LM051 Final Year Project

Interim Submission

The Design and Implementation of Modern 3D Rendering Engines

Author: Rónán Quill

Student ID: 17040434

Supervisor: Dr. Patrick Healy

Date: 20/01/2021

Table of Contents

Table of Figures

Table of Equations

Table of Code Samples

1. Introduction

My project, The Design and Implementation of Modern 3D Rendering Engines, explores the various techniques that go into developing a modern 3D rendering engine that can render realistic images at an interactive framerate, i.e. 60 FPS.

Over the last number of years, there have been significant advancements in graphics APIs, hardware and techniques. Old shading systems such as Blinn-Phong have been superseded by vastly more complex and visually appealing physically-based rendering systems. Simplistic forward rendering architectures have been replaced by combinations of deferred shading and advanced versions of forward rendering. Furthermore, ray tracing has started to make its way into the world of real time rendering. I intend to explore how all of these are coming together to allow considerable improvements in realism in 3D environments.

My project is being developed using C++ as my programming language and I have chosen Vulkan as my graphics API, along with Vulkan I am using GLSL as my shader language. Vulkan is the successor to OpenGL and provides a modern way to target multiple platforms with one graphics API. However, I also intend to support the DirectX12 graphics API in the future. By using C++ and Vulkan I will be able to run my rendering engine on several platforms such as Windows, Linux, Mac OS, Android devices and the Nintendo Switch. I will explore the Vulkan API in more detail in a following section of this report.

For this interim submission I set out 3 major milestones which I wanted to achieve:

1. Render 3D Objects using the Vulkan API;
2. Enhance the shading model to utilise Physically Based Rendering;
3. Implement a tiled deferred shading rendering engine architecture;

Throughout this report, I will detail the research and implementation steps that I undertook to achieve each of these goals along with some rendered images created by my implementation of these concepts.

A big challenge of this submission was picking what not to include, entire master thesis' have been devoted to topics covered in subsections of this report so at times it was difficult to keep things concise but still clear. I have chosen not to cover some of the core concepts of computer graphics covered in the Computer Graphics 1 or Computer Graphics 2 modules as the report would be far too long. However, I feel like I have included enough detail to give a good understanding of the theory and implementation of the covered techniques.

2. Related Works / Inspirations

There exists countless examples of rendering engines currently released today, some open source like Unreal Engine 4 or Godot, most are closed source like those from major AAA game studios like Rockstar's RAGE and EAs Frostbite. What is very interesting about these rendering engines is that they all tackle rendering in ways and using different techniques. Real time rendering does not have a "correct" agreed-upon set of steps you need to follow to render a scene in a realistic way, this keeps the field interesting by making experimentation and research a worthwhile experience.

A major inspiration for this project is the Halcyon rendering engine created by EAs SEED research division. Most rendering engines value performance over flexibility, this is very understandable when you need to extract as much from your hardware as possible to render a complex scene at a high FPS. However, the Halcyon engine is designed around facilitating rapid prototyping which allows the team at SEED to continually explore and evaluate rendering techniques. And as they put it "scout ahead" for new techniques (Wihlidal, 2019).

Halcyon was one of the first engines to present the idea of hybrid rendering systems, a system which combines traditional rasterization for rendering 3D meshes with ray tracing to provide global illumination techniques such as soft shadows, reflection & ambient occlusion. Many more engines have started to follow in their footsteps and hybrid rendering systems are beginning to be adopted by all major studios.

Now is a great time to explore the modern topics of rendering engines as the advent of real time ray tracing techniques in these hybrid rendering systems have allowed major leaps in image quality.

Figure 2.1 Images of BWM Car, on the left, is a real-life photo, on the right is a 3D model rendered with Unity (Barré-Brisebois & @ZigguratVertigo, 2019).

The above picture is a prime example of the image quality that is currently possible, on the left is a real-world picture of a BWM car, on the right is a 3D model of the same car rendered in the Unity game engine in real-time.

3. Vulkan

Vulkan is an extremely complex graphics. There is a lot more work involved in utilising Vulkan than there was in using OpenGL, most online resources will recommend you avoid Vulkan unless you are after high-performance graphic. I will now cover what Vulkan is and the steps that are needed to render a simple triangle using Vulkan.

3.1. Introduction

Vulkan is one of the 2 main new generation graphics APIs, the other being DirectX12. Vulkan is developed by the Khronos group and is presented as a highly efficient, cross-platform API for modern GPUs that can be used for both graphics rendering and general GPU computing. For this project, I am focusing solely on 3D graphics rendering but this does overlap with GPU computing through the use of compute shaders in certain techniques such as tiled shading.

Vulkan is supported on several platforms such as Windows, Linux, Android, and the Nintendo Switch. By using a library called MoltenVK, you can support MacOS & IOS as this provides a Vulkan wrapper around Apples Metal graphics API (The Khronos Group, 2021).

Vulkan was first released in 2016 and has started to gain a lot of traction lately as game developers learn how to utilise Vulkan and what it has to offer. Vulkan is the successor API to OpenGL but it follows a completely different design philosophy than OpenGL. Vulkan interacts with the graphics hardware at a much lower level than OpenGL, requiring you to be extremely explicit about every action you want the GPU to undertake. Many responsibilities are shifted from the GPU driver writers to rendering engineers such as CPU-GPU synchronisation and graphics pipeline creation. This allows problems of the past to be avoided where poor driver implementation caused rendering engines to perform poorly on certain hardware.

In the past only one CPU thread was responsible for submitting work to the GPU, this quickly started to become a major bottleneck as scenes became more complex. With Vulkan, there is a heavy focus on creating work for the GPU from multiple threads by creating command buffers which record sets of commands for the GPU to execute. Typically, all the commands needed to render an entire scene are recorded to these command buffers and then submitted to a command queue that passes the commands onto the GPU. Unlike in OpenGL, you must ensure that you do not attempt to write to any resource that is currently being used by the GPU. In the past, the GPU driver would work out a solution for you to prevent the program from crashing, now you are responsible to ensure that the resource is free to write to (Foley, 2015).

Due to how new the Vulkan API is, it was quite a struggle to effectively learn it. A lot of Vulkan tutorials and resource contain errors and race conditions that the authors have missed. Most popular resources for learning Vulkan focus on simple static scenes which avoids a lot of the complexities that are involved with managing the dynamic scenes which I aimed to support. And most online forums for Vulkan are quite barren as it seems relatively few have undertaken the task to learn it, even large established studios like Valve have published incorrect information on using Vulkan which directly contradicts GPU manufacturer guidelines.

This has been an enjoyable challenge though and allowed me to come up with some interesting solutions to tackle memory management and pipeline creations in Vulkan, which I intend to showcase in my final project submission.

3.2. Rendering Using Vulkan

Step 1: Create the Vulkan instance

The Vulkan Instance (VkInstance) is the connection between your application and the Vulkan library (Vulkan Tutorial, n.d.). The Vulkan API uses the VkInstance object to store all per-application state. It is the VkInstance which starts the process of loading and initialising the GPUs driver software. This instance object must be created before performing any other Vulkan operation (LunarG, Inc, 2019). It also has additional features like allowing you to specify debug call-backs that can be triggered whenever Vulkan performs an action or encounters an error.

Step 2: Acquire a physical device handle

A VkPhyiscalDevice in Vulkan represents the actual GPU that you have selected in the system. Typically, in a system there will only be one dedicated GPU however in laptops there can be 2 GPUs, an integrated one in the CPU for low power usage and a standalone dedicated GPU. You need to be sure to select the correct GPU in these devices to ensure that your rendering engine achieves its max performance.

Step 3: Create up the logical device

The logical device (VkDevice) represents the actual GPU driver and it is how we communicate with the GPU. Most commands in Vulkan require the VkDevice to be provided as a parameter (Vulkan Guide, 2020).

Step 4: Create the window surface and swap chain

To actualyl show the results of GPU render work we need a surface to present the rendered image to. A swap chain is a list of images that are accessible to the operating system to display to the screen, you can create a swap chain with a specified number of images, in most cases 3 images are used to utilise triple buffering (Vulkan Guide, 2020). To create a swap chain, you need to supply it with a windows surface to present this rendered image on, I am utilising GLFW which is a cross-platform windowing framework to provide this functionality.

Step 5: Acquire the swap chain images and create frame buffers

To render to these images, we need to create a render target to render to, in Vulkan these are known as frame buffers (VkFrameBuffer). A frame buffer holds an array of images which the fragment shaders can write to. To render to the swap chain, we request handles to the images managed by the swap chain and create frame buffers for these images.

