timeframe and the output quality of both solutions. This will then lead to the recommendation as to what graphical API is best suited for developing graphical applications for Apple devices.

1.1 Objectives

The following list contains the objectives for this project:

- **OBJECTIVE 1** Learn Metal by reading through the Apple Official Documentation
- **OBJECTIVE 2** Learn OpenGL by watching through the Computer Graphics course on Edx
- **OBJECTIVE 3** Render a scene in Metal using deferred shading which has HDR lighting and contains 3d objects with stencil reflections and shadow mapping, walls rendered using parallax mapping, hundreds of objects rendered using instancing and simulated water surface
- **OBJECTIVE 4** Render a scene in OpenGL using deferred shading which has HDR

lighting and contains 3d objects with stencil reflections and shadow mapping, walls rendered using parallax mapping, hundreds of objects rendered using instancing and simulated water surface

- **OBJECTIVE 5** Compare the performance of both solutions by measuring FPS
- **OBJECTIVE 6** Compare the difficulty of coding of both solutions though code comparisons
- **OBJECTIVE 7** Compare the time-efficiency of both solutions, i.e. time to learn each of them
- **OBJECTIVE 8** Compare the visual quality through a questionnaire

The research into the development in both APIs will be carried out first, followed by the simultaneous implementation of the scenes in both APIs. This will be done in an incremental manner, creating prototypes along the way to quickly test results and evaluate them. The comparison will be done once both scenes are fully created based on the objectives of this dissertation, which will then be followed up by the discussion and conclusion.

1.2 Motivation

The main motivation behind this project is to learn more about the computer graphics and explore it deeper than the university programme allows and to do something different than usual desktop application development and web development which I have already plenty of experience with especially from my year in the industry. The aim is then to use the experience gathered during this project to open better career options and possibly try a career as a game developer, which always require having some kind of background experience in game programming techniques. Also what interests me about this project is its complexity in an area that is for me rather unknown as of yet. I would like then use this opportunity through this project to stretch myself mentally and see how well I am able to cope with this kind of complexity.

1.3 Overview of the Report

This report starts with the literature review that explores the work and foundations done in computer graphics and investigates the importance of shaders for computer graphics programming. The literature review then looks at the possible reasons why Apple introduced their own graphical API and looks at the importance of iOS platform especially giving a reason as to why it is worthy to carry out this project.

This is then followed by laying out the methodology used in this project. This part of the dissertation will include a description of the methodology, tools, standards, languages, algorithms, and techniques that are being used in the project.

The next chapter will discuss the requirements for this project, their justification and the design decisions that will be used during the implementation.

This will be followed by a description of how the requirements were implemented and any issues encountered.

Testing and analysis will be done after both scenes were implemented and comparison will be made to find out how each API is performing.

After that a critical evaluation of this final year project will be carried out reviewing the process and looking at whether all the objectives were achieved.

This report will finish with the conclusion discussing the findings and any suggestions for further work.

Chapter 2

Literature Review

This literature review discusses the developments in computer graphics in order to show the importance of being able to program custom shaders and their effect on the field of computer graphics in terms being able to render realistic 3D graphics scenes. This follows up by looking at the graphical APIs used by Apple devices and the significance of iOS as a mobile platform and the reason why the comparison between the OpenGL and Metal is important.

To avoid any misunderstandings the following definition will be used for shading: "the term shading refers to the combination of light, shade (as in shadows), texture and color that determine the appearance of an object." (Hanrahan and Lawson, 1990). Whereas, a shader will be defined as a piece of code that runs directly on a video card specifying what effects will be applied to objects in a scene, i.e. specifying what kind of shading should be applied (Hergaarden, 2011).

2.1 Computer Graphics

At the dawn of computer graphics there were not any standards or protocols that would govern the creation of graphical applications. Each video card manufacturer supplied with their hardware a low-level API to provide access to their cards. It was then up to the individuals to make use of these very proprietary APIs and algorithms to produce any graphical output. The issue with this approach was portability as all these APIs were tied to a single

vendor, not compatible with any other manufactures.This then resulted in having to write a specific code for each specific platform (Bailey and Cunningham, 2011).

This frustrating and extremely inefficient way of developing graphical applications then led to efforts to create and define graphics standards that would be supported across several different devices providing the portability missing from the first approach (Bailey and Cunningham, 2011).

The movement to create graphical standards was accompanied by the vast advancements on the hardware field as well. The 3Dfx Voodoo graphics card was introduced in the year 1995 and was considered a first consumer-level 3D hardware accelerated graphics card marking a shift from an inefficient software rendering to hardware rendering making a real-time 3D scene rendering a possibility on personal computers (Bailey and Cunningham, 2011; St-Laurent, 2004).

Even though the hardware performance and standards were advancing and improving there was an issue. The quality and creativity of graphics applications was restricted by so-called fixed function pipeline that the new standards provided. These new open standards only provided a predefined set of limited functions and a fixed shading model. This resulted in restricting developers in how they could affect the rendering to suit their needs resulting in synthetic-looking graphics. (Bailey and Cunningham, 2011; St-Laurent, 2004).

2.1.1 Shaders

There was a discontent with this fixed functionality among researches and developers alike. Lastra et al. (1995) argues that the goal of computer graphics should be "directed towards making computer-generated images appear as realistic as possible." To achieve this goal however was not possible with fixed function pipeline as a wide variety of surfaces cannot be represented with a limited number of functions and capabilities (Cook, 1984; Whitted and Weimer, 1982; Harrahan and Lawson, 1990)

Cook (1984) says that artistic control is desirable as the computer graphics become more sophisticated and proposes a new way for influencing the shading of the geometry through what he calls a shader tree that provided a way of controlling a light source, surface reflections and atmospheric effects through simple expression trees that were parsed and executed to

influence the shading. The work by Cook (1984) is described by Olana and Lastra (1998) as the foundation for writing custom shaders upon which further advancements were done. Cook's (1984) work implies that the more freedom there is in creating the graphics the more realistic the rendered scenes look.

Another attempt to customise shading was done by Perlin (1985) who created a Pixel Stream Editor that was used to create effects such as clouds, fire water, stars and other similar phenomena. The Shader Tree by Cook (1984) was a good foundation but the criticism was that the set of expressions was rather small (Olana and Lastra, 1998; Perlin, 1985). The Pixel Stream Editor improved upon this work by providing a full high level programming language to influence the rendering output, providing control structures such as looping and conditional statements, logical and mathematical operators, and the ability to carry out asynchronous operations.

Hanrahan and Lawson (1990) took the inspiration in both the Shader Trees and the Pixel Stream and tried to improve upon this work by introducing a new shading language. Hanrahan and Lawson (1990) points out the ineffectiveness of the Pixel Stream by saying "In the pixel stream model, shading is a post-process which occurs after visible surface calculations. Unfortunately, this makes his language hard to use within the context of a radiosity or ray-tracing program, where much of the shading calculation is independent of surface visibility." The language presented in this paper was introduced as an algorithm-independent and platform-independent language making it suitable for any implementation. The language ended up being introduced in RenderMan, a 3D graphics tool used by Pixar for the creation of their movies.

All the mentioned work so far shows the attempt and aim from the researches to move from the fixed function pipeline of computer graphics to one that is highly customisable and provides a high level language that can be used to influence the rendering in much richer ways than the fixed function pipeline could ever do.

The first attempt to use programmable shaders in a real-time application was done by Rhoades et al. (1992) where a real-time procedural texture mapping was done in the Pixel-Planes 5 engine.

Lastra et al. (1995) then introduces a prototype shading architecture called PixelFlow

and argues that the advancements in the hardware performance are now good enough to start using custom shaders in real-time environments creating a breakthrough for computer graphics which then leads to many shading languages and standards for desktops being created. There was Nvidia that worked on the Cg shading language. There was Microsoft with its Direct3D that developed HLSL language. There was OpenGL with its GLSL shading language. (Olano and Lastra, 1998).

The significance of the research into and the introduction of programmable shaders in computer graphics is obvious as they act as a tool in order to create as realistic graphics as possible that match the vision of the programmers. Firstly, this is important as Cook (1984) says "some applications require a high degree of realism as an end in itself." Secondly, this is important for game developers as being able to fully customise and program shaders in any way possible results in being able to create unique graphics that can differentiate a game or a graphical application from its competitors. Research conducted by Consumer Electronics Association claims that 75% of players consider graphics important when buying a game (Usher, 2014). This is then backed up by the result of the study on the role of structural characteristics in video-game play motivation carried out by Westwood and Griffiths (2010) suggests that out of 6 different types of gamers, 4 regard the quality of graphics as the most important aspect of gaming.

2.2 OpenGL and Metal

The push to use shaders in real-time rendering and on consumer desktops lead to the evolution of OpenGL to support programmable shaders in version 4.0, which has been a widely supported version by Apple on all of their devices. However, that ceased to be the case in 2014 when Apple decided to introduce their own graphical API called Metal. According to Apple (2016), Metal is described as "a framework supports GPU-accelerated advanced 3D graphics rendering and data- parallel computation workloads. Metal provides a modern and streamlined API for fine-grained, low-level control of the organisation, processing, and submission of graphics and computation commands, as well as the management of the associated data and resources for these commands. A primary goal of Metal is to minimise the

CPU overhead incurred by executing GPU workloads."

However, Metal did not become significant until 2017 when the second version was introduced introducing a plethora of new features matching and even trying to go beyond what OpenGL does. This was a significant step that came together with Apple introducing their own in-house designed GPU in their latest iPhones and together with this they deprecated OpenGL. (Begbie and Horga, 2018).

Dilger (2017) argues that the reason behind Apple adopting their own proprietary graphical API instead of the widely used OpenGL is the ability to be in control of every aspect of their devices. Apple develops their own OS, they design their own CPUs, they design their own GPUs and now have their own graphical API as well. Dilger (2017) further argues that the reason for doing this is the aim for better performance and optimisation through controlling every aspect of their ecosystem as this allows them to "customise iOS (and later macOS and tvOS) development to the subset of graphics capabilities that Apple actually shipped, throwing away lots of irrelevant overhead to radically optimise how efficiently Metal code could use available GPUs."

2.3 Summary

There are over one billion active iOS devices worldwide according to official Apple press information. The global gaming market is supposed to reach 115 billion dollars in 2018 worldwide, with 50 billion dollars coming from mobile games (MediaKix, 2018). The App Store is the most profitable mobile store where people spend almost twice as much as compared to Google Play (Perez, 2018). The A12 chip in the newest iPhones is on par with CPUs in laptops, outperforming Skylake-based chips in MacBooks when it comes single core performance (Owen, 2018). Needless to say that the A12 chip completely outperforms any Android competition as well. On Geekbench the A12 chip scores 11,515 points as compared to Samsung Galaxy Note 9 which scores 8,876 points (Smith, 2018).

The conclusion of this is simple. The new API introduced by Apple is a significant step for developers. The performance of iOS devices now allows programmable shaders to be used on mobile devices and to create graphics that were not possible before. The superior

market share of iOS devices and the importance of graphics in video games and applications means that Metal is something developers should not ignore.

However, due to these changes being recent there is a knowledge gap when it comes to the comparison of the still available OpenGL and the modern Metal and this is the void this dissertation is trying to fill - to look at how these two APIs compare with each other and what is the viable route for developers to take, whether it is worth it for them to stay with OpenGL or whether the new features and capabilities of Metal are worth of spending time learning it.

Chapter 3

Methodology

The aim of this project is to provide a comparison of the OpenGL and Metal graphical APIs through the creation of graphical scenes using both APIs in order to find out an ideal API for the development of graphics applications for iOS and MacOS. This aim generates several questions that will need to get answered before the project can be carried out.

Firstly, are there shaders that cannot be implemented in both APIs? This needs to be sorted as we want to create a comparison that is fair and just. We to have both scenes containing same or similar effects as that would show how well both APIs compare under the same conditions.

Secondly, what is an appropriate development methodology for this project? A development methodology needs to be chosen in order to provide a good framework that would allow the development to be quick and flexible in order to be able to evaluate the progress quickly and be able to react to changes quickly.

Thirdly, what development environments and programming languages to use to create scenes that would result in similar and fair conditions for both APIs? We want to use possibly same tools and same programming languages with both APIs to again allow for fair and just comparison.

Fourthly, how can we analyse the performance of both APIs? This is important as the performance of applications is crucial to provide a good usability and good user experience. Poor performance can result in lost revenue, reduced competitiveness and bad customer relations.

Fifthly, what effective qualitative methods can we use to measure the visual quality of both scenes? Even though the performance is important and crucial, the visual quality is as important. Having an API that is slightly faster but the resulting quality is considerably inferior should not probably result in recommending it.

The hypothesis as of now is that Metal should perform fairly better than OpenGL. This is based on the fact Metal is a lower-level custom API made specifically for Apple chips. The assumption is this should result in better optimisation hence applications running faster. Though the low-level approach of Metal might add an extra complexity from the programming point of view resulting in longer development.

3.1 Shaders and Effects

Unlike OpenGL Metal does not provide any geometry shaders therefore these will not be used in any of the scenes.In case of OpenGL a type of shaders that is missing from its set are compute shaders which are present in Metal. None of these shaders hence will be used in the scenes generated by both APIs. However, the lack of these in each API might be taken into the consideration during the comparison process to see whether the lack of or inclusion of these have any effects on the performance or graphical output quality of these APIs.

As the objectives state both scenes will use deferred shading, HDR lighting, stencil reflections, shadow mapping, parallax mapping, 100 objects rendered through instancing, reflecting water surface. This was a purely arbitrary choice. The aim was to create two exactly the same or as similar as possible scenes. This set of effects could include anything else or omit some of the effects as long as both scenes are the same or similar. The goal was to choose a set of effects in order for the scenes to be hardware demanding in order to have enough data to compare the performance and visual quality of both solutions.

3.2 Development Methodology

Agile methodology will be the methodology of choice for this project. The agile methodology is focused on developing software in an iterative and incremental way (Sakolick, 2018). This

will go hand in hand with the objective 3 and objective 4 that specify that scenes are going to consist of several independent effects allowing to break this project into small deliverables. Implementing each effect in each of the APIs will be one independent iteration allowing to quickly assess the progress of the project and whether the result is being aligned with the aim of this dissertation.

The agile methodology allows a fast development of software while being able to react to changes quickly (Sakolick, 2018). there is a possibility of technological issues arising when implementing the effects in either of the APIs. Maybe something will not be possible in one of the APIs, maybe some of the OpenGL functions are not supported anymore by Apple. Being able to do the quick iterations followed by an evaluation will spot such issues quickly.

3.3 Development Environment

As this is a project aimed at developing for Apple devices there is not many options as to what tools to use. Mac OS is needed for any compilation work and cannot be substituted by any other operating system. The environment used for the development in Metal is going to be Xcode as a preferred IDE for this project. As Metal is Apple's own API, no multi-platform development will be done in this project making Xcode a perfect IDE for this purpose as it was developed by Apple supporting all their technologies and containing all the SDKs and libraries needed to develop in Metal. There are alternatives in terms of using different code editing tools such as Visual Studio Code or Netbeans products, however this boils down to personal preferences and does not offer any performance advantages as for both of these alternatives Xcode still needs to be installed as it provides all the necessary libraries and SDKs. There are no alternatives as to how Metal is implemented, Apple's implementation of Metal is the only one.

The same preferences will apply to the development of the OpenGL scene. The Xcode will have to be used as it again provides an implementation of OpenGL and all the necessary libraries. We want to ensure the environments for the development in both APIs is same or as similar as possible to allow for a fair comparison.

There is an alternative for both of these APIs to actually develop and write shaders in

some of the gaming engines. One example of this being Unity which uses either OpenGL or Metal for its shaders on Apple devices and that can be written in its own high-level shader language which then gets compiled to one of those APIs. However, this is not an ideal solution as the compilation between the shader language used by Unity into a native shader language used by Metal or OpenGL could result in a poorly optimised code possibly having the poor impact on the results of this dissertation.

3.4 Programming language

Regarding the actual shader writing there is only one option for each API. The Metal uses a shading language called Metal Shading Language or MSL that is based on the C++ 14 language, which will be used in this project. The OpenGL uses for its programmable shaders GLSL or OpenGL Shading Language which is based on the C language, which will be used in this project. Both of these APIs support and provide only one type of shading language, hence there are no other options to consider in order to directly write shaders in any of these APIs.

Another consideration that needs to be done is the type of language that is going to be used to configure and set up both APIs in the project. In order for any shading to be done, both APIs need to be initialised in the application. Currently, Apple supports two languages to develop applications for their products - Objective-C and modern Swift. The interoperability is well supported between these two languages and the development for iOS and MacOS devices heavily relies on this interoperability (Developer.apple.com, 2018). As the Objective-C has been used for the past 30 years there is still a dependency on lots of the libraries that are only written in Objective-C (Developer.apple.com, 2018). For this reason both of these languages will be used in this project as some of the Objective-C libraries will probably need to be used.

The is a choice though between Swift and Objective-C when initialising the APIs in the startup method. As of now the choice is to use Swift to initialise both APIs. The development of applications in Swift is considered generally to be faster than in Objective-C (Popko, 2018; Williams, 2018). This is due to its higher-level nature, automatic memory

management and cleaner syntax which makes programming more efficient (Popko, 2018; Williams, 2018). However quite surprisingly, the performance of Swift seems to be better than the performance of Objective-C, quantitatively it seems to be 3-4x faster (Squires, 2014; Kirichok, 2015). As a result of this Swift will be used to initialise both APIs and for any other cases that are not dependant on using Objective-C libraries.

3.5 Measuring the Performance

The is a choice though between Swift and Objective-C when initialising the APIs in the startup method. As of now the choice is to use Swift to initialise both APIs. The development of applications in Swift is considered generally to be faster than in Objective-C (Popko, 2018; Williams, 2018). This is due to its higher-level nature, automatic memory management and cleaner syntax which makes programming more efficient (Popko, 2018; Williams, 2018). However quite surprisingly, the performance of Swift seems to be better than the performance of Objective-C, quantitatively it seems to be 3-4x faster (Squires, 2014; Kirichok, 2015). As a result of this Swift will be used to initialise both APIs and for any other cases that are not dependant on using Objective-C libraries.

No external tools will be used to measure the performance as the Xcode provides enough of the profilers to measure the information needed for this dissertation. Though there is a possibility as the research progresses that need will arise to use some external tools for this purpose.

3.6 Measuring the Visual Quality

There are now ways to measure the quality of graphics applications through objective and automatic metrics (Aydin et all. 2010; Lavou and Mantiuk, 2015). Aydin et al. (2010) argues that subjective measurement of video quality is a long, expensive and laborious process especially when used on a large sets of data. However, in the case of this dissertation this will not be necessary and is out of the scope. Firstly, all that is needed to be evaluated is the quality of only two scenes, hence there will not be a large data set. Secondly, as the target

are primarily mobile devices we are more interested in the subjective opinions of potential users and buyers than in the scientific way of measuring video quality.

Mantiuk et al. (2012) present and analyse four subjective methods to asses the image quality. There is a single stimulus method where a quality of the image is rated on a scale. There is a double stimulus method where two pictures are presented and are rated on a scale at the same time. Then there is a similarity judgements method where the quality difference represented on a scale is done between two images. And the last but not least, there is a forced choice method where simply a higher quality image is chosen out of the two, which according to Mantiuk et al. (2012) is the most accurate method out of all of them. Mantiuk et al. (2012) state that the forced choice method is also the most time-efficient.

From the description of each of these methods and from the results of the study done by Mantiuk et al. (2012) the method that will be used in this dissertation to asses the visual quality of the scenes is going to be the forced choice method as it nicely fits with the two scenes that will be created and compared. If for any reason this method yields inconclusive results the similarity judgements method will be used to provide more information about what subjects think of the quality of both scenes.

Chapter 4

Requirements and Design

This chapter will discuss the requirements for this project and the justification of why they are essential in order to meet the aims of this work. The second section of this chapter will provide an UML diagram and the discussion of the design decisions made during the implementation.

4.1 Requirements

The aim of this dissertation is to provide a comparison between OpenGL and Metal on the Apple platform, specifically the iOS platform. The requirement therefore is to create two equivalent 3D environments using both APIs which will consist of equivalent graphical techniques. The choice of these graphical techniques was influenced by the target platform of this dissertation which is iOS. As already discussed in the literature review part of this dissertation the iOS's most popular items are games (Taylor, 2018) where hyper realistic graphics are the most important aspects of video games (Usher, 2014; Westwood and Griffiths, 2010). Therefore, the requirements for both of the scenes were tailored towards the techniques that would be used in video games which would result in creating realistic looking scenes.

There are several ways to achieve realism in 3D scenes and a lot of research has been carried out to assess what makes a 3D scene appear realistic to observers. According to Phong (1975), Longhurts et al. (2003) and Frisvad et al. (2007) the level of realism in 3D environments corresponds to the use of light. Creating a model of illumination that imitates

the illumination of the real world as close as possible results in realistically looking scenes. The illumination model then provides a way for displaying such phenomena like shadows which can "fool people into believing that they are looking at a photograph of a real scene" and reflections (Longhurts et al, 2003).

As highly realistic illumination models are computationally expensive and therefore not possible on all hardware, the realism of 3D scenes can be increased through "artistically retouching images to provide details such as curbs, dirt and roughness of surfaces" or to be exact the realism can be increased through the use of textures and materials (Longhurts et al, 2003).

Based on these findings of what makes a 3D environment appear realistic what follows is a discussion of individual requirements.

4.1.1 Requirement 1: Illumination

With illumination being the biggest contributor to the realism of a 3D scene, the first requirement will be that each scene must have an illumination model included. Each scene will contain an illumination model that will approximate the real world illumination.

The most realistic illumination model is achieved through so-called ray tracing. It is a technique where each light ray is simulated and its path is traced. These rays then move around the scene and reflect from objects before this information is taken to create a final image (Caulfield, 2018). However, as this dissertation is aimed specifically at iOS mobile devices, such technique is not practically possible to render on mobile devices in real time. For example, in an application called Path Tracer, rendering a scene with three boxes with ray tracing is very slow.

Figure 4.1 shows the beginning of the rendering in Path Tracer.

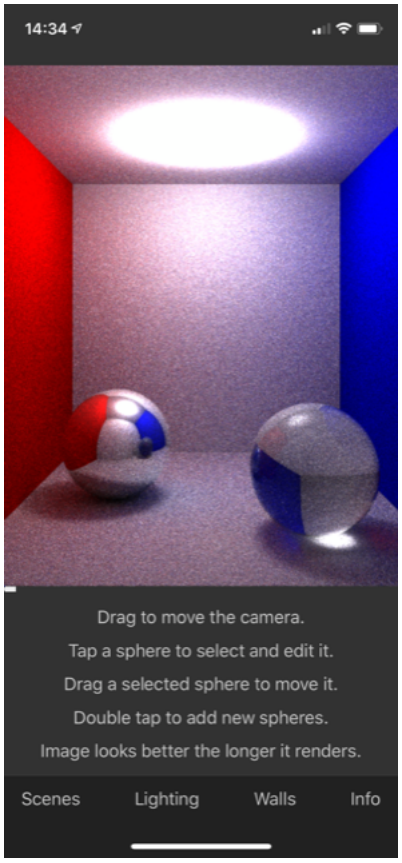


Figure 4.1: *Path Tracer rendering a scene*

Figure 4.2. shows the finished render. The time it took to render this simple scene was exactly one minute. Therefore, this is not very usable on phones as of now.

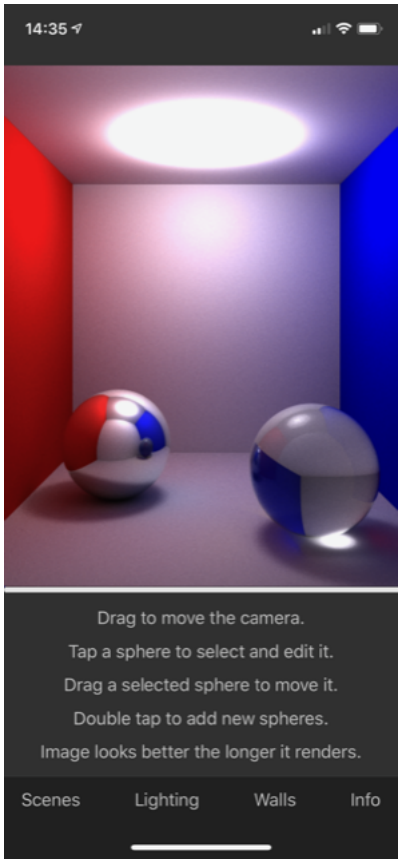


Figure 4.2: *Ray tracing done*

For this reason a Phong's reflection model will be used to provide the illumination for both scenes. The aim of Phong's reflection model is to give a good approximation of the real world illumination (Phong, 1975).

4.1.2 Requirement 2: Shadow Mapping

The second requirement will be the inclusion of shadows which will help with the effort to make the scenes look realistic. Inclusion of shadows helps to add to the authenticity of the scene making people believe they are looking at a real environment (Longhurst et al, 2003). A scene without shadows results in looking flat and not natural. (Everitt et al, n.d.)

An approximation technique will be used to create realistically looking shadows. This technique is called shadow mapping which is achieved through a two-pass rendering in order to render shadows of objects. According to Everitt et al. (n.d.) shadow mapping is a hardware efficient technique that does not put much extra strain on hardware, making it a

good choice for mobile devices.

4.1.3 Requirement 3: Reflection

The third requirement that will help to achieve realism in the scenes is reflection. Reflection, together with shadows and illumination, is another phenomenon that can be naturally found in real-world environments which contributes to make virtual worlds be perceived as realistic (Caliskan and Cevik, 2015).

A reflection mapping will be used instead of ray tracing due to its speed advantage (Greene, 1986). This technique is referred to as environment mapping or reflection mapping.

4.1.4 Requirement 4: Deferred rendering

The fourth requirement that will help to achieve realism in the scenes is deferred rendering. Having multiple lights in a scene adds to the computational complexity and with the traditional way of rendering, lots of inefficient calculations are made due to the situations where light is applied to polygons that will ultimately remain hidden (Mallett and Yuksel, 2018).

Deferred rendering will eliminate these inefficient calculations resulting in developing the ability to render lots of light sources that will ultimately contribute to the scenes looking realistic (Mallett and Yuksel, 2018; Longhurst et al, 2003)

4.1.5 Requirement 5: Textures with Parallax Mapping

The fifth requirement that will help to achieve realism in the scenes is the addition of textures with parallax mapping.

According to Longhurst et al. (2003) a physical environment consists of lots of imperfections which includes phenomena such as dirt, dust and scratches. Different objects also have different materials. Some objects have surfaces that are rough while other objects have surfaces that are smooth. The exclusion of these details in virtual worlds breaks the illusion of realism.

An efficient way to provide these details is through the use of textures that would be wrapped around objects giving an impression of different materials (Amanatides, 1987).

Parallax mapping is just an extension of normal texturing that allows to express elevated surface and is among the techniques that is an inexpensive way to improve surface details contributing to the realism of the scene (Gao et al., 2008).

Figure 4.3 shows that surface elevation effect is achieved without adding any new polygons hence having minimal impact on the performance, whereas Figure 4.4 shows the surface elevation effect that parallax mapping creates.

Figure 4.3: *No extra polygons added*

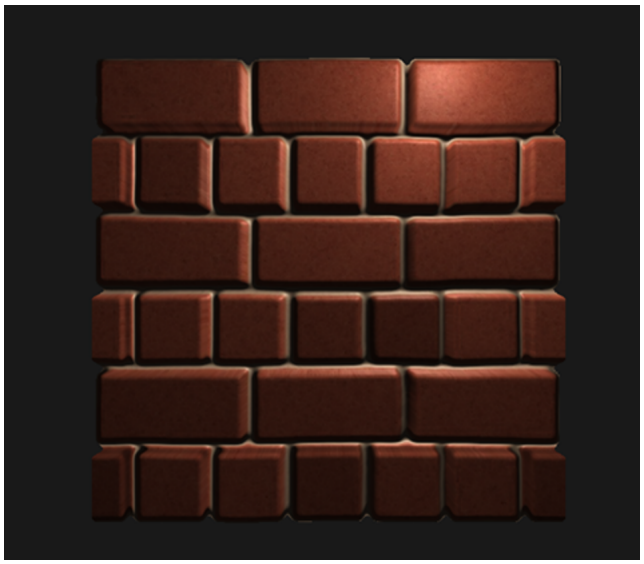


Figure 4.4: *Parallax mapping effect*

4.1.6 Requirement 6: Instancing

The sixth requirement that will help enhance realism is instancing. This dissertation is centered mainly around video game development hence the ability to be able to render lots of game objects is very important. Being able to render a world that is full of enriching

objects such as trees, buildings and plethora of other objects is very important in order to present a credible realistic world that improves the way in which the world is being perceived (Carucci, 2006).

