Sheffield Hallam University

Faculty of Arts, Computing, Engineering & Sciences

Department of Computing

Project (Technical Computing)

[55-604708]

2018/19

Author:	**Stephen Marc Wilks**
Year Submitted:	**2019**
Supervisor:	**Thomas Sampson**
Degree Course:	**BSc (Honours) Computer Science**
Title of Project:	**Situational Analysis of Modern Rendering Techniques for Real-Time 3D Graphics**

Confidentiality Required?

NO ☐

YES ☐

Abstract

When developing real-time 3D graphics applications, it is important to use the most optimal performing rendering technique that suits the needs of the application. Performance of the application can be severely impacted if the wrong technique is chosen. Realising that the wrong technique was chosen late in development could potentially be catastrophic. This project investigates forward, deferred, forward plus, and tiled deferred rendering techniques, giving detailed overviews of each and presents performance data for them under various situations. This aim of this investigation is to provide developers with the information required to give confidence when a decision needs to be made as to which rendering technique should be used.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

At the heart of any real-time 3D graphics application, a rendering technique is used to define the steps taken to render the 3D scene. There are various rendering techniques to choose from and the decision as to which is used rests in the hands of the developer. Each technique has different benefits, drawbacks, and features that developers need to be aware of in order to choose the most suitable and best performing technique for their application. Despite the copious amount of research and development surrounding rendering techniques, there are few resources that provide a developer with enough information to confidently identify which is most suitable for their use case. Being uninformed could result in the wrong technique being chosen which may lack required features or could lead to significant performance losses. Changing rendering techniques late in development could break existing components of the render pipeline and potentially cause other problems such as assets needing to be reworked. The time that this may require isn't always viable on commercial applications with tight time budgets such as Video Games. Therefore, it is of upmost importance that during project planning, the correct rendering technique is chosen.

1.2 Real-Time Rendering

Real-time 3D graphics applications, as the name implies, use real-time rendering which is a focus on producing and presenting images rapidly in real-time. It is used in highly interactive applications where user interaction can affect the next image that is generated. This cycle of generating images and displaying them must be fast enough that the user does not notice

individual images and has a smooth and seamless experience (Akenine-Möller et al. 2018).

The rate at which the images are generated and displayed is called frame rate. There is typically a goal to achieve a frame rate of at least 30 frames per second. Below this threshold it begins to become painfully obvious as the next image is presented, therefore diminishes the experience and is unacceptable (Akenine-Möller et al. 2018).

Graphical fidelity in real-time rendering is limited by the processing power that it takes to render each frame and achieve the frame rate target of the application. The reason that computer-generated imagery in films can appear photo-realistic at times is because offline rendering is used. Films have no requirement to deliver a particular number of frames each second in real-time, and therefore can be rendered offline to be displayed to a consumer as an array of static images. An example of the power required to render a single frame in a film is that in the Pixar film *Monsters University*, it took 29 hours to render a single frame using the combined power of 2,000 computers (D. Takahashi 2013).

1.3 Aims & Objectives

The aim of this project is to investigate forward, deferred, forward plus, and tiled-deferred rendering techniques for real-time rendering, providing enough information for developers to give them confidence in selecting the most appropriate of these techniques based on the requirements of their application.

The project deliverable aims to support the investigation by facilitating performance analysis of the target rendering techniques under varying conditions. It should implement the target rendering techniques and allow for run-time adjustment of a 3D scene to achieve this. Which render technique is being used should be switchable at run-time in order to easily assess how each technique performs under the varying conditions. The performance of the application should be presented in real-time to give instant feedback as to the effect that the changes make. The ability to perform captures of performance data to a file should also be supported to allow for offline data analysis.

Chapter 2

Research

2.1 Rendering Techniques

2.1.1 Forward

Forward rendering is the simplest rendering technique. There is only a single lighting pass that rasterizes the objects in the scene and during shading, the lighting on the object is calculated for every light (Oosten 2015). This results in the algorithmic complexity seen in figure (2.1) which relates directly to the number of objects in the scene and the light count. This highlights the fact that forward rendering scales poorly when introducing many lights to a typical 3D scene.

$$\mathcal{O}(object\_fragment\_count \times light\_count) \tag{2.1}$$

This technique suffers from pixel overdraw which can be expensive in a complex scene. It is caused by repeating lighting calculations on all objects that rasterize a particular fragment. This results in a huge number of unnecessary lighting calculations (Harada, McKee, and Yang 2012).

2.1.2 Deferred

Deferred rendering attempts to solve the complexity and overdraw issues in traditional forward rendering. It works by deferring, as the name suggests, the lighting calculations until the geometry has been rendered. This means that there are two passes, geometry and lighting.

In the geometry pass, the objects in the scene are rasterized into several images with each image containing different geometrical information. This concept is referred to as a geometry

buffer, or G-Buffer (Saito and T. Takahashi 1990). An example of a fully composed G-Buffer can be seen in figure 2.1.

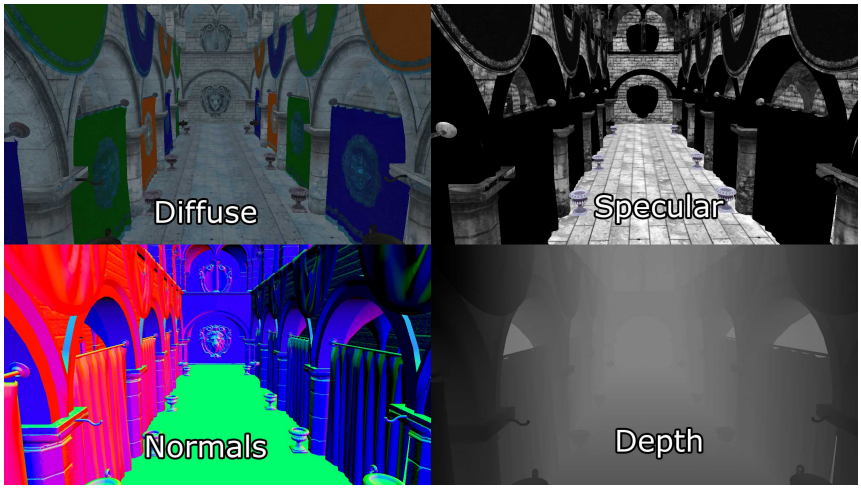


Figure 2.1: G-Buffer Composition Example (Oosten 2015)

Once the G-Buffer is filled with the relevant geometrical data, it is then sampled during the lighting pass, allowing for lighting to be able to be calculated for each fragment. This results in the algorithmic complexity seen in figure (2.2).

$$\mathcal{O}(screen\_resolution \times light\_count) \tag{2.2}$$

Although this technique already proves to be a large optimization over forward rendering because each fragment only calculates lighting once, there are ways to further reduce this cost. Optimizations such as *Efficient Light Volumes*, *Stencil Masking*, and *Dynamic Branching* all have the same goal of reducing the number of fragments considered for lighting, resulting in a reduction of lighting calculations, and therefore reducing the overall cost (Koonce 2008).

One of the biggest problems that this technique has is its high memory and bandwidth requirements due to the usage of the G-Buffer. If there are four render targets in the G-Buffer, that means that four times as many bytes are being written as a traditional forward renderer which only has a single render target. Also, the G-Buffer needs to be sampled many times during the lighting pass which increases the bytes read significantly (Koonce 2008). There are ways to reduce the size of the G-Buffer so that it has a smaller memory footprint such as tighter packing of geometry information, however the high memory and bandwidth requirements are largely unavoidable.