Step 6: Create a render pass

In Vulkan, all rendering happens inside of a rendering pass (VkRenderPass). It is not possible to do rendering commands outside of a render pass; however, it is possible to do compute commands without using a render pass. A render pass renders into a specified framebuffer. A render pass also

contains sub passes which specify different stages for rendering. These are very useful in mobile GPUs as they allow first-class support of tiled rendering which is heavily used by mobile GPUs.

Step 7: Create a graphics pipeline

Vulkan requires you to specify in advance all states that you require to have set for rendering.

A VkPipeline allows you to prespecify the following:

- Shaders, i.e. Vertex Shader, Fragment Shader
- Vertex shader input description
- The input assembly i.e. describe what kind of geometry will be created from these vertices such as triangles or lines.
- The viewport to render to
- Rasterization information e.g. face culling options
- Multisampling options
- Blend states for framebuffers
- Depth buffer options

Step 8: Create a GPU command queue

To allow the GPU to accept rendering commands we must create a queue (VkQueue), which are execution ports for the GPU. Different kinds of queue exist on most GPUs, in this project I focus on the graphics/present queue which is responsible for accepting commands relating to rendering an image. However, you could also create a compute queue which accepts commands for performing general-purpose GPU computation (Vulkan Guide, 2020).

Step 9: Create a command pool and command buffer

To submit rendering commands to these queues we must first record these commands into command buffers (VkCommandBuffer). However, to obtain a command buffer we must first create a command pool which acts as an allocator for command buffers (Vulkan Guide, 2020). In Vulkan to support multithreaded rendering, you must create at least one command pool per thread as they are not thread-safe. You also need a separate command buffer per thread as only one thread can record to a command buffer at a time.

Step 10: Fill the command buffer

To fill these command buffers we need to:

1. Put the command buffer into a recording state (vkBeginCommandBuffer);
2. Begin a render pass (vkCmdBeginRenderPass);
3. Specify which rendering pipeline to use (vkCmdBindPipeline);
4. Bind resources like vertex buffers, index buffers, textures and uniform buffers to the graphics pipeline to provide information to the shaders;
5. Then record a draw command (vkCmdDrawIndexed).More than one draw command can be recorded in a single command buffer and it is indeed encouraged to record as many commands as you can before submitting work to the GPU.

6. After all your commands have recorded you must end the render pass (vkCmdEndRenderPass)
7. Finally end the recording state of the command buffer (vkEndCommandBuffer)

Step 11: Submit to the GPU

Before submitting these commands to the GPU queue we need to ensure synchronization between the CPU and the GPU and that we do not attempt to render to an image that is currently in use by the GPU (Vulkan Guide, 2020). Vulkan uses concepts such as fences and semaphores which facilitate this synchronization. When these synchronization structures have signalled that the requested swap chain image is ready we submit our commands to the presentation queue with vkQueueSubmit followed providing this queue to the swap chain along with the index of image which you wish to render to with vkQueuePresentKHR.

Following all of this, you will be presented with a rendered image representing everything that was recorded in the command buffers.

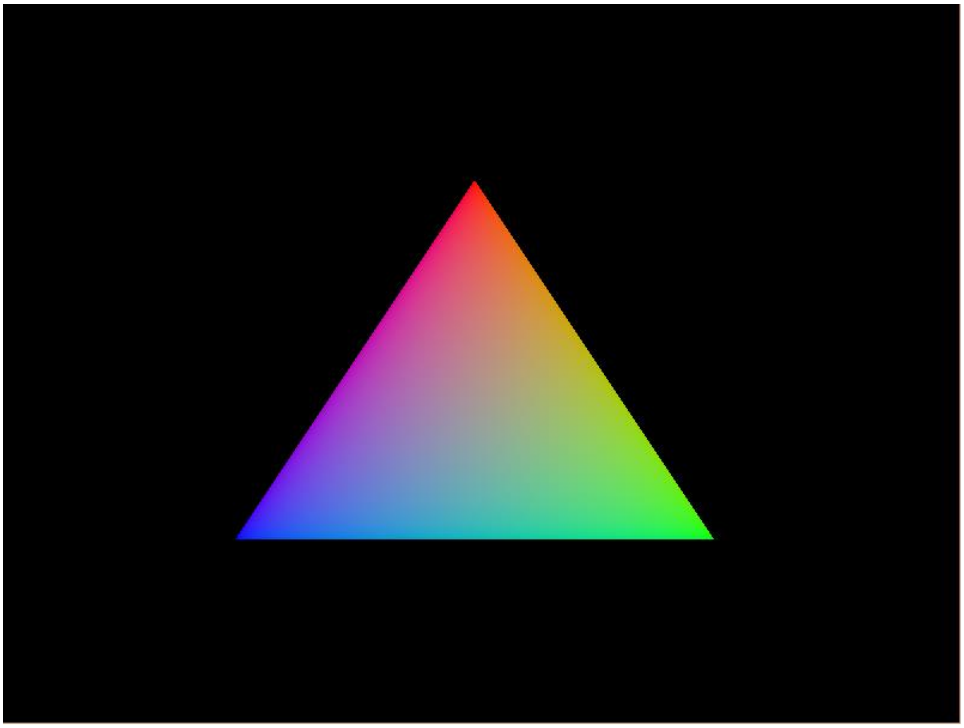

Figure 3.1 Simple triangle rendered using Vulkan (in-engine)

As you can see it is quite an involved task to render an image in Vulkan and what is describe above only renders simple 2D images. To support 3D scenes, we need to extend this system to support depth buffers and to support complex scene we need to implement a system to manage our meshes and materials and to automate their command recording. I will not cover these concepts in this report however they have been implemented as part of the coding section of my project.

4. Physically Based Rendering

Following on from my Vulkan implementation I sought to implement a modern material shading system. As physically based rendering is the shading system present on all modern rendering engines and it is almost unheard of in today's age for a rendering engine to not support it, I chose to examine this topic and add an implementation to my project.

4.1. Introduction

Physically based rendering or PBR is an approach to rendering that renders materials using the physical properties of lights and materials based on real-world values rather than artistic parameters.

In real-time applications such as games, it is not possible to achieve exact physical representation of materials like what is done in the movie industry as it is too computationally expensive. Instead, we utilise approximations that still produce realistic and visually pleasing results.

For a PBR shading model to be considered physically based, it must satisfy the following 3 conditions: (Learn OpenGL, 2016)

1) Be based on the microfacet surface model;
2) Be energy conserving, outgoing light energy should never exceed the incoming light energy (excluding emissive surfaces);
3) Use a physically based BRDF;

Physically based rendering requires a completely different material and lighting system than the one traditionally used in Blinn-Phong shading. In Blinn-Phong shading a material would be defined by the following parameters:

- **Ambient colour:** RGB value describing its ambient lighting colour
- **Diffuse colour:** RGB value describing its diffuse lighting colour
- **Specular colour:** RGB value describing its specular reflection colour
- **Shininess exponent:** floating point exponent describing the shininess of the material

None of these concepts have a physical basis in reality and are instead artistic parameters that are selected through quite a subjective process to create a desired material. This process also required you to create a different material for different lighting conditions such as a day/night cycle to get a consistent look.

Lights too would also be defined by artistic parameters rather than physical units which further complicated the asset authoring pipeline and sometimes lighting details would be baked into material textures.

One of the core principles of PBR is the decoupling of material and lighting information, this ensures visual consistency between all objects in a scene regardless of lighting conditions (Lagarde & Rousiers, 2014).

Two major workflows currently exist for physically based rendering, the metallic-roughness workflow and the specular-glossiness. I chose to use the metallic-roughness workflow as most sources, both academic and practical, use metallic-roughness.

The metallic-roughness workflow exposes the following parameters for a material:

- **Albedo:** RGB value describing the colour of the material with no lighting information or if the surface is metallic it contains the base reflectivity of the material.
- **Metalness:** Floating point value ranging from 0.0 to 1.0. 0.0 represents a non-metallic surface and 1.0 represents a fully metallic surface. Values in between these ranges are used to represent a metallic surface with surface imperfections such as a dust covering or rust.
- **Roughness:** Floating point value from 0.0 to 1.0 which describes the microgeometry's roughness with 0.0 representing a perfectly smooth surface that will perfectly reflect incoming light, while values ranging up to 1.0 will cause the incoming light to scatter on contact.

These parameters are usually provided through texture maps with a mesh mapped to the textures through its UV vertex coordinates. Additionally, a normal map is also generally included which describes per-pixel surface normals for a material.

4.2. The Physics of Light

As physically based rendering models an approximation of lights interaction with a material it is important to define and understand some of the principles of the physics of light. Light or visible light is electromagnetic radiation within the portion of the electromagnetic spectrum that can be perceived by the human eye (Wikipedia, n.d.). Light waves are emitted when the electrical charges in an object oscillate. Part of the energy that caused the oscillations—heat, electrical energy, chemical energy—is converted to light energy, that is radiated away from the object. In rendering, such objects are treated as light sources (Akenine-Möller, et al., 2018).