However, rendering hundreds or thousands of objects directly puts a big strain on the GPU. Therefore, instancing is used to achieve rendering of multiple objects in an efficient way (Carucci, 2006).

4.1.7 Requirement 7: Water Surface

The final requirement that aids with the naturalism of a virtual world is creating a water surface. This requirement was chosen as having a water surface is a common natural phenomenon found in virtual worlds. Water simulation provides another entity that enriches realism in 3D scenes (Kryachko, 2006).

Furthermore, simulation of a water surface belongs to a very complex topic in computer graphics (Kryachko, 2006; Tan and Feng, 2008; Li and Xu, 2009). Therefore, it would be interesting to see how each API handles complex simulation such is the water surface.

4.2 Design

As a considerable part of this dissertation is based on experimentation a detailed UML is not possible to create as the knowledge for what will be needed in detail is not known upfront.

However, the initial stages of this work were based on carrying out lots of prototypes to get accustomed with the technologies as quickly as possible. During this stage a pattern that is used when working with both APIs emerged. The building blocks of any graphical application are:

- Camera
- Renderer
- Meshes
- Lights

- Shaders
- Helper classes

These together with the requirement definitions were used to create a UML diagram. The UML diagram only provides a rough layout. The details will differ in both APIs and the level of detail comes with the development of the project.

Figure 4.5 shows the UML design based on the requirements of this dissertation.

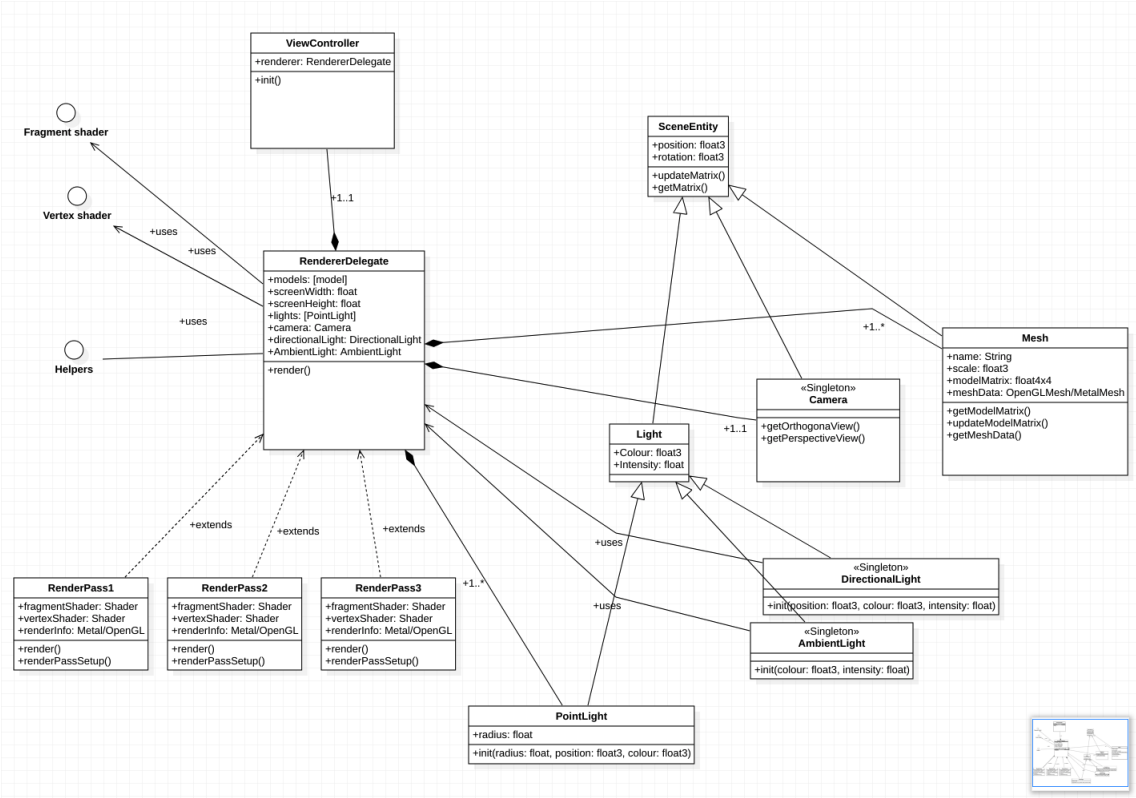


Figure 4.5: *UML diagram*

What follows is the description of the classes in Figure 4.5.

4.2.1 View Controller Class

In order to initialise a window to draw into for both OpenGL and Metal, a delegate pattern must be used. Within Metal, it is the MetalView class that needs to be delegated, whereas with OpenGL, it is the GLKView class that needs a delegate class to be specified. Therefore, the ViewController will inherit from the MetalView or the GLKView and will also serve as an entry point into the iOS application. Furthermore, the ViewController will instantiate a renderer class that will serve as a delegate for either the MetalView or GLKView.

4.2.2 RendererDelegate class

The RendererDelegate class will function as a delegate for either the MetalView or GLKView. The RendererDelegate class will be responsible for rendering everything to the screen. It will contain a list of models to draw, a set of lights to provide illumination and a singleton camera object to view the scene among many other fields such as screen settings and other initialisations that are required.

4.2.3 RenderPass classes

The RenderPass classes will act as extensions of the RendererDelegate class. An extension is a technique in Swift that allows us to split one class over several different files. This will be useful as one of the requirements specifies that multi-pass rendering will be needed to carry out shadow mapping and deferred rendering. This will not result in any performance benefits. However, it will result in code that is better organised and easier to read which will make the debugging and future proofing easier as having all the render passes in the same file would result in code that is thousand lines long making it unreadable.

4.2.4 SceneEntity class

The SceneEntity class will only be an abstract class that will be used by other entities in the application. This class will need to exist as the camera, lights and meshes all need a position, a rotation and matrices built from these two values. Therefore, having one class from which all of classes inherit will reduce code duplication and will allow us to specify methods that will be shared by all these instances.

4.2.5 Light class

The Light class will inherit from the SceneEntity class. A light object will only need a position and a matrix from its parent class. The Light class will specify a colour and an intensity as its two fields. The intensity will specify the contribution made by the specific light towards the overall illumination of the scene.

PointLight, DirectionalLight and AmbientLight classes will inherit from this Light class. Having them in separate classes allows us to force the specific initialisation through their specialised constructors. The DirectionalLight class and the AmbientLight class will be declared as singletons.

4.2.6 Mesh class

The mesh class will represent actual objects in the scene. It will inherit from the SceneEntity class as all the objects in the scene will need to have a position and rotation specified. Additionally, it will have extra fields such as a name, scale and meshData.

4.2.7 Camera class

The camera class will inherit from the SceneEntity class as position and rotation are needed to place the camera in the scene. As there can only be one camera in the scene, the Camera class will only allow a singleton instantiation.

4.2.8 Shaders

Shaders will refer to any collection of shaders.

4.2.9 Helpers

Helpers will refer to all the helper classes needed in both implementations. Helpers will be virtual classes with sets of static functions that will be used throughout the projects.

4.3 Summary

This chapter specified the requirements that will be implemented in both APIs with the effort to make the scenes look realistic. Based on these a UML diagram was created giving a layout to how the requirements should be implemented. However, as lots of the implementation required is unknown as of now a lot is expected to change when it comes to the overall code design.

Chapter 5

Implementation

In this chapter we will discuss how each graphical technique was implemented. A description of each technique is provided together with code samples for both Metal and OpenGL focusing on the main aspects of the implementation. In case of only syntactic differences only a Metal code is provided.

The implementation of the following techniques is going to be described: illumination, instancing, reflection, shadow mapping, deferred rendering and textures with parallax mapping. Furthermore, the issues encountered during the implementation will be discussed as well.

5.1 Illumination

Illumination is based on Phong's illumination model. The characteristics of the light source location, the observer's location and the specular properties of the object are taken into consideration in order to calculate the illumination intensity. The model consists of three types of reflections - a diffuse reflection, a specular reflection and an ambient reflection. Once these are calculated, they are multiplied by the objects colour to get the final illumination (Phong, 1975).

5.1.1 Diffuse Reflection

Diffuse reflection defines the amount of light an object receives regardless of the viewing angle. Diffuse reflection is calculated through the dot product of the normalised surface normal - a perpendicular vector to an object's surface defining the slope of the surface - and the normalised direction of the light ray (Listing 5.1).

If the light ray is pointing in the opposite direction to the surface normal, the result of the dot product is -1 which indicates that the object's surface should receive the full intensity of the diffuse reflection (Figure 5.1).

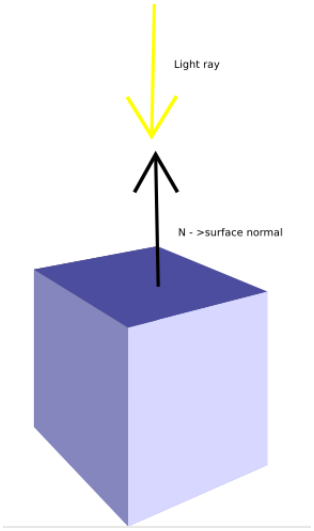


Figure 5.1: *Top surface will receive full diffuse reflection*

If the surface normal is pointing completely away from the light ray the value of the dot product is 1, indicating no diffuse reflection intensity should be received by that particular surface (Figure 5.2).

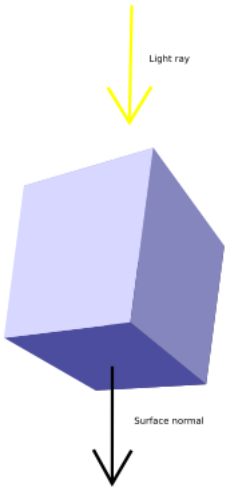


Figure 5.2: *Bottom surface will not receive any light*

The result then needs to be multiplied by the light's colour to attain the amount of diffuse reflection that will contribute to the final surface colour (Listing 5.2).

CODE

```
// norrmalises the values and clamp it between 0 and 1 as negative numbers
// are not used for RGB colours
float diffuseReflectionIntensity =  saturate(dot(normalize(
    normalizedLightDirection),normalize(normalizedNormalDirection)));
```

Listing 5.1: Dot product

```
float3 diffuseReflectionColor = lightColor * diffuseReflectionIntensity;
```

Listing 5.2: Getting the final colour

5.1.2 Specular Reflection

Specular reflection defines the shininess of an object based on the observer's location, in our case the camera.

Specular reflection needs a light ray direction vector and a surface normal to calculate a reflection vector (Figure 5.3 and Listing 5.3).

The reflection vector is then used in the dot product with a direction vector of the camera to find out whether any specular reflection is received by the observer. This is the same process as diffuse reflection - specular reflection will be strongest when the reflection vector and the camera direction vector point towards each other.

The result is then taken to the power of the specular shininess which is a material value provided by the object and multiplied by the material specular colour of the object (Listing 5.4).

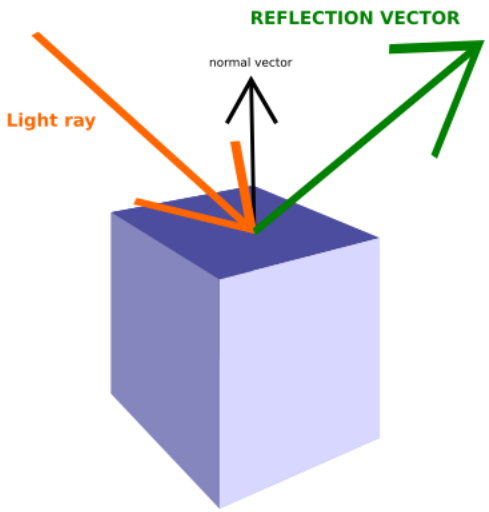


Figure 5.3: *Reflection of the light ray*

CODE

```
float3 reflectionVector = reflect(lightDirection, normal);
```

Listing 5.3: Calculate the reflection vector

```
// gets the dot product and raise to the power of shininess
float SpecularIntesity = pow(saturate(dot(cameraDirection, reflectionVector)),
    materialShininess);
// gets the resulting colour
float3 specularReflectionColor = materialSpecularColour * SpecularIntensity
```

Listing 5.4: Get the intensity and final colour

5.1.3 Ambient Reflection

Ambient reflection refers to the overall illumination in a scene. It works based on the assumption that no surface is ever completely black due to light rays bouncing all around in various directions contributing at least a little bit of light to every surface.

Ambient reflection is defined through an intensity value to define how much of the light's colour should be added to every fragment regardless of its position (Listing 5.5).

CODE

```
1 float3 ambientReflection = lightColour * AmbientIntensity;
```

Listing 5.5: Adding ambient reflection intensity

5.1.4 Result

Figure 5.6 shows the diffuse, specular and ambient reflection effect when applied to an object.

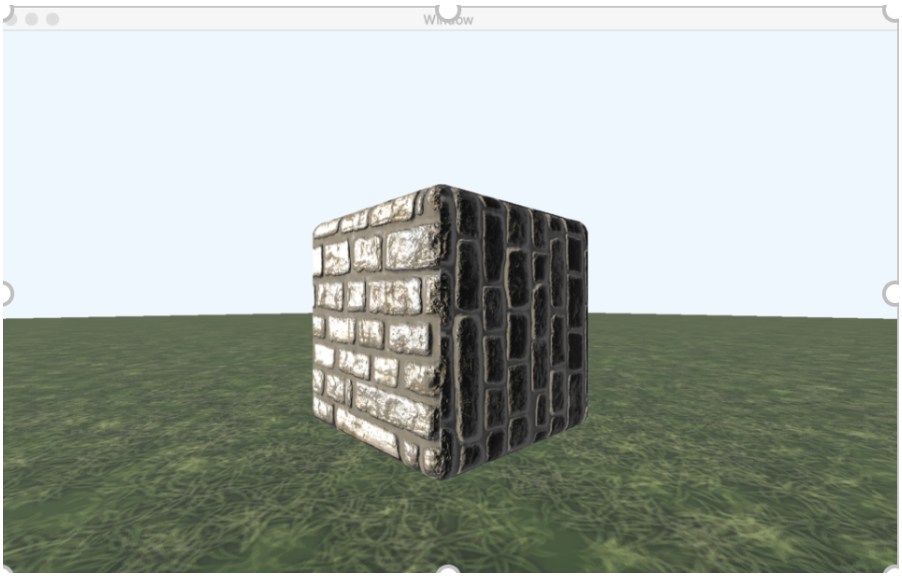


Figure 5.4: *Illimunation model applied to an object*

5.2 Instancing

The second technique that was implemented was instancing which allows to render lots of objects with the same vertices efficiently. Each rendered object is called an instance.

Only one instance of vertex buffer is sent to the GPU at the beginning of rendering. This vertex buffer is then used for every subsequent instance being rendered out.

As each instance would have a different position, an array of these values for every instance is sent to the GPU as well (Listing 5.6 for Metal; Listing 5.8 for OpenGL). This is then accessed using a per-instance identifier in a vertex shader (Listing 5.7 for Metal; Listing 5.9 for OpenGL).

CODE (Metal)

```
// creates a GPU buffer with positions
let instancePositionsBuffer = RenderViewDelegate.device.makeBuffer(bytes:
    instancePositions, length: MemoryLayout<float3>.stride * instances.count)

// sends the buffer to the GPU
 renderEncoder.setVertexBuffer(instancePositionsBuffer, offset: 0, index: Int(
    instancesBuffer.rawValue))
```

Listing 5.6: Sending an array of positions to the GPU

```
vertex VertexOut vertexShader(... ,constant float3 *instancePositions [[buffer
    (1)]],uint instanceId [[instance_id]], ...){
float3 instancePosition = instancePositions[instanceId];
}
```

Listing 5.7: Accessing each instance

CODE (OpenGL)

```
// creates an array of instance positions
let instancePosition : Float3 = [...]

// sends the positions to the GPU
for (index, instance) in instancePositions.enumerated(){
glUniform4f(glGetUniformLocation(renderHandle, "instancePositions[\(index)]"),
    instance.x, instance.y, instance.z, instance.w)
}
```

Listing 5.8: Sending an array of positions to the GPU

```
vec4 instancePosition = instances[gl_InstanceID];
```

Listing 5.9: Accessing each instance

5.2.1 Result

Figure 5.5 shows 1000 cubes rendered using instancing.

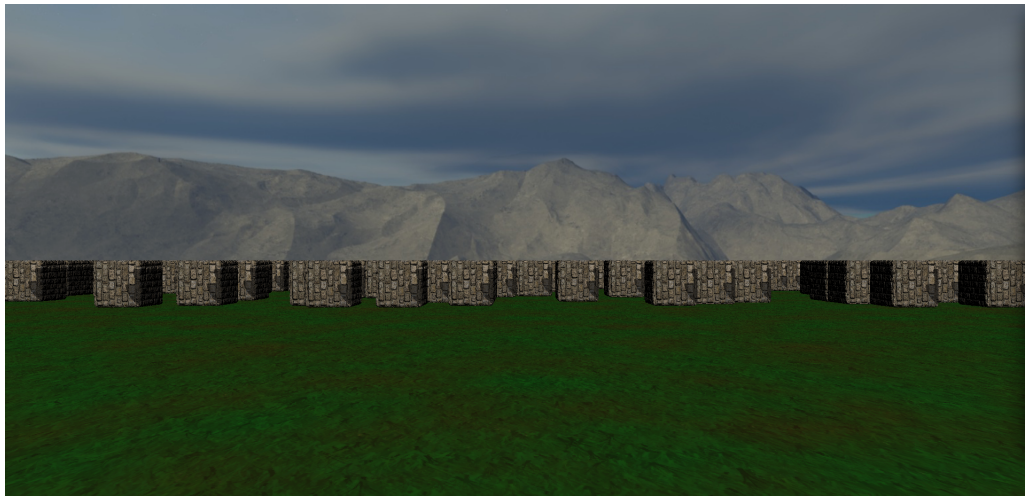


Figure 5.5: *Cubes rendered using instancing*

5.3 Reflection

Reflection was implemented through reflection mapping using a cube texture. A cube texture is a texture that consists of 6 different images where each image represents a different axis (Figure 5.6).

Figure 5.6: *Cube map texture*

A cube map texture is a 3D texture that requires a vector of three components in order to be able to sample it.

In order to create an illusion of reflection a camera direction vector is reflected along with the surface normals (Listing 5.10 for Metal; Listing 5.12 for OpenGL). The resulting reflection vector, which is a three component vector, is used to sample the cube map texture. The samples are then applied to the object, making it look like it is reflecting its environment (Listing 5.11 for Metal; Listing 5.13 for OpenGL).

CODE (Metal)

```
float3 cubeTextureCoordinates = reflect(cameraDirectionVector, normalVector);
```

Listing 5.10: Getting the reflection vector

```
constexpr sampler txtSampler(filter::linear);
float4 cubeTextureSampled = cubeTexture.sample(txtSampler,
    cubeTextureCoordinates);
return objectColor * cubeTextureSampled;
```

Listing 5.11: Sampling and applying the cube texture

CODE (OpenGL)

```
vec3 cubeTextureCoordinates = reflect(cameraDirection, normal);
```

Listing 5.12: Getting the reflection vector

```
vec4 cubeTextureSample = texture(cubeTexture, cubeTextureCoordinates).rgba;
fragmentColor = objectColor * cubeTextureSample;
```

Listing 5.13: Sampling and applying the cube texture

5.3.1 Result

Figure 5.7 shows the cube-map reflection effect.

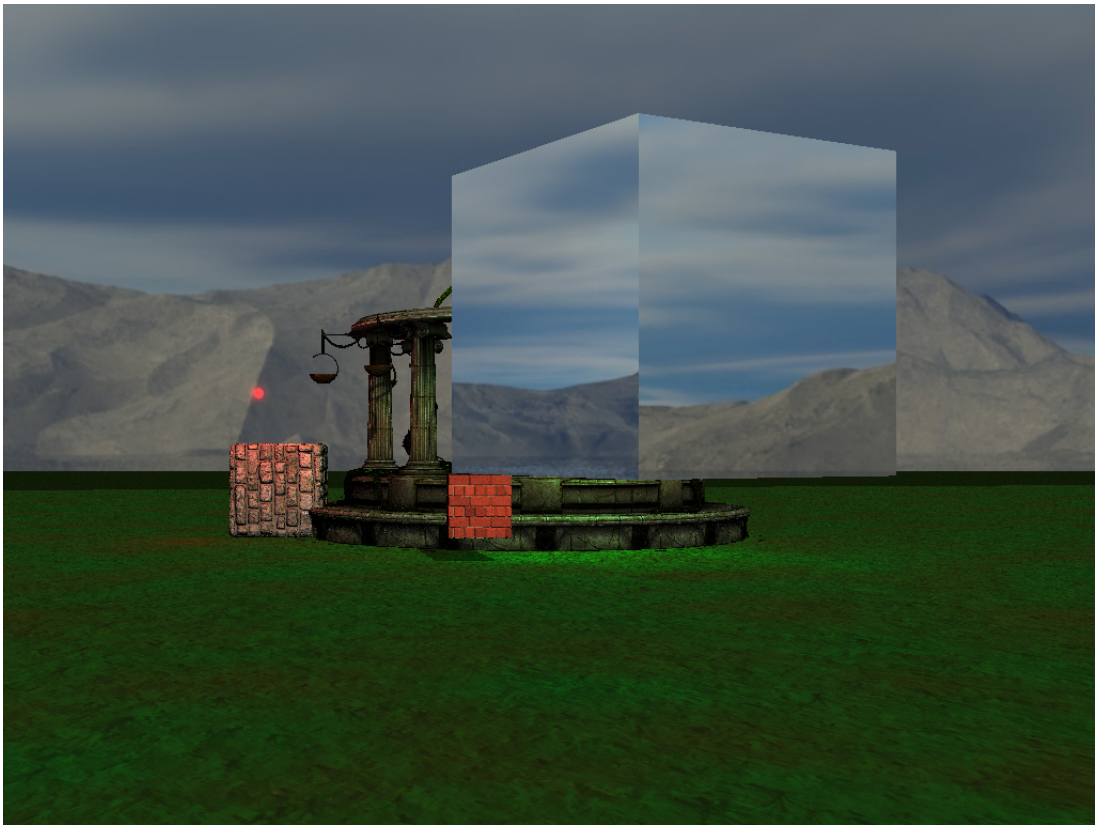


Figure 5.7: *A cube reflecting its environment*

5.3.2 Shadow Mapping

Shadow mapping is achieved through rendering a scene multiple times. In the first render, a scene is transformed as if it was seen from a light source position which is achieved by multiplying all the objects' positions by a light space matrix. This then allows us to create a depth map visualising everything light rays are hitting (Listing 5.14 for Metal; Listing 5.17 for OpenGL).

After that the depth map is sent to the next render pass which is responsible for rendering to the screen. During this process the depth map created in the first pass is sampled to see whether the current fragment that is being rendered is hidden from the light and hence should be covered in shadows. This is done by comparing the Z-buffer value sampled from the depth map with the current Z-buffer value of the fragment. If the fragment's current Z-buffer value is higher than the Z-buffer value from the depth map it means that the fragment should be covered in shadows (Listing 5.16).

In order to be able to get the fragment's Z-Buffer value in the depth map from the first pass, a matrix that transforms the fragment's position into the light coordinate space is needed (Listing 5.15).

CODE (Metal)

```
//creates a depth texture to write to for the first pass
let textureDescriptor = MTLTextureDescriptor.texture2DDescriptor(pixelFormat:
    .depth32Float, width: Int(RenderViewDelegate.device.width), height: Int(
    RenderViewDelegate.device.height), mipmapped: false);
ShadowMappingAttributes.shadowTexture = RenderViewDelegate.device.makeTexture(
    descriptor: textureDescriptor);

// attaches the texture to the first pass
ShadowMappingAttributes.shadowRenderPassDescriptor.depthAttachment.texture =
    ShadowMappingAttributes.shadowTexture;
```

Listing 5.14: Depth texture creation

```
let orthoProjectionForLigth = GLKMatrix4MakeOrtho(-10.0,10.0, -10.0, 10.0,
    view.minClip, view.maxClip);
let lightView = GLKMatrix4MakeLookAt(DirectionalLight.position.x,
    DirectionalLight.position.y, DirectionalLight.position.z , 0.0, 0.0, 0.0,
    0.0, 1.0, 0.0);
let lightSpaceMatrix = GLKMatrix4Multiply(orthoLightProjection, lightView);
```

Listing 5.15: Creating a light space matrix

```
float shadow_sample = shadow_texture.sample(s, txtCoordinatesLightSpace);
float current_sample = in.shadowPosition.z ;

if(current_sample > shadow_sample){
        out.albedo.a = 1;
}
```

Listing 5.16: Sampling and comparing Z-Buffers

CODE (OpenGL)

```
// creates a depth texture
glGenFramebuffers(1, &depthMapFBO);

glGenTextures(1, &depthMapTexture);
glBindTexture(GLenum(GL_TEXTURE_2D), depthMapTexture);

glTexImage2D(GLenum(GL_TEXTURE_2D), 0, GL_DEPTH_COMPONENT32F, GLsizei(width),
    GLsizei(height), 0, GLenum(GL_DEPTH_COMPONENT), GLenum(GL_FLOAT), nil);

// Attaches the texture
glBindFramebuffer(GLenum(GL_FRAMEBUFFER), depthMapFBO);
glFramebufferTexture2D(GLenum(GL_FRAMEBUFFER), GLenum(GL_DEPTH_ATTACHMENT),
    GLenum(GL_TEXTURE_2D), depthMapTexture, 0);
```

Listing 5.17: Depth texture creation

5.3.3 Result

Figure 5.8 shows the result of shadow mapping.

Figure 5.8: *An object casting a shadow*

5.4 Deferred rendering

Deferred rendering builds on the technique used in shadow mapping - rendering a scene using several passes. In this case three render passes are used to carry out deferred rendering.

The first pass is still responsible for rendering the depth map used for shadow calculations as before. However, the second pass is now rendering into 3 pre-defined textures (Figure 5.9) instead of rendering to a screen. One of these textures is used to store position information, the second texture is used to store normals and the third texture is used to store colour information (Listing 5.18 for Metal, Listing 5.21 for OpenGL). The second pass is called a G-buffer pass (Listing 5.19).