Another problem is that this technique does not have transparency support because it only shades the nearest surfaces (Hargreaves and Harris 2004). There are ways around this

issue, one common fix being to have an additional that uses traditional forward rendering for transparent objects (Shishkovtsov 2005).

This technique is incompatible with hardware-assisted aliasing because it requires analysis of 3D geometry. Deferred rendering uses images therefore it is immune to the effects of hardware-assisted aliasing (Koonce 2008). The most common solution to this problem is to use a post-processing solution such as FXAA, an algorithm that is used to smooth the jagged edges of an image.

2.1.3 Forward+

Forward+ rendering, also known as tile-based forward rendering, was presented by AMD as an alternative to deferred rendering. It attempts to solve the high memory and bandwidth issues associated with a traditional deferred renderer (Harada, McKee, and Yang 2012).

The difference to a traditional forward renderer is that it adds two additional passes prior to the lighting pass, those being a depth pre-pass and a light culling pass (Harada, McKee, and Yang 2012).

In the depth pre-pass, the objects in the scene are rasterized into a depth image. The purpose of the depth image is to be used during light culling to reduce pixel overdraw in the lighting pass which is especially expensive in a forward+ renderer. This is done similarly to the geometry pass filling the G-Buffer in the deferred renderer, although because the forward+ renderer only uses a single image instead of several, it requires much less memory and bandwidth (Harada, McKee, and Yang 2012).

The light culling pass splits the screen into tiles, keeping a list of the indices of the lights that are visible within the tiles. This list of indices is then used in the lighting pass to reduce lighting calculations (Harada, McKee, and Yang 2012).

The lighting pass is the almost the same as the traditional forward renderer aforementioned, however instead of calculating the light on each object for all of the lights in the scene, it only uses the lights that contribute to each particular fragment (Oosten 2015).

The benefit of using this technique over a deferred renderer is that, because it is based on a forward renderer, support for transparency and hardware-assisted aliasing is not lost (Harada, McKee, and Yang 2012).

2.1.4 Tile-based Deferred

Tile-based deferred rendering is a modification to a traditional deferred renderer that primarily aims to solve the bandwidth problems (Lauritzen 2010).

The difference to a traditional deferred renderer is that the lighting pass is replaced with a compute shader. This shader divides the screen into screen-space tiles, culls the lights per tile, and then uses the G-Buffer to compute lighting for each pixel using only the contributing lights per tile (White and Barré-Brisebois 2011). The output is exactly the same as the lighting pass in a traditional deferred pipeline, just achieved differently. This single pass approach means that the G-Buffer only has to be read once which results in constant and absolute minimal bandwidth (Andersson 2009).

The fact that this technique relies on a compute shader could prove to be a problem. Compute shaders were introduced fairly recently, therefore not all machines support them (Andersson 2009).

This technique has the advantage over traditional deferred rendering of being able to do efficient MSAA (Lauritzen 2010). It does not however, fix the problem that a traditional deferred renderer has regarding transparency.

2.2 Measuring Performance

When measuring performance in any application, it is of high importance that the most consistent and effective data is collected. There are two common measures of performance in real-time 3D graphics applications application, those being frame rate and frame time. Frame rate is the measure of how many frames are completed within a specific time period, typically a second. Frame time is how long, typically in milliseconds, that a frame takes to complete. When comparing the two, the difference is that frame time is a linear measure, whereas frame rate is not, therefore it is best to use frame time. The nature of frame time being a linear measurement means that it is easier to work with when measuring how different changes affect overall performance (ARM 2011).

2.3 Analysis

The research conducted in this chapter has provided a detailed overview of various rendering techniques and how to measure performance in the type of applications that requires them.

It has covered the main benefits, drawbacks, and features of various rendering techniques

and should enable a developer to begin narrowing down the suitable rendering techniques for their application. The underlying concepts of each rendering technique were also described which will form the basis for the implementations in the project deliverable that are described later in this report.

When analyzing the performance of real-time 3D graphics applications, research has shown that frame time is the best metric to capture and this will be used for the performance analysis later in this report.

Chapter 3

Planning

3.1 Version Control

Version control, specifically Git, was used for the project deliverable to keep track of the change history. Version control systems are a necessity when it comes to developing software, and prove especially useful when a change breaks existing systems. The ability to compare the differences between a broken version and a working version can help massively in identifying the cause of the breakage (Azarian 2013). To maintain a healthy change history, at every change worth noting, a commit is to be made.

3.2 Programming Language

Due to it being the language of choice in the game industry, an industry synonymous with real-time 3D graphics and a desire to extract every last bit of performance, C++ was chosen for the project deliverable. C++ is a high performance language that has a certain caveat that must be considered when using it due to its low-level nature. That caveat is manual memory management, something that isn't an issue in high level languages such as Java or C#. These languages both have garbage collection systems that clean up allocated memory automatically. Automatic garbage collection can be either a negative or a positive depending on the software that it is being used to develop. Managing memory manually in low-level languages allows for much higher performing software, however if it's not done right, it can be slower than languages that do it automatically and cause a range of other problems.

3.3 Graphics API

Considering that one of the project goals was for the project deliverable to be cross-platform, the options were narrowed down to OpenGL and Vulkan. New and innovative graphics APIs such as Vulkan have complex low-level APIs that take much longer to learn because of all of the boilerplate code required to extract every bit of performance. For this reason, time constraints meant that a decision was made to use OpenGL for the project deliverable.

When writing an application using OpenGL, there is still a lot of boilerplate code for each operating system relating to creating a window, receiving input, and timing, however there are libraries to avoid it. Libraries such as SDL and SFML are bigger libraries that abstract all low level OpenGL and provide a more high level experience, but also include built-in features such as audio and networking. There are also lightweight libraries such as GLFW and FreeGLUT which provide a more low-level experience and only abstract away minimal boilerplate code. FreeGLUT is based on the GLUT API therefore it's quite old whereas GLFW is a more modern library. Therefore the decision was made to use GLFW as it provides a modern API and gives low level control over OpenGL whilst abstracting away the boilerplate input, window handling, and timing code that doesn't affect the renderer.

Because OpenGL is constantly evolving, the core version of OpenGL on a machine may not support the latest functionality. Rather than managing the pointers to the OpenGL functions manually, there are third party libraries that can be used in order to avoid the headache and give quick access to all of the latest functionality (Shreiner et al. 2013). There are a number of these libraries available such as *GLEW*, *glad*, and *gl3w*. *GLEW* was chosen as the extension loader for the project deliverable due to the fact that it's the most well documented example out of the options available.

3.4 Build System

When working on software using multiple platforms, it's not ideal to maintain builds for each platform. Because the project deliverable may be developed using multiple platforms, a solution for this problem was required. A tool that allows for a build configuration to be defined so that software can be built everywhere is called a build system. Without using a build system, the builds for all platforms have to be handled manually (Jorge 2018).

A requirement from the build system was that it generated the relevant build scripts which ruled out Make and Ninja (Jorge 2018). Besides those, the two most stand out build systems

that generate build scripts were Premake and CMake.

CMake is a fast and very popular build system tool in the C++ world. The biggest obstacle of CMake is that the build configuration file is written in a language unique to CMake. This results in excessively verbose build configuration files and requires learning, albeit fairly simple, a new language (Jorge 2018). Premake on the other hand, is a tool that uses LUA for the build configuration file which results in a simple and easy to read syntax that is familiar to many developers (Perkins 2018). For this reason, it was decided that Premake would be used for the project deliverable.