In physically based rendering we model lights interaction with surface irregularities that are in the size range of 1 – 100 wavelengths. We focus on this range as surface irregularities smaller than 1 wavelength have no effect on light and surface irregularities larger than 100 wavelengths effectively tilt the surface rather than affecting its local flatness. To translate this into rendering terms, surface irregularities much larger than a wavelength change the local orientation of the surface (Akenine-Möller, et al., 2018).

When light strikes a surface point it is reflected in a specific direction by that surface point however in rendering each pixel represents several surface points that reflect light in various directions. To calculate the final appearance of a pixel we need to aggregate the results of all these different reflection directions. Rather than modelling this explicitly, we treat it statistically and view the surface as having a random distribution of surface normals. As a result, we model the surface as reflecting and refracting light in a continuous spread of directions. The width of this spread, and thus the blurriness of reflected and refracted detail depends on the surface roughness (Akenine-Möller, et al., 2018).

When light is refracted it continues to interact with the inner volume of an object, for metallic objects most incident light is reflected and the rest is absorbed. However non-metallic objects exhibit a wide variety of scattering and absorption behaviours (Akenine-Möller, et al., 2018). Materials with low scattering and absorption are transparent, allowing all refracted light to travel through the object. While in opaque objects, light undergoes several scattering and absorption events until some of it is re-emitted back from the surface. This is known as subsurface scattering.

The subsurface scattered light exits the surface at varying distances from its entry point. The distribution of the entry-exit distances depends on the scattering properties of the material. If the entry-exit distances are small compared to the distance between shading samples, they can be assumed to be effectively zero for shading purposes, allowing subsurface scattering to be combined with surface reflection into the local shading model, with outgoing light at a point depending only on incoming light at the same point. However, since subsurface-scattered light has a significantly different appearance than surface reflected light, we divide them into separate shading terms. The specular term models surface reflection, and the diffuse term models local subsurface scattering (Akenine-Möller, et al., 2018).

If the entry-exit distances are large compared to the distance between shading samples, then specialized rendering techniques are needed to model the visual effect of light entering the surface at one point and leaving it from another (Akenine-Möller, et al., 2018). These specialized techniques will be explored in a later section.

4.3. Microfacet Theory

Physically based rendering techniques base themselves on the microfacet theory. Microfacet theory is the mathematical analysis of the effects of microgeometry on a materials reflectance. (Akenine-Möller, et al., 2018). This theory describes that any surface at a microscopic scale can be described by tiny perfectly reflective mirrors called microfacets (Learn OpenGL, 2016).

Figure 4.1 Microfacet example (Learn OpenGL, 2016)

The roughness of a surface changes the orientation of these mirrors and thus causes light to scatter along a variety of directions on rougher surfaces, resulting in a more widespread specular reflection. In contrast, smooth surfaces will reflect light in the same direction across the surface giving rise to a smaller and sharper specular reflection (Learn OpenGL, 2016).

Figure 4.2 Microfacet light scattering example (Learn OpenGL, 2016)

4.4. The BRDF

A material model, referred to as a Bidirectional Scattering Distribution Function (BSDF), can be decomposed into two parts: A Reflectance part, the Bidirectional Reflectance Distribution Function (BRDF), and a Transmittance part, the Bidirectional Transmittance Distribution Function (BTDF).

During this chapter, I will focus on the BRDF as translucence materials will be explored in a later chapter.

The BRDF forms part of what is known as the rendering equation (Equation 4.1). Where $f_r(x, \omega_i, \omega_o, \lambda, t)$ represents the BRDF.

$$L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_\Omega f_r(x, \omega_i, \omega_o, \lambda, t) L_i(x, w_i, \lambda, t)(\omega_i \cdot \boldsymbol{n}) d\omega_i$$

Equation 4.1 The Rendering Equation

The Cook-Torrance BRDF is a function that can be plugged into the rendering equation in place of f_r. With this BRDF we model the behaviour of light in two different way making a distinction between diffuse reflection and specular reflection providing an individual BRDF for each (CodingLabs.net, n.d.).

The diffuse BRDF model the local scale subsurface scattering for dielectric materials. While it is ignored for metallic objects as metallic objects do not refract light and thus have no diffuse reflection (Learn OpenGL, 2016).

The specular BRDF models the specular reflection of a material and is defined by the Cook-Torrance approximation (Equation 4.2).

$$f(l, v) = \frac{D(h)F(v, h)G(l, v, h)}{4\,(n \cdot l)(n \cdot v)}$$

Equation 4.2 Cook-Torrance Approximation (Karis, 2013)

This BRDF is composed of three functions and a normalization factor. These are defined as the Normal **D**istribution function, the **F**resnel equation and the **G**eometry function (Learn OpenGL, 2016):

- **Normal distribution function:** This approximates the amount that the surface's microfacets align with the halfway vector (a unit vector at half-angle between the view direction vector and light direction vector). It is influenced by the surface's roughness. It is the primary function approximating the surface microfacets.
- **Fresnel equation:** The Fresnel equation describes the ratio of surface reflection at different viewing angles.
- **Geometry function:** This describes the self-shadowing property of the microfacets. When a surface is relatively rough, the surface's microfacets can overshadow other microfacets reducing the amount of light that the surface reflects.

We integrate the BRDFs over all relevant light sources to arrive at the final colour value of a surface material.

4.5. Direct Lighting Implementation

I will now cover my implementation of calculating the effect of direct light sources on a material.

4.5.1. Shaders

The shaders, written in GLSL, first require me to define a material system which accepts the following material parameters:

```
struct PBRMaterial
{
    vec3 albedo;
    float metalness;
    float roughness;
};
```

Code Sample 4.1 PBR material parameters

To begin with, I started with a single directional light source as this is the simplest light source containing just a colour and direction.

I grouped my PBR lighting calculations into a single function CalculateDirectLighting, which takes in the material parameters along with the view vector, light vector and surface normal, although it is considered bad practice to be using single variable names, for my PBR shader code I chose to use a naming convention that matches the commonly used mathematical notation for formulating BRDFs as this made the link between the equations and code much clearer.

```
vec3 CalculateDirectLighting(PBRMaterial material, vec3 V, vec3 L, vec3 N)
{
    vec3 H = normalize(V + L);

    vec3 diffuse = DiffuseBRDF(material.albedo);

    vec3 kS;
    vec3 specular = SpecularBRDF(material, N, V, H, L, kS);

    //Ensure energy conservation
    vec3 kD = vec3(1.0) - kS;

    //Remove diffuse component for metalics
    kD *= 1.0 - material.metalness;

    return kD * diffuse + specular;
}
```

Code Sample 4.2 Direct lighting calculation shader function

4.5.2. The Diffuse BRDF

For the diffuse BRDF I explored a couple of options. In his paper, Real Shading in Unreal Engine 4, Brian Karis from Epic Games compares Lambertian diffuse (Equation 4.3) to Disney diffuse (Equation 4.4), presented by Brent Burley from Disney (Burley, 2012) and concludes that there are only minor visual differences between the two thus making Disney diffuse not worth the extra computational cost and that Disney diffuse leads to added challenges when incorporating image-based lighting.

$$f(l, v) = \frac{c_{diff}}{\pi}$$

Equation 4.3 Lambertian Diffuse

$$f_d = \frac{baseColor}{\pi}(1 + (F_{D90} - 1)(1 - \cos\theta_l)^5)\,(1 + (F_{D90} - 1)(1 - \cos\theta_v)^5)$$

$$F_{D90} = 0.5 + 2\cos\theta_d^2 roughness$$

Equation 4.4 Disney Diffuse (Burley, 2012)

However, Sébastien Lagarde & Charles de Rousiers from Dice concluded in their paper, Moving Frostbite to Physically Based Rendering (Lagarde & Rousiers, 2014), that this conclusion is incorrect and leads to a disconnect with the specular reflection of the object as the roughness of the surface has not been taken into account. They concluded that Disney Diffuse is the correct solution.

Stephen McAuley in his presentation about physically based rendering at Ubisoft (McAuley, 2019) stated that they also used simple Lambertian diffuse up until around 2018. Going off of all this information I felt that the simple implementation of Lambertian diffuse was a good choice as a first step and other methods could be tried at a later stage of the project.

```
//Lambertian
vec3 DiffuseBRDF(vec3 albedo)
{
    return albedo / Pi;
}
```

Code Sample 4.3 Lambertian diffuse implementation

4.5.3. The Specular BRDF

For the specular BRDF I utilised the Cook-Torrance BDRF approximation (Equation 4.2) as all sources I came across show that this is a well-entrenched industry standard. To begin I followed the implementation and techniques set out by Brian Karis (Karis, 2013) to start me off.