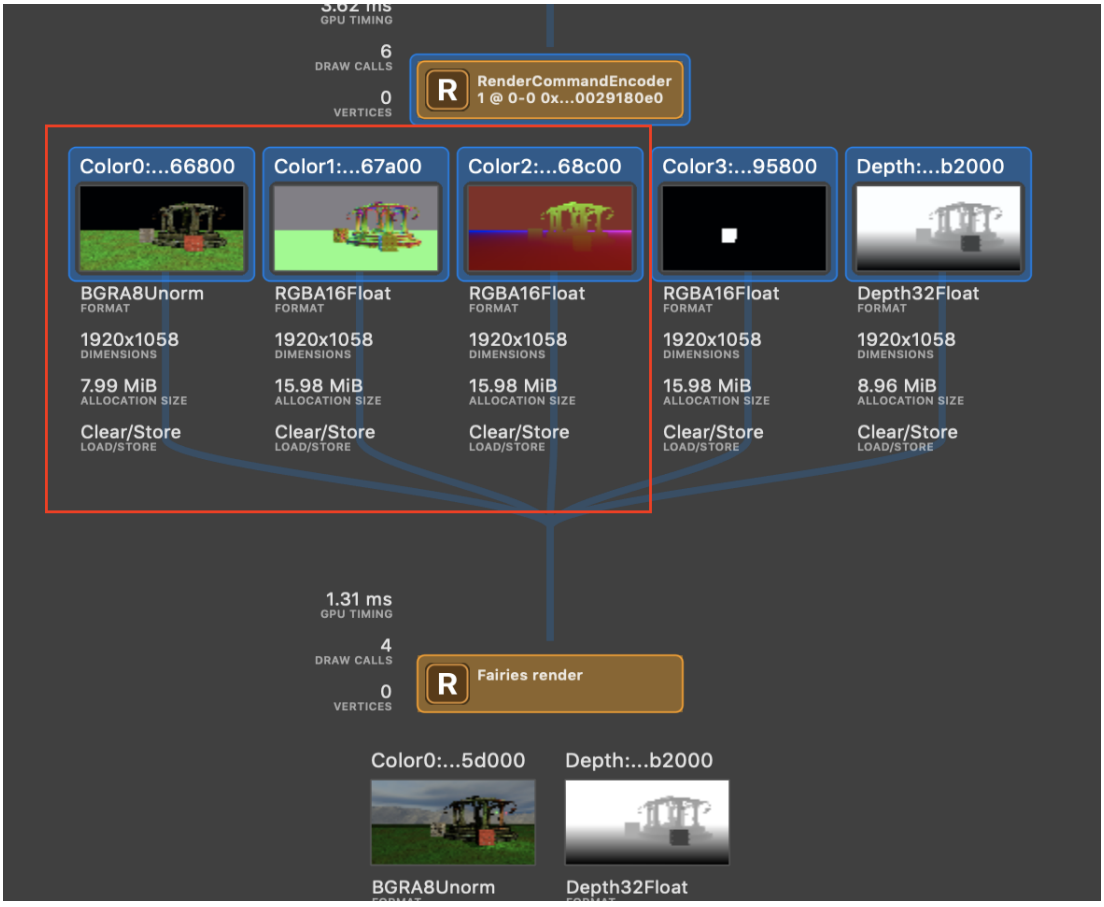


Figure 5.9: *The three textures created in the G-Buffer pass going into the composition pass*

These 3 textures are then sent into the third pass where they are sampled and the samples are written into a screen texture (Listing 5.20). The third pass is called a composition pass.

The main difference here is that illumination calculations are carried out in the third pass instead of the second pass and the samples from the textures are plugged into the calculations. This means that illumination is only carried out for objects that will end up on the screen resulting in better performance as no calculations are done for hidden fragments.

CODE (Metal)

```
gBufferPassAttributes.colorMap.setTexture(texture: MeshHelper.
    create2dTextureForRendering(pixelFormat: gBufferPassAttributes.colorMap.
    pixelFormat, size: size, storageMode: .private))

gBufferPassAttributes.normalMap.setTexture(texture: MeshHelper.
    create2dTextureForRendering(pixelFormat: .rgba16Float, size: size,
    storageMode: .private))

gBufferPassAttributes.positionMap.setTexture(texture: MeshHelper.
    create2dTextureForRendering(pixelFormat: .rgba16Float, size: size,
    storageMode: .private))

// Attaching the textures to the G-Buffer pass

gBufferPassDescriptor.colorAttachments[0].texture = gBufferPassAttributes.
    colorMap.texture;

gBufferPassDescriptor.colorAttachments[1].texture = gBufferPassAttributes.
    normalMap.texture;

gBufferPassDescriptor.colorAttachments[2].texture = gBufferPassAttributes.
    positionMap.texture;
```

Listing 5.18: Creating G-Buffer textures

```
// write colour information based on whether an object has a texture or not
if(hasColorTexture)
{
    out.colourTexture = float4(baseColorTexture.sample(textureSampler, in.
    textureCoordinate * int(*tiling)).xyz, 1.0);
```

```
}
else
{
    out.colourTexture = float4(material.colour, 1.0);
}


// write normal information based on whether an object has a normal texture or
    not
if(hasNormalTexture)
{
  float3  normalValue;
  normalValue = normalTexture.sample(textureSampler, in.textureCoordinate *
   int(*tiling)).xyz;
  normalValue = normalValue * 2 - 1;
  normalDirection  = float3x3(in.worldTangent, in.worldBitangent, in.
   worldNormal) * normalValue;
}
else
{
    normalDirection = in.worldNormal;
}

out.normalTexture = normalDirection;


// write position information
out.positionTexture = float4(in.fragmentPosition, 1.0);
```

Listing 5.19: Rendering into the textures

```
// fragment body
float4 colourTexture = albedoTexture.sample(s, in.texCoords);
float3 normalDirection = normalTexture.sample(s, in.texCoords).xyz;
float3 fragmentPosition = positionTexture.sample(s, in.texCoords).xyz;
float3 baseColor = albedo.rgb;

```

```
// illumination calculation as explained in the Illumination section
```

Listing 5.20: Sampling the textures in the composition pass

CODE (OpenGL)

```

// generate buffer
glGenFramebuffers(1, &gBufferFBO);
glBindFramebuffer(GLenum(GL_FRAMEBUFFER), gBufferFBO)


////////////////////// POSITION TEXTURE //////////////////////////
glGenTextures(1, &positionTexture);
glBindTexture(GLenum(GL_TEXTURE_2D), positionTexture);

glTexImage2D(GLenum(GL_TEXTURE_2D), GLint(0), GLint(GL_RGB16F), GLsizei(
    Renderer.width), GLsizei(Renderer.height), 0, GLenum(GL_RGB), GLenum(
    GL_FLOAT), nil);

glFramebufferTexture2D(GLenum(GL_FRAMEBUFFER), GLenum(GL_COLOR_ATTACHMENT0),
    GLenum(GL_TEXTURE_2D), positionTexture, 0);

////////////////////// NORMAL TEXTURE ////////////////////////////////
glGenTextures(1, &normalTexture);
glBindTexture(GLenum(GL_TEXTURE_2D), normalTexture);

glTexImage2D(GLenum(GL_TEXTURE_2D), 0, GLint(GL_RGB16F), GLsizei(Renderer.
    width), GLsizei(Renderer.height), 0, GLenum(GL_RGB), GLenum(GL_FLOAT), nil
    );

glFramebufferTexture2D(GLenum(GL_FRAMEBUFFER), GLenum(GL_COLOR_ATTACHMENT1),
    GLenum(GL_TEXTURE_2D), normalTexture, 0);


////////////////// COLOUR TEXTURE ////////////////////////////////
glGenTextures(1, &albedoSpecTexture);
```

```
26 glBindTexture(GLenum(GL_TEXTURE_2D), albedoSpecTexture);
27
28 glTexImage2D(GLenum(GL_TEXTURE_2D), 0, GLint(GL_RGBA), GLsizei(Renderer.width)
       , GLsizei(Renderer.height), 0, GLenum(GL_RGBA), GLenum(GL_UNSIGNED_BYTE),
       nil);
29
30 glFramebufferTexture2D(GLenum(GL_FRAMEBUFFER), GLenum(GL_COLOR_ATTACHMENT2),
       GLenum(GL_TEXTURE_2D), albedoSpecTexture, 0);
31
32 // ATTACH ALL 3 TEXTURES TO THE FRAMEBUFFER
33
34 glDrawBuffers(3, [GLenum(GL_COLOR_ATTACHMENT0), GLenum(GL_COLOR_ATTACHMENT1),
       GLenum(GL_COLOR_ATTACHMENT2)]);
```

Listing 5.21: Creating G-Buffer textures

5.5 Textures with Parallax Mapping

A set of three textures is used to implement parallax mapping. The first texture is a colour texture that gives an object its appearance (Figure 5.10).

Figure 5.10: *Colour texture*

The second texture is a normal texture that affects surface normals to create the effect of roughness (Figure 5.11).

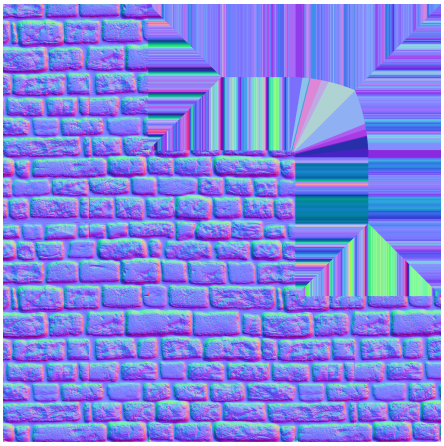


Figure 5.11: *Normal texture*

The third texture is a depth texture that specifies how far to displace texture coordinates to create the effect of displacement (Figure 5.12).

Figure 5.12: *Depth texture*

The idea is then to use the displaced texture coordinates to sample the normal texture and the colour texture which then results in the displacement effect.

The displaced texture coordinates are attained by sampling the depth texture using the original texture coordinates and then multiplying the sampled height with the camera direction vector and subtracting the result from the original texture coordinates (Listing 5.22).

CODE

```
1
2 float2 displacement = viewDirection.xy * depthTexture.sample(s,  in.
    textureCoordinate).g
3 in.textureCoordinate = in.textureCoordinate - displacement;
4
5 // use the in.textureCoordinate to sample the colour texture and normal
    texture as before
```

Listing 5.22: Getting the displaced coordinates

5.5.1 Result

Figure 5.13 shows the effect of parallax mapping.

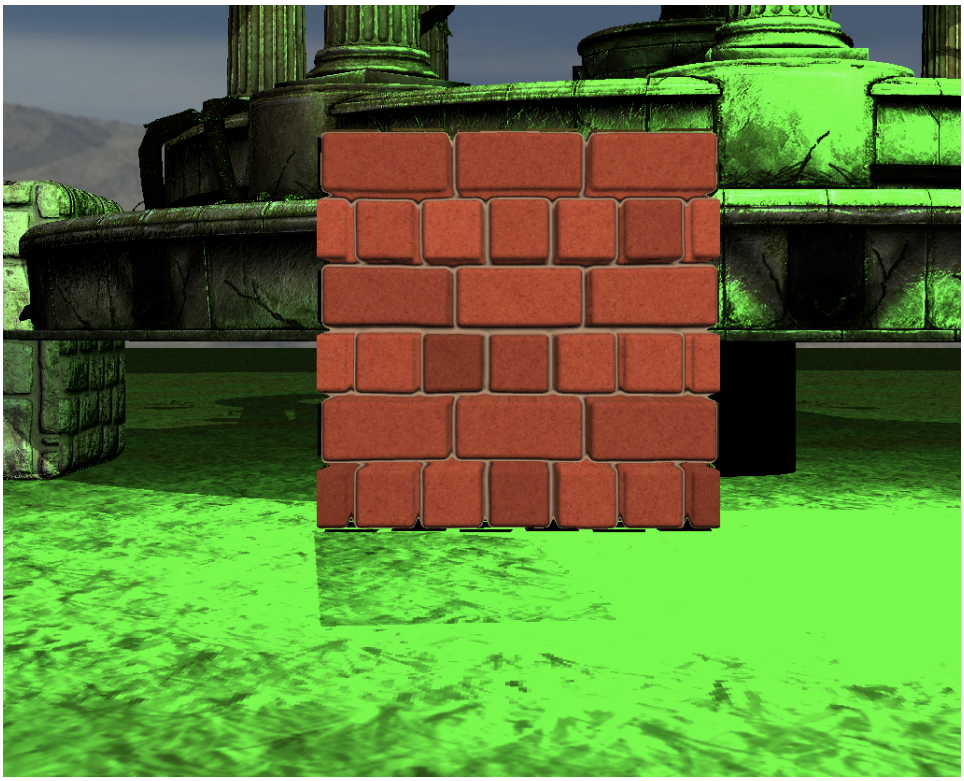


Figure 5.13: *A cube rendered using parallax mapping*

5.6 Issues

This section will discuss the issues encountered during the implementation of both scenes.

5.6.1 Model Loading

Being able to load .obj 3D models into the OpenGL scene proved to be the most challenging part of the implementation.

The GLKit that Apple provides as a helper library to use with OpenGL contains all the necessary mathematical functions, initialisation functions and texture loading functions. However, it has absolutely no functions that would allow loading of 3D models.

To conform to the requirement that both scenes must be the same in order to be able to carry out a fair comparison, it was necessary to be able to use .obj models in the OpenGL scene as well, as that is what Metal defaultly supports.

There were two possible approaches through which this could be performed.

One approach would be to try to parse .obj files and then use it to initialise vertex buffers. The structure of .obj files is not very complicated. However in terms of the time it would take, this was not desirable as it would negatively impact the project's timeframe. Building a parser would create a risk of not being able to spend enough time on the project's objectives.

The second approach would be to try to make use of the Model I/O framework that allows import and export of 3D models which was used when implementing the Metal scene. As there was already experience with Model I/O the second approach was more fitting.

Model I/O Approach

When working with Object I/O during the Metal implementation it was noticed that the interface provides functions to get a memory address of every vertex buffer attribute. The idea was then to make use of the Swift's main advantage which is a well supported direct memory access due to its compatibility with Objective-C through its bindMemory() function that can access a specified memory address and bind it to the specified data type (Figure 5.14).

The plan was to use the Model I/O functions to get memory addresses of all the attributes and then use the bindMemory() function to bind the memory to the data structures used in the OpenGL shaders.

The issue is that the interface does not provide a function to find out how many elements are in each attribute buffer. Hence, there is no way to tell when to stop advancing through the memory. However, it did provide a possible way to solve this just by using a slightly different approach.

Instead of accessing the attributes individually, Object I/O also provides a memory address of the whole vertex buffer that combines the position, normal and texture coordinates attributes together. This whole buffer, unlike the individual attribute buffers, has the overall count necessary for advancing as well. It lacks the size of an individual vertex. However,

bindMemory(to:capacity:)

Binds the memory to the specified type and returns a typed pointer to the bound memory.

Declaration

```
@discardableResult func bindMemory<T>(to type: T.Type, capacity count: Int) ->
    UnsafePointer<T>
```

Parameters

type
The type T to bind the memory to.

count
The amount of memory to bind to type T, counted as instances of T.

Figure 5.14: *A function to bind to a memory location*

that can be calculated by dividing the total size of the buffer by the number of vertices in the buffer.

This then provided the solution (Listing 5.23). The memory address of the vertex buffer allowed us to carry out a memory bind. Advancing was achieved by calculating the vertex size by dividing the overall buffer size by the number of vertices. It was then bound to a data structure that reflected the individual attributes and the loop was stopped once it reached the vertex count.

```
1 // data structure reflecting individual attributes
2 struct Vertex {
3     // position
4     var x: GLfloat;
5     var y: GLfloat;
6     var z: GLfloat;
7 
8     // normals
9     var nx: GLfloat;
```

```
10      var ny: GLfloat;
11      var nz: GLfloat;
12
13      // texture coordinates
14      var tx: GLfloat;
15      var ty: GLfloat;
16
17  }
18
19  // binding process for vertices
20  for index in 0...mdlMesh.vertexCount-1
21  {
22
23  vertices.append( mdlMesh.vertexBuffers[0].map().bytes.advanced(by: index *
        vertexSize).bindMemory(to: VertexTest.self, capacity: MemoryLayout<
        VertexTest>.stride).pointee)
24
25  }
26
27  // get indices -> same process as when getting vertices but
28  // need to acount for submeshes as well
29  for (index, submesh) in (mdlMesh.submeshes?.enumerated())! {
30   let mdlSubmesh = submesh as! MDLSubmesh
31    var indices : [GLuint] = []
32    for index in 0...mdlSubmesh.indexCount-1
33    {
34    indices.append(mdlSubmesh.indexBuffer.map().bytes.advanced(by:  MemoryLayout
       <GLuint>.stride * index).assumingMemoryBound(to: GLuint.self).pointee)
35    }
36
37  }
```

Listing 5.23: Final solution

A very short solution compared to having to write the parser suggested earlier. Another advantage is the ability to make use of the Object I/O framework before doing all the binding. For example, the framework allows us to automatically calculate tangents and bitangets of

any object without having to do this manually.

There might be a performance penalty when loading an object, however run-time performance is not influenced. Initialisation performance does not matter as it is not measured in this project.

5.6.2 Platform Specific Issue

During the implementation of shadow mapping for the OpenGL scene a hardware specific issue was encountered.

When a depth texture is created in the first pass, it is then necessary to switch to the default framebuffer in order to be able to draw into the screen texture in the second pass.

Based on the OpenGL documentation the default framebuffer should have an ID of 0. Switching to the default framebuffer then is performed as in Listing 5.24.

```
1 glBindFramebuffer(GL_FRAMEBUFFER, 0);
2
3 ... // second pass render
```

Listing 5.24: Binding a default framebuffer

This however did not work. A purple screen was displayed instead indicating something is wrong There was no crash, no indication of what could be wrong. No error message was shown.

Research through the OpenGL documentation revealed a method called glGetError() which returns an integer of the last error. It needs to be placed where the error is expected to happen. Therefore, the function was placed right after binding the default framebuffer. The error message “1286” was printed to the console, which based on the OpenGL documentation means “Invalid FrameBuffer Operation” indicating an issue with the currently bound framebuffer.

Another function called glCheckFramebufferStatus() was then used to find out what exactly was wrong with the current framebuffer. The error message in this case printed out “33305” which means “GL FRAMEBUFFER UNDEFINED” or that the “default framebuffer does not exist”.

Before moving on, a static helper function was created to provide a user-friendly description of the provided error (Listing 5.25). This saved time by not having to look up the documentation every time an error occurred.

```
static func getGlErrorDescription(error : GLenum){
        switch (error) {
          case GL_NO_ERROR:
print("description")

 case GL_INVALID_ENUM:
print("description")

case GL_INVALID_VALUE:
print("description")

GL_INVALID_OPERATION:
print("description")

..
// the rest of the enums

}
```

Listing 5.25: Helper error function

The error message indicated the default framebuffer does not exist. This prompted more research in Apple's OpenGL documentation which revealed that iOS does not implement window-system-provided framebuffers and instead a framebuffer object is used to draw into and can be acquired by the .bindDrawable() method.

Therefore, in order to draw into the screen texture instead of calling the glBindFramebuffer(GL FRAMEBUFFER,0) method to get the default framebuffer, the bindDrawable() method needs to be called instead which will bind the Apple's own implementation of the default buffer.

This issue stresses the importance of having to have deep knowledge about the target platform in computer graphics as implementations can differ.

5.7 Summary

This chapter provided a description of how each technique was implemented. The implementation was identical in lots of cases for both APIs. Namingly, the shaders usually only differed in their syntax. The main differences were encountered in how the client's code was initialised in both APIs.

This chapter also discussed the issues encountered. The major issues were experienced during the implementation of the OpenGL scene due to specific problems only encountered on the Apple platform.

Chapter 6

Analysis and Results

In this chapter a comparison between Metal and OpenGL will be carried out. The chapter will begin with a description of how each scene was set up in order to provide the fairest results possible. This will then be followed by profiling each scene in order to find out system resources each scene takes up and framerates each scene is running at.

The second part of this chapter will discuss both APIs in terms of difficulty of coding both solutions through discussing coding styles of both APIs. The time it took to learn each API will be discussed together with this as well.

6.1 Performance Comparison

The performance comparison of both APIs was carried out using Xcode profiling instruments. The values obtained were then compared to find out what API performs better.

Several approaches were used to produce as similar scenes as possible to provide a fair comparison. However, as Metal and OpenGL are two different APIs not everything can be done in an equivalent way.

First approach was to ensure both scenes are rendered from the same camera point capturing the same number of objects. This was achieved through setting camera attributes, projection matrices and objects' attributes with equivalent values. Objects, such as temples and point lights, were randomly generated with the generator set up to render the same amount of objects with the positions fitting all into to the screen height and width.

The second approach was to use visual inspection to ensure both scenes look identical by fine tuning the values. Figure 6.1 shows the values set for scene objects in order to produce similar results.

	OpenGL	Metal
Camera position	(0,0,10)	(0,1,-10)
Projection angle	45*	45*
Near plane	0.1	0.1
Far plane	100	100
Light position	(1,3,11)	(1,4,-11)
Number of lights	50	50
Temple origin position	(0,-1,0)	(0,0,0)
Temple scale	0.001	0.001
Number of temple instances	50	50
Parallax box position	(0,1,0)	(0,2,0)
Reflection box position	(0,0,3)	(0,1,-3)
Reflection box scale	0.5	0.5
Ground position	(0,-1,0)	(0,0,0)
Ground scale	10	10

Table 6.1: *Scene set up for both APIs*

The differences in the values are caused by different coordinate systems used by both APIs. Metal uses a left handed coordinate system with its positive Z-axis pointing into the screen. While OpenGL uses a right handed coordinate system with its positive Z-axis pointing away from the screen.

Furthermore, Apple provides OpenGL with a library called GLKit which offers built-in functions for constructing orthographic and perspective projections and includes other functions for manipulating matrices and carrying out mathematical operations. Such library is not available for Metal forcing a custom implementation of these constructs resulting in slight differences between both scenes.

Both implementations use 3 render passes to produce a final render. There is one vertex

shader used for the first pass in both APIs. In both APIs, the G-Buffer pass has unique shaders for each submesh due to the need to have a unique set of textures and material values. The composition pass uses one vertex shader and one fragment shader. The reflection cube and the sky box use one vertex shader and fragment shader each.

Furthermore, all the target render textures in both APIs were set to the same format and to the same size.

No explicit optimisation efforts were made in any of the APIs. Both APIs were used straight "out of the box".

Figure 6.2 shows the Metal scene, while Figure 6.1 shows the OpenGL scene

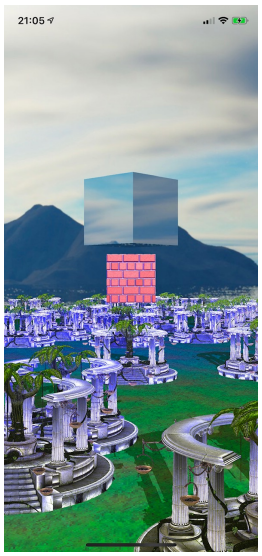


Figure 6.1: *OpenGL testing scene*

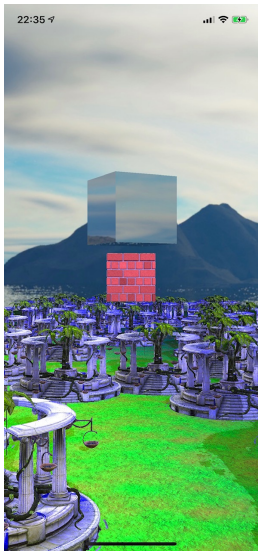


Figure 6.2: *Metal testing scene*

6.1.1 Profiling and results

This section will provide results obtained from profiling both scenes and discussion of those results.

CPU usage

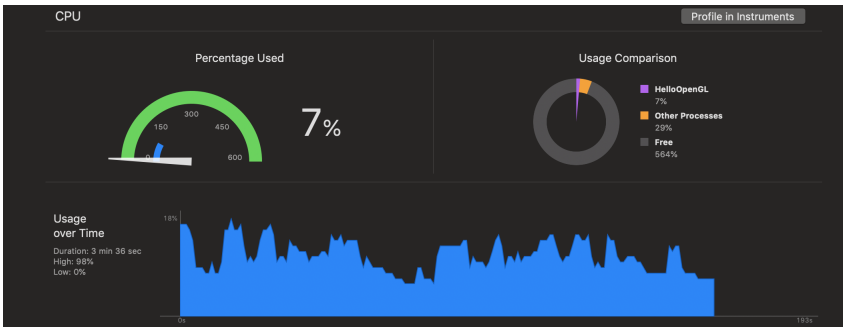


Figure 6.3: *OpenGL CPU Usage*

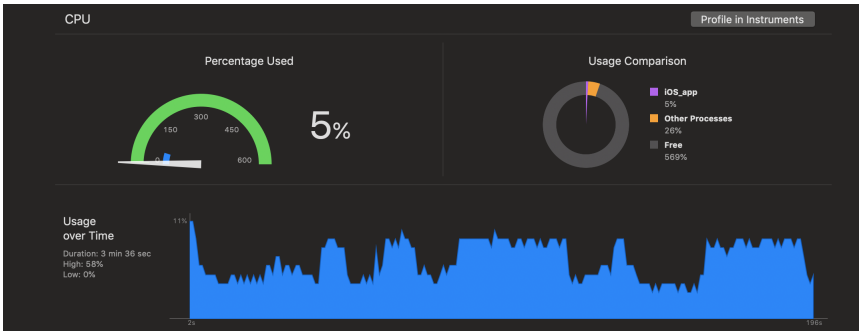


Figure 6.4: *Metal CPU Usage*

CPU usage was measured by leaving the scenes running for the same amount of time. Figure 6.3 shows CPU usage while running the OpenGL scene. At peak usage the OpenGL scene was using up to 18% of CPU, while the lowest usage was around 7%.

Figure 6.4 shows CPU usage while running the Metal scene. The peak usage was at 11%, while the lowest usage at certain points was only 3% of CPU.

GPU usage

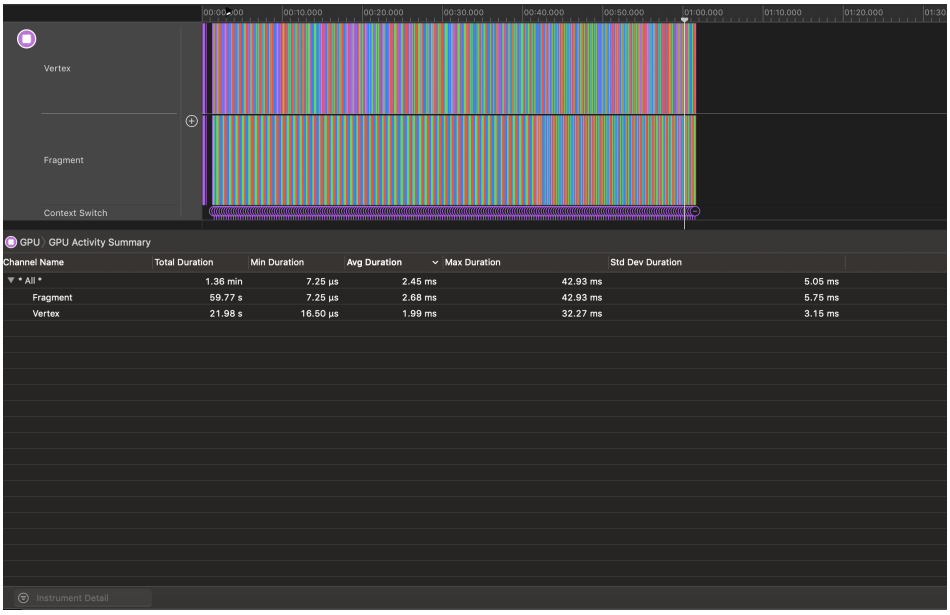


Figure 6.5: *OpenGL GPU Usage*

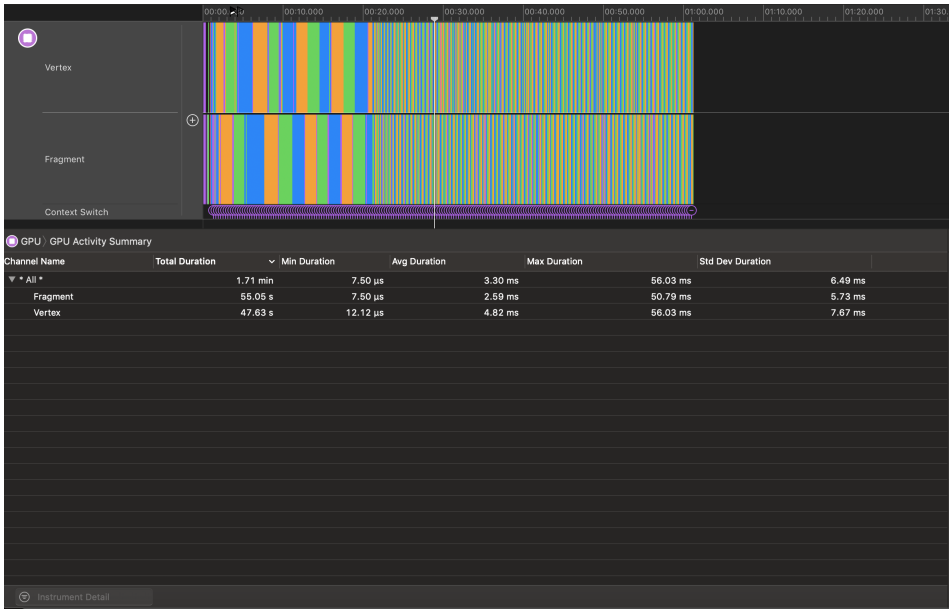


Figure 6.6: *Metal GPU Usage*

Figure 6.5 displays GPU usage while running the OpenGL scene. The average duration of fragment shaders was 2.68ms. The average duration of vertex shaders was 1.99 ms. Figure 6.6 displays GPU usage while running the Metal scene. The average duration of fragment shaders was 2.59ms. While the average duration of vertex shaders was 4.82 ms.