3.5 Mathematics Library

Although building a math library is a great way to understand the underlying mathematical concepts, there are libraries that can be used for the math operations in the project deliverable that are guaranteed to be written correctly and are highly efficient. It is easy to make small mistakes when writing code related to mathematics, and very difficult to debug. It requires a significant amount of time to implement and test the correctness of all of the math functions. Due to the time constraints of the project, the decision was made to use a math library.

There were a few mathematics libraries that came under consideration because of their support for vector and matrix operations such as Eigen, GLM (OpenGL Mathematics), and GMTL (Generic Math Template Library). The fact that GLM is targeted to users of OpenGL and is specifically designed for graphics programming made it an easy choice. The benefit of using GLM within an OpenGL project is that it is based on the GLSL specifications, therefore it uses the same naming conventions and functionalities. This means that code translates quite well between C++ and GLSL (Creation 2018). GLM will handle all of the operations related to vector and matrix math within the project deliverable.

3.6 User Interface

Implementing a user interface is a time consuming process. There are various third party libraries that handle this very well. It was decided that using a library for the user interface was the best option given the time constraints.

The requirement for the user interface library was that it contained the functionality that would be needed to display real-time performance statistics and change various settings of the project deliverable with elegance and ease. Through extensive research, it appeared that

several libraries including *NanoGUI*, *Nuklear*, and *dear imgui* were suitable for the basic user interface needs of the project deliverable. The decision was made to use *dear imgui* out of the available options for the project deliverable, this was influenced by the fact that it is used in such a large quantity of commercial software (Cornut 2019).

3.7 Model Loader

In order to place realistic 3D models into the scene within project deliverable, 3D model files would need to be loaded. There are hundreds of 3D file formats available and they support a varying range of features from just geometry and material information to including 3D animation (Chakravorty 2019). For the project deliverable, only basic geometry information was required. After looking at the various formats, it was clear that *OBJ* had the most basic format and met the requirements. For this reason, the decision was made to use the *OBJ* format.

Chapter 4

OpenGL

OpenGL is a cross-platform application programming interface (API) used to communicate with the graphics hardware. The API gives a developer access to over 500 commands documented by the OpenGL specification that can be used to create interactive 2D and 3D applications (Shreiner et al. 2013). The implementation of the OpenGL libraries are typically provided by the manufacturer of the graphics hardware (Vries 2015).

When developing an application using OpenGL, it's important to be aware that OpenGL is effectively a large state machine; often referred to as the OpenGL context. A good example of this is: if a developer want to draw lines instead of triangles, they simply need to use an API command to change a state in OpenGL. On the next draw command OpenGL will now know that it needs to draw lines and not triangles (Vries 2015).

Early versions of OpenGL pre 2.0 used a fixed-function pipeline, this meant that the developer had no choice but to use a fixed shading model. Since OpenGL version 2.0, developers have had access to a programmable pipeline. The current programmable pipeline can be seen in in figure 4.1, the blue boxes are the programmable shader stages.

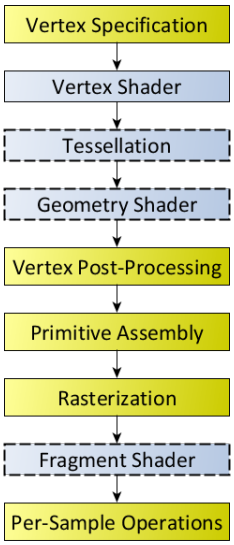


Figure 4.1: OpenGL Rendering Pipeline (Wiki 2019)

A C/C++ like language called GLSL is used to write the shader code. As of today, developers have vertex, fragment, geometry, and tessellation control/evaluation shaders to take advantage of (Shreiner et al. 2013). As of OpenGL 4.3 there is a compute shader that can be used to perform arbitrary computations independent of the rendering pipeline. Any number of these shaders together make up a shader program which will be compiled, linked, and executed on the GPU, often in parallel. The processor count on the graphics hardware determines how many can be executed at once (Wolff 2011).

Chapter 5

Implementation

5.1 Rendering Techniques

To implement the multiple rendering techniques, the design decision was made to use an abstract base class *Renderer* that each rendering technique implementation class extends. The *Renderer* class contains two key structures, *Light* and *RenderCommand*, these are both used in all rendering techniques. The *Light* structure holds the position, radius, and color of a light in the scene. The *RenderCommand* structure represents an object to be rendered in the scene, it holds a transformation matrix for where the object should be rendered, and information about the object mesh to be rendered. The *Renderer* abstract class contains a pure virtual function named *Render* that performs all of the work for each rendering technique. The *Render* function takes four parameters, the camera's projection and view matrix, a *std::vector* of render commands, and a *std::vector* of lights. These parameters contain all of the generic data required for rendering.

For each rendering technique, a list of *Light*s are needed in one or more of the shader programs. To supply the shader programs with the lights in the scene, an SSBO is used to send an arbitrary length array of *Light* objects.

5.1.1 Forward

The implementation of the forward renderer is straight forward. There's a single lighting pass that iterates over all of the objects in the scene, rasterizing them using a shader that accumulates light on the object for all the lights in the scene.

5.1.2 Deferred

The implementation of the deferred renderer is slightly more complex than the forward renderer and now consists of two passes, geometry and lighting.

The G-Buffer is represented as an FBO with the structure seen in figure 5.1. This structure is unoptimized and is only used in this implementation for simplicity because of time constraints. There are ways to make the G-Buffer more efficient which will be discussed later in this paper.

Prior to the start of the geometry pass, the G-Buffer FBO is bound to mark the starting point of where the G-Buffer is to be written to. For the geometry pass itself, all of the objects in the scene are rasterized using a shader program that writes the diffuse, normals, and world positions of the geometry to each of the relevant render targets of the G-Buffer FBO.

Now that the G-Buffer has been filled with necessary geometry information, it is time for the lighting pass. This implementation uses the *Stencil Masking* optimization previously mentioned in the *Information Review* chapter of this report. To do this, a feature of OpenGL is used called a *Stencil Buffer*. By rendering the light volumes and performing *Stencil Testing*, the fragments that aren't affected by light can be discarded, therefore the lighting is only calculated for pixels rasterized by light volumes. These lighting calculations are performed in the light accumulation render target of the G-Buffer. The result is that the light accumulation render target of the G-Buffer now contains a fully composite and lit scene texture. Now that the G-Buffer is filled, the G-Buffer FBO is unbound as it no longer needs to be written to.

Finally, a full-screen quad is rendered to the screen which displays the fully composite and lit scene texture from the light accumulation target of the G-Buffer.

Attachment	**Format**	**Values**
Color 0	GL_RGB16F	Diffuse r, g, b, Unused a
Color 1	GL_RGB16F	Normal r, g, b, Unused a
Color 2	GL_RGB16F	World Position r, g, b, Unused a
Color 3	GL_RGBA16F	Light Accumulation r, g, b, a
Depth Stencil	GL_DEPTH24_STENCIL8	Depth r, g, b, Stencil a

Figure 5.1: G-Buffer Structure (Deferred Renderer)

5.1.3 Tile-based Extensions

5.1.3.1 Light Culling

Both of the tile-based rendering techniques in this project use the same algorithm for light culling. The algorithm used in this project is based on the work presented by DICE at the Game Developers Conference in their presentation entitled *DirectX 11 Rendering in Battlefield 3* (Andersson 2011).