Each component of the numerator offers different techniques with varying results, Brian Karis put together a fantastic resource (Karis, 2013) of some of the various options available and elaborated on the benefits of each, I used this resource to pick the appropriate implementation for each component.

```
vec3 SpecularBRDF(PBRMaterial material, vec3 N, vec3 H, vec3 V, vec3 L, out vec3 FOut)
{
    float NDF = DistributionGGX(N, H, material.roughness);

    vec3 F0 = vec3(0.04);
    F0 = mix(F0, material.albedo, material.metalness);
    vec3 F = FresnelSchlick(max(dot(H, V), 0.0), F0);

    FOut = F;

    float G = GeometrySmith(N, V, L, material.roughness);

    //Calculate Cook-Torrance BRDF
    vec3 numerator = NDF * F * G;
    float denominator = 4 * max(dot(N, V), 0.0) * max(dot(N, L), 0.0);

    return numerator / max(denominator, 0.001);
}
```

Code Sample 4.4 Cook-Torrance BRDF shader function

4.5.4. The Normal Distribution Function

For the Normal Distribution Function, Karis concluded that Disney use of GGX/Trowbridge-Reitz (Equation 4.5) presented by Burley to be an ideal candidate as it produces a natural appearance due to the longer "tail" that it produces for the specular reflection. α here corresponds to the roughness of the surface, while h refers to the halfway vector between the surface normal and the light direction.

$$NDF_{GGXTR}(n, h, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2}$$

Equation 4.5 GGX/Trowbridge-Reitz (Learn OpenGL, 2016)

When roughness is low, a highly concentrated number of microfacets are aligned to the halfway vector over a small radius. Due to this high concentration the NDF displays a very bright spot. On a rough surface where microfacets are aligned in much more random directions, the microfacets align with the halfway vector in a much lower concentration thus producing greyer flatter results (Learn OpenGL, 2016).

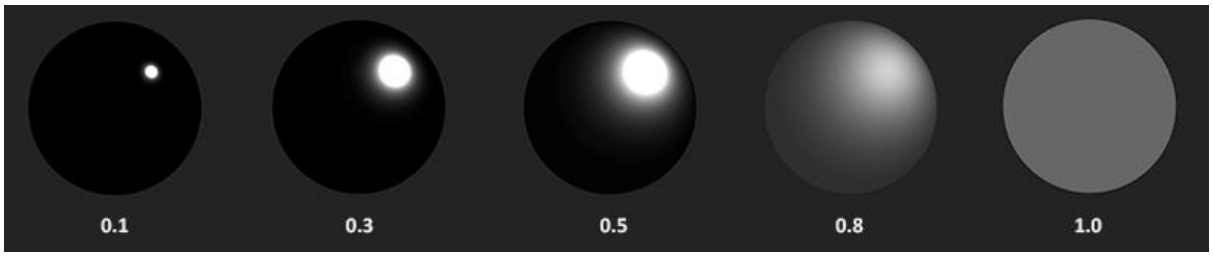

Figure 4.3 Effects of NDF as roughness increases (Learn OpenGL, 2016).

This resulted in the following implementation by me:

```
//GGX (Trowbridge-Reitz)
float DistributionGGX(vec3 N, vec3 H, float roughness)
{
    float a = roughness*roughness;
    float a2 = a * a;
    float NdotH = max(dot(N, H), 0.0);
    float NdotH2 = NdotH * NdotH;

    float denom = (NdotH2 * (a2 - 1.0) + 1.0);
    float denom2 = denom * denom;

    return a2 / (Pi * denom2);
}
```

Code Sample 4.5 GGX/Trowbridge-Reitz implementation

4.5.5. Fresnel Reflection

For the Fresnel reflection, the industry standard is the Schlick approximation.

$$F_{Schlick}(h, v, F_0) = F_0 + (1 - F_0)(1 - (h \cdot v))^5$$

Equation 4.6 Fresnel reflection Schlick approximation (Learn OpenGL, 2016).

In this equation F_0 represents the base reflectivity of the surface, which is calculated using the materials index of refraction. There are a few caveats to the Schlick approximation, the first is that the equation is only really defined for dielectric materials. For conductors (metals) calculating the base reflectivity with indices of refraction doesn't properly hold and instead, we would need to use a different Fresnel equation.

As this is inconvenient, we further approximate the equation by pre-computing the surface's response at normal incidence (F_0) at a 0-degree angle as if looking directly onto a surface. We interpolate this value based on the view angle, as per the Fresnel-Schlick approximation, allowing us to use the same equation for both metals and non-metals. However, we need to tint the base reflectivity for a metallic surface to achieve realistic results (Learn OpenGL, 2016).

```
vec3 F0 = vec3(0.04);
F0 = mix(F0, material.albedo, material.metalness);
vec3 F = FresnelSchlick(max(dot(H, V), 0.0), F0);
```

Code Sample 4.6 F_0 calculation

To simply things, most rendering engines define a base reflectivity that is averaged around most common dielectrics. This is usually 0.04, however, a potential improvement would be to expose this parameter to be configurable.

```
//Schlick
vec3 FresnelSchlick(float cosTheta, vec3 F0) // costTheta is the LDotH or the anlge between the half vector
{
    return F0 + (1.0 - F0) * pow(max(1.0 - cosTheta, 0.0), 5.0);
}
```

Code Sample 4.7 Fresnel Schlick approximation implementation

4.5.6. Geometric Attenuation (Geometry Function)

The geometric attenuation function approximates the relative surface area where its micro surface details overshadow each other causing light rays to be occluded. This is based off of the roughness parameter of a surface's material (Learn OpenGL, 2016).

The geometry function that I used is a combination of the GGX and Schlick-Beckmann approximation known as Schlick-GGX (Equation 4.7).

$$G_{SchlickGGX}(n, v, k) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k}$$

Where *k* is a remapping of roughness according to

$$k = \frac{(Rougness + 1)^2}{8}$$

Equation 4.7 Schlick-GGX (Karis, 2013)

This remapping was put forward by Burley in his Disney PBR paper as it reduced perceived "hotness" and was more usual for artists (Karis, 2013).

To effectively approximate the geometry, we need to take account of both the view direction (geometry obstruction) and the light direction vector (geometry shadowing). We can take both into account using Smith's method (Learn OpenGL, 2016):

$$G(n, v, l, k) = G_{sub}(n, v, k)G_{sub}(n, l, k)$$

Equation 4.8 Geometric Attenuation: Smith Method

Which lead to my implementation:

```
float GeometrySchlickGGX(vec3 N, vec3 V, float roughness)
{
    float r = (roughness + 1.0);
    float k = (r * r) / 8.0;

    float NdotV = max(dot(N, V), 0.0);

    float num = NdotV;
    float denom = NdotV * (1.0 - k) + k;

    return num / denom;
}

float GeometrySmith(vec3 N, vec3 V, vec3 L, float roughness)
{
    return GeometrySchlickGGX(N, V, roughness) *  GeometrySchlickGGX(N, L, roughness);
}
```

Code Sample 4.8 Geometry attenuation smith method implementation

4.5.7. Energy Conservation

After calculating the results of the diffuse BRDF and the specular BRDF there a couple of more steps you must undertake to ensure that you comply with the laws of physics.

We need to ensure that our BRDF is energy conserving and does not emit more light than it has received, except in the case of emissive surfaces. We do this by calculating the ratio that our diffuse BRDF and specular BRDF contributes to the reflectance equation and ensure that their combined ratio is not above 1.0. kS here is equal to the Fresnel value of our specular BRDF as this gives us the ratio of specular light for a surface.

Secondly, because metallic surfaces do not refract light, we must disregard or further scale the diffuse component if the material is metallic.

```
//Ensure energy conservation
vec3 kD = vec3(1.0) - kS;

//Remove diffuse component for metalics
kD *= 1.0 - material.metalness;

return kD * diffuse + specular;
```

Code Sample 4.9 Energy conservation shader calculation

4.5.8. Results

Below is the result of applying the direct lighting BRDFs to an array of spheres with a base albedo of RGB(255, 0, 0) i.e. Red, with metalness increasing along the X-axis in increments of 0.2 and roughness increasing along he Y-axis in increments of 0.2. Notice the lack of diffuse lighting in the bottom left on the fully metallic material with 0 roughness and the growing specular highlight as roughness increases.