Frame Rate

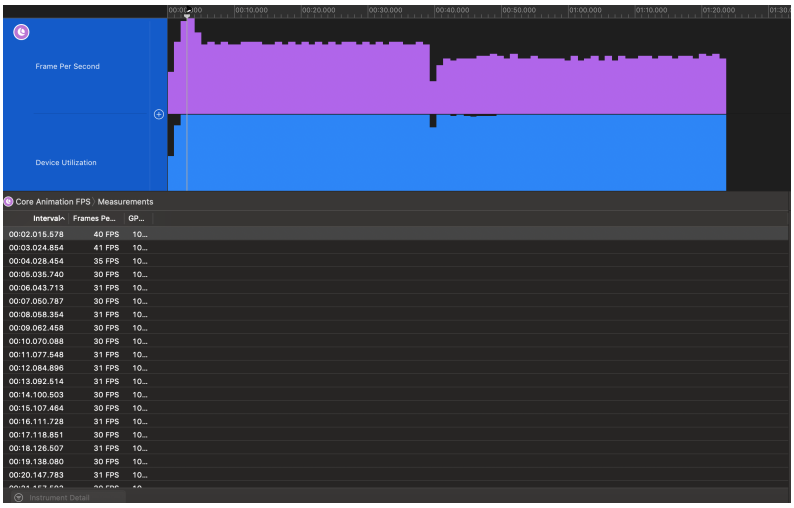


Figure 6.7: *OpenGL FPS at beginning of the rendering*

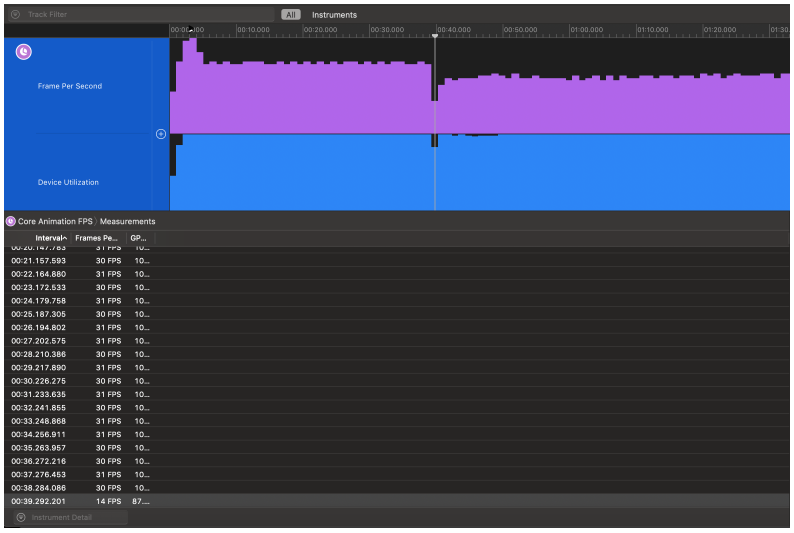


Figure 6.8: *OpenGL FPS drop in the middle*

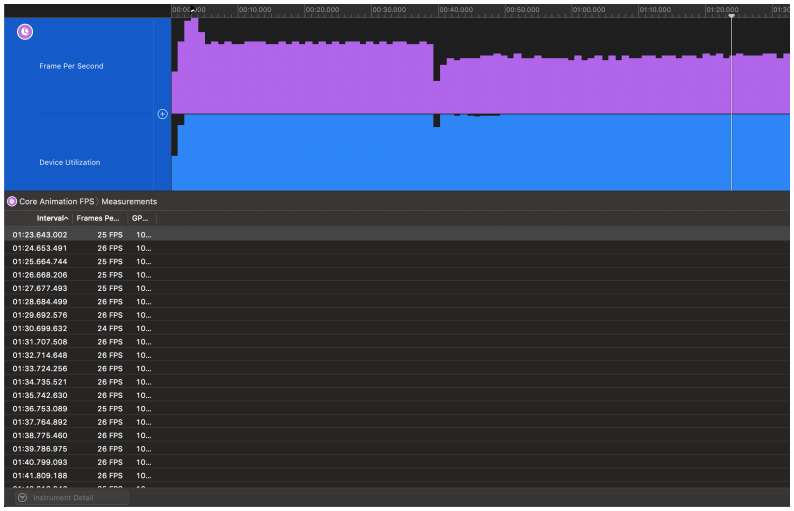


Figure 6.9: *OpenGL stabilised FPS*

Figure 6.7 displays framerate at the start of rendering the OpenGL scene. The scene starts at 40FPS and drops down to around 30FPS. Then in the middle there is a major FPS drop as shown in Figure 6.8 where the framerate drops to only 14 frames. After that the framerate settles at around 26FPS (Figure 6.9).

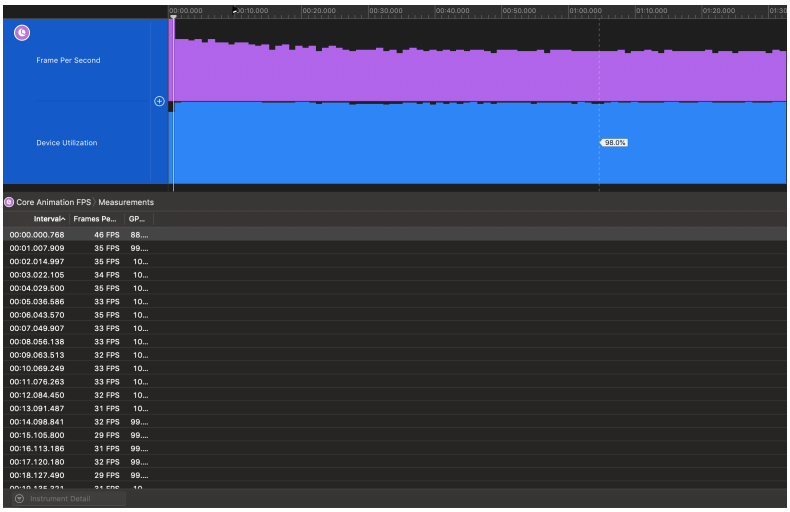


Figure 6.10: *Metal FPS beginning of rendering*

Figure 6.10 displays framerate at the start of rendering the Metal scene. The rendering starts at 46FPS and drops down to around 33FPS. There are no major drops in the framerate for the duration of the profiling. After a minute and half the framerate settles at around 28FPS (Figure 6.11).

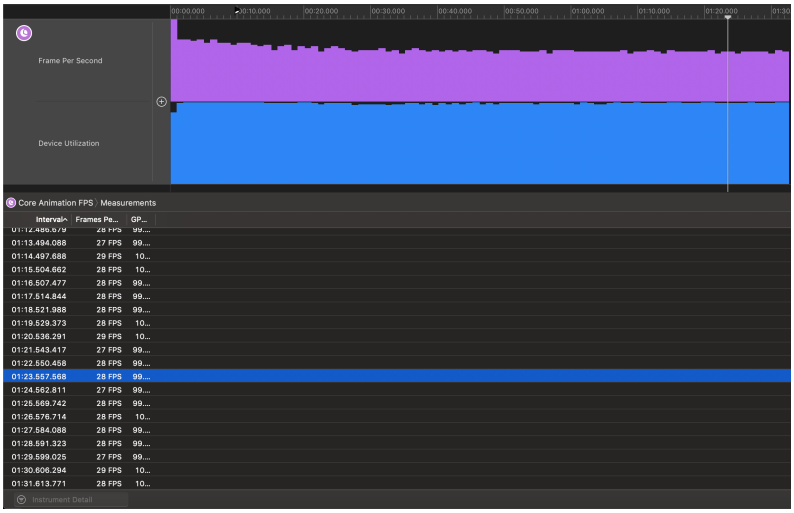


Figure 6.11: *Metal stabilised FPS*

6.1.2 Discussion

CPU utilization is where the main difference between the APIs occur. At peak times Metal is almost 64% more efficient than OpenGL, while the lowest usage in Metal is almost 134% more efficient. This shows a clear advantage for Metal when it comes to utilizing CPU processing and it is exactly what Apple promised when they introduced Metal in 2014 - lower CPU overhead resulting in using CPU more efficiently. This is a major benefit as freed up CPU for other computations and to make applications run faster.

However, when it comes to GPU utilization the results are inconclusive. The execution of fragment shaders in both APIs is almost identical. However, the average duration of vertex shaders in Metal was 4.82ms which is over twice as much than in OpenGL. As there is no difference in fragment processing, the assumption here is the Metal vertex shaders are performing worse due to bad coding practices in the Metal implementation. However, there is not enough evidence and data to conclude any results from this. This would require more investigation as to what is exactly happening.

When it comes to the frame rate of each scene, the frame rate is higher in the Metal scene by around 3 frames per second. Both scenes start with high frame rates which then drop and stabilize at much lower values. This indicates an issue with bad optimisation in both implementations. Possibly there is an issue with bad resource management in both APIs -

the system resources might be getting clogged up by improper resource management.

The difference in frame rates is not as high as maybe expected considering how better Metal utilizes CPU processing. This is possibly caused by the issue with the vertex shaders in the Metal scene not performing well. If the issue of slow vertex shaders in Metal is due to poor optimisation, then it would be expected for the frame rate of the Metal scene to be considerably higher after some optimisation is done.

Overall, the Metal scene seems to be performing better than the OpenGL scene even despite the issues with the shaders.

However, there are several limitations to this performance comparison. Firstly, as already mentioned bad practices due to inexperience in computer graphics programming are most certainly present in both solutions negatively impacting and skewing the performance results. Secondly, the approach to create as equivalent code as possible might not be the best approach as it might not be exploiting unique advantages of each API hence making any of the two APIs to underperform. A possibly better approach would be to create new 3D scenes using both APIs harnessing their strongest features and then do comparison based on that. However, that would require a deeper knowledge of both APIs which could not be gained due to the project's timeframe.

6.2 Code Comparison

This section will discuss coding styles of both APIs. The differences between certain between the APIs will be pointed out and their advantages and disadvantages discussed. This section will also discuss the time it took to learn each API.

6.2.1 Creational Patterns

In order to discuss differences in coding styles between the two APIs, a pattern that is employed by each of them needs to be mentioned as that dictates the coding process in both APIs

OpenGL uses what is called a handle pattern. Render objects such as programs, textures, buffers and others are created through dedicated OpenGL functions that return a handle or

a unique ID that is referencing that particular object. Listing 6.1 shows a process of creating a texture object in OpenGL. Even though a texture is an image, the type of the texture object is an unsigned integer or Gluint.

```
// A handle containing an ID referencing a colour texture
var colourTexture = Gluint();

let path = Bundle.main.path(forResource: fileName, ofType: fileExtension)!

let options : [String : NSNumber] = ["GLKTextureLoaderOriginBottomLeft" :
    NSNumber(integerLiteral: 1)];

do {
    let info = try GLKTextureLoader.texture(withContentsOfFile: path, options
    : options)
    // returns a handle ID that is assigned to colourTexture variable
    colourTexture = info.name;

}
catch {
    fatalError("Could not load a texture");
}
```

Listing 6.1: OpenGL handle pattern

Together with this OpenGL also heavily relies on the use of enum values to alter the behaviour of functions. Listing 6.2 shows this approach with GLenum() wrapper being used to pass an enum value to a function. The enum values are global values and in order to find out which ones to plug into a specific function OpenGL documentation must be consulted as the actual function header does not explicitly specify what enum values to use.

```
glActiveTexture(GLenum(GL_TEXTURE20));
glTexParameteri(GLenum(GL_TEXTURE_2D), GLenum(GL_TEXTURE_WRAP_S), GL_REPEAT);
glTexParameteri(GLenum(GL_TEXTURE_2D), GLenum(GL_TEXTURE_WRAP_T), GL_REPEAT);
glTexParameteri(GLenum(GL_TEXTURE_2D), GLenum(GL_TEXTURE_MIN_FILTER),
    GL_NEAREST);
glTexParameteri(GLenum(GL_TEXTURE_2D), GLenum(GL_TEXTURE_MAG_FILTER),
```

```
    GL_LINEAR);
glBindBuffer(GLenum(GL_ARRAY_BUFFER), 0);
glBindBuffer(GLenum(GL_ELEMENT_ARRAY_BUFFER), 0);
```

Listing 6.2: OpenGL overuse of generic enums

Metal on the other hand heavily employs a descriptor pattern. A descriptor object is a light weight function-less object that contains attributes that need to be assigned. The descriptor object is then used to create the actual object based on the provided attributes. The created object is then locked out and cannot be modified any further. Listing 6.3 shows a process of creating a texture in Metal. The type of the texture object is MTLTexture, clearly indicating what kind of object it is.

Metal also relies on enum values to alter rendering functions. However, enums are not globally specified, rather they are all implemented in individual structures with each function specifying which enum structure can be used with that specific function.

```
// a variable holding a texture
var colourTexture : MTLTexture;

// a descriptor object for the texture that will be used to create the actual
    texture object
let descriptor = MTLTextureDescriptor.texture2DDescriptor(pixelFormat:
    pixelFormat, width: Int(size.width), height: Int(size.height), mipmapped:
    false);


// assigning the attributes
descriptor.storageMode = storageMode;

descriptor.usage =  [.renderTarget,.shaderRead,.shaderWrite];

// creating the texture object based on the descriptor object
colourTexture =  RenderViewDelegate.device.makeTexture(descriptor: descriptor)
    !
```

Listing 6.3: Metal descriptor pattern

The handle pattern employed by OpenGL makes code much harder to read. With the handle pattern, all the OpenGL render objects are either of type GLuint (unsigned integer) or Glint (signed integer) which contain reference IDs returned by invoked handle functions. This makes it hard to discern objects from each other as no type is explicitly defined having to rely strongly on naming of the variables. Whereas in Metal all the objects are strongly typed clearly indicating of what type each variable is. The usage of global enum values makes developing slower by having refer to the documentation often.

The descriptor pattern makes it convenient to clearly see the attributes of the object that is being created. However, this does result in longer code as a descriptor object must be defined for all the Metal render objects. The descriptor pattern also means the object's attributes cannot be modified once created. If an object with a different state is needed, a descriptor must be created with the different state again from which a new object must be initialised. Though a content of those objects can be changed. For example, a different texture can be attached to a texture object. However, the attributes such as the texture storage cannot be changed.

6.2.2 Shader Objects

Rendering in OpenGL starts with creating a program object that has a vertex shader and fragment shader attached to it. Once the shaders are attached they need to be linked up and compiled through a handle function which returns a handle ID of the program object which is then used for referencing that particular pair of shaders once rendering.

The equivalent of the OpenGL's program object in Metal is a render pipeline object. Similarly to the OpenGL's program object, a render pipeline object contains a vertex shader and a fragment shader. However, it also contains information about target textures and vertex descriptor for the objects that will rendered using this render pipeline object.

The difficulty with the OpenGL's program is the necessity to manually compile and link up the shaders requiring to write a special class for this purpose. Metal has an advantage here by providing a one-line function that compiles the shaders automatically through just providing their names.

The additional information that needs to be provided for the render pipeline object results

in the ability to precompile the shaders as compared to OpenGL where shaders are compiled on the fly. Furthermore, the requirement to link a vertex structure and target textures with the shaders provides a better coupling and structure through having all this information in one object. In OpenGL, a vertex structure is defined separately from the shaders which if it is poorly results in code that is hard to maintain.

Shaders

On the shader side, the shading language used in OpenGL (GLSL) is based on C language. Metal Shading Language (MLSL) is based on C++ language. As C++ is based on C language, syntax-wise there are lots of similarities between GLSL and MLSL making the actual shader code almost identical and easy to port from one to the other. However, there are clear advantages to MLSL being based on C++ such as having the ability to create classes and use other object oriented features.

The built-in functions provided by both APIs in shaders are identical as well, however in some cases a different naming is used but providing the same functionality. This is another similarity that makes it easy to port shaders from one API to the other.

However, there are a few differences between the APIs in how shaders are being constructed.

Firstly, Metal has the ability to give its shaders custom names (Listing 6.5) and allows to include several shaders in one file as compared to OpenGL where each shader requires being in an individual file and named "main()" (Listing 6.4). Being able to use different names and several shaders in one file allows for better code structure and provides more freedom in how to organise code.

```
1 //either a vertex function or fragment function
2 void main()
3 {
4 }
```

Listing 6.4: OpenGL shader header

```
1 //vertex shader function called vertex_depth_noinstance
2 vertex float4 vertex_depth_noinstance()
```

```
{
}

//fragment shader function called fragmentCompositionPass
fragment float4 fragmentCompositionPass()
{
}
```

Listing 6.5: Metal shader header

OpenGL shaders can get very quickly disorganised by having its input and output attributes set up as loose variables (Listing 6.6) as compared to Metal where any incoming data is defined as parameters of a shader function and any output data is defined as a return type of a shader function (Listing 6.7).

```

uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 texCoord;
layout (location = 3) in vec3 bitangents;
layout (location = 4) in vec3 tangents;

out vec2 outTexCoord;

void main()
{
}
```

Listing 6.6: OpenGL loose variables

```
//fragment shader with parameters defined attributes
fragment float4 fragmentCompositionPass(FragmentBuffer in [[stage_in]],
                                        depth2d<float> shadow_texture [[texture(
    shadowTextureBuffer)]])
```

```
4 {
5 }
```

Listing 6.7: Metal shader parameters

Metal API is compatible with the Apple's SIMD library. The SIMD library allows code sharing between Metal shaders and Swift client code. This is useful for defining common data structures. The SIMD library allows to define these data structures in one file which then can be imported into the Swift code and the shading code simultaneously (Figure 6.12). This results in reducing layout mismatches that can occur when having to define the same data structure multiply times as is the case with OpenGL where a data structure needs to be defined over multiple different files if it needs to be used in shaders and client code (Figure 6.13).

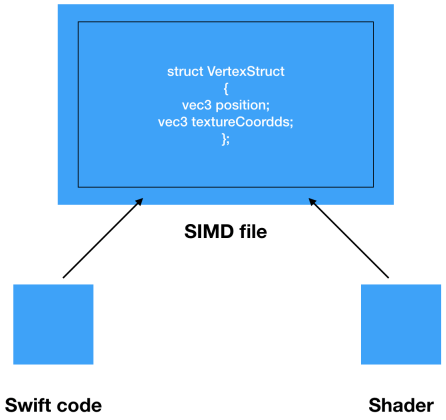


Figure 6.12: *Metal SIMD*

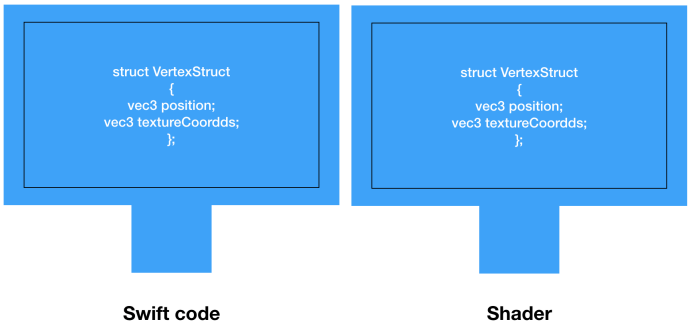


Figure 6.13: *OpenGL duplicate structures*

Xcode and Metal dramatically improves the debugging of shaders by being able to set up breakpoints and step through the shaders line by line. Such ability is not available for OpenGL making debugging of shaders much harder and a longer process.

6.2.3 Render Pass

The flow of rendering in each API is illustrated in Figure 6.14. This follows the same pattern. Metal code tends to be longer. However, that allows a more explicit control than in OpenGL.

In OpenGL firstly a framebuffer is created to which target textures are attached. This framebuffer ID handle then needs to be binded in order to draw to the target textures. Once the framebuffer is binded, buffers, textures and other render objects are sent to a GPU after which everything is rendered out to the textures. If different texture targets are required, a framebuffer must be changed and this process repeated.

Metal, on the other hand, requires getting a command queue which is used throughout the application lifetime. From the command queue a command buffer must be created. There is usually one command buffer per render pass. After that a command encoder is created from a render pass descriptor. A render pass descriptor is equivalent to the OpenGL framebuffer. A render pass descriptor defines target textures to render into. Unlike OpenGL, there is more explicit control in how the textures should be stored and when they should be disposed allowing to better optimise the performance. However, there is always an option to use default values. The command encoder must be explicitly started, all the render commands

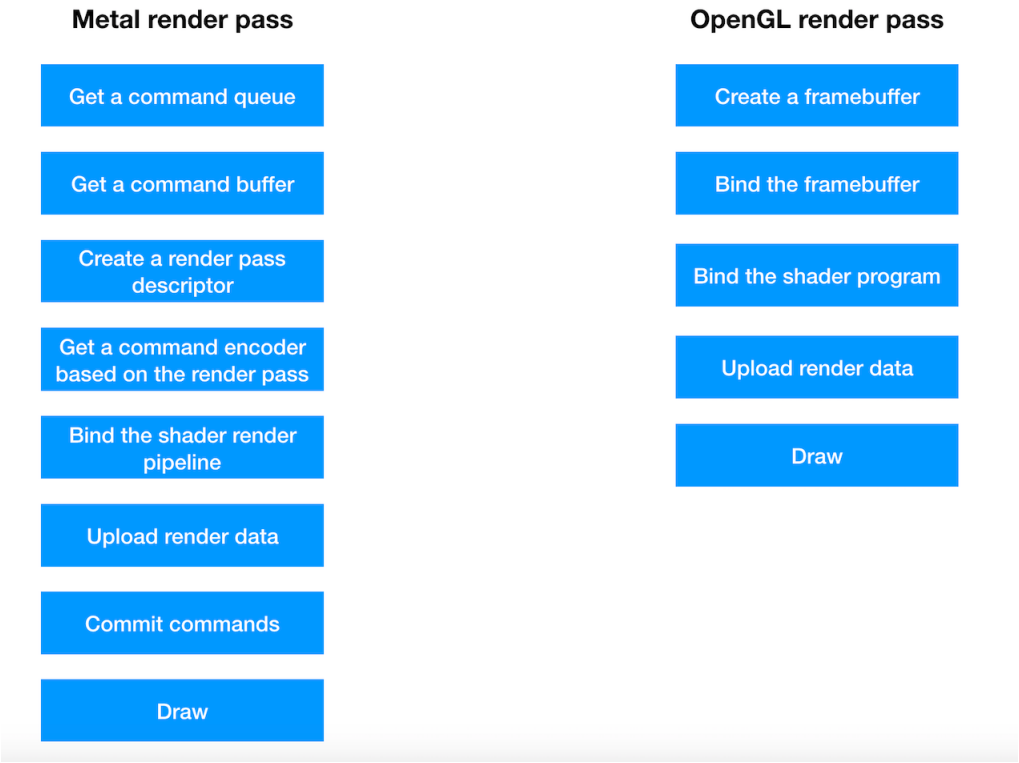


Figure 6.14: *Render pipeline in both APIs*

provided and then explicitly ended by committing the render commands after which the objects are rendered into the target textures.

The process is very similar in both APIs. Even though the Metal implementation is longer there is no extra complexity as compared to OpenGL. On the other hand, a more explicit control in Metal makes it more clear what is happening at every step of rendering.

Once it comes to the actual render body where buffers, textures are uploaded to the GPU, the process is almost identical. Apart from syntactic differences (Listing 6.8). The functions have just different names however the process of sending buffers, textures to the GPU and then calling a rendering method is identical.

```
1 //OpenGL -> set a program shader object
2 glUseProgram(shadowsPassProgram);
3 //Metal -> set a render pipeline object
4  renderEncoder.setRenderPipelineState(shadowPassPipeline);
5
6
```

```
// OpenGL -> sending a buffer to the GPU
glBindBuffer(GLEnum(GL_ARRAY_BUFFER), vertexBuffer);
// Metal -> sending a buffer to the GPU
renderEncoder.setVertexBuffer(vertexBuffer, offset: 0, index: Int(
    instancesBuffer.rawValue))


//OpenGL -> send a texture to the GPU
glBindTexture(GLEnum(GL_TEXTURE_2D), colorTexture);
//Metal -> send a texture to the GPU
renderEncoder.setFragmentTexture(colorTexture, index: 0);


//OpenGL -> draw
glDrawElementsInstanced(GLenum(GL_TRIANGLES), GLsizei(indexCount), GLenum(
    GL_UNSIGNED_INT), nil, GLsizei(instanceCount));
//Metal -> draw
renderEncoder.drawIndexedPrimitives(type: .triangle, indexCount: indexCount,
    indexType: indexType, indexBuffer: indexBuffer.buffer, indexBufferOffset:
    indexBuffer.offset, instanceCount: instanceCount);
```

Listing 6.8: Metal shader parameters

6.2.4 Discussion

This code comparison highlighted that both APIs are rather similar to each other. There are syntactic differences between the APIs and some aspects being implemented differently. However, the high-level concepts are identical.

Metal offers more control and requires everything being explicitly typed out. Metal forces more structure through its descriptor pattern forcing developers to adopt it which results in having less freedom in how to organise code. This, however, results in highly readable and well maintainable implementation even without having to plan the structure of the application upfront. Furthermore, Metal offers a more modern approach to shader writing through its C++-like language, custom named shaders, and exclusion of loose variables.

OpenGL results in more compact and shorter code. Its handle pattern and reliance on global enum values results in code that was found to be less readable. There is less structure enforced in OpenGL as compared to Metal. However, this requires lots of planning and creation of UML diagrams in order to avoid making code poorly structured.

In terms of the time it took to learn each API, the results are similar as well. Each scene was being developed by learning the particular API at the same time. Metal took roughly 2 months to learn, with OpenGL taking around 1 month to learn. However, the decreased time in learning OpenGL was caused by already knowing Metal. As already mentioned the concepts and high-level approach of both APIs is identical which means that learning one API considerably helps with learning the other API. Skills and techniques learnt in OpenGL can be applied in Metal programming and vice versa.

For this project Metal was considered to be faster to work with and easier to use.

6.3 Summary

This chapter provided a comparison between Metal and OpenGL in terms of their performance with Metal outperforming OpenGL. Furthermore, this chapter also provided a code comparison between the two APIs. Both APIs are rather similar with Metal being the preferred API during this project.

Chapter 7

Critical Evaluation

This chapter will contain a personal evaluation of the whole project. The following topics will be discussed: review of project's objectives, a review of the plan, evaluation of the product, lessons learnt, and the reflection on the first two deliverables.

7.1 Review of the Objectives

This section will review the project's achievements against the objectives. For the description of objectives refer to the project proposal.

7.1.1 Objective 1 & Objective 2: Learn about Metal and OpenGL

I successfully managed to learn all I needed about both APIs in order to create the final product. For OpenGL I did not end up using the Computer Graphics course on Edx as I felt learning through video tutorials is much longer process than learning through reading. Instead, I decided to use online tutorials and text books. I would consider achievements of these objectives successful even though I took a different route to do so.

7.1.2 Objective 3 & Objective 4: Render a scene in both APIs

I managed to create both scenes in both APIs with almost all the graphical techniques apart from the water surface which ended up being above my mathematical skills. I believe I have the necessary skills in both APIs to be able to implement a water surface, however mathematically it would have required me to learn lots of new concepts - specifically integral and differential calculus - which unfortunately was not possible due to the timeframe of the project. Hence I was forced to omit the water surface from both scenes.