The screen is divided into 32 x 32 tiles. Each tile has an array of light indices and holds a maximum of 1024 lights. A depth texture of the scene is used to determine the minimum and maximum depths per tile, this is used during culling to avoid pixel overdraw. A frustum is calculated for each tile and then all of the lights are checked against each tile frustum to see if the light volume intersects with them. If there is an intersection, the index of the light is added to a per-tile array of visible light indices. This array of visible light indices is then used to reduce the number of lighting calculations although, how it is used differs between the tile-based rendering techniques in this project. The specifics of how the visible light indices array is used will be looked at in the descriptions of the implementations for the each rendering technique.

5.1.3.2 Forward+

The forward+ renderer implementation consists of three passes. A depth pre-pass, light culling pass, and a lighting pass.

The depth pre-pass uses an FBO similarily to the G-Buffer in the deferred renderer. Instead of rasterizing the scene to multiple render targets, it is only written to a depth texture to be used in the light culling pass.

The light culling pass is done using a compute shader that implements the algorithm aforementioned in the *Light Culling* section. In the compute shader, following the light culling algorithm code, the array of visible light indices is written to an SSBO to be used in the lighting pass.

The lighting pass iterates over all of the objects in the scene, rasterizing them using a shader that accumulates light on the object. Only the lights that are visible in the current fragments screen space tile contribute to the light accumulation, this information is retrieved from the SSBO of visible light indices supplied by the light culling pass.

5.1.3.3 Tiled Deferred

The implementation for the tiled-deferred renderer is nearly the same as the traditional deferred renderer aforementioned. The only change is to how the light accumulation render target of the G-Buffer is filled. In this rendering technique, the light accumulation render target is now filled using a compute shader that both culls the lights and performs the lighting in a single pass. The light culling algorithm in the compute shader is implemented as explained in the *Light Culling* section. Following the light culling algorithm code, the geometry information of the G-Buffer is used to accumulate the light at each pixel for the lights that are visible in the current pixels screen space tile. The result of these lighting calculations is a fully composite and lit scene texture, which is written to the light accumulation render target.

Changing the lighting pass in this way means that there is no longer a requirement for the depth stencil G-Buffer attachment used in the implementation of traditional deferred rendering, as no stencil testing is required. This results in a small change to the G-Buffer being that the depth stencil attachment is dropped for just a depth attachment. The revised G-Buffer structure for this implementation can be seen in figure 5.2.

Attachment	**Format**	**Values**
Color 0	GL_RGB16F	Diffuse r, g, b, Unused a
Color 1	GL_RGB16F	Normal r, g, b, Unused a
Color 2	GL_RGB16F	World Position r, g, b, Unused a
Color 3	GL_RGBA16F	Light Accumulation r, g, b, a
Depth	GL_DEPTH_COMPONENT32	Depth r, g, b, Unused a

Figure 5.2: G-Buffer Structure (Tiled-deferred Renderer)

5.1.4 Run-time Switching

Using the *Renderer* base class for the various rendering techniques allows for run-time switching of rendering techniques with ease. By allocating memory and instantiating pointers to all rendering technique, a further pointer to a generic *Renderer* object can be held and retargeted depending on the active technique. Through having a generic *Render* function that is used in each *Renderer* implementation, the generic pointer can call that function regardless of which technique the pointer is currently pointing to.

5.2 Capturing Performance Data

To get the time it took to process a frame, the *glfwGetTime* function of GLFW is used. This function returns a *double* which is the time in seconds since GLFW was initialized. By instrumenting the main loop at the start and end with variables that initializes as *glfwGetTime*, the end variable can be subtracted from the start variable to give a *double* which is the frame time in seconds.

To perform a data capture, upon pressing the *Capture* button on the user-interface, a *boolean* flag is set to capture the next one thousand frames. At the end of each frame, the frame time is stored in a *std::vector* that holds up to 1000 *double*s. Once the *std::vector* contains 1000 frame times, they are written to a comma-separated values file using *std::ofstream*. The *boolean* flag for capturing frame times is then set to false and the previous frame times are cleared from the *std::vector*.

Chapter 6

Testing

6.1 Strategy

To analyse the performance of the various rendering techniques, the testing strategy is to configure object and light count of the scene being rendered then collect average frame time over 1000 frames for each variation. This is to be repeated for each rendering technique using enough variations of light and object count that a range of interesting results is given. Although the object and light counts vary, for reliable and repeatable data, everything else such as camera position will remain the same. All of the dynamic lights in the scene are in view of the camera so that the worst case of performance is being tested.

The objects in the scene will be a *Stanford Bunny*, a popular model used for demonstrations in Computer Graphics. Using this model gives a realistically complex object that is reflective of what might be seen in typical real-time 3D graphics applications.

As identified in the *Information Review* section, the best measure of performance in real-time 3D graphics applications is frame time. This is the measure of performance that will be used for gathering the performance results in testing. When analyzing the test result graphs, it is important to remember that with frame time **lower is better**.

The results for each test were gathered on a machine with the specifications in figure 6.1.

CPU	Intel Core i5-8600k @ 3.60GHz
GPU	NVIDIA GeForce GTX 1060 6GB
RAM	16.0 GB @ 2.66MHz
Display Size	1920x1080
Operating System	Windows 10 Pro 64-bit

Figure 6.1: Test Machine Hardware

6.2 Results

6.2.1 Test 1

In this test, it is clear that forward rendering is the best performing. Considering that the complexity of a traditional forward renderer scales with the number of objects in a scene, having only a single object means that the computational expense stays low. Because the computational expense is low, the overhead of the other pipelines make them worse if rendering just a single object.

The deferred renderer appears to have a completely linear line and with a single object, performs poorly as the light count increases. This is due to the overhead of the light stencil optimization because light volumes are being rendered twice for the number of lights in the scene.

The overhead of the tiled-deferred renderer is significantly higher than the others, although as the light count increases, it gets worse at a lesser rate than the other rendering techniques. The forward+ renderer on the other hand, shows a strikingly similar line to that of the forward renderer when a single object is in the scene.

Although this is an unrealistic scenario in a real application, it demonstrates nicely the complexity of a forward renderer, and the overhead incurred from the other rendering techniques.

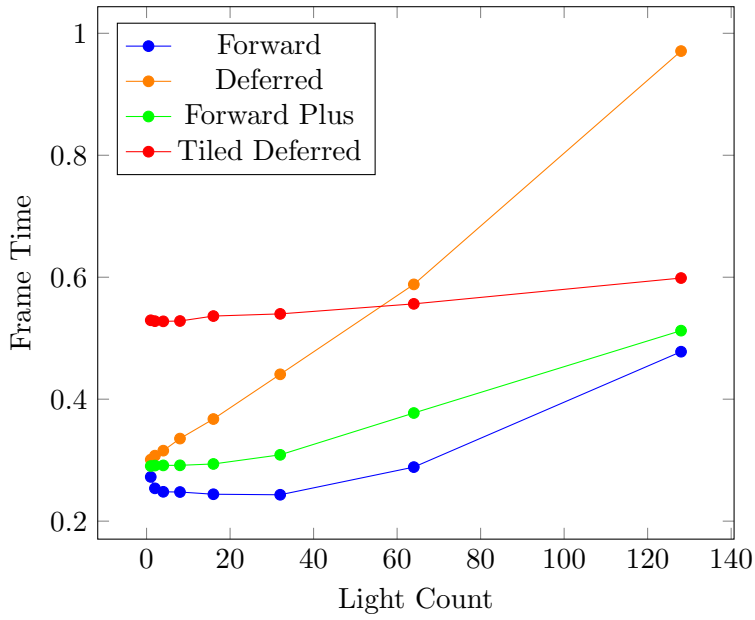


Figure 6.2: Test Scene 1 (1 Object)

6.2.2 Test 2

In the second test, with just four objects in the scene it, the fact that the algorithmic complexity of forward rendering is directly related to object count is showing as the inline steepens.