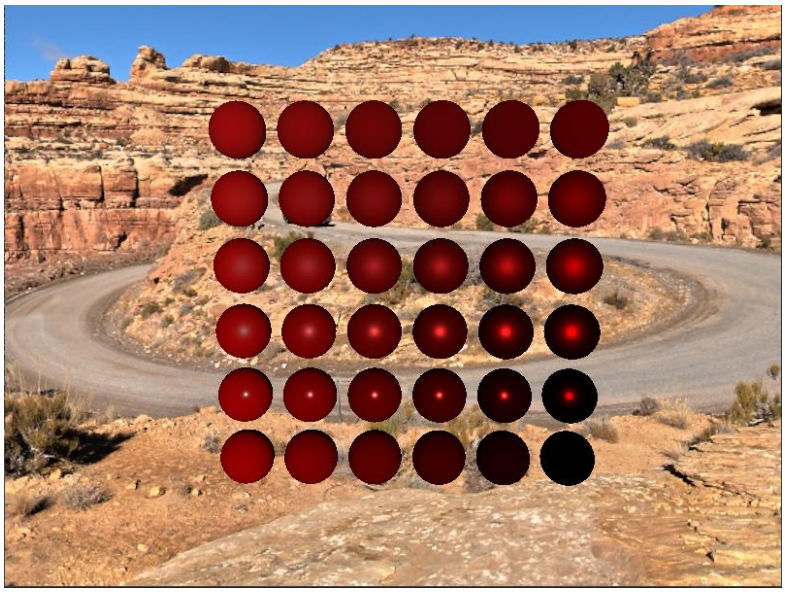

Figure 4.4 Results of applying direct lighting PBR BRDFs to a range of materials

4.6. Indirect / Environment Lighting

When only modelling direct lighting as seen in the previous results you miss out on the major visual improvements that PBR introduces, particularly so for metallic surfaces as they have no diffuse reflectance. With direct light alone you do not get any interesting specular reflection, only a highlight of the colour of the light source. To correct this, I will now go over some techniques to model this environment light.

In the reflectance equation (Equation 4.9), there is no difference between a direct light source and indirect light sources. All incoming light sources have a radiance value and the reflectance equation integrates over them (Akenine-Möller, et al., 2018).

$$L_o(p,\omega_o) = \int_\Omega (k_d \frac{c}{\pi} + k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}) L_i(p,\omega_i) n \cdot \omega_i d\omega_i$$

Equation 4.9 Cook-Torrance Reflectance Equation (Learn OpenGL, 2016)

The task, therefore, comes down to how do we represent these radiance values for the overall environment of a scene. One approach, first presented by Karis (Karis, 2013), is to use a pre-processed HDR environment texture map which models each pixel of the texture as a light source. This technique is the pioneering technique for environment lighting and is used by many rendering engines. However, this technique is beginning to fall out of use as it is deemed to be too static for interactive environments and it is quite difficult to source suitable texture maps (Lagarde & Rousiers, 2014). Instead, a growing trend is to develop a system to generate a physical based sky and atmosphere that contains a vast amount of information about your environment such as: altitude, air composition, longitude, latitude, time of year and weather effects. Which all contribute to the final lighting (Unity Technologies, n.d.). As this is a very advanced technique, I chose Karis' image-based lighting technique using environment maps. This is a good baseline to work from and still produces visually pleasing results with a physical basis.

4.6.1. Environment Map / Image-Based Lighting

Solving the reflectance integral requires you to sample an environment map from all possible directions ω_i over the hemisphere Ω which is far too expensive to do in real-time (Learn OpenGL, 2016). To overcome this, we pre-compute as much of integral as we can and store this data in separate texture maps which will be sampled to retrieve the information.

Like with direct lighting, the reflectance equation (Equation 4.10) can be split into induvial components for diffuse and specular lighting.

$$L_o(p,\omega_o) = \int_\Omega (k_d \frac{c}{\pi}) L_i(p,\omega_i) n \cdot \omega_i d\omega_i + \int_\Omega (k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}) L_i(p,\omega_i) n \cdot \omega_i d\omega_i$$

Equation 4.10 Cook-Torrance Reflectance Equation Split into Diffuse Component and Specular

If we assume *p* is the centre of the environment map then for diffuse lighting this gives an integral that only depends on the direction ω_i as the diffuse lambert term is a constant term (the colour c, the refraction ratio k_d, and π are constant over the integral) (Learn OpenGL, 2016).

Knowing this we can precompute a new cube map known as the irradiance map, that for each sample direction ω_o stores a precomputed sum of all indirect diffuse light of the scene hitting some surface aligned along a direction. The result of this can be seen below.

Figure 4.5 Irradiance map

For the specular component, we are not so lucky and instead the integral depends on the incoming light direction and the incoming view direction. Epic Games' split sum approximation (Equation 4.11) solves this by splitting the pre-calculation into 2 individual parts that we can later. The split sum approximation splits the specular integral into two separate integrals:

$$L_o(p, \omega_o) = \int_\Omega L_i(p, \omega_i) d\omega_i * \int_\Omega f_r(p, \omega_i, \omega_o) n \cdot \omega_i d\omega_i$$

$$f_r(p, \omega_i, \omega_o) = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

Equation 4.11 Split sum approximation

The first term, when precalculated, is stored in what is known as the pre-filtered environment map. This is similar to the irradiance map, but we take roughness into account. For increasing roughness levels, the environment map is convoluted with more scattered sample vectors, creating blurrier reflections. For each roughness level we convolute, we store the sequentially blurrier results in the pre-filtered map's mipmap levels.

The sample vectors and their scattering amount are generated using the normal distribution function of the Cook-Torrance BRDF that takes a normal and view direction as input. Epic Games makes a further approximation to the integral by assuming the view direction to be equal to the output sample direction ω_o . The effect of this is that we don't get grazing specular reflections when looking at specular surface reflections from an angle this is however generally considered an acceptable compromise (Learn OpenGL, 2016). Karis points out that this is the largest source of error and break from reality in imaged based lighting (Karis, 2013).

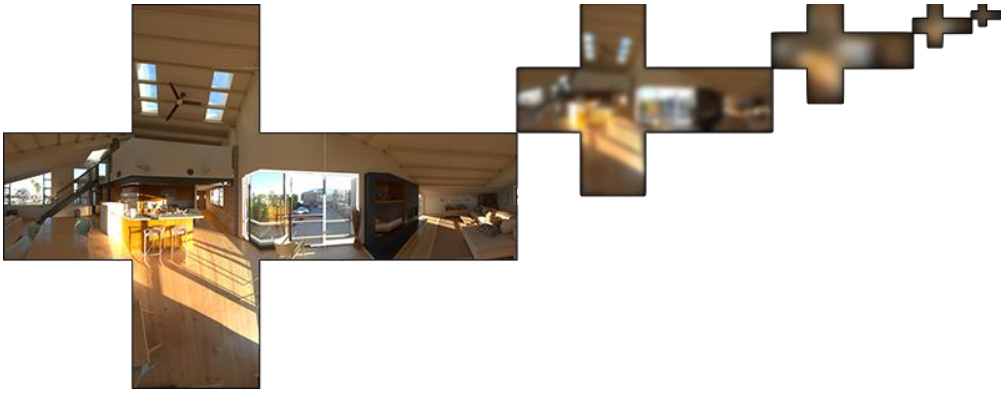

Figure 4.6 pre-filtered environment map (Learn OpenGL, 2016)

The second term of the split sum approximation is the BRDF part of the specular integral. This is the same as integrating the specular BRDF with a solid-white environment (Karis, 2013), we store the response of the BRDF as a 2D lookup table in a texture known as the BRDF integration map.

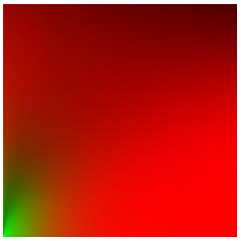

Figure 4.7 BRDF integration map (Learn OpenGL, 2016)

This integration map is constant across all environment maps so it can be calculated once and reused.

Both the pre-filtered environment map and the BRDF integration map can then be combined in a shader to calculate the specular integral.

4.6.2. Calculating the Environment Light Contribution

To calculate the indirect lighting contribution for a surface material we need to sample these 3 generated textures.

```
layout(set = 1, binding = 3) uniform samplerCube radianceMap;
layout(set = 1, binding = 4) uniform samplerCube irradianceMap;
layout(set = 1, binding = 5) uniform sampler2D integrationMap;
```

Code Sample 4.10 Environment lighting texture map shader bind points

As the irradiance map holds all of the scenes indirect diffuse light, we sample this irradiance map along the surface normal and combine the value with the material albedo to get the diffuse portion of the indirect light

```
vec3 irradiance = texture(irradianceMap, N).rgb;
vec3 diffuse    = irradiance * material.albedo;
```

Code Sample 4.11 Environment light diffuse lighting calculation

Like with direct lighting we must still scale this diffuse light according to the specular light contribution so that we obey the laws of energy conservation.