In terms of the reflection, I only used the most basic form. This came down to not having enough time to implement anything more complicated due to having to spend the time on writing the actual dissertation and working on my other assignments.

This unfortunately comes down to a poor choice of requirements to implement. I created this list of techniques to implement at the beginning of the project where my knowledge of graphical APIs was minimal hence I had no knowledge of how long each technique would take nor how complex each technique was. Maybe a better approach would have been to implement one complicated graphical technique, such as the water surface, instead of focusing on the quantity.

7.1.3 Objective 5 & 6 & 7: Compare both APIs

I managed to measure the performance of both solutions on iPhone X using the Xcode's profiling instruments after which I carried out the performance comparison as well. There were optimisation issues which skewed the performance comparison in terms of GPU utilization. There was a lack of time to find out what exactly went wrong.

Code comparisons were also carried out. The "time-efficiency" comparison was moved into the code comparison section and mentioned in the discussion as it did not have enough content to stand on its own because "time-efficiency" was rather hard to measure and quantify.

7.1.4 Objective 8: Compare the visual quality

I did not meet this objective. During the development of both scenes it became obvious that there is no difference in how both APIs are rendering the scenes. Apple did not make any claims that Metal would result in better quality graphics. Both scenes look identical hence having a questionnaire would seem like a wasteful time spent that would not contribute to any useful information. Hence, after the discussion with my supervisor it was decided to leave out this objective completely.

7.2 Review of the Plan

The initial plan was to learn about OpenGL and Metal simultaneously and then start the implementation towards the end of the December. However, this plan changed before the end of November. Constant switching between learning OpenGL and learning Metal proved to be distracting and confusing. Hence, after the discussion with my supervisor I decided to focus on Metal first and implement the Metal scene first after which I would focus on OpenGL and implement the OpenGL scene next.

Another reason to focus on a single API first was that after the first few weeks I realised the concepts and ideas behind each graphical technique will be the same so after I learned and implemented the Metal scene, OpenGL will be easier and faster to learn.

I did not start implementing the Metal scene until I fully learnt and fully understood all the computer graphics concepts in Metal. Hence, the implementation of the Metal scene got pushed until end of January. However, it took one month less to implement than planned. The implementation of the OpenGL scene got pushed till the beginning of the March and took one month less to implement as well.

The delay was also caused by the coursework for other modules at the end of December and beginning of January. When I planned the first Gantt chart I did not take into account the coursework for other modules properly and did not consider how much time they will take. Hence, the coursework took a considerate amount of time in December and January where I could not focus on the final year project.

Furthermore, the delay was caused by the early struggles with completely new concepts.

The project progressed faster once the initial concepts were fully comprehended.

The delays in the implementations did not have any negative effects on Objectives 5,6,7 and 8. These objectives were supposed to be finished by the first week of April. However, this was a decision made on the assumption that analysis and discussion needs to be handed in the second week of April. That was not the case as the analysis and discussion was not to be handed in until the last day of April. Furthermore, the objectives took less time than planned as I was making notes while implementing the scene making the last part of dissertation easier and faster to produce.

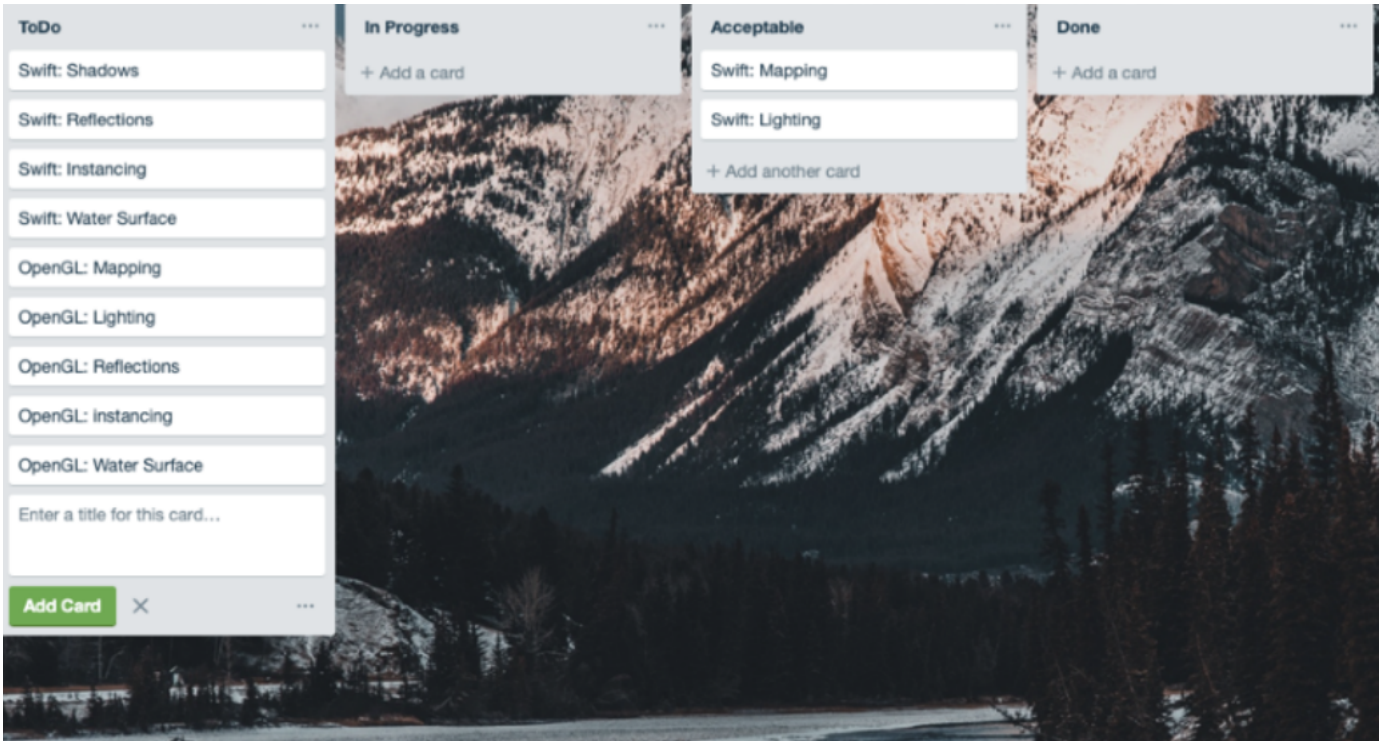


Figure 7.1: *Initial Trello board*

It is necessary to say the actual Gantt chart was not used or modified throughout the project. Personally, it provided a good overview at the beginning, however I personally did not find it useful past that. Maybe one of the reasons for this was that the Gantt chart was created using an online tool which was not very intuitive and any modifications to the Gantt chart proved to be cumberstone.

What proved to be more useful during the development of the project was a Trello board (Figure 7.1) created in a Kanban style which I used throughout the project and improved

upon as well. The Trello board reflected the Gantt chart and contained all the objectives that needed to be done. However, unlike the Gantt chart, the Trello board was easy to edit and provided a better overview of what I should be focusing on at that particular moment, what needs to be done and what has already been completed.

7.3 Evaluation of the Product

As I managed to almost complete all the implementation requirements I think the product turned out to be good. Especially at the beginning of the development of the Metal scene I carried out regular weekly refactoring sessions to keep the code readable and easy to modify. I created a fair amount of static helper functions throughout to make creation of common objects easier. For example, creation of common render objects such as pipelines, textures is now done through a single static function call.

I am also happy with the decision to add a Git repository to my projects early on which made modifications and experimentations easier.

However, as the time progressed and some obstacles occurred there was less time to carry out regular code refactoring resulting in an unorganised code. As I had to finish some features quickly instead of planning it first I just coded it to make it work. This especially applies to the OpenGL scene. OpenGL proved to be more difficult due to it being an old API. Therefore, one of the weakness of the final product is disorganised code that especially in the OpenGL case can be confusing, hard to read and hard to manage.

Unfortunately, this all comes down to not having enough time. The decision was to either keep the code neat and readable while not being able to implement all the requirements. Or, implement all the requirements at the price of less organised code. I deemed it was more important to finish all the requirements.

7.4 Lessons Learnt

The first lesson I learned is the importance of refactoring. If I kept the same focus on the refactoring throughout the project the performance comparison would be more accurate.

Both API solutions have altogether a few thousand lines of code now. Adding new features to any of the solution is quite time-consuming and hard. A time spent on refactoring would make the code more future-proof.

I also learnt that it is necessary to tweak the development methodology in order to suit the nature of the project instead of trying to make the project work with the development methodology chosen at the beginning. I chose the Agile methodology with a 1 month sprint as my initial development methodology. At the beginning of the development I tried to fit my project around this methodology. However, it did not seem to work. I felt I needed something more flexible and something without the time pressure of the sprint.

Hence, I switched to Kanban instead which gave me the flexibility I needed at the beginning of the project where a lot of development was based on experimentation which did not suit having any kind of sprint. However, as my knowledge of computer graphics programming got better, the development was done in a more organised way. I noticed I am able to finish new features on a weekly basis. Hence, I added a 1 week sprint to my Kanban methodology to provide a way to review the process weekly and to be able to see how fast the project is moving towards the end goal. I stuck with this combination of Kanban and Agile until the end of the project.

7.5 Reflection on the Previous Deliverables

I am happy with the first deliverable. I do not feel there is anything that I feel like I would want to add to it.

Regarding the second deliverable, I wish the word count was more flexible or that the code samples did not count towards the word count as in my case where I had to present codes from both APIs that proved to be quite an obstacle. Unfortunately, the result of that is I feel I did not include everything I needed and had to compromise on the quality of that deliverable.

7.6 Summary

Overall, this was a good project where I managed to learn a completely different style of programming compared to what we were learning at the university. I am disappointed that I could not implement all the graphical techniques. However, this project provides a good ground to explore more graphical programming in the future.

Chapter 8

Conclusions

This work set out to compare OpenGL and Metal on Apple devices through creating two identical scenes. This project successfully managed to compare the two APIs and draw a conclusion that the introduction of Metal by Apple was a step that was beneficial due to Metal performing better than OpenGL and also Metal was found to be easier to use. Therefore, Metal would be a recommended API to use when developing graphical applications for Apple devices especially when deciding whether to learn OpenGL or Metal.

For developers that already know OpenGL and are reluctant to switch to Metal, they can still use OpenGL to keep developing for Apple devices. The similarity of both APIs makes it fairly easy for OpenGL developers to port their code to Metal, should Apple completely remove OpenGL support from their devices.

On PCs and other platforms OpenGL is being replaced by a new API called Vulkan. Vulkan is not supported officially by Apple. However, the Khronos Group, a consortium responsible for OpenGL and Vulkan, is bringing Vulkan to MacOS and iOS through an SDK that will compile Vulkan commands into Metal commands through an intermediate layer hence allowing developers using Vulkan to deploy their applications on Apple devices. It would be interesting to see how this Vulkan-to-Metal approach is performing in comparison to pure Metal making this comparison a good choice for future work.

To conclude, Metal is a modern graphical API that was found to be performing better and easier to use than OpenGL during the implementation of this project.

References

Max, N. (1981). Vectorized Procedural Models for Natural Terrain. ACM SIGGRAPH Computer Graphics, 15(3), pp.317-324.

Whitted, T. and Weimer, D. (1982). A Software Testbed for the Development of 3D Raster Graphics Systems. ACM Transactions on Graphics, 1(1), pp.43-58.

Cook, R. (1984). Shade trees. ACM SIGGRAPH Computer Graphics, 18(3), pp.223-231. Perlin, K. (1985). An Image Synthesizer. ACM SIGGRAPH Computer Graphics, 19(3), pp. 287-296.

Hanrahan, P. and Lawson, J. (1990). A language for shading and lighting calculations. ACM SIGGRAPH Computer Graphics, 24(4), pp.289-298.

Rhoades, J., Turk, G., Bell, A. and State, A. (1992). Real-Time Procedural Textures. ACM SIGGRAPH Computer Graphics, pp.95-100.

Anselmo Lastra, Steven Molnar, Marc Olano, and Yulan Wang. (1995). Real-time programmable shading. In Proceedings of the 1995 symposium on Interactive 3D graphics (I3D '95). ACM, New York, NY, USA, pp. 59-66.

Olano, M. and Lastra, A. (1998). A Shading Language on Graphics Hardware: The PixelFlow Shading System. ACM SIGGRAPH Computer Graphics, pp.159-168.

St-Laurent, S. (2004). Shaders for Game Programmers and Artists. Boston, Massachusetts: Thomas Course Technology PTR.

Aydin, T., Čadík, M., Myszkowski, K. and Seidel, H. (2010). Video quality assessment for computer graphics applications. ACM Transactions on Graphics, 29(6), p.1.

Westwood, D. and Griffiths, M. (2010). The Role of Structural Characteristics in Video-Game Play Motivation: A Q-Methodology Study. Cyberpsychology, Behavior, and Social Networking, 13(5), pp.581-585.

Bailey, A., Cunningham, S.(2011). Graphics Shaders: Theory and Practice, Second Edition. Boca Raton, Florida: A K Peters/CRC Press.

Hergaarden, M. (2011). Graphics Shaders. VU Amsterdam.

Mantiuk, R., Tomaszewska, A. and Mantiuk, R. (2012). Comparison of Four Subjective Methods for Image Quality Assessment. Computer Graphics Forum, 31(8), pp.2478-2491.

Usher, W. (2014). 75% Of Gamers Say That Graphics Do Matter When Purchasing A Game. [online] CINEMABLEND. Available at: https://www.cinemablend.com/games/75-

Gamers-Say- Graphics-Do-Matter-Purchasing-Game-64659.html [Accessed 23 Nov. 2018].

Lavoué, G., Mantiuk, R. (2015) Quality Assessment in Computer Graphics. Visual Signal Quality Assessment – Quality of Experience (QoE). Springer, pp.243-286.

Kirichok, S. (2015). Is Swift Faster Than Objective-C?. [online] Yalantis.com. Available at: https:// yalantis.com/blog/is-swift-faster-than-objective-c/ [Accessed 23 Nov. 2018].

Apple. (2016). About Metal and This Guide. [online] Available at: https:// developer.apple.com/ library/archive/documentation/Miscellaneous/Conceptual/ MetalProgrammingGuide/Introduction/ Introduction.html [Accessed 23 Nov. 2018].

Mandrigin, I. (2016). iOS App Performance: Instruments & beyond – Igor M – Medium. [online] Medium. Available at: https://medium.com/@mandrigin/ios-app-performance-instruments- beyond-48fe7b7cdf2 [Accessed 23 Nov. 2018].

Dilger, D. (2017). One Apple GPU, one giant leap in graphics for iPhone 8. [online] AppleInsider. Available at: https://appleinsider.com/articles/17/04/17/one-apple-gpu-one-giant-leap-in- graphics-for-iphone-8 [Accessed 23 Nov. 2018].

Begbie, C., Horga, M.(2018). Metal by Tutorials. Razeware LLC.

Developer.apple.com. (2018). Swift | Apple Developer Documentation. [online] Available at: https:// developer.apple.com/documentation/swift [Accessed 23 Nov. 2018].

Mediakix. (2018). The $50B Mobile Gaming Industry: Statistics, Revenue [Infographic]. [online] Available at: http://mediakix.com/2018/03/mobile-gaming-industry-statistics-market-revenue/#gs. 7KsHhUM [Accessed 23 Nov. 2018].

Perez, S. (2018). Apple’s App Store revenue nearly double that of Google Play in first half of 2018. [online] TechCrunch. Available at: https://techcrunch.com/2018/07/16/apples-app-store-revenue- nearly-double-that-of-google-play-in-first-half-of-2018/?guccounter=1 [Accessed 23 Nov. 2018].

Owen, M. (2018). Apple's A12 Bionic comes close to desktop CPU performance in benchmarks. [online] AppleInsider. Available at: https://appleinsider.com/articles/18/10/05/ apples-a12-bionic- comes-close-to-desktop-cpu-performance-in-benchmarks [Accessed 23 Nov. 2018].

Smith, C. (2018). The iPhone XS and XS Max just destroyed the iPhone X – and every Android phone – in benchmarks. [online] BGR. Available at: https://bgr.com/2018/09/18/ iphone-xs-max- vs-android-benchmarks-review-says-apple-wins/ [Accessed 23 Nov. 2018].

Popko, A. (2018). Swift vs Objective C: iOS' Programming Languages Compared. [online] Netguru.co. Available at: https://www.netguru.co/blog/swift-vs-objective-c-ios-programming- languages-compared [Accessed 23 Nov. 2018].

Sacolick, I. (2018). What is agile methodology? Modern software development explained. [online] InfoWorld. Available at: https://www.infoworld.com/article/3237508/agile-development/what-is- agile-methodology-modern-software-development-explained.html [Accessed 23 Nov. 2018].

Squires, J. (2018). Apples to apples, Part II. [online] Jesse Squires. Available at: https://www.jessesquires.com/blog/apples-to-apples-part-two/ [Accessed 23 Nov. 2018].

Williams, H. (2018). Objective-C vs Swift. [online] Techworld. Available at: https://www.techworld.com/careers/objective-c-vs-swift-which-programming-language-should-i-learn-3672233/ [Accessed 23 Nov. 2018].

Taylor, H. (2018). Games account for 75% of App Store spending. [online] GamesIndustry.biz. Available at: https://www.gamesindustry.biz/articles/2018-06-01-games-account-for-75-percent-of-app-store-spending [Accessed 6 Apr. 2019].

Usher, W. (2014). 75% Of Gamers Say That Graphics Do Matter When Purchasing A Game. [online] CINEMABLEND. Available at: https://www.cinemablend.com/games/75-Gamers-Say- Graphics-Do-Matter-Purchasing-Game-64659.html [Accessed 23 Nov. 2018].

Westwood, D. and Griffiths, M. (2010). The Role of Structural Characteristics in Video-Game Play Motivation: A Q-Methodology Study. Cyberpsychology, Behavior, and Social Networking, 13(5), pp.581-585.

Frisvad, J., Christensen, N. and Falster, P. (2007). The Aristotelian rainbow. Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia - GRAPHITE '07.

Longhurst, P., Ledda, P. and Chalmers, A. (2003). Psychophysically based artistic techniques for increased perceived realism of virtual environments. Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa - AFRIGRAPH '03.

Phong, B. (1975). Illumination for computer generated pictures. Communications of the ACM, 18(6), pp.311-317.

Caulfield, B. (2018). What's the Difference Between Ray Tracing, Rasterization? | NVIDIA Blog. [online] The Official NVIDIA Blog. Available at: https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/ [Accessed 6 Apr. 2019].

Owens, B. (2013). Forward Rendering vs. Deferred Rendering. [online] Game Development Envato Tuts+. Available at: https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342 [Accessed 6 Apr. 2019].

Everitt, C., Rege, A. and Cebenoyan, C. (n.d.). Hardware Shadow Mapping. [online] Inst.cs.berkeley.edu. Available at: http://inst.cs.berkeley.edu/~cs283/fa10/lectures/shadow_mapping.pdf [Accessed 6 Apr. 2019].

Caliskan, A. and Cevik, U. (2015). Overview of Computer Graphics and algorithms. 2015 23nd Signal Processing and Communications Applications Conference (SIU).

Greene, N. (1986). Environment Mapping and Other Applications of World Projections. IEEE Computer Graphics and Applications, 6(11), pp.21-29.

Mallett, I. and Yuksel, C. (2018). Deferred adaptive compute shading. Proceedings of the Conference on High-Performance Graphics - HPG ’18.

Lee, W., Shin, Y., Lee, J., Kim, J., Nah, J., Jung, S., Lee, S., Park, H. and Han, T. (2013). SGRT. Proceedings of the 5th High-Performance Graphics Conference on - HPG ’13.

Amanatides, J. (1987). Realism in Computer Graphics: A Survey. IEEE Computer Graphics and Applications, 7(1), pp.44-56.

Gao, R., Yin, B., Kong, D., Zhang, Y. and Si, H. (2008). An improved method of parallax mapping. 2008 8th IEEE International Conference on Computer and Information Technology.

Carucci, F. (2006). GPU Gems 2. Upper Saddle River, N.J.: Addison-Wesley. Kryachko, Y. (2006). GPU Gems 2. Upper Saddle River, N.J.: Addison-Wesley.

Li, X. and Xu, M. (2009). Water simulation based on HLSL. 2009 IEEE International Conference on Network Infrastructure and Digital Content.

Tan, X. and Feng, X. (2008). An Approach for 3D Water Surface Simulation. 2008 International Conference on Computer Science and Software Engineering.

Appendices

Appendix A

Project Logbook

Log Book

18/9/2018

Project kick off lecture

24/09/2018

I am researching into what kind of project to do. I would like to do something with computer graphics but unsure exactly what – possibly creating a 3d scene as a tech demo of what is possible in OpenGL. Other possibilities include machine Learning and neural networks. The complexity of these projects is a problem though, hence I will be waiting for Tuesday when there are group and supervisor meetings to further discuss this with any of the lectures.

26/09/2018

After the Tuesday meetings I have a better idea of what I want to do now. I want to do definitely something with computer graphics.

02/09/2018

Another meeting with my supervisor. The idea is now to do a comparison between writing shaders in Unity and writing shaders in Metal - Apple newest graphical API. I would carry this out through writing a few shaders in Unity, a few shaders in Metal and then doing a comparison.

03/09/2018

Researching the possibility of doing the comparison between Unity shaders and Metal shaders. However, this does not seem to be a good idea.

https://forum.unity.com/threads/how-does-metal-integrate-with-unity.542429/

In this post I am finding out that Unity is using Metal anyway. Whatever shaders are written in Unity this is then compiled to Metal. This makes the initial idea for the project a bit useless.

Trying to come up with something else. Taking the initial idea at the beginning to do something in OpenGL and then comparing that with Metal. Metal replaced OpenGL in MacOS and iOS, however OpenGL can still be used.

https://www.extremetech.com/computing/270902-apple-defends-killing-opengl-opencl-as-developers-threaten-revolt

New idea then is doing a comparison between OpenGL and Metal.

08/09/2018

Researching the new project idea. Mainly looking at the following links:

https://www.upwork.com/hiring/mobile/opengl-vs-metal-mobile-graphics-api/

https://blogs.unity3d.com/2014/07/03/metal-a-new-graphics-api-for-ios-8/

https://www.linkedin.com/pulse/pros-cons-using-apples-metal-graphics-rendering-api-edvinas-šarkus

https://news.ycombinator.com/item?id=17231747

Seems interesting, challenging enough and a useful comparison to do. Metal and OpenGL are currently competing graphical APIs for the Apple platform. Starting to draft out my project proposal for tomorrow's meeting.

10/09/2018

The draft had a good response and Metal vs OpenGL got approved. Researching now good sources for OpenGL and Metal.

Metal has a good Apple official documentation with lots of samples.

https://developer.apple.com/metal/

I have a good experience with Apple documentations, they were always good detailed, well explained. Going with that for Metal learning.

https://www.edx.org/course/computer-graphics

For OpenGL I am going to take out a course on Edx. It has some Maths refresher as well which is going to be useful.

https://learnopengl.com

This also seems to be a good source for OpenGL information.

15/10/2018

Working on my project proposal, finishing all up for the review tomorrow. Researching some examples of what is possible with OpenGL and Metal.

18/10/2018

Finished off the rest of the proposal after getting a feedback. No big changes needed just a bit more defined objectives. Hence, I had to research for a while of what I could include in my OpenGL scene. This would now include a deferred shading, parallax mapping, shadow mapping, water surface, and instancing.

26/10/2018

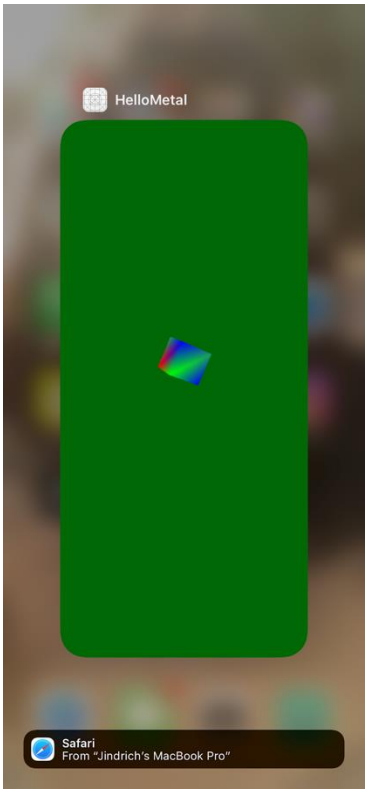
I started learning the Metal API by creating a simple iOS application that drew out a triangle to the screen. I used this resource for the tutorial: https://www.raywenderlich.com/7475-metal-tutorial-getting-started. The result was jus a triangle with three different colours on a green background.

30/10/2018

There was another meeting with the supervisor. There was a discussion about a literature review and topics for the review.

31/10/2018

This was another time I spent learning more about Metal. This time I am following the second part of the tutorial. https://www.raywenderlich.com/728-metal-tutorial-with-swift-3-part-2-moving-to-3d. This time the result is a 3D rotating cube. There is a lot of setup for Metal to even start drawing primitive shapes. It is a very low-level API for sure. Not having issues with any of the programming part. It is a long process but not hard to understand. However, I will need to look at some Math specifically at linear algebra. As a source for that I will probably going to use Khan Academy or the Edx tutorial

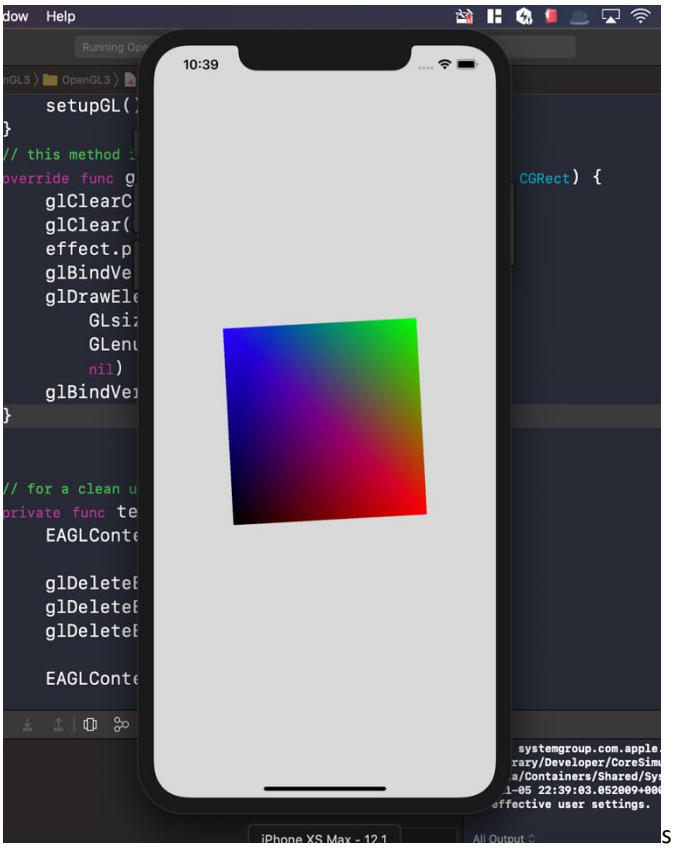


3/11/2018

After having done some tutorials in Metal l will be looking now at OpenGL finally. It is not as easy to find good tutorials for OpenGL on MacOS. Especially since the OpenGL has been deprecated the number of tutorials for the latest version of OpenGL for MacOS is very low.