The line that represents the deferred renderer starts and ends in exactly the same place as in the first test, this is because the deferred renderers algorithmic complexity scales with screen resolution, not object count. The rendering of the light volumes for light stencil optimization will result in the same overhead regardless of object count, resulting in a consistently linear line.

Both tile-based rendering techniques are already beginning to show their worth as the light count rises. Although forward+ outperforms tiled-deferred until 128 lights, the steeper angle of the line representing forward+ suggests that at very high light counts, tiled deferred will perform better.

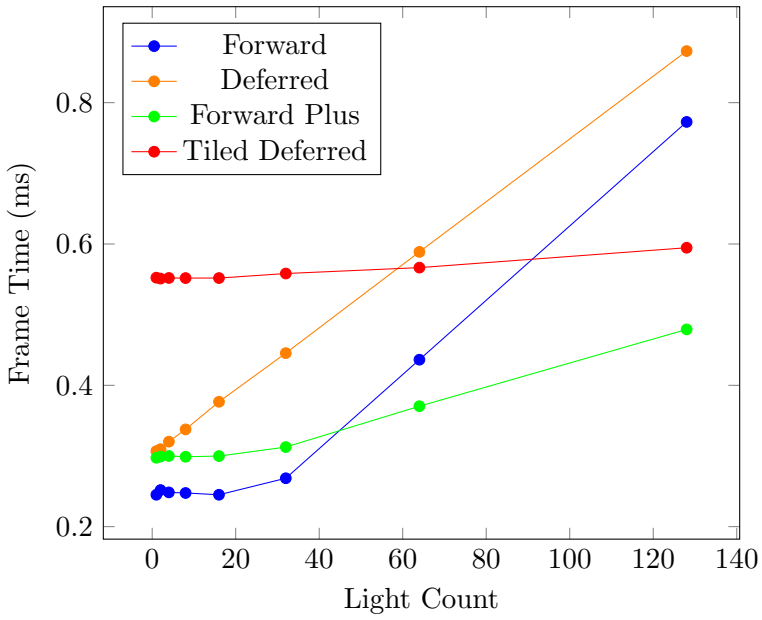


Figure 6.3: Test Scene 2 (4 Objects)

6.2.3 Test 3

In test three, at just 16 objects in the scene, traditional forward rendering is becoming the worst performing rendering technique at lower light counts with every additional object.

The initial overhead of both of the tile-based renderers have risen by a very similar amount. This is due to them using the same techniques for culling lights which incurs some overhead.

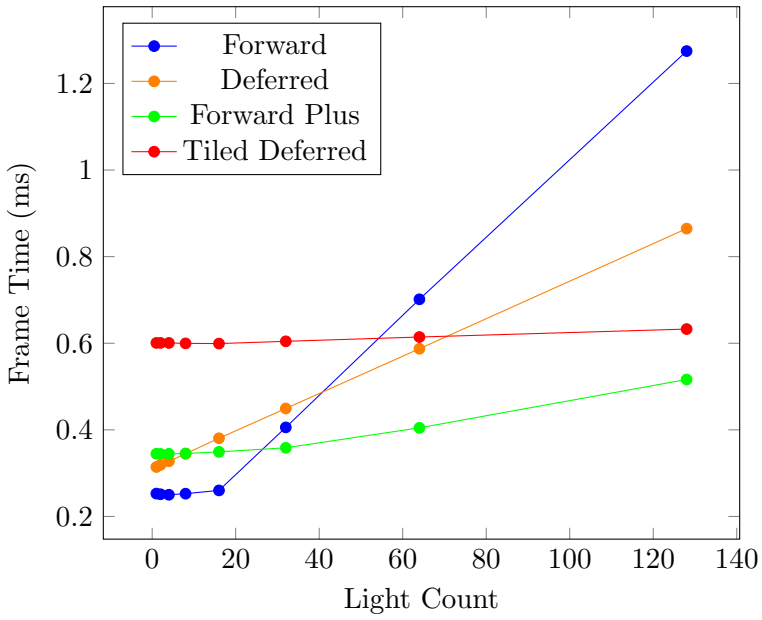


Figure 6.4: Test Scene 3 (16 Objects)

6.2.4 Test 4

In test four, with an object count of 64, forward+ outperforms tiled-deferred up until at least 128 dynamic lights. There is a small constant incline to the line that represents forward+, this shows that at a very high dynamic light count, tiled deferred will outperform forward+.

Unsurprisingly, the traditional forward renderer continues to become less viable for many dynamic lights as the object count grows.

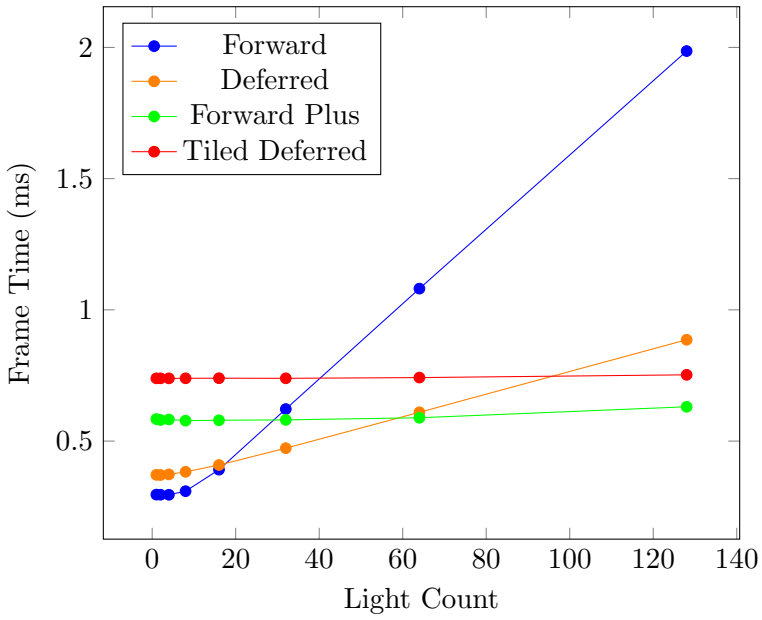


Figure 6.5: Test Scene 4 (64 Objects)

6.2.5 Test 5

In test five, with a relatively large object count of 256, tiled deferred now outperforms forward+. Looking at the angle of the lines and the results from the previous test, the assumption can be made that with this many objects or more, tiled deferred will always outperform forward+.

Surprisingly, the traditional deferred renderer is the best performing up until 128 lights. It is clear though by the angle of the line that as the light count rises, it will again be outperformed by the tile-based rendering techniques.