To do this we use a modification of the Fresnel Schlick equation used indirect lighting however this time we modify the equation to directly take surface roughness into account as before we used the half vector but in this case we do not have a single half vector as we are receiving indirect light from all direction in a hemisphere orientated around the surface normal (Learn OpenGL, 2016).

```
vec3 FresnelSchlickRoughness(float cosTheta, vec3 F0, float roughness)
{
    return F0 + (max(vec3(1.0 - roughness), F0) - F0) * pow(max(1.0 - cosTheta, 0.0), 5.0);
}
```

Code Sample 4.12 Fresnel Schlick approximation with roughness taken into account for environment lighting

The result of the Fresnel equation gives us the ratio of specular light that the surface is receiving, like before we can then work out the ratio to apply to the diffuse light, also like with direct lighting we disregard/scale the diffuse lighting for metallic surfaces.

```
vec3 F0 = vec3(0.04);
F0 = mix(F0, material.albedo, material.metalness);
vec3 F = FresnelSchlickRoughness(max(dot(N, V), 0.0), F0,  material.roughness);

vec3 kS = F;
vec3 kD = 1.0 - kS;
kD *= 1.0 - material.metalness;
```

Code Sample 4.13 Energy conversion calculations for environment-based lighting

To calculate the specular lighting contribution, we sample the pre-filtered environment map (radiance map) along the reflection vector of the surface normal. We also sample an appropriate mip level of the radiance map based on the surface roughness which will give rougher surfaces blurrier specular reflections (Learn OpenGL, 2016).

```
vec3 R = reflect(-V, N);

const float MAX_MIP_INDEX = 6.0;
vec3 prefilteredColor = textureLod(radianceMap, R,   material.roughness * MAX_MIP_INDEX).rgb;
```

Code Sample 4.14 Sampling the radiance map according to surface roughness value

We then sample the BRDF integration lookup texture given the surface roughness and the angle between the normal and view vector.

With this value, we can work out the specular component of the environment lighting by multiplying it with the radiance map sample, Fresnel value and the sample of the integration map.

```
vec2 envBRDF  = texture(integrationMap, vec2(max(dot(N, V), 0.0),  material.roughness)).rg;
vec3 specular = prefilteredColor * (F * envBRDF.x + envBRDF.y);
```

Code Sample 4.15 Calculating the specular BRDF for environment lighting

To get the total indirect lighting contribution we add the diffuse and specular components together with the diffuse component scaled by the diffuse lighting ratio.

```
return (kD * diffuse) + specular;
```

Code Sample 4.16 Total environment lighting contribution

Finally, to get the final lighting value of a surface we combine the direct lighting contribution with the indirect lighting contribution to give the pixel its colour.

```
vec3 directLight = CalculateDirectLighting(mat, V, L, N) * perFrameInfo.lightColour * NdotL;

vec3 indirectLight = CalculateIndirectLighting(mat, V, L, N);

vec3 finalLighit = directLight + indirectLight;
```

Code Sample 4.17 Combining direct lighting contribution and environment light contribution to final surface colour

These environment lighting techniques then need to be combined with other global illumination techniques such as reflections, shadows and ambient occlusion to arrive at the final representation of a material in a scene. These topics will be covered in later sections of the project.

4.7. Results

Below are the final results of applying physically based rendering techniques to different material parameters. In the image (Figure 4.8) the surface metalness increases along the X-axis. The surface roughness increases along the Y-axis giving you an overview of the result of the BRDFs with different material parameters. This presents a huge contrast to the direct lighting example and highlights the importance of environment light and reflections for metallic materials.

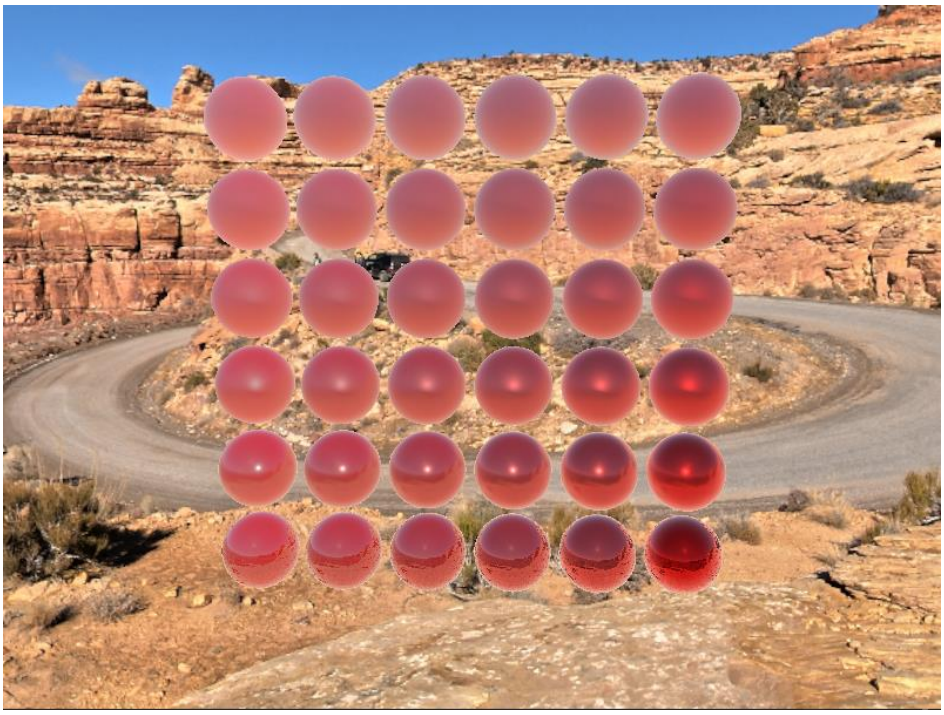

Figure 4.8 Results of PBR shading including environment lighting

5. Deferred Shading

Up to this point, I had been utilising what is known as a forward rendering architecture. This alone is a quite basic and outdated architecture and quickly becomes a bottleneck when implementing complex scenes with many lights. To overcome this, I set out to explore and implement a different rendering architecture known as deferred shading.

5.1. Introduction

Deferred shading is a technique which is used to support a very large amount of lights in a scene. In a typical forward shading architecture, you render an object and calculate its surface colour according to all lights in the scene and you repeat this process for every object in the scene. Forward rendering on its own can be quite wasteful in highly complex scenes as fragment shader output can be overwritten when objects overlap, thus wasting the time we spent on the lighting calculations for that fragment (Learn OpenGL, 2016).

Deferred shading on the other hand involves performing all visibility tests and surface property evaluations before performing any material lighting computations (Akenine-Möller, et al., 2018). This surface property evaluation involves writing basic attributes of the mesh such as the position, normal and PBR material properties of an object to full screen render targets. These render targets, collectively known as the G-Buffer, hold these surface properties and correspond pixel to pixel to the final render target. This stage of generating the G-Buffer is known as the geometry pass. After the G-Buffer is generated we invoke a lighting pass which reads in the values of the G-Buffer as inputs into our PBR BRDFs for each pixel and outputs the final colour of the pixel. This essentially changes the complexity of rendering a scene from $O(geometry * lights)$ to $O(geometry + lights)$ (Neverender, 2010).

For traditional forward rendering, the vertex and pixel shader programs retrieve each light's and material's parameters and computes the effect of one on the other. Forward shading needs either one complex vertex and pixel shader that covers all possible combinations of materials and lights, or shorter, specialized shaders that handle specific combinations. Long shaders with dynamic branches often run considerably slower, so a large number of smaller shaders can be more efficient, but they also require more work to generate and manage. Since all shading functions are done in a single pass with forward shading, it is more likely that the shader will need to change when the next object is rendered, leading to inefficiency from swapping shaders (Akenine-Möller, et al., 2018).

Deferred shading allows a strong separation between lighting and material definition. Each shader is focused on parameter extraction or lighting, but not both. Shorter shaders run faster, both due to length and the ability for the GPU to optimize them. The number of registers used in a shader determines occupancy, a key factor in how many shader instances can be run in parallel. This decoupling of lighting and material also simplifies shader system management. For example, this split makes experimentation easy, as only one new shader needs to be added to the system for a new light or material type, instead of one for each combination (Akenine-Möller, et al., 2018).