5/11/2018 (time spent: 2-3 hours)

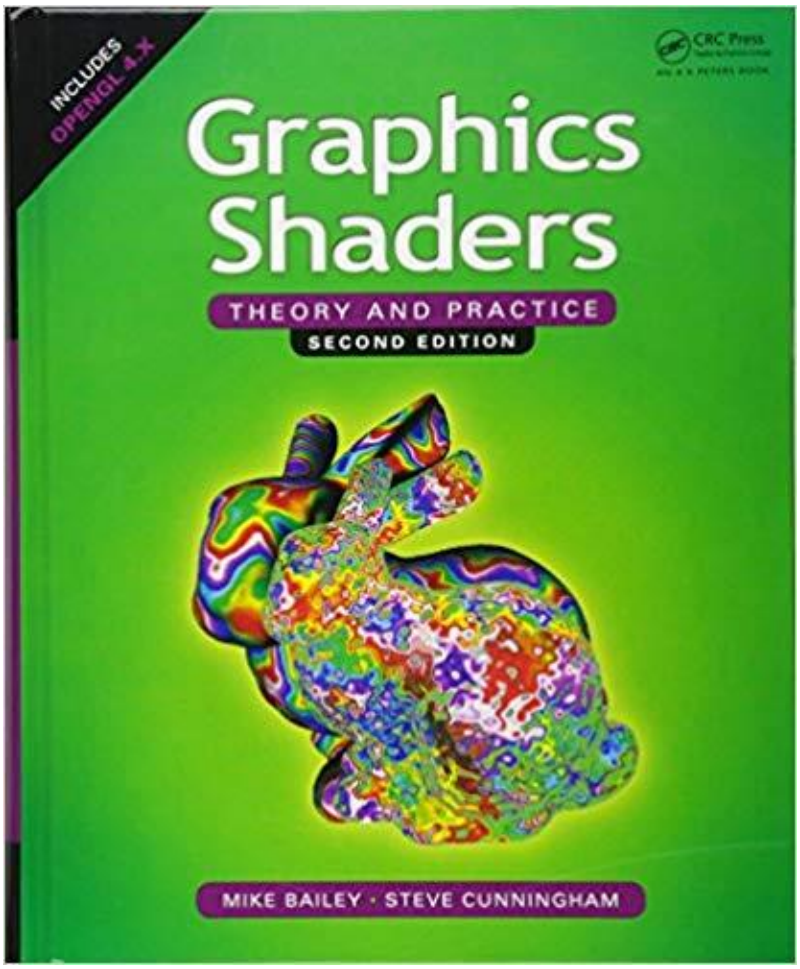
I finally found a good tutorial on https://www.raywenderlich.com/5146-glkit-tutorial-for-ios-getting-started-with-opengl-es. The result is similar to the stuff I did in Metal. Again it was fairly easy to understand. The only thing I got stuck on a little bit was Swift. I will need to learn a little bit about that. Specifically, I will need to have a look at extensions and pointers. As I have done the similar thing in both APIs I am now able to do some little comparison. OpenGL seems to be a bit "uglier" than Metal. Metal seems to be as low-level but a bit more logical, organized and feels better throughout. With OpenGL the setup process is a bit more confusing and complicated.



S

9/11/2018 (time spent: 2-3 hours)

I started working on my next deliverable, specifically the literature review. Firstly, I started with searching for good sources that I could use to write my literature review. As a book I found a good resource called Graphics Shaders: Theory and Practice by Mike Bailey and Steve Cunningham. This book looks at writing shaders in OpenGL 4.0 which is a very useful as that is the version of the OpenGL used in MacOS and iOS.



For Metal I found a resource on raywendelrich.com. A book is called Metal by Tutorials.

In terms of papers I found a few resources as well. I found a paper called Shade Trees by Robert L. Cook which basically laid out foundation for shaders. Another useful paper I found is called Graphics Shaders by Mike Hergaarden from Amsterdam University which talks about the evolution of shaders in graphics programming.

10/11/2018 - 11/11/2018 (time spent: 5 hours)

These two days I spent writing my literature review with the sources I found the other day. I got to around 1000 words so far.

17/11/2018 - 18/11/2018 - (time spent: 10 hours)

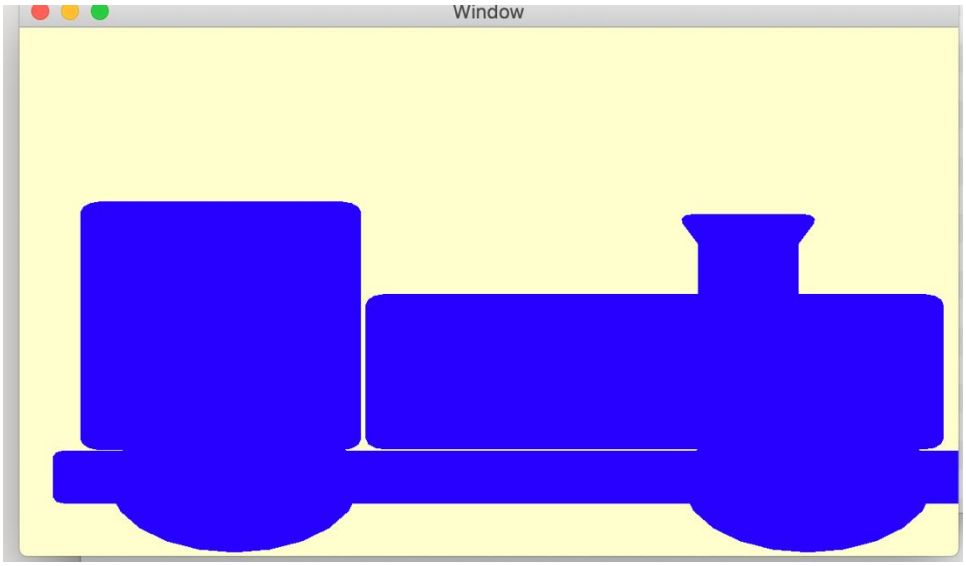
Same as last weekend, the days were spent writing my literature review.

20/11/2018 - 22/11/2018 - (time spent: 20 hours)

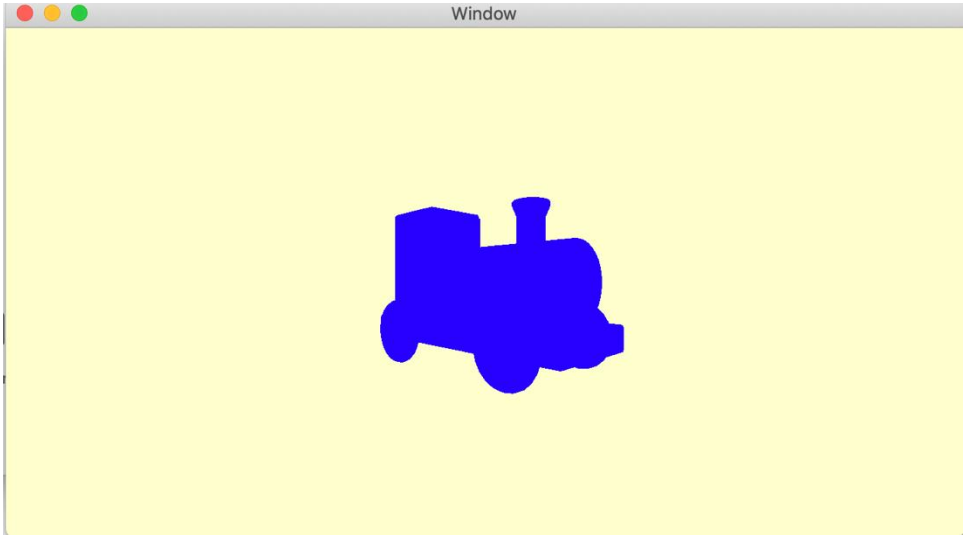
Finishing and submitting my literature review.

30/11/2018 - 3/12/2018 (time spent: 10 hours)

After the literature review was finished and after I did some work on the assignments for other modules I returned back to researching and learning about Metal. This time I focused on importing a 3D model and displaying it on the screen as compared to my previous tutorial showcase where I rendered only a triangle with some simple vertex and fragment functions.

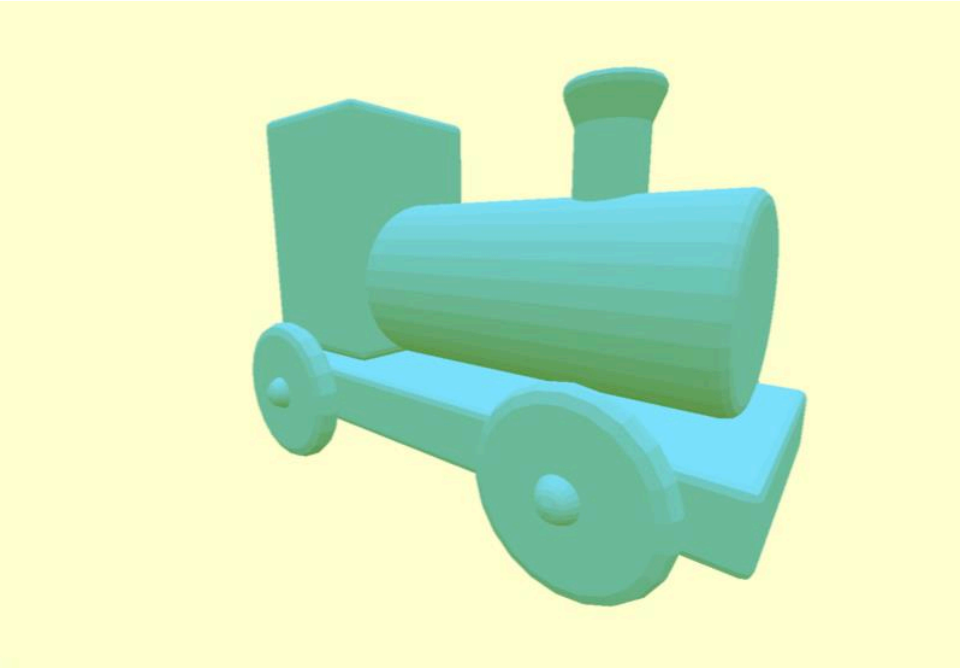


This was a first attempt at importing a 3D model. The first render only looks like a 2D image as no transformations were done to the train object.



The next try was to render the model so that it looks like a 3D object. This was done through transformations, rotations and scaling. The model had to be transformed from the object space to the projection space to have it

displayed like this which was done through matrix transformations. The model is still lacking any kind of lighting and is only being displayed with a blue base color.



The train now has a depth and looks like a 3D object. The visuals were achieved through a fragment shader. The object's normals are being taken into account and an interpolation is happening between a light blue colour and a dark blue colour. If the object's normal is pointing up then a fully light blue colour appears. For normals pointing directly down a dark blue colour appears. For anything in between a mix of these two colours is created.

This result was showcased to my supervisor on Tuesday 4th of December.

Week 12

The next week was spent by working on an assignment for mobile development.

Week 13, Week 14, Week 15

Christmas Vacation

Week 16 and Week 17

Spent these weeks working on several assignments.

2019

17/1/2019 - 20/1/2019 (time spent: 15 hours)

One of the issues while working on the tutorials before was my limited knowledge of Swift, Objective-C and Metal Shading Language. Hence, I decided to dedicated 4 days to spend some time learning about the features of these languages I did not understand that well.

First thing was to learn about Swift Extensions. I used this link: https://cocoacasts.com/four-clever-uses-of-swift-extensions and the official Apple documentation. Extensions are used to add methods to classes that cannot be directly accessed and they can also be used as a good way to provide a multiple inheritance.

Furthermore, I also learned about the way Swift notations can be used to pass around a memory reference rather than a value. This comes in handy when working with Objective-C code as C-based languages make use of memory references a lot.

I also looked at so-called lazy variables which are variables that are only evaluated once they are used which results in better optimisation. **https://medium.com/@abhimuralidharan/lazy-var-in-ios-swift-96c75cb8a13a**

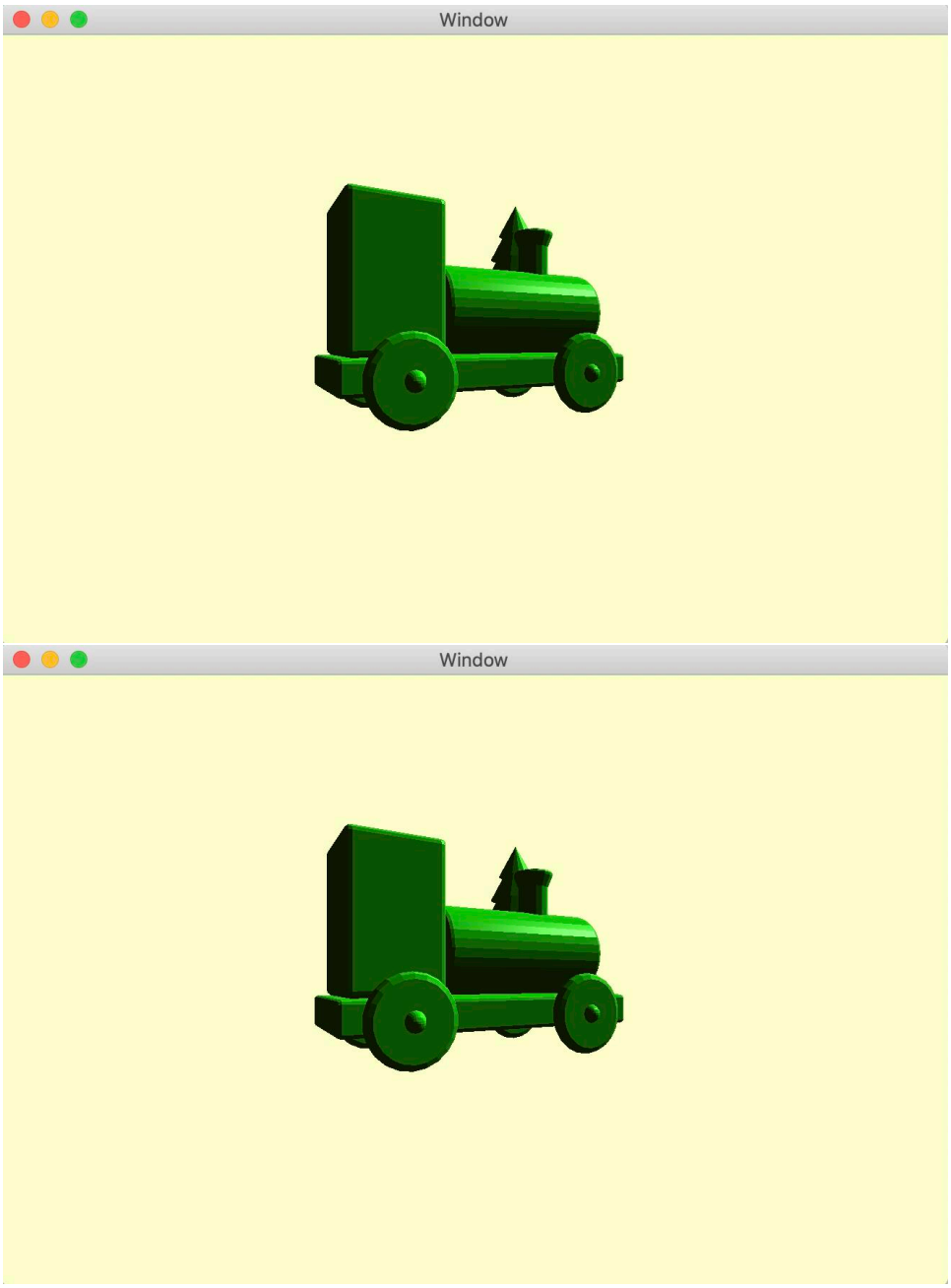
I read through the Metal Shading Language Specification at **https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf** to find out about some of the attributes used in vertex shaders and fragment shaders.

25/1/2019 (time spent: 4 hours)

I used this day to revise and go through what I have already learned about Metal to lock in the knowledge.

26/1/2019 -28/1/2019 (time spent: 15 hours)

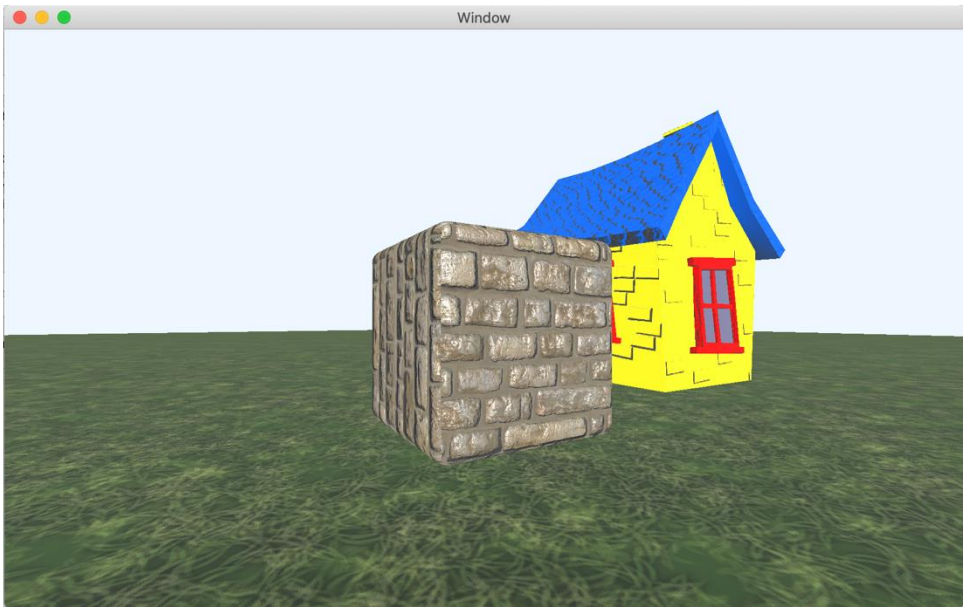
I carried on with making the lighting a little bit better. I learned about the Phong's reflection model. I learned about its 3 parts : diffuse lighting, specular lighting and ambient lighting and how all they add up to create a sense of global illumination and reflection based on object's normals and its material. The reflection and the lighting can be seen in the picture below:



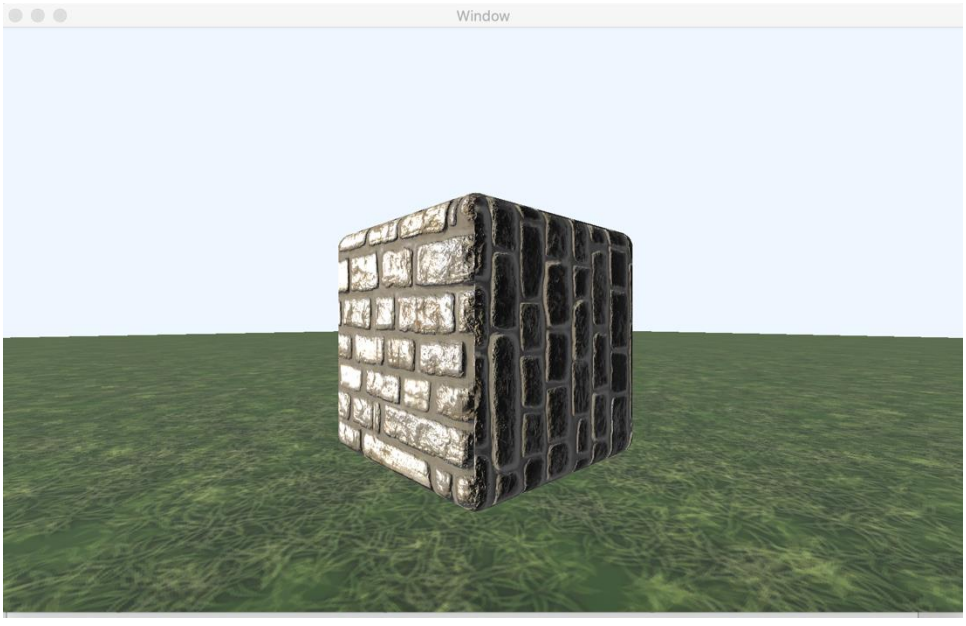
29/1/2019 - 30/1/2019 (Time spent: 5 hours)

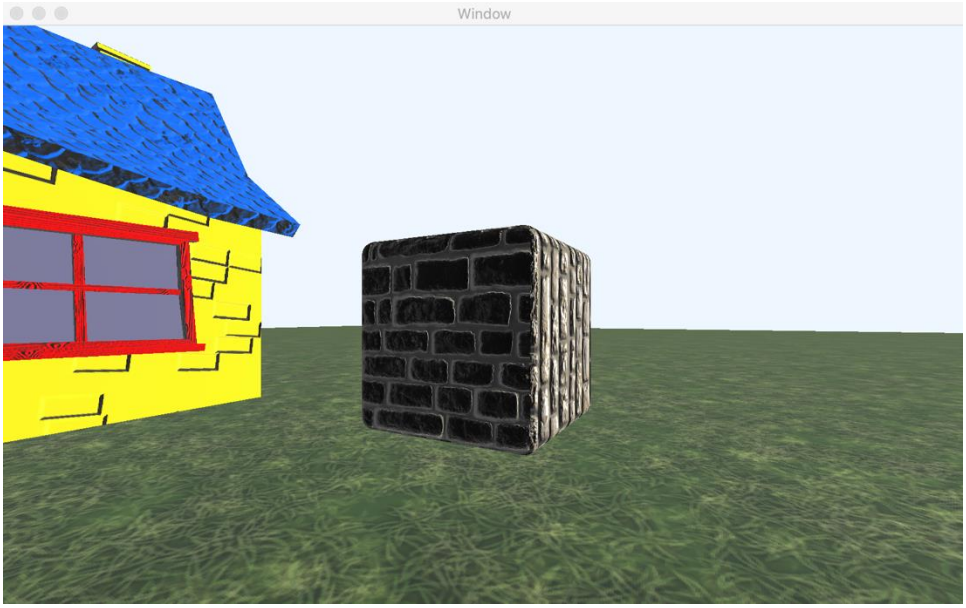
These two days I spent on learning about textures. I started with applying a base color texture that just simple gives a model an appearance to make it recognizable. Such effort can be seen in the picture below. It is a simple texture provided with the model that is just wrapped around the object.

From this I moved onto normal textures. Normal textures are a set of textures that is used together with the base textures to provide the surface of the model with more details without having to change its geometry. It uses normals that are coming out of each fragment surface and then the camera view to create an effect of depth. This kind of effect can be seen in the pictures below on the cube and on the cottage house. The cube and the cottage are absolutely smooth objects without polygons coming out of them. However, using normal texture mapping an effect of roughness is achieved.



Another improvement is the lighiting model used. Before I only used a generic lighting that affected every model in the same way. Now I am also reading from the model file its material specifications (whether it is a shiny material or a rough material etc.) and using this to affect the lights based on the material used. The effects can be seen in the pictures below:





31/1/2019 (Time spent: 1 hour)

Watched video from Apple's WWDC conference about the differences between OpenGL and Metal APIs.

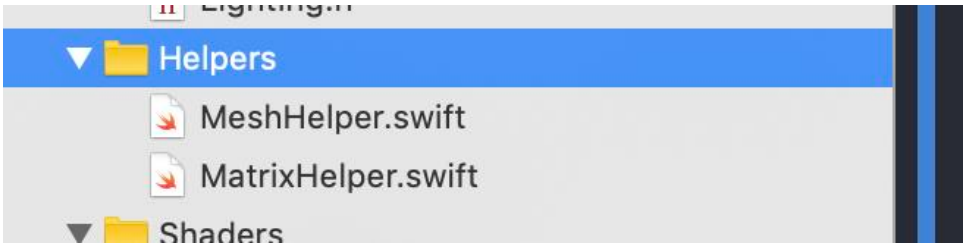
https://developer.apple.com/videos/play/wwdc2018/604/

Not a bad video. However, the content was rather too high-level without getting into low-level details. Though it was a useful time spent as it allowed me to see a summation of the main differences between these two competing APIs.

1/2/2019 - 3/2/2019 (Time spent: 12 hours / 14 hours)

Up until now the way I was creating the scene in Metal was through experimentation and prototyping. The main goal was to get the results I want while sacrificing the quality of the code and the structure of the code. Hence for this weekend, I decided to do some code refactoring. I thought it would be a good idea to do this now and not wait until end of the project as at that point the code base would be quite big making the refactoring task much harder.

However, refactoring still took quite bit of time and is not finished yet, though very close to finishing as of now. The main goal was to restructure classes by moving the code around into more logical structures I created a few helper classes with static functions to streamline some processes. As is the example in the picture below:

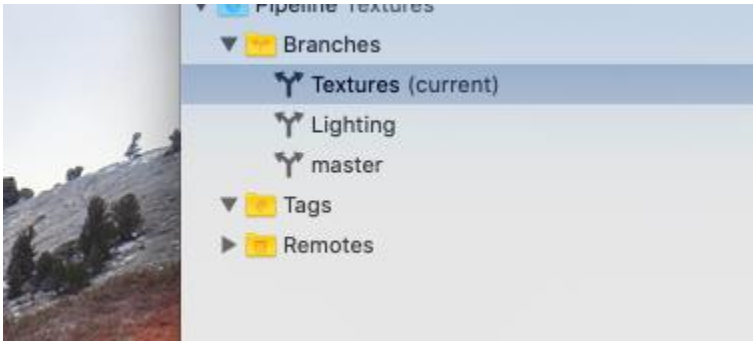


MeshHelper contains static functions in regards to creating new Metal objects such as setting up a render pipeline, loading textures, getting vertex descriptors. While the MatrixHelper contains static functions to make manipulation with matrices a little bit easier.

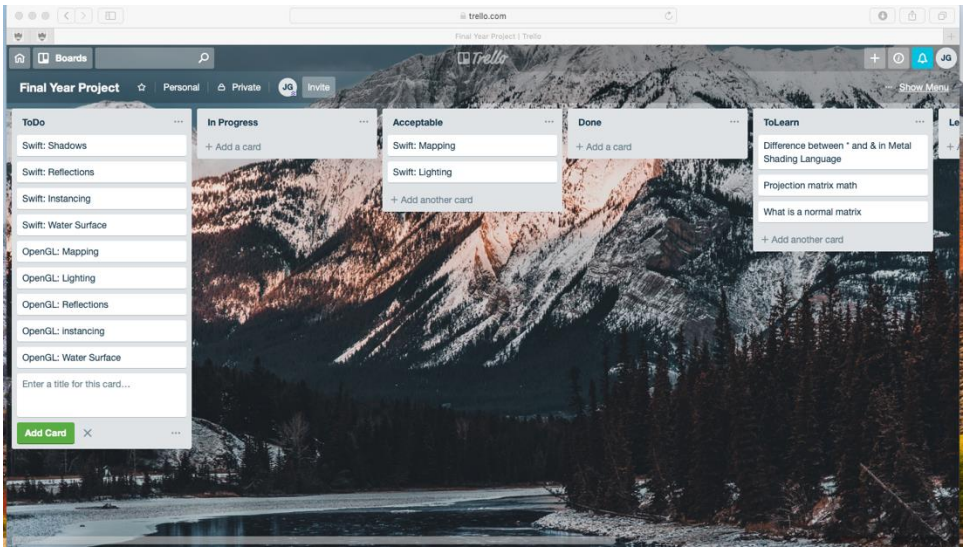
At this point I am aware that in the future I will for sure reorganise the code again. But doing the refactoring now resulted in a cleaner code which will make future refactorings easier.

Reorganising, deleting and optimising code resulted in the code breaking up quite often during this weekend which is the factor behind why it took so long.

Together with the refactoring I also introduced a source control in my project which was not present at the beginning. Personally, I think that at the beginning there was no need for the source control. It would not hurt to have it but at the same time the project was not large in any way hence it was not hard to maintain. However, that has changed and now I find it very useful to track changes in the code. This also I feel provided me with better flexibility when it comes experimenting as I do not have to worry about breaking point to the point of having to spent hours trying to get the working build back. With the source control, I can just easily undo all the changes I made in a matter of seconds.



The current source control set up is in the picture above. Currently the plan is to branch out of master for every new feature I work on for the scene. Once I am happy with the way everything works I merge the branch back into the master and branch out a new branch with a new task.



During this weekend I also created myself a Trello board to make me better organised. Currently for the actual project I set up ToDo, In Progress, Acceptable and Done columns through which I moved the tickets. I took a little bit of inspiration from the Kanban assignment for Software Quality Management as I feel the structure of Kanban and the flexibility of Kanban with no sprints and other similar aspects fits this project very well.