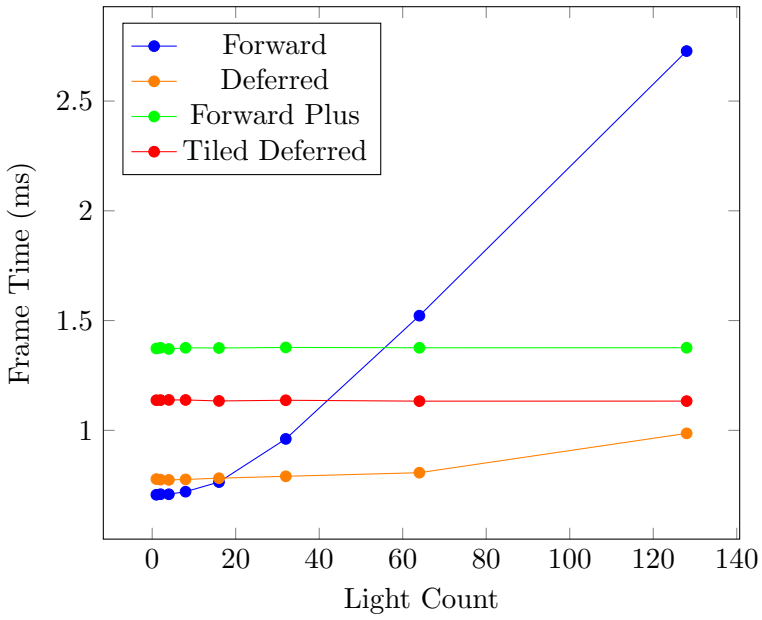


Figure 6.6: Test Scene 5 (256 Objects)

6.3 Summary

From the results seen in this chapter, it is clear that there is not a single rendering technique that outperforms the rest in all scenarios. This shows the importance of selecting the correct rendering technique for the target application to get optimal performance. The results from this chapter should be used in tandem with the overviews of each rendering technique from the *Research* chapter. Using these together will allow for an informed decision to be made as to which rendering technique best fits the target application based on both features and performance.

Chapter 7

Conclusion & Evaluation

The focus of this project was to provide a guide for developers that is used when a decision needs to be made as to which rendering techniques best suits their specific real-time 3D graphics application. It is my belief that this project has successfully provided the information necessary to facilitate this, meeting the initial aims and objectives of the project. As intended, the project deliverable allows for the collection of performance data for each rendering technique and has proven invaluable to the success of the project. By using the information regarding features and performance of the various rendering techniques, it allows for a developer to identify the rendering technique that best fits their target application. The importance of choosing the correct technique has been backed up in the *Testing* chapter of this report by the test results collected using the project deliverable. The test results demonstrate how significant the performance difference of each rendering technique can be under the same conditions.

Despite the success of the project, time constraints had an adverse affect on the overall substance of the project because the investigation could have been much deeper. There's plenty more research on various other rendering techniques and work in the area that could have been covered in this project. What is covered in this project is all that the time limitations allowed for.

7.1 Future Work

7.1.0.1 Recovering World Position from Depth

In a deferred rendering pipeline, it is important to keep the geometry buffer as small as possible so that it requires less bandwidth and memory space to store and retrieve the data.

For the implementation in the project deliverable, the world positions are stored in the G-Buffer, however there's a simpler and cheaper way (Wenzel 2006). It is possible to avoid storing world space positions into the G-Buffer by reconstructing the world position using the depth texture (Pettineo 2009). Adding this optimization would better reflect modern rendering engines that tightly pack data into their G-Buffer.

7.1.0.2 Compact Normal Storage

Besides avoiding storing the world positions into the geometry buffer as aforementioned in the previous section, it is possible to reduce size of the geometry buffer in the deferred rendering pipeline by encoding the geometry normal data (Pranckevičius 2009). One way to can achieve this is by storing only the normals X and Y, then use those to reconstruct the Z thus reducing the size of the geometry buffer, although a small amount of precision is lost, it is preferred in favour of lowering the memory that G-Buffer uses (Valient 2009).

7.1.0.3 Optimized Light Culling

The current tile-based light culling algorithm used in the project deliverable is fairly trivial. This could be improved to refine the culling results and reduce the shading cost as much as possible. One prevalent issue with the current algorithm is that the math used is not accurate enough when testing large radius lights against small frustums. This issue presents the problem that the lights being tested are not culled away properly. The problem can be overcome by using axis-aligned bounding boxes instead of frustums (János 2018).

7.1.0.4 Transparency Support

Transparency is a feature that is seen in most real-time 3D graphics applications, regardless of rendering technique. Adding support for transparency in a deferred rendering pipeline can be complicated and time constraints of this project did not allow for it. It would be beneficial to add support for transparent objects to the project deliverable because it would allow for gathering of results within a scene that contains both opaque and transparent objects, better reflecting what might be scene in a typical 3D scene.

7.1.0.5 Additional Rendering Techniques

Recently, there has been further developments into tile-based rendering techniques with the introduction of cluster-based rendering techniques. Cluster-based techniques builds on tile-

based techniques by further subdividing the volumes associated with the tiles. The subdivision of the view frustum is more accurate than the tile-based implementation seen in the project deliverable, and therefore reduces the number of light calculations even further. Cluster-based rendering technique are used in large commercial projects such as Avalanche Engine, the game engine that powers the video game titled Just Cause (Persson 2013). It would be beneficial to support additional rendering techniques such as cluster-based to add more depth to this project, providing more options for developers.

7.1.0.6 Further Platform Support

By using OpenGL as the graphics API of choice, there is a lot of platform freedom due to it being a platform independent API. However, each platform has slightly different requirements. This means that support must be explicitly written into the project deliverable. It would be highly beneficial to have the option to test on other platforms to see how the test case results vary depending on the platform and hardware it is deployed on. The results could possibly be of such variance from one platform to another that certain rendering techniques may not be viable. For example, mobile devices typically have very limited resources when comparing to a modern desktop computer. The limited hardware of mobile devices may not have to GPU processing power may not be able to support the high memory and bandwidth requirements of a deferred renderer. However, it is only speculation that this would be the case without actual tests which is why it is important to perform them.

7.2 Personal Reflection

Looking back on the time that I have dedicated to this project, I can confidently say that I have enjoyed learning about such an interesting topic. I chose it because of my interest in video game programming and game engines. This proved to be a wise decision because it meant that I maintained interest throughout the project.

The part of the project that I enjoyed the most was the implementation of the rendering techniques in the deliverable. It proved extremely interesting to dive deep into how each of the rendering techniques worked and has me wanting to learn even more. Implementing each of the rendering techniques was a lot of fun and I intend to continue this work after this project by researching and implementing further rendering techniques that I was not able to cover within the time constraints of this project

During the implementation of the rendering techniques, I enjoyed learning about areas of

modern OpenGL that I had never encountered before. My interest in video game programming meant I had prior experience with OpenGL, however certain parts of OpenGL that I had never used such as compute shaders proved very intriguing and left me wanting to learn more. I was aware of the issue with sending large arrays of data to a shader program, however I had no exposure to SSBOs which I got to learn about and realized they are a very useful feature of the modern API. This project has added much more depth to my understanding of OpenGL which will help me with other graphics APIs as well because they typically support very similar sets of functionality. It has improved my general understanding of Computer Graphics and what modern graphics APIs are capable of. It has given me the desire to undertake more personal learning in this field and I intend to investigate innovative graphics APIs such as Vulkan. It has influenced the direction that I would be willing to take in my career also, I would be open to roles relating to graphics programming that I wasn't necessarily interested in before.