Deferred shading is not without its downside though. Two important technical limitations of deferred shading involve transparency and antialiasing. Transparency is not supported in a basic deferred shading system, since we can store only one surface per pixel. One solution is to use forward rendering for transparent objects after the opaque surfaces are rendered with deferred

shading. An advantage of forward shading methods is that antialiasing schemes such as MSAA are easily supported. Forward techniques need to store only N depth and colour samples per pixel for N × MSAA. Deferred shading could store all N samples per element in the G-buffers to perform antialiasing, but the increases in memory cost, fill rate, and computation make this approach expensive (Akenine-Möller, et al., 2018). To overcome this antialiasing problem, you can instead utilise other antialiasing techniques such as FXAA as a post-process on your final rendered image. Deferred shading also has a considerable impact on VRAM usage although this is less of an issue on modern GPUs which typically have 4GB+ of VRAM

5.2. The G-Buffer

As mentioned previously the G-Buffer is a collection of textures which store model data that is used as input into the lighting pass. A simple G-Buffer layout for a PBR rendering pipeline would contain the following properties:

- A 3D vector representing world position;
- A 3D normal vector representing the surface normal in world space;
- RGBA albedo colour;
- Metalness value;
- Roughness value;

This is how such a G-Buffer layout would look in memory:

Layer Content	R	B	G	A	Channel Size
World Position	X	Y	Z	-	32 Bit
Surface Normal	X	Y	Z	-	32 Bit
Albedo	R	G	B	A	8 Bit
Metalness	X	-	-	-	8 Bit
Roughness	X	-	-	-	8 Bit

Figure 5.1 Basic G-Buffer

As you can see this is exceptionally wasteful as we have a lot of empty channels in our render targets. For a render target size of 1920 * 1080 (1080p) this would take up over 87MB of VRAM and for 4K resolution, this would be over 350MB.

An immediate optimisation that can be seen is to pack the metalness and roughness parameters into the same render target but onto different channels. Another optimisation that is not so obvious is that the world position layer is redundant as the world position can be reconstructed from the scene's depth buffer.

The world position can be reconstructed by reversing the process by which the depth value was calculated. Given the screen position and depth value we first convert the screen position from NDC space to Clip space by remapping it from [0 -> 1] to [-1 -> +1]. Plugging the depth value in as the Z coordinate of the clip space we can convert to view space by multiplying this position by the inverse project matrix and then performing the perspective divide by diving by the W component. Then to get to world space we multiply this view space coordinate by inverse view matrix.

```glsl
// Convert screen space coordinates to world space.
vec4 ScreenToWorld(vec2 screenPos, float depth, mat4 InverseProjection, mat4 InverseView)
{
    vec2 transformedScreenPos = vec2(screenPos.x, screenPos.y) * 2.0f - 1.0f;

    // Convert to clip space
    vec4 clip = vec4(transformedScreenPos, depth, 1.0f);

    // Convert to view space
    vec4 view = InverseProjection * clip;
    view /= view.w;

    // Convert to world space
    vec4 world = InverseView * view;

    return world;
}
```

Code Sample 5.1 World position reconstruction from using depth buffer value

By utilising these optimisations, we can roughly half the G-Buffer's memory requirements, which has the resulting layout:

Layer Content	R	B	G	A	Channel Size
Surface Normal	X	Y	Z	-	32 Bit
Albedo	R	G	B	A	8 Bit
MetalicRoughness	Metalness	Roughness	-	-	8 Bit

Figure 5.2 Optimised G-Buffer

The memory requirements for this layout are now 47MB for 1080p and 189MB for 4K. This is still a rather basic G-Buffer compared to that of most modern architecture however it serves its purpose for this project and can be expanded on as needed by new techniques such as ambient occlusion. Also, there exists further optimisation techniques for reducing the memory requirements of the surface normals down to 10 bits per channel, however, this will be explored at a later stage.

Below is an example of the G-Buffer used by Ubisoft in Rainbow 6 Siege.

	R8	G8	B8	A8
RT0	world normal (RGB10)			GI
RT1	base color (sRGB8)			config (A8)
RT2	metalness (R8)	glossiness (G8)	cavity (B8)	aliased value (A8)
RT3	velocity.xy (RGB8)			velocity.z (A8)

Figure 5.3 An example of a possible G-buffer layout, used in Rainbow Six Siege. In addition to depth and stencil buffers, four render targets (RTs) are used as well (Akenine-Möller, et al., 2018).

5.3. Geometry Pass Implementation

To perform the geometry pass of my deferred rendering implementation we need to create a separate Vulkan graphics pipeline per mesh type and material combination. For example, in my case as a starting point, I created a pipeline to accept static meshes with PBR parameters specifying its materials. Another example of a combination that would be needed is animated meshes which have PBR textures as its material.

Creating the G-Buffer is relatively simple and involves just writing the required properties of the mesh material combo to the G-Buffer frame buffers in the fragment shader.

```
layout (location = 0) out vec4 outNormal;
layout (location = 1) out vec4 outAlbedo;
layout (location = 2) out vec4 outMetalicRoughness;

layout(set = 0, binding = 1) uniform PBRMaterialParameters{
    vec4 albedo;
    float metalness;
    float roughness;
} material;

void main()
{
    outNormal = vec4(inNormal, 1.0);
    outAlbedo = material.albedo;
    outMetalicRoughness = vec4(material.metalness, material.roughness, 0.0, 0.0);
}
```

Code Sample 5.2 Geometry pass fragment shader for static meshes with PBR parameter material

The resulting G-Buffer would appear as follows:

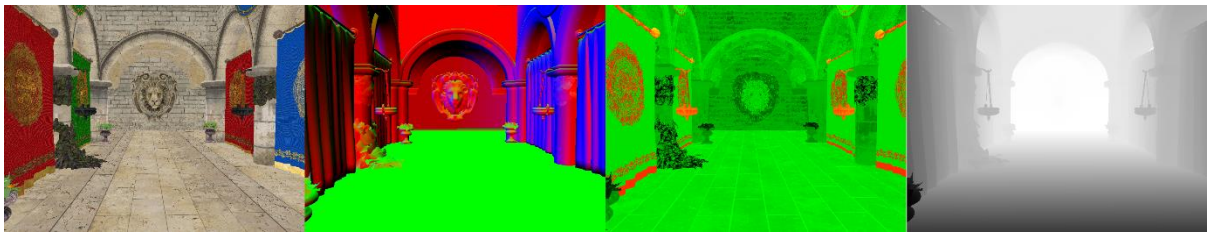

Figure 5.4 G-Buffer contents after geometry pass, from left to right: albedo layer, surface normal layer, metallic-roughness layer and dept buffer of Sponza scene (in-engine)

5.4. Lighting Pass Implementation

For the lighting pass, we just need a separate pipeline per light type. As of now, I support 3 different light sources: point lights, directional lights and environment light. More light types will be added as the project progresses.

Each of these light shaders take in the G-Buffer as input and use its content to calculate the result of the PBR BRDFs. We utilise one draw call per light and blend the results into the final render target. This allows us to only need to calculate the PBR BRDF once per light per pixel.

This alone will allow a large number of lights in a scene, but it is quite wasteful as the light shaders are executed on every pixel even if the light turns out to have no effect on the surface at that pixel. To further improve performance, we can utilise light volumes. Lights volumes are spheres or other shapes which represent the bounds of where the effect of this light will be applied. By utilising light

volumes, we can force the pixel shader to only execute on pixels that overlap with this bounding volume thus reducing wasted BRDFs evaluations.

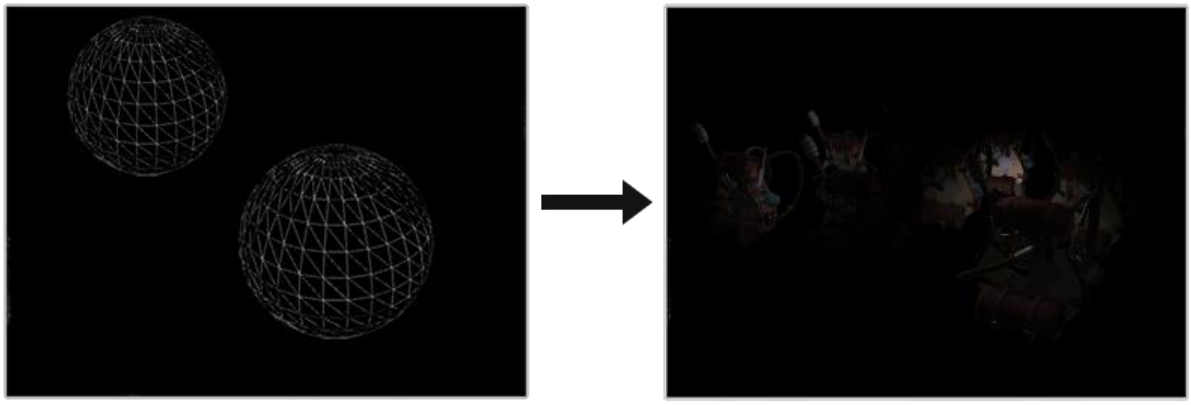

Figure 5.5 Light volumes (Learn OpenGL, 2016).