I also set up a column for things to learn and things to look up and ask about.

Together with this I also started a document where I describe what I implemented and in what way I implemented it which I hope will be useful for another deliverable.

4/2/2019 - 6/2/2019 (Time spent: 9 hours)

After the refactoring phase on my final year project I went on to finish another part of the objective for the Metal part of the project. This time I was looking at instancing which is a technique of drawing multiple different objects without overloading the computation resources unnecessarily.

Basically the approach is to send the vertices only for one object and then access the same memory location while drawing the other instances of the same object. That way the memory usage is reduced as if only one object was rendered. As part of this some restructuring of my code was necessary to make this instancing fit into my project as my rendering pass had to be changed to take this into account and to send new buffers to the GPU.

7/2/2019 (Time spent: 1 hour)

On this day there was a Thursday final year project lecture and a meeting with my supervisor. The lecture topic was about the architectural patterns in software development. This was kind of a summary of the architectural pattern lecture we had in the second year and was aimed rather at people who did not take that module rather on people who already know something about the patterns. Furthermore, the nature of my project might no need any of this so the lecture was rather useless for me. Maybe

there will be a possibility to implement some of the patterns as I go through the project but that depends on the circumstances.

In the meeting with my supervisor I discussed recent changes and additions to my project. There was nothing else discussed. It was only a showcase of what I have done during the winter break.

11/2/2019 - 13/2/2019 (Time spent: 9 hour)

Another objective was to render objects with shadow mapping, i.e. to create an illusion of objects casting shadows and that is what I implemented in these three days. Considered time was spent on again on refactoring the code to accommodate the shadow mapping feature.

The finally result can be seen in the picture above. The whole structure is casting a shadow and even the pillars on to each other.

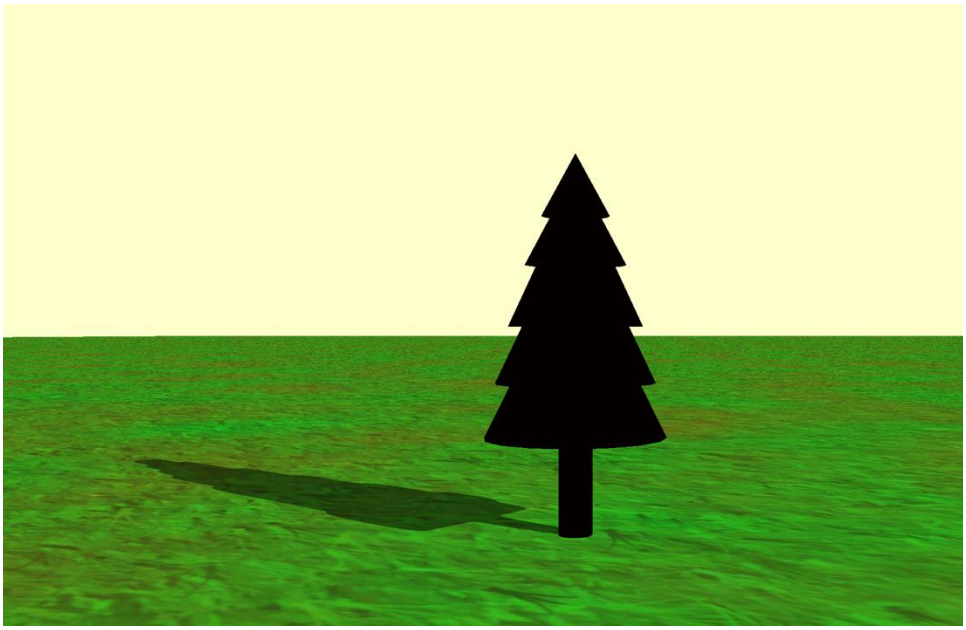
The process is achieved through doing a second render pass. In the first pass a camera is set to where the directiona light is. A scene using only depth information is rendered (no colours is needed). This scene is saved as a depth texture. The second pass which then displayes the actual scene to the screen uses this depth texture to calculate the shadows. Each fragment's depth information is compared to this shadow depth texture and based on that the lighting is adjusted.

14/2/2019 (Time spent: 5 hours)

There was another lecture today. This time the topic was testing and evaluation. How these two tasks are carried out depends a lot on the nature of the project. So the lecture itself was very generic. This was then followed by a meeting with my supervisor and a discussion about the lecture took place. In my

project the testing will not be a typical testing done in a software engineering as my acceptance test is quite subjective as all I care about is whether the effects being rendered look the way they should (does light correctly illuminates objects in the scene etc.) The evaluation will be in the form of measurements of how well each solution is running.

The rest of the day was spent fixing an unpleasant bug in my code.

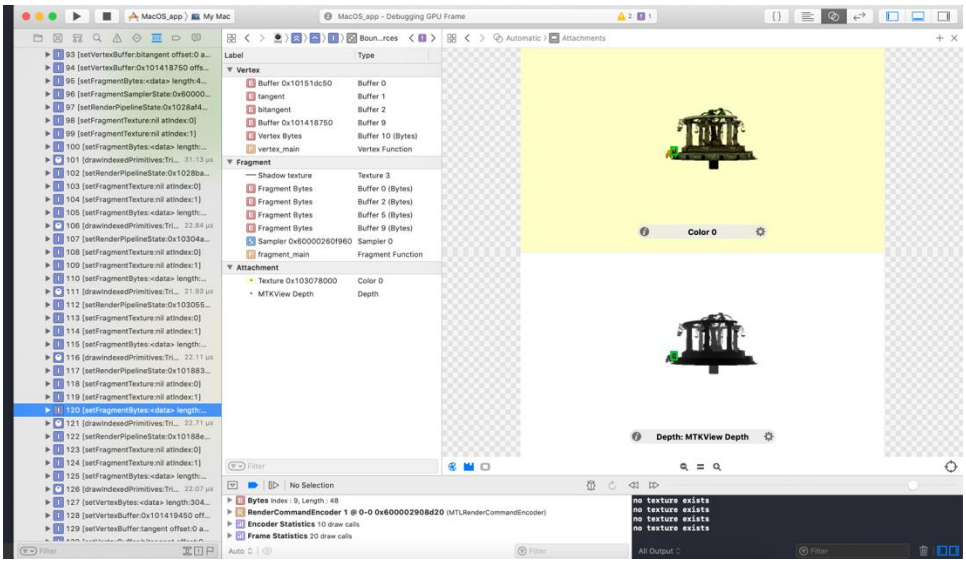


The issue can be seen in the picture above. Basically, all the objects that I had in my scene started to look completely black as the tree above. This was my first big bug that I had to fix and highlighted the unpleasant process of fixing bugs in computer graphics as it is not as easy as setting breakpoints as in a normal software engineering project.

What I first noticed was that the ground, which is an external object as well, was rendered correctly with a correct lighting applied. This made me think there must be something wrong with the normals in the tree. They would either be completely missing, or they would be pointing inwards.

The code at this point already probably has around 1000 lines. There are numerous different buffers being sent to the GPU and numerous matrix calculations being carried out to calculate lighting, textures, displacements etc.

Firstly, I thought there is an issue with the way the submeshes are being loaded and that their materials are not being processed correctly as it is a fairly complex process that involves lots of data manipulation and lots of settings having to be filled in. However, after some investigation there was no problem with this.



Finally after some time I found out about a debugger in XCode that makes it a little bit easier to debug graphical APIs. As the picture shows above the debugger allows to see all the data that was uploaded to the GPU and all the client's instructions as they are being carried out. Unfortunately it is does not let you investigate what is happening in the shader functions however it is good enough to confirm there is nothing wrong the client's side which was my finding as well. All the data was loaded correctly to the GPU and all the data was correct as well. The instructions being carried out in the command encoder were correct as well.

This turned my attention towards the shader functions especially towards the fragment shader which is responsible for the final look of the object. This revealed that the normal direction computation was taking into consideration that every object contains a normal map. However, the tree did not contain a normal texture, hence it was skewing with the direction the normals were pointing to.

17/2/2019 (Time spent: 3 hours)

CODE REVIEW AND REFACTORING

As the project progresses, it turned out to be a good idea to do a weekly code refactoring. Arbitrary I chose to do it every Sunday. I feel the code refactoring is very important as with each new feature implemented the code base grows significantly and with each feature dependent on the code written before it is important to organize the code to be able to navigate through the code quickly. Also usually when implementing new stuff there is lots of experimentations going on and sometimes I forget to delete temporary code. Hence, regular code refactoring is a good way to get rid of code sections that are no longer needed and were just left behind by accident.

18/2/2019 - 19/2/2019 (Time spent: 4 hours)

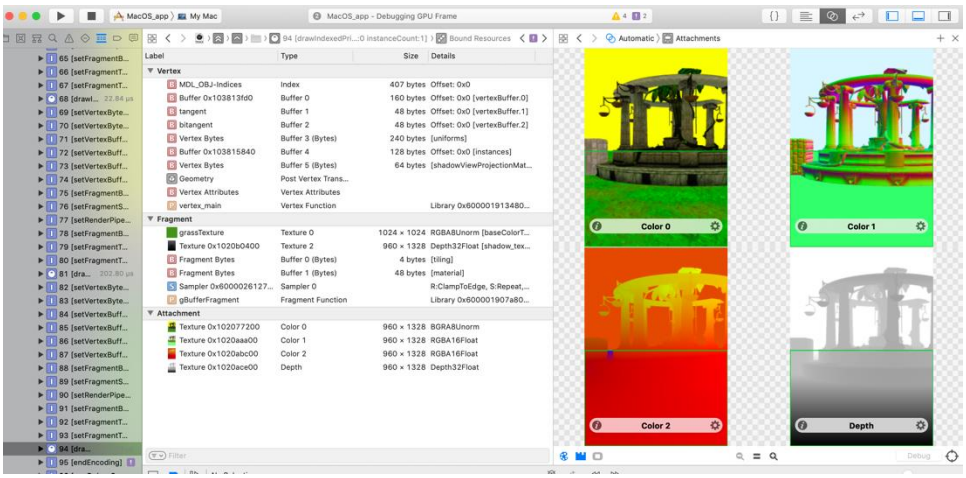
DEFERRED RENDERING

These two days I spent learning about how to implement deferred rendering. Visually there is no difference in the way the resulting scene looks. What changed is the way the scene is being rendered to the screen.

Up until now the scene was rendered using a forward rendering only, where the scene is being rendered and the lights are being applied as the vertices and the fragments are being processed. This is fine as long as there are not lots of light sources deployed in the scene.

However, once the number of light sources increases in the scene, the performance goes down significantly as the lighting needs to be calculated for fragments that will not end up in the final scene.

With deferred rendering several render passes are created. The first pass is the shadow pass that I created last week in order to create a shadow map. The second pass is a GBuffer pass where 3 textures are created. There is a color texture, normal texture and position texture created in this pass.



(the textures created during the G buffer pass)

These three textures are sent to the last pass that is called the composition pass, which takes these three textures, combines them together and then applies the lighting. This means the lighting is only applied to the fragments that are displayed on the screen. This results in saving computational power and allows to render hundreds of lights much more efficiently.

Visually there is no difference. However, with the deferred rendering I can now draw lots of point lights in the scene if I want to and still keep a good performance. Which is rather essential on the smartphones.

20/2/2019 (Time spent: 3 hours)

BUG POINTLIGHTS TEXTURE

One thing I wanted to add to the scene in order to test the deferred lighting properly and to make the scene look a little bit better was textured point lights so that they are represented in the world visually.

This however proved to be an issue as I struggled to display them properly. Firstly, to test this properly I only created coloured circles at where the point lights were located.

The picture shows the issue I was struggling with. The red point light is supposed to be behind the pillar, hence the red circle should not be displayed.

I thought the issue was caused because the rendering of these point lights happened at the composition pass. So I moved the rendering to the GBuffer pass. The rendering worked fine there and the depth was calculated correctly, however a different issue was created. Because the lighting is applied in the composition pass, the point light textures were affected by the lighting. That was not desirable and hence the rendering could not be done in the GBuffer pass.

So my next thinking was to draw the point lights in the composition pass and use the textures created in the Gbuffer pass to calculate the correct depth. So in order to do that I uploaded the depth texture from the Gbuffer pass to the same shader function that is used to render the lights. I sampled this depth texture and then compared the sample with the position of the lights. However, that did not work as well. The lights were still displaying without any depth information. I will need to figure this one out on some other day.

21/2/2019 (Time spent: 1.5 hours)

Data Analysis lecture and Meeting

Today there was a lecture about data analysis. However, this proved to be a disappointment. The lecture turned out to be rather an advert for the Msc. course in data mining. The lecture was followed by a meeting with my supervisor. I showed the deferred rendering and then set out a plan for the next few weeks where I will start looking at the implementation of the second scene in OpenGL.

23/2/2019 (Time spent: 4 hours)

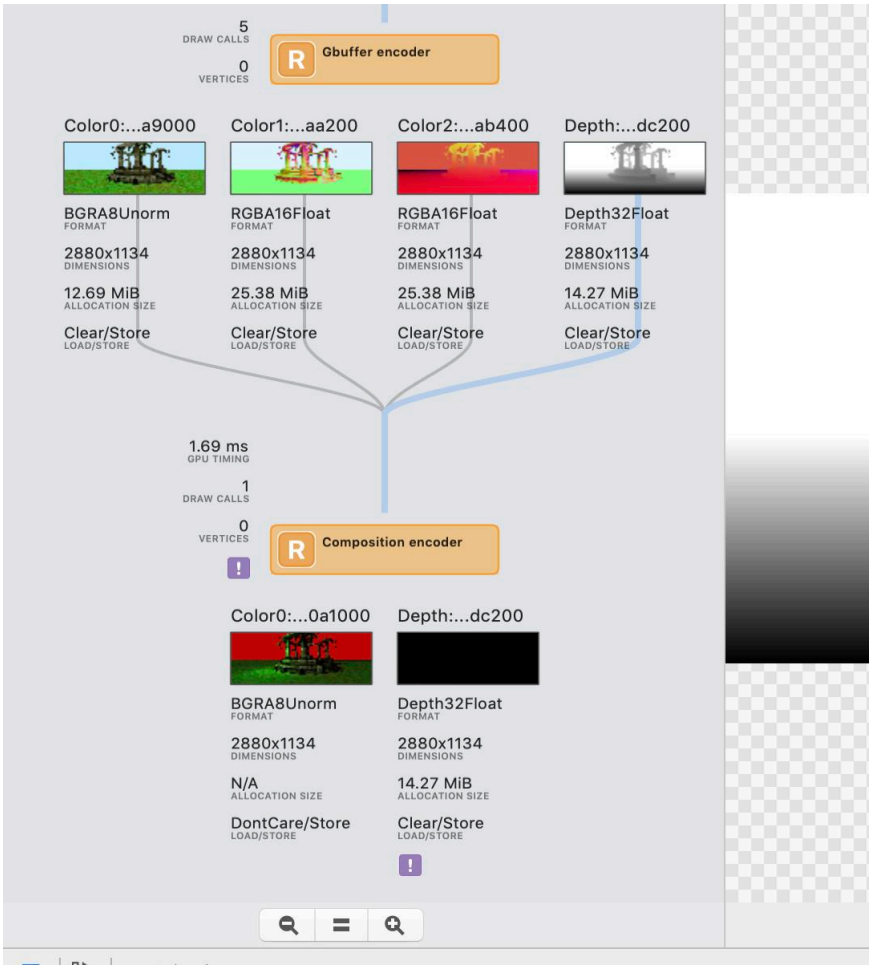
BUG POINTLIGHTS TEXTURE

I was still stuck as to how to draw the point light textures correctly.

At this point I went back to reading how depth testing works in Metal. The depth stencil testing is done after every fragment computation. Based on the MTLDepthStencilState configuration that is attached to

every MTLRenderPipeline, a fragment's z value is evaluated with the content of the depthbuffer. This is done automatically. So for me there was surely no need to do it manually as I was trying to do last Wednesday.

At this point I went to have look at a debugger and see what is being passed around on the GPU.



And that is when I noticed the depth texture in the composition pass is completely black. Hence there is no way the renderer can do the depth comparison as there is nothing to compare against. Hence, there must be an issue the depth texture not displaying correctly in the composition pass. Since it is completely black, the fragment functions cannot compare against anything as to how to draw the point light textures. Hence in order to fix this the composition pass must be writing correctly to the depth textures, which then can be used by Metal when drawing the point lights textures.

After analyzing the code the issue was with the MTLRenderPipeline for the composition pass not being set up correctly. It did not have a depth texture capability turned on. Once I turned it on, the point lights textures were displayed correctly.

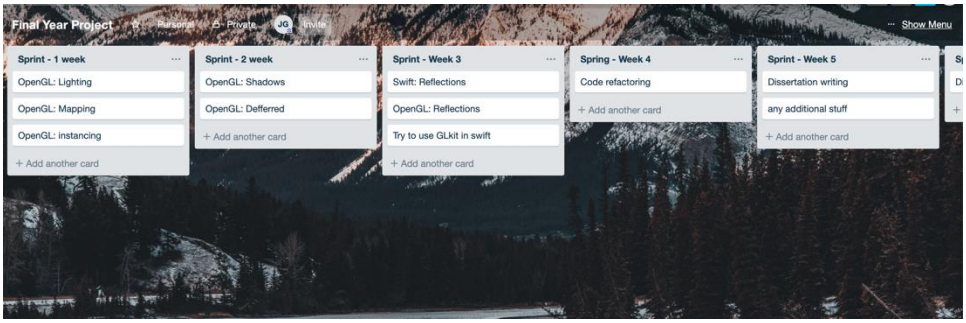
24/2/2019 (Time spent: 3 hours)

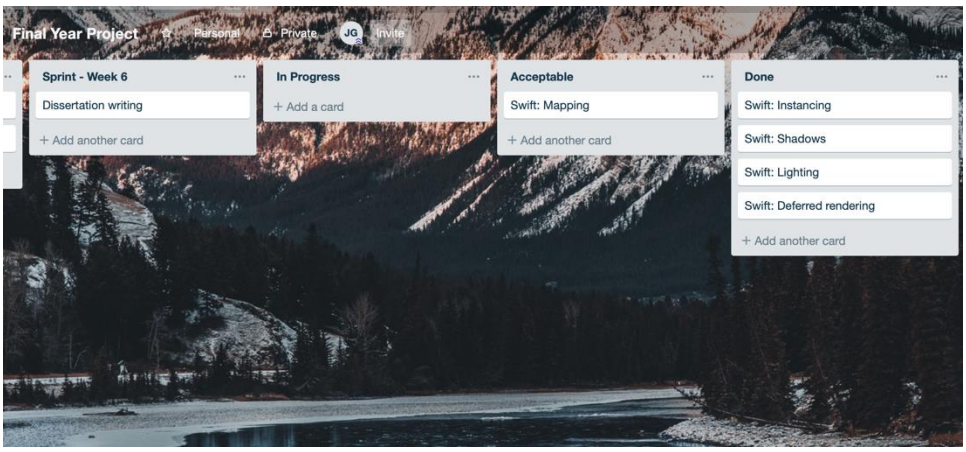
Typical Sunday code refactoring as mentioned before. No new features were added, it was just an attempt to make the code better readable. The code base is already thousands line long. So any kind of code refactoring is a lengthy process and a continuous testing needs to be done often to test that after any kind of refactoring the application is still working as expected. For example, any kind of change in the way buffers are being sent to the GPU can completely make the application stop working. Hence this must be a careful and long process.

28/2/2019 (Time spent: 2 hours)

As it is the end of the month and to stick with the plan I will now move on to the OpenGL implementation and I will finish whatever is left for the Metal part in the first week of April or last week of March depending on how things will go.

As I started the OpenGL implementation I also took some time to review my Trello board and analyze the timeframe for the next deliverables. What I noticed in my previous “iterations” is that what I am doing is sort of a one week sprint where each week I delivered a new feature which was then followed by the refactoring of the code every Sunday. As there is around 6 weeks left for the next deliverable I decided to split the 6 weeks into a weekly sprints as well. The example is depicted in the pictures below.





This is showing the next 6 weeks and what is in progress, what is in an acceptable state and what is completely done. The last two weeks (Week 5 and Week 6) are dedicated to the dissertation writing. Though this might change and take considerably less time because as I was implementing the features I

was writing a document alongside with it explaining what I have done. This means that a certain part of the dissertation is already done.

Hence, in this week I will have to deliver OpenGL lightning, parallax mapping and instancing.

2/3/2019 (Time spent: 6 hours)

This Saturday I officially started the implementation in the OpenGL. The process was not as easy as expected. The ideas behind the graphical techniques are the same, however using OpenGL to achieve those techniques is slightly different from Metal.

This is mainly due to OpenGL being based on C/C++ languages. While Metal provides a user friendliness when it comes to communicating with shaders, operating buffers etc.

OpenGL requires a developer to work with memory addresses. Thankfully Swift was designed in such a way to allow for this and it is very easy to turn it into a kind C++ language when it comes to memory management. However, when every single field is of type GLint it becomes after a while quite hard to keep track of everything.

Unlike Metal, OpenGL requires a custom shader compilation code. And unlike Metal, there is no default way in OpenGL to load 3D models. At least as of now I did not find anything that I could use for model loading.

3/3/2019 (Time spent: 3 hours)

Spent on implementing lighting in OpenGL scene. The process was exactly the same as with the Metal implementation.

6/3/2019 (Time spent: 5 hours)

This day was spent on texturing and mainly creating a fake 3D effects through normal maps and depth maps. As with the lighting the shaders were very similar to how this is done in Metal.

The only main problem was having to manually add tangents and bitangents. In Metal through the MDLObject I was able to generate these for any object automatically. However as of now I still do not have means to import an MDL object into OpenGL hence I had to define the vertices of the cube and its normals, tangents and bitangents manually. This is fine for a simple cube that has 4 faces only. However, this is not really possible with more complex objects.

7/3/2019 (Time spent: 3 hours)

Today we had a presentation about the last two deliverables (the testing evaluation and the final year project presentation). This was a useful presentation that clarified better what I have to do for those last two deliverables.

This was then followed by a meeting with my supervisor. My main questions were centered around the way I should be writing the requirements part of the dissertation as my project is not a usual design & implementation project. Some of the points taken from the meeting with my supervisor:

- I should mention the difference in debugging in Metal and OpenGL
- Requirements -> need to say that I am doing a comparison between Metal and OpenGL through using these following techniques etc. (which are requirements), then need to say why those techniques.
- I could use the UML that I used for my refactoring of the code
- There should be probably around 2000 words per each section in the deliverable

This was then followed by me implementing the last deliverable of this sprint which was instancing. This did not take too long to implement and it was actually more straight forward than in Metal. With this completed I managed to finish what I set out to finish in this sprint. The final product of this sprint can be seen in the picture below.

After this I made a few attempts to import .obj and .mtl objects. I managed to understand the data structure returned from the MDLObject a little bit better. However, as of now I still do not have a working code to be able to load models into OpenGL.

9/3/2019 (Time spent: 2 hours)

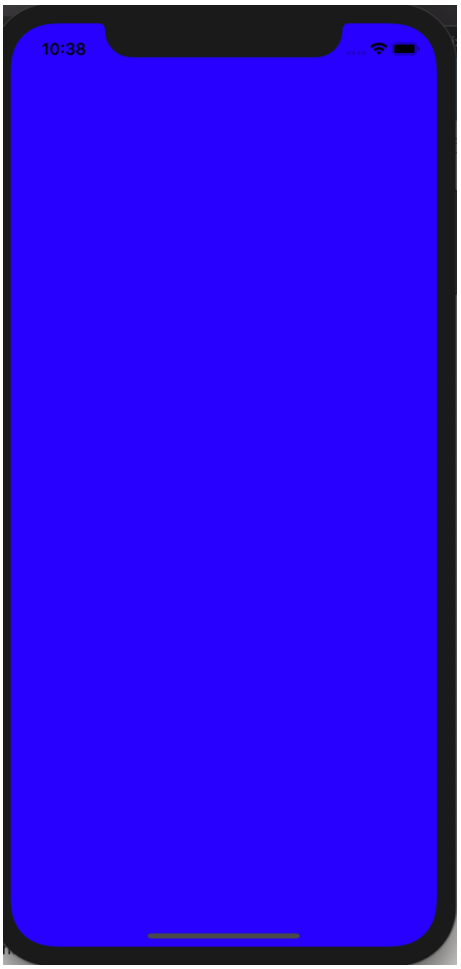
These two hours on Saturday I used to start writing my dissertation. The next deliverable will consist of requirements, design and implementation. Hence, I started working on the requirements and the design. I wrote a very rough draft so far, putting no organisation to anything. I will tidy this up during the week.

12/3/2019 (Time spent: 6 hours)

Finally after a two days break I started coding again. This week has two objectives: implement shadow mapping and deferred rendering.

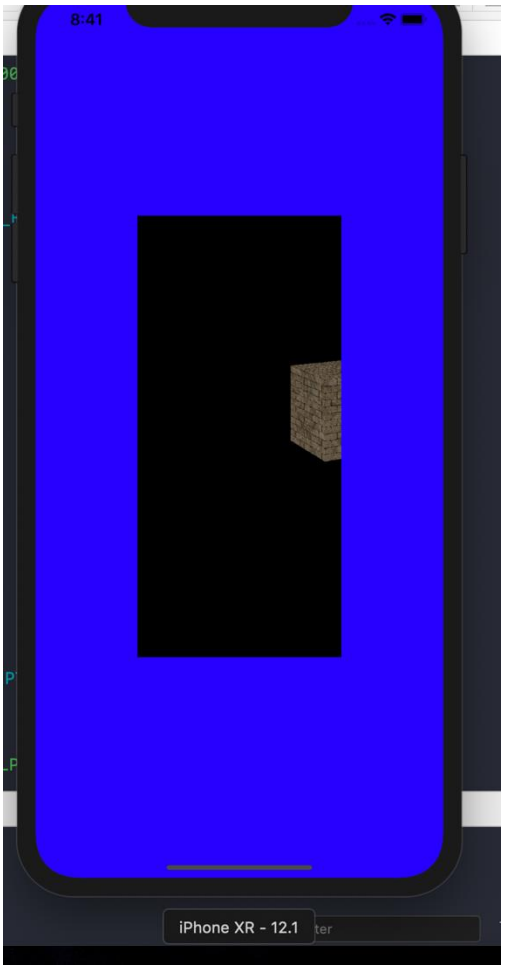
This Tuesday I started implementing a shadow mapping. The process is very similar to when I was implementing the shadow mapping for Metal. There is obviously a need to do a multi-pass rendering, which unfortunately requires a little bit of the reorganization of the code as everything until now was written using only one-pass rendering. So before I could fully start implementing the shadow mapping I had to refactor the code a little bit to allow for multi-pass rendering.

Once that was done, I had to create a first pass where I created a depth texture. Then I had to create a second pass where I created shadows from sampling the first texture. The first issue was that the display was displaying just a purple colour. This obviously did not look right and made it clear there is definitely some error somewhere.



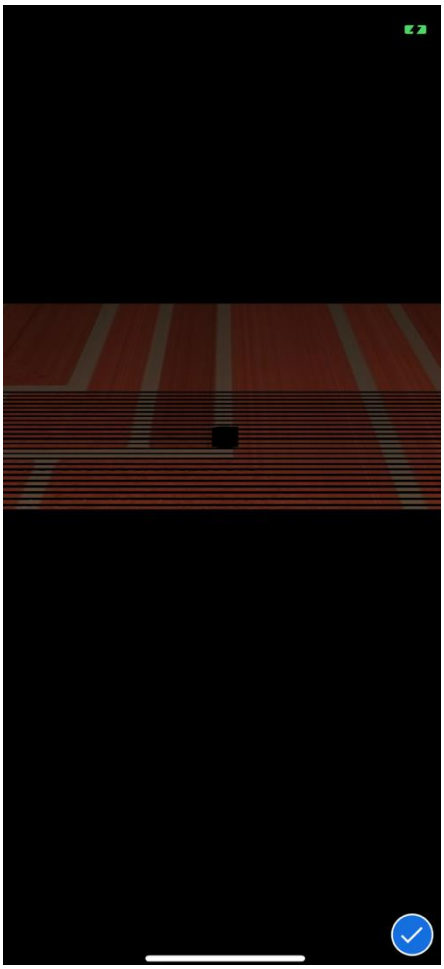
So I researched this issue and found a method called getError() in OpenGL that returns a current error that was returned somewhere in the code. The error indicated there is some issue with frame buffers. After more research and after stumbling upon an old Apple OpenGL ES documentation I found out that default framebuffer id on iOS is one number higher than the number of custom frame buffers created in the code. So this was the first issue. This took long time to solve as everywhere else a 0 is used for the default buffer. So once I fixed that the purple screen disappeared.