One of my biggest worries coming into the project was that I have always had problems with report writing. I attended a writing workshops and that helped quite a lot. The practical experience of writing a large report however, has been the most beneficial part and I believe that I have significantly improved in this area from the beginning of the University year.

Chapter 8

Bibliography

Akenine-Möller, Tomas et al. (2018). *Real-Time Rendering*. 4th. A. K. Peters, Ltd., p. 1.

Andersson, Johan (2009). *Parallel Graphics in Frostbite - Current and Future*. SIGGRAPH. URL: http://s09.idav.ucdavis.edu/talks/04-JAndersson-ParallelFrostbite-Siggraph09.pdf.

— (2011). "Direct X Rendering in Battlefield 3". In: URL: http://www.dice.se/wp-content/uploads/2014/12/GDC11_DX11inBF3_Public.pptx.

ARM (2011). *Use frame time instead of FPS for measurements*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0555a/BEIGDEGC.html.

Azarian, Irma (2013). *A Review of Software Version Control: Systems, Benefits, and Why it Matters*. URL: https://www.seguetech.com/a-review-of-software-version-control-systems-benefits-and-why-it-matters/.

Chakravorty, Dibya (2019). *8 Most Common 3D File Formats in 2019*. URL: https://all3dp.com/3d-file-format-3d-files-3d-printer-3d-cad-vrml-stl-obj/.

Cornut, Omar (2019). *Software using dear imgui*. URL: https://github.com/ocornut/imgui/wiki/Software-using-dear-imgui.

Creation, G-Truc (2018). *GLM*. URL: https://glm.g-truc.net/0.9.9/index.html.

Harada, Takahiro, Jay McKee, and Jason C. Yang (2012). *Forward+: Bringing Deferred Lighting to the Next Level*, pp. 1–4. URL: https://takahiroharada.files.wordpress.com/2015/04/forward_plus.pdf.

Hargreaves, Shawn and Mark Harris (2004). *Deferred Shading*. URL: http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf.

János, Turánszki (2018). *Optimizing tile-based light culling*. URL: https://wickedengine.net/2018/01/10/optimizing-tile-based-light-culling/.

Jorge, Julien (2018). *An overview of build systems (mostly for C++ projects)*. URL: https://medium.com/@julienjorge/an-overview-of-build-systems-mostly-for-c-projects-ac9931494444.

Koonce, Rusty (2008). *Deferred Shading in Tabula Rasa*. Ed. by Hubert Nguyen. Addison-Wesley, pp. 429–457.

Lauritzen, Andrew (2010). *Deferred Rendering for Current and Future Rendering Pipelines*. SIGGRAPH. URL: https://software.intel.com/sites/default/files/m/d/4/1/d/8/lauritzen_deferred_shading_siggraph_2010.pdf.

Oosten, Jeremiah van (2015). *Forward vs Deferred vs Forward+ Rendering with DirectX 11*. URL: https://www.3dgep.com/forward-plus/.

Perkins, Jason (2018). *Premake*. URL: https://premake.github.io/.

Persson, Emil (2013). *Practical Clustered Shading*. SIGGRAPH. URL: http://www.humus.name/Articles/PracticalClusteredShading.pdf.

Pettineo, Matt (2009). *Scintillating Snippets: Reconstructing Position From Depth*. URL: https://mynameismjp.wordpress.com/2009/03/10/reconstructing-position-from-depth/.

Pranckevičius, Aras (2009). *Compact Normal Storage for small G-Buffers*. URL: http://aras-p.info/texts/CompactNormalStorage.html.

Saito, Takafumi and Tokiichiro Takahashi (1990). *Comprehensible Rendering of 3-D Shapes*, pp. 197–206.

Shishkovtsov, Oles (2005). *Deferred Shading in S.T.A.L.K.E.R.* Ed. by Matt Pharr. Addison-Wesley, pp. 143–166.

Shreiner, Dave et al. (2013). *OpenGL Programming Guide*. 8th. The Khronos OpenGL ARB Working Group, pp. 2–3.

Takahashi, Dean (2013). *How Pixar made Monsters University, its latest technological marvel*. URL: https://venturebeat.com/2013/04/24/the-making-of-pixars-latest-technological-marvel-monsters-university/.

Valient, Michal (2009). *The Rendering Technology of Killzone 2*. Game Developers Conference. URL: https://www.guerrilla-games.com/read/the-rendering-technology-of-killzone-2.

Vries, Joey de (2015). *Learn OpenGL*, pp. 17–22.

Wenzel, Carsten (2006). *Real-time Atmospheric Effects in Games*. SIGGRAPH. URL: https://developer.amd.com/wordpress/media/2012/10/Wenzel-Real-time_Atmospheric_Effects_in_Games.pdf.

White, John and Colin Barré-Brisebois (2011). *Five Rendering ideas from Battlefield 3 and Need For Speed: The Run*. SIGGRAPH. URL: https://www.ea.com/frostbite/news/more-performance-five-rendering-ideas-from-battlefield-3-and-need-for-speed-the-run.

Wiki, OpenGL (2019). *Rendering Pipeline Overview*. URL: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Rendering_Pipeline_Overview&oldid=14511.

Wolff, David (2011). *Getting Started with GLSL 4.0*. PACKT, pp. 5–47.

Chapter 9

Glossary

FBO OpenGL Framebuffer Object. 15, 16

FXAA Fast approximate anti-aliasing. 5

G-Buffer Geometry Buffer. 4, 5, 6, 15, 16, 17, 27

GLSL OpenGL Shading Language. 10, 13

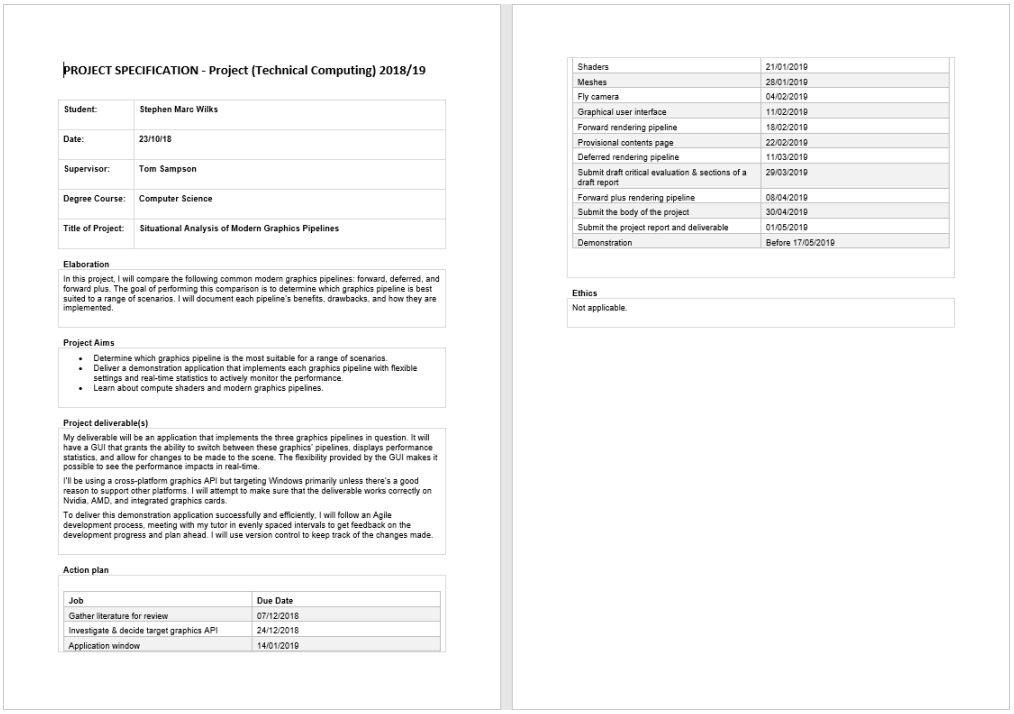
GPU Graphical Processing Unit. 13, 28

MSAA Multisample anti-aliasing. 6

SSBO OpenGL Shader Storage Buffer Object. 14, 16, 29

Appendix A

Project Specification Document



PROJECT SPECIFICATION - Project (Technical Computing) 2018/19

Student:	**Stephen Marc Wilks**
Date:	**23/10/18**
Supervisor:	**Tom Sampson**
Degree Course:	**Computer Science**
Title of Project:	**Situational Analysis of Modern Graphics Pipelines**

Elaboration

In this project, I will compare the following common modern graphics pipelines: forward, deferred, and forward plus. The goal of performing this comparison is to determine which graphics pipeline is best suited to a range of scenarios. I will document each pipeline's benefits, drawbacks, and how they are implemented.

Project Aims

- Determine which graphics pipeline is the most suitable for a range of scenarios.
- Deliver a demonstration application that implements each graphics pipeline with flexible settings and real-time statistics to actively monitor the performance.
- Learn about compute shaders and modern graphics pipelines.

Project deliverable(s)

My deliverable will be an application that implements the three graphics pipelines in question. It will have a GUI that grants the ability to switch between these graphics' pipelines, displays performance statistics, and allow for changes to be made to the scene. The flexibility provided by the GUI makes it possible to see the performance impacts in real-time.

I'll be using a cross-platform graphics API but targeting Windows primarily unless there's a good reason to support other platforms. I will attempt to make sure that the deliverable works correctly on Nvidia, AMD, and integrated graphics cards.

To deliver this demonstration application successfully and efficiently, I will follow an Agile development process, meeting with my tutor in evenly spaced intervals to get feedback on the development progress and plan ahead. I will use version control to keep track of the changes made.

Action plan

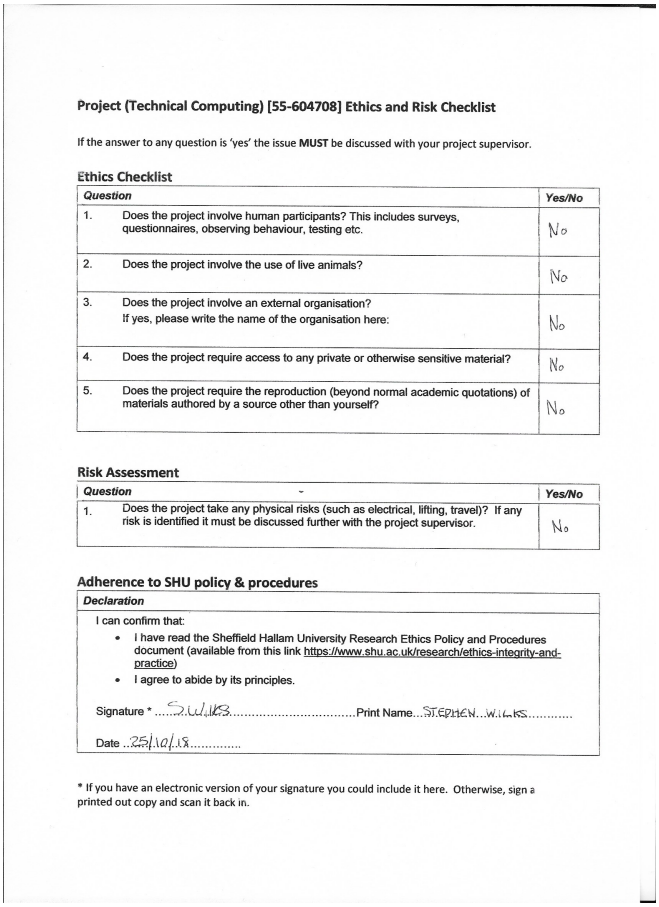
Job	Due Date
Gather literature for review	07/12/2018
Investigate & decide target graphics API	24/12/2018
Application window	14/01/2019
Shaders	21/01/2019
Meshes	28/01/2019
Fly camera	04/02/2019
Graphical user interface	11/02/2019
Forward rendering pipeline	18/02/2019
Provisional contents page	22/02/2019
Deferred rendering pipeline	11/03/2019
Submit draft critical evaluation & sections of a draft report	29/03/2019
Forward plus rendering pipeline	08/04/2019
Submit the body of the project	30/04/2019
Submit the project report and deliverable	01/05/2019
Demonstration	Before 17/05/2019

Ethics

Not applicable.

Appendix B

The Ethics Form



Project (Technical Computing) [55-604708] Ethics and Risk Checklist

If the answer to any question is 'yes' the issue **MUST** be discussed with your project supervisor.

Ethics Checklist

Question		*Yes/No*
1.	Does the project involve human participants? This includes surveys, questionnaires, observing behaviour, testing etc.	No
2.	Does the project involve the use of live animals?	No
3.	Does the project involve an external organisation? If yes, please write the name of the organisation here:	No
4.	Does the project require access to any private or otherwise sensitive material?	No
5.	Does the project require the reproduction (beyond normal academic quotations) of materials authored by a source other than yourself?	No

Risk Assessment

Question		*Yes/No*
1.	Does the project take any physical risks (such as electrical, lifting, travel)? If any risk is identified it must be discussed further with the project supervisor.	No

Adherence to SHU policy & procedures

Declaration

I can confirm that:

- I have read the Sheffield Hallam University Research Ethics Policy and Procedures document (available from this link https://www.shu.ac.uk/research/ethics-integrity-and-practice)
- I agree to abide by its principles.

Signature * ..... Print Name ... STEPHEN WILKS .....

Date ... 25/10/18 .....

* If you have an electronic version of your signature you could include it here. Otherwise, sign a printed out copy and scan it back in.

Appendix C

Using the Deliverable

The table seen in figure C.1 documents the entire directory structure of the project deliverable to assist with navigation.

Directory	**Description**
src	Contains the implementation source code.
include	Contains the implementation header code.
media	Contains the models and shaders.
dependencies	Contains the third party dependencies.
dist	Contains a runnable version of the deliverable.

Figure C.1: Deliverable Directory Structure

To run the project deliverable, navigate to the *dist* directory in the directory structure of the project deliverable. Identify *Deliverable.exe* and make sure that a *media* folder is present in the same directory as the executable. If there is no *media* directory present in the same directory as the executable, it will not function correctly, so take the *media* directory from the root directory of the directory structure and place it where the executable resides in *dist*. With the project deliverable executable and *media* directory in place, double-click the executable and the project deliverable will run successfully so long as the machine has the ability to use compute shaders.

With the project deliverable running, all of the available controls can be seen on the user interface that is located to the left hand side of the window. The user interface can be used to manipulate the scene and change the back-end rendering technique, as well as watch real-time render statistics and how performance is affected as the scene is changed. Using the

Capture button will perform a capture of the next 1000 frame times and write them to a comma-separated value file *capture.csv* in the same directory as the executable.