This finally made the rendering work fine. HOWEVER. The rendering still did not look right.



The size of the rendered screen was very small and the cube seemed to be too big. There was an issue with the texture size into which OpenGL was rendering. As previously this was down to a specific Apple implementation. On the new Retina displays the process of getting the correct size is slightly different because of the scale factor. There is a different property that must be used to get the correct screen size. Most of the resources for iOS OpenGL are quite old, hence this was not mentioned and was creating visual glitches.

However, even after this fix, the shadows still did not work correctly.



13/3/2019 (Time spent: 3 hours)

Today I carried on with trying to fix the shadow mapping. And the same issue was still present and I could not figure out why. The rendering of shadows was just completely messing up.

14/3/2019 (Time spent: 4 hours)

Finally, after so much time spent on being stuck I managed to fix the issue with the shadow mapping.

The axis orientation is different from Metal and that is something I did not realise having spent so much time with Metal. The Z axis is flipped as compared to Metal which caused the issue with shadow mapping not working properly. Because the Z axis was flipped, the shadows were displayed on the other side of the screen basically, hence I was not able to see them.

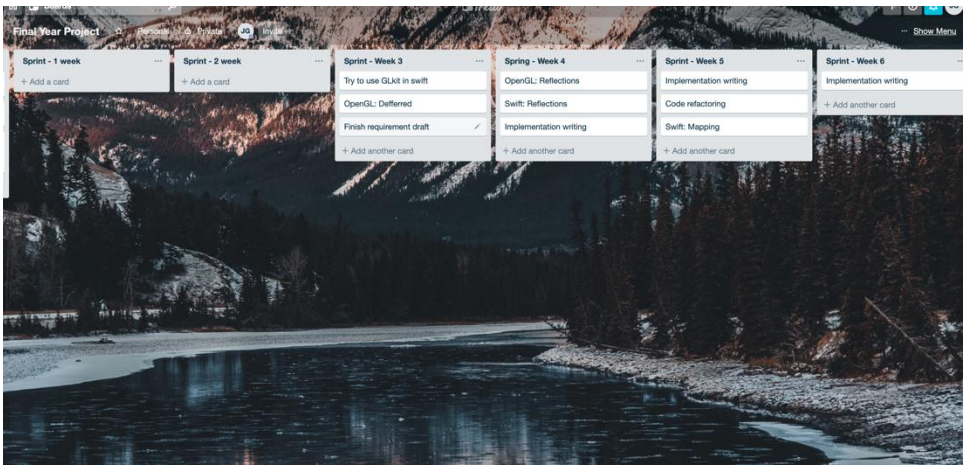
I managed to find out that this is the issue as I found a new way of debugging OpenGL on iOS. Once the iPhone is connected to the laptop, XCode is able to capture the GPU frames. This is exactly the same as GPU frame capturing with Metal. However, in Metal there is no need to connect a physical device to the laptop as it is enough to just run a simulator to access the GPU frame capturing.

The weird shadow glitch that created parallel lines was caused by the orthographic projection not being large enough.

Working shadows as per picture.

15/3/2019 (Time spent: 2 hours)

I used the time this Friday to update my Trello board and review what needs to be done.

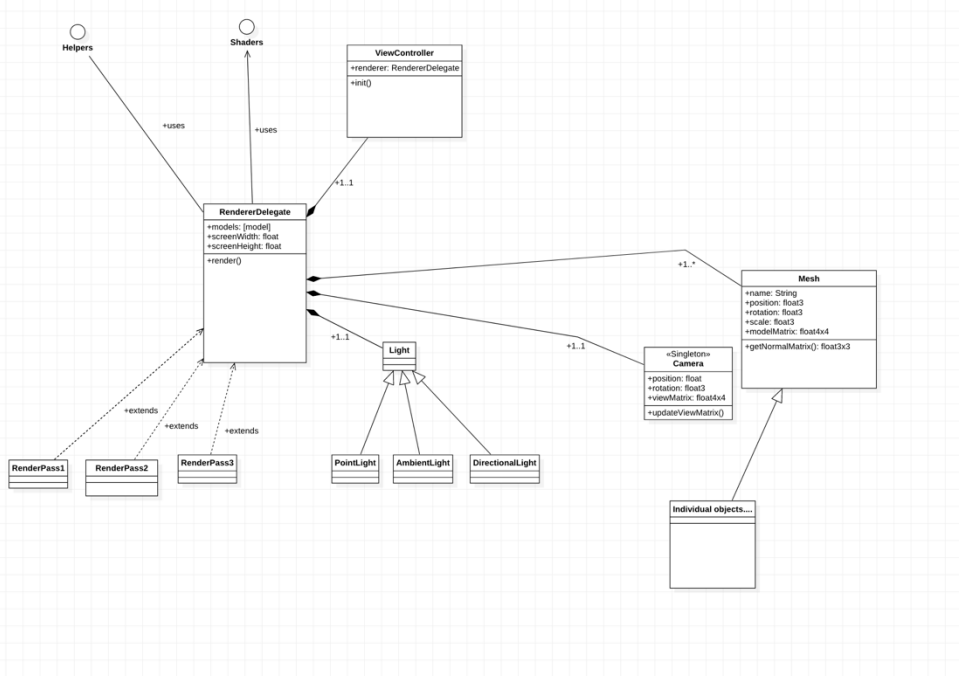


After the struggle with the shadow mapping this week I had to move the deferred rendering into next week sprint unfortunately. So there is a little bit of a delay.

I also added more details in regards to what needs to be written for my dissertation. This week I need to finish the draft for the requirements and at the beginning of the next week I need to start typing my implementation part of the dissertation.

It would be ideal in the week 3 sprint to finish the deferred rendering early so that I can move some stuff from week 4 into week 3.

The rest of the time I spent on moving my UML diagram from a physical form into a digital form through a software called StarUML. I need to still flesh out the details but so far so good.



16/3/2019 (Time spent: 6 hours)

Unusually this Saturday I spent writing the requirement and design part of my dissertation. The goal is to have it finished this weekend which is on track as of now. I wrote around 1700 words for the requirements and design. All that is left is to proofread it and send it to my supervisor so that he can give me a feedback on Thursday during our next meeting.

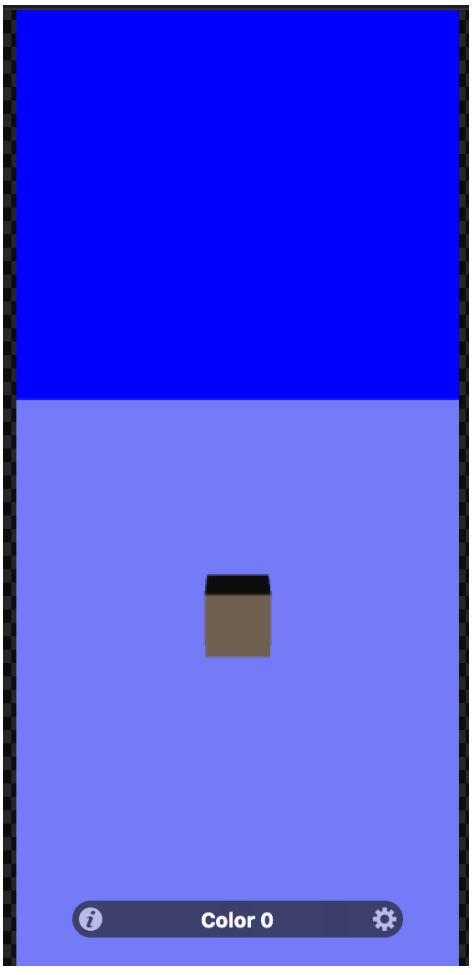
I decided to write the dissertation a little bit earlier to ensure I can put enough quality in to it as well. Instead of focusing too much on the product and then getting behind with the dissertation.

18/3/2019 (Time spent: 2 hours)

I spent this day finishing the requirements part of my dissertation. I had to rework the class diagram a little bit and add a bit more description and then proof-read my work. After that I sent the first draft of the requirements to my supervisor.

19/3/2019 - 21/3/2019 (Time spent: 8 hours)

I started working on the deferred rendering. The initial process was the same again as with the Metal. The principles are identical to how Metal works. There are slight differences in implementation for sure though. The whole set up of the deferred rendering took some time. When I finally set everything for the deferred rendering a first issue appeared:



After the shadow pass and the G-Buffer pass and sending all the necessary textures to the composition pass, the scene was displayed with weird colours as above.

I used the new debugging feature I discovered when I was working on shadow mapping and it revealed the textures that are being sent to the composition stage looks bit skewed for some reason.

Upon this I examined the vertex shaders and fragment shaders to ensure the correct processing is done. I also compared this to my shaders in Metal, which are quite identical, apart from a different syntax.

Everything seemed fine.

This took lots of time to figure out but when I was examining the textures send between buffers I noticed they look as if they were blending.

So that took me to examine the render logic I had for shadow, g-buffer and composition passes, which is where I noticed the issue – I was not clearing the buffers properly between individual passes, which caused the textures skewing up. Once I cleared the buffers properly, the problem disappeared.

In the rest of the time I spent converting all the techniques I implemented so far to make them work with the deferred rendering. This took some time as I was doing it carefully in order to ensure I do not break what I already implemented before and works.

22/3/2019 (Time spent: 4 hours)

This Friday I spent time implementing parallax mapping for the Metal scene, which I did not do before. I only did a normal mapping, but skipped the parallax mapping at that time as I did not have a correct depth textures to use for the parallax mapping. This time I finally got ahold of some depth textures but had to use Photoshop to turn them into height textures to make them work in the Metal scene.

The rest of the time I spent time trying to implement the parallax mapping for the Metal scene. I did not finish it this Friday.

23/3/2019 (Time spent: 2 hours)

This Saturday I spent time on trying to make the model loading work for OpenGL. I tried different approaches to how this could be done. Unfortunately, there are absolutely no resources for this online. Hence, I was left to my own trying. What I did found was an old presentation from Apple in 2015 at WWDC where they mentioned the MDLMesh is able to work with OpenGL. However, as 2015 was already a year when Metal existed, the details about how MDLMesh can be used with OpenGL were not provided. Link for that presentation [WWDC15 Graphics and Games - Applehttps://devstreaming-cdn.apple.com/.../602_managing_3d_assets_with_model_io.pdf?...]

So there was an encouragement in knowing that there must be some way to make this work.

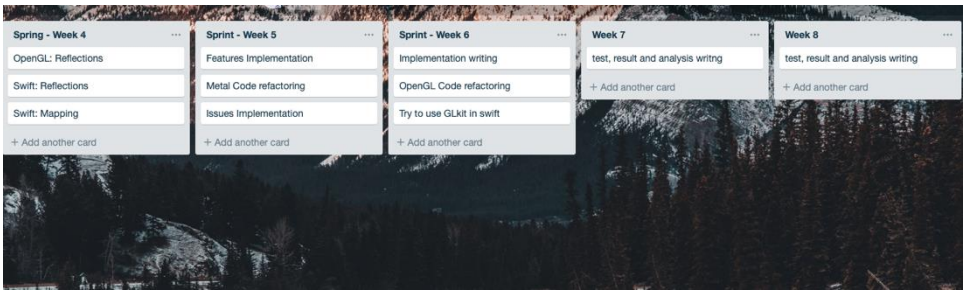
One of the approaches was to try to plug the vertex buffer from the MDLMesh straight into the vertex buffer object for OpenGL. This however failed because the data types of individual vertices in the vertex buffer are different to what OpenGL expects.

The MDLMesh allows to access individual attributes in the vertex buffer individually. So I tried that as well. Basically, I tried to access each individual vertex attribute in the memory, use its data type and the data type's memory size and tried to load it into data types OpenGL can understand. The issue here is

the MDLMesh does not provided a number of how many there are. Hence, when accessing this through memory is not possible as it is not known when to stop.

24/3/2019 (Time spent: 3 hours)

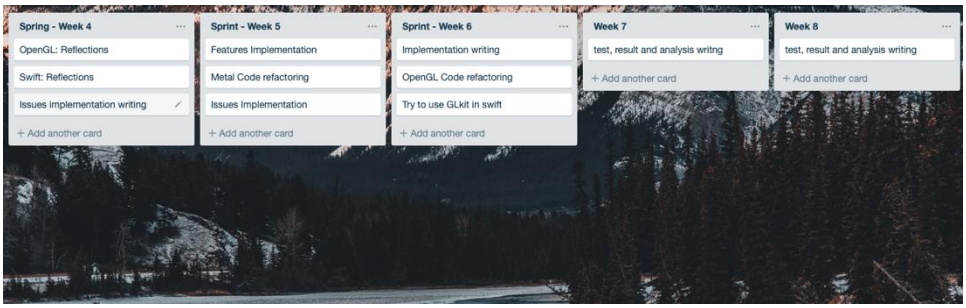
On Sunday I carried on with trying to make the model loading for OpenGL work.



The updated Trello board for the next week' sprint. The plan is to finish reflection for both Metal and OpenGL. And then finish the parallax mapping for the Metal scene.

25/3/2019 (Time spent: 3 hours)

Parallax for metal done



Since I managed to finish parallax mapping for Metal today I updated the Trello board again. My limit was 3 tasks per week, but since I finished the parallax mapping earlier than I thought I put a new card into this week's sprint – to write the issue part of the implementation part, which I will be working on tomorrow.

26/3/2019 (Time spent: 2.5 hours)

Today I just spent the time writing my dissertation. I focused on the issues I had during the implementation. I discussed the model loading issue for OpenGL in detail.

Currently I have around 1000 words written for that. So with the design and requirements part which has 1700 words I have altogether 2700. There are still 2300 words left to do for my next deliverable.

27/3/2019 (Time spent: 2.5 hours)

I started implementing the reflection technique for Metal. There are several ways to do it. In my objectives I didn't specify actually how I will do it. So I started with the simplest one as there is a time pressure now to finish soon so I can focus on the dissertation and other assignment.

The simplest type of reflection is through a sky cube map where an object only reflects a sky cube map around it but does not reflect any surroundings objects.

If the object needs to reflect the surrounding objects as well, 6 renders need to be done from the object that is supposed to reflect where each render creates a part of the cube map. Through the research I done Metal does offers a simplified and more efficient process of this, however there might not be time to do this.

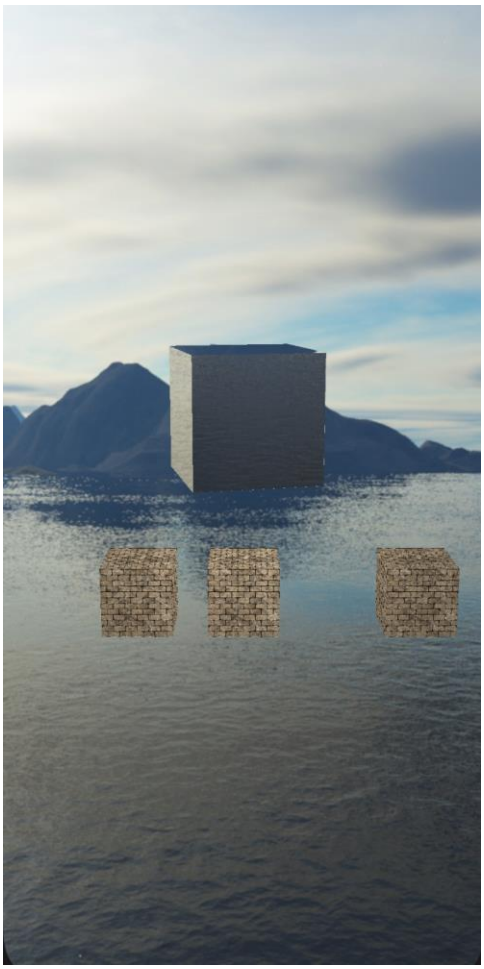
I managed to finish the cube map reflection for Metal today.

// add picture

28/3/2019 (Time spent: 2.5 hours)

Today I spent implementing the cube map reflection for OpenGL. The process is very similar, with few dissimilarities. Everything went fine until the end where I encountered a flickering issue. This issue was not present in the simulator, however it was present when connecting a physical device to the laptop and running it on it. The research indicated this could be a problem with the iOS implementation of OpenGL. And there were a few fixes that I tried out but none of them worked.

After looking around the code, the issue was with the normalMatrix that I creates in the shader by transposing and inverting a model matrix. Doing this on the CPU instead and then sending it to the GPU, instead of doing it on the GPU in the shader, seeme to have fixed the issue:



This means that all that techniques I set out to implement in each of the scenes are done now. Apart from the water surface, however, that is out of the scope due to the maths involved.

29/3/2019 (Time spent: 2.5 hours)

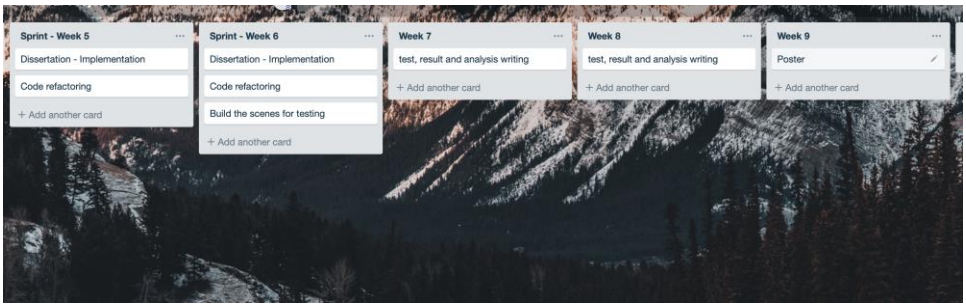
Dedicated the time to write about the implementation part of my dissertation. Described all the techniques briefly as there is a limiting word count limit.

30/3/2019 (Time spent: 1 hour)

Written a few bits for the implementation part.

31/3/2019 (Time spent: 3 hour)

Spending the day to work on the dissertation and the log book.



Updated the Trello board. The rest of the weeks will be about the dissertation writing. This week and the next week I will write the implementation. The last two weeks I spent writing the evaluation and test results.

Code-wise I will do some code refactoring as towards the end I was trying to implement the techniques as quickly as possible.

Next week I will assemble the scenes for the testing and evaluation.

The log book now will be updated sparingly as there is no point writing about much since everything was now implemented.

29/4/2019

The last month was spent on writing the dissertation. No more implementation was done.

Project finished

Appendix B

Project Proposal

Comparison of OpenGL and Metal graphical APIs

Student: [redacted]
Supervisor: Norman Murray

Aim

The aim of this project is to compare the Metal and OpenGL graphical APIs for MacOS and iOS. This would be carried out through creating two exactly the same graphical scenes showcasing a set of rendering concepts in both APIs and then comparing the efficiency, user-friendliness, complexity, timeframe and the output quality of both solutions. This will then lead to the recommendation as to what graphical API is best suitable for developing graphical applications for MacOS and iOS.

Background

The Metal and OpenGL are currently the only competing graphical APIs for MacOS and iOS devices. OpenGL is an open source cross-platform graphical API for creating 2d and 3d graphics. It is a graphical standard implemented in GPU drivers. This API was developed in 1992 and since then became the most updated graphical API. It provides a low-level access to the GPU with the main benefit the code runs on many platform without having to do any extra conversions.

Both APIs can be used to write shaders, which are programs of small size that run directly on the GPU and are responsible for displaying and manipulating pixels to create graphical effects, such as reflections, lighting and to create transformations . GPU processing is used due to its parallel architecture, which is useful in computer graphics due to the need to process lots of pixel transformations and computations at once.

The Metal API is very similar to OpenGL in that it provides a very low-level access to the GPU. However unlike the OpenGL, it is a graphics API completely developed by Apple and only supported on Apple devices, hence not providing any cross-platform development. The Metal API is currently supported on all Apple devices including Macbook, iPhones and Watches. For shader writing it is using a C++ like language. This API was released for the first time in 2014 for OS X El Capitan and then replaced by Metal 2 in 2017 with the release of Mac OS High Sierra. It is officially the preferred graphical API for the Apple ecosystem.

There is a need for this comparison between these two competing APIs due to the latest changes made by Apple in their latest MacOS Mojave. As of September 2018 the OpenGL standard has been deprecated in MacOS and iOS. From now on Apple will not be supporting OpenGL anymore and no new versions are going to be released. Even though the standard is now deprecated, developers can still write their applications using OpenGL, which then poses a question whether to keep using OpenGL when developing for the Apple ecosystem or whether to jump on and use Apple's own proprietary graphical API.

Who will benefit?

The project is aimed at any developers developing multi-platform graphical applications that they expect will run on Apple products and have been using OpenGL as their main graphical API. The developers should benefit from this project as it should help them make decisions as to whether it is worth spending time and learning the Metal API kit when developing for iOS and MacOS or whether it would be sufficient right now for them to stay with OpenGL.

Another group of people benefiting from this project will be developers just starting to get into computer graphics. This project should give them an idea what both APIs offer and which one they should focus on.

This project would also be useful for project leaders giving them some ideas around how much time each technology takes to learn which then they can incorporate into their own project planning.

Motivation

The main motivation behind this project is to learn more about the computer graphics and explore it deeper than the university programme allows and to do something different than usual desktop application development and web development which I have already plenty of experience with especially from my year in the industry. The aim is then to use the experience gathered during this project to open better career options and possibly try a career as a game developer, which always require having some kind of background experience in game programming techniques.

Also what interests me about this project is its complexity in an area that is for me rather unknown as of yet. I would like then use this opportunity through this project to stretch myself mentally and see how well I am able to cope with this kind of complexity.

Objectives

- **OBJECTIVE 1:** Learn Metal by reading through the Apple Official Documentation
- **OBJECTIVE 2:** Learn OpenGL by watching through the Computer Graphics course on Edx
- **OBJECTIVE 3:** Render a scene in Metal using deferred shading which has HDR lighting and contains 3d objects with stencil reflections and shadow mapping, walls rendered using parallax mapping, hundreds of objects rendered using instancing and simulated water surface
- **OBJECTIVE 4:** Render a scene in OpenGL using deferred shading which has HDR lighting and contains 3d objects with stencil reflections and shadow mapping, walls rendered using parallax mapping, hundreds of objects rendered using instancing and simulated water surface
- **OBJECTIVE 5:** Compare the performance of both solutions by measuring FPS
- **OBJECTIVE 6:** Compare the difficulty of coding of both solutions though code comparisons
- **OBJECTIVE 7:** Compare the ‘time-efficiency” of both solutions, i.e. time to learn each of them
- **OBJECTIVE 8:** Compare the visual quality through a questionnaire

Optional Objectives

- **Optional Objective 1**: Learn about skeletal animations using OpenGL and Metal through their official documentations
- **Optional Objective 2:** Implement a skeletal animation of an object into the scene created for this project in OpenGL and Metal

Development Requirements

The project will be developed on the MacBook Pro 13 released in 2017. The laptop has an i5 2.3Ghz Intel CPU, 16 GB of RAM and 256 SSD. The laptop is running the MacOS Mojave.

The second version of the MetalKit will be used to write the shaders, hence the requirement is MacOS High Sierra though it is recommended to use MacOS Mojave to provide the best compatibility.

The OpenGL version 3.2 will be used to write the shaders as that is the latest version support by Apple.

Xcode version 10 will be used to develop the actual product. Xcode is freely available and can be downloaded from the Apple Store. The Xcode can be used to write both OpenGL shaders and Metal shaders. It also have debugging tools needed to evaluate the performance of both solutions.

Microsoft Word and Microsoft Excel will be used to write the report and create graphs.

TeamGantt was used to create the Gantt chart and will be used throughout the project to track its progress. This can be accessed through a web browser and is free to use for personal projects.

Google Drive will be used to write a log book, which will be shared with the supervisor. This will be accessed through a web browser

All the software used for this project is available in the university computer rooms.

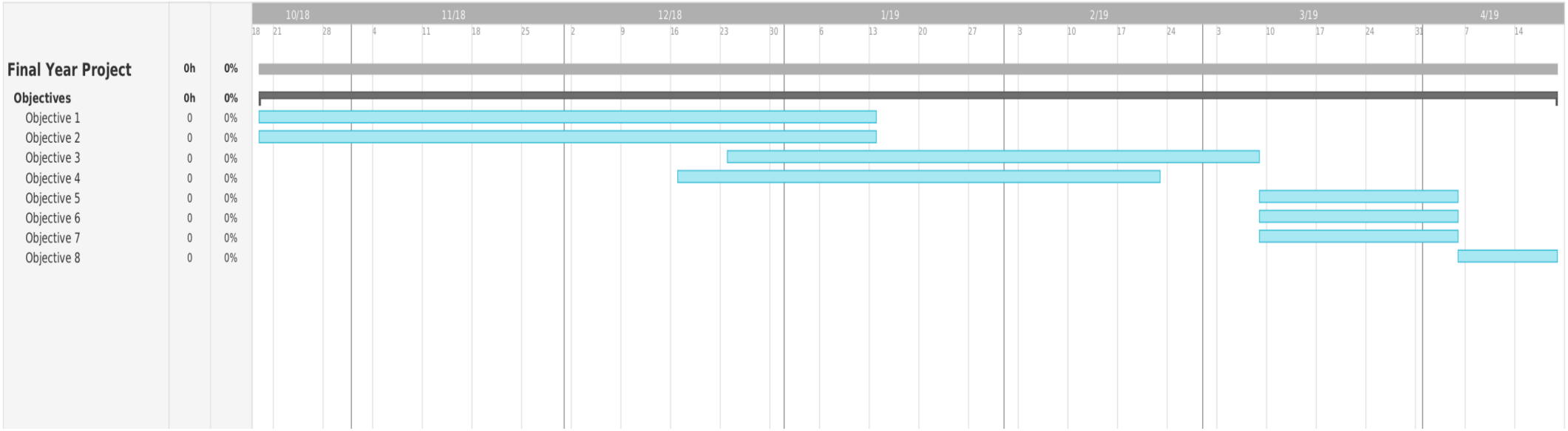
Methodology

Agile methodology will be the methodology of choice for this project. The agile methodology is focused on developing software in an iterative and incremental way. The methodology allows a fast development of software while being able to react to changes in the requirements. It emphasis a functional software over documentation, cooperation over tools and processes, ability to react to changes, and delivering individual small pieces instead of a whole product at once

This methodology seems to be a great choice for this project mainly due to how well it copes with changes and provides a good flexibility. This project is being carried out with very low knowledge background in OpenGL, Metal and shader writing in general. Hence, it is very likely that when it comes to writing selected shaders in each of the graphical API some changes might be necessary. This could either due to the complexity of the proposed shaders or due to the short timeframes. Either way, using the agile methodology would then allow to react to these issues and adjust some of the requirements.

Also the implementation part of this project is based on rather small deliverables instead of a one big product. This is then again where the advantage of the agile methodology will proved to be useful.

Gantt chart



- **OBJECTIVE 1:** 19/10/2018 - 13/01/2019, 12 weeks, 3 days
- **OBJECTIVE 2:** 19/10/2018 - 13/01/2019, 12 weeks, 3 days
- **OBJECTIVE 3:** 25/12/2018 - 08/03/2019, 10 weeks, 5 days
- **OBJECTIVE 4:** 17/12/2018 - 22/02/2019, 9 weeks, 5 days
- **OBJECTIVE 5:** 09/03/2019 - 05/04/2019, 4 weeks
- **OBJECTIVE 6:** 09/03/2019 - 05/04/2019, 4 weeks
- **OBJECTIVE 7:** 09/03/2019 - 05/04/2019, 4 weeks
- **OBJECTIVE 8:** 06/04/2019 - 19/04/2019, 2 weeks