5.5. Combining Forward and Deferred Rendering

Due to the drawback of lack of transparency support with deferred shading, it is often necessary to combine deferred shading and forward shading together to completely render the scene, as transparent objects are not supported yet this is preparation for when they will be implemented further down the line

The process of rendering a scene used both deferred shading and forward shading involves the following:

1) Render to the G-Buffer;

```
// ------------------------------------------------------------------------------------------
// First Step: Render G-Buffer
// ------------------------------------------------------------------------------------------
VulkanDebug::BeginMarkerRegion(commandBuffer, "GBuffer Generation Pass", glm::vec4(0.0f, 1.0f, 0.0f, 1.0f));
deferredRenderer.RenderGBuffer(commandBuffer, logicalDevice, commandPool, scene);
VulkanDebug::EndMarkerRegion(commandBuffer);
```

Code Sample 5.3: Deferred Shading: G-Buffer Generation

2) Copy over the depth buffer from the g-buffer to the swap chain depth duffer;

```
vkCmdBlitImage(commandBuffer,
    deferredRenderer.gBuffer.depth.image.textureImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL,
    depthImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
    1, &imageBlit, VK_FILTER_NEAREST);
```

Code Sample 5.4: Deferred Shading: Copy depth buffer

3) Perform the lighting pass;

```
// ------------------------------------------------------------------------------------
// Perform Lighting pass on gbuffer
// ------------------------------------------------------------------------------------
VulkanDebug::BeginMarkerRegion(commandBuffer, "Lighting Pass", glm::vec4(1.0f, 1.0f, 1.0f, 1.0f));
deferredRenderer.PerformLightingPass(commandBuffer, logicalDevice, commandPool, scene, inheritanceInfo);
VulkanDebug::EndMarkerRegion(commandBuffer);
```

Code Sample 5.5: Deferred Shading: Lighting Pass

4) Perform forward rendering;

```
// ------------------------------------------------------------------------------------
// Normal forward rendering
// ------------------------------------------------------------------------------------
VulkanDebug::BeginMarkerRegion(commandBuffer, "Forward Rendering Pass", glm::vec4(0.0f, 0.5f, 0.2f, 1.0f));
forwardRenderer.PerformRenderPass(commandBuffer, logicalDevice, commandPool, scene, sampler, inheritanceInfo);
VulkanDebug::EndMarkerRegion(commandBuffer);
```

Code Sample 5.6: Deferred Shading: Forward Rendering

5.6. Tiled Shading

In basic deferred shading, each light is evaluated separately and the values are blended together on the final render target (Akenine-Möller, et al., 2018). This alone offered optimisation over forward rendering when the scene contained less than 100 light sources. However, with scenes that contain hundreds or even thousands of light sources, the cost of evaluating each light source and blending the results begins to add up and quickly becomes a bottleneck.

Tiled shading is a technique created to overcome this issue. The core idea of tiled shading is to divide up the render target into tiles of pixels (Figure 5.6), i.e. 16 x 16 grids of pixels, we then create a list of lights that affect each tile. The lists of lights are then iterated over in a single fragment shader invocation and the final colour value for a pixel is calculated, thus saving the cost of multiple shader calls and the cost of blending together their results.

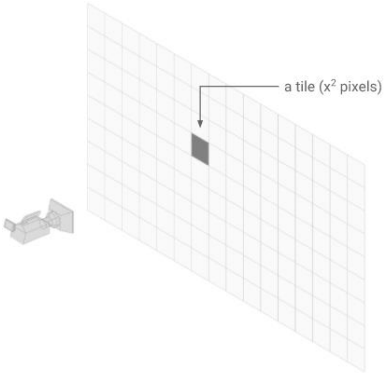

Figure 5.6 Example of render target tiles (Unity Technologies, 2019)

To create this list of for a tile we cull the light volume against the view frustum that is created by the tile from the camera (Figure 5.7). This tile frustum can be further optimized by sampling the minimum and maximum depth value of that from the depth buffer and using these values to calculate the near and far plane of our frustum. To perform the culling, we utilise compute shaders on the GPU as this task can be easily parallelized. Each tile is assigned to a different thread of a size

which allows the whole tile to be processed in one go, i.e. for a 16 x 16 tile we utilise a thread group of size 32. Once these lists of lights have been created, we then pass them onto the fragment shader which loops through the lists for each tile and calculates its final colour.

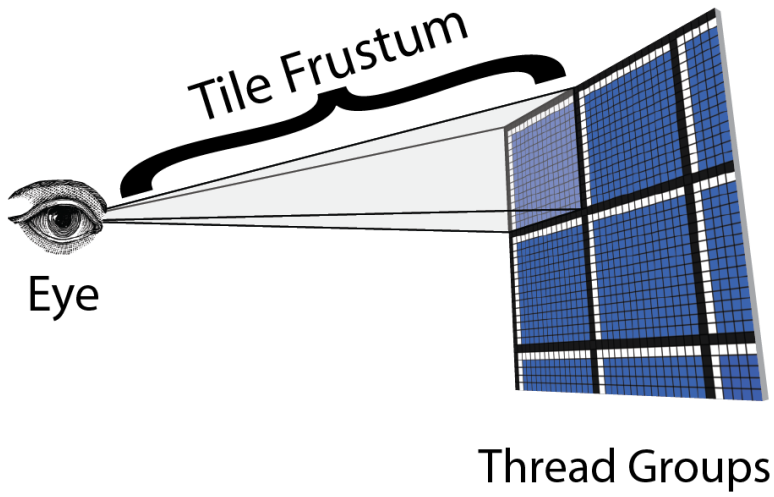

Figure 5.7 Example of tile frustums in tiled shading (Jeremiah, 2015)

To test the effectiveness of this technique, I created a scene with 1000 light sources and compared normal deferred shading to tiled deferred shading.

For normal deferred shading, the lighting pass took 381ms to execute on the GPU, far too slow for real-time applications when the light count is this high. After implementing a basic version of tiled shading, the lighting pass was reduced to 6ms and the light culling stage took 21ms. This presented a fantastic improvement and I expect to see further gains in performance as I optimized my light culling stage.

Figure 5.8 In-engine scene with 1000 light sources @ 30FPS

The light culling stage of tiled shading does not only have to be used with deferred shading, it can also be utilised with forward rendering in what is known as forward+ rendering, in certain situations this can even be as fast as deferred shading.

5.7. Clustered Shading

Clustered Shading is an advancement on tiled shading and is the focus of current work at the time of writing. Tiled shading splits the render target into 2D tiled, while clustered shading splits the whole view frustum into 3D cells called clusters. Unlike tiled shading where tile frustums were created according to the min and max depth recorded in a cell, clustered are independent of scene geometry (Akenine-Möller, et al., 2018).

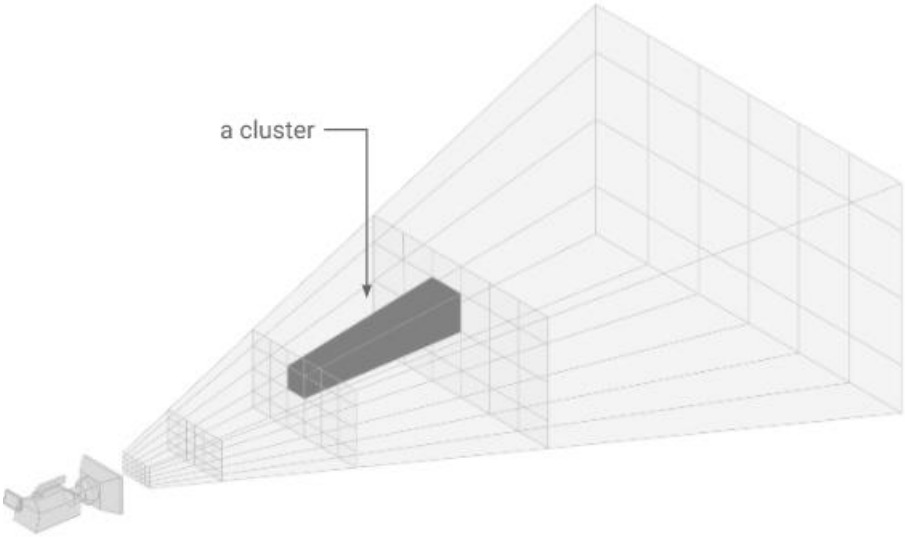

Figure 5.9 Clustered shading cluster example (Unity Technologies, 2019).

Similar to tiled shading, lists of lights are created by culling light volumes against these clusters. However, these lights are now a 3D array that will be queried according to an object's position so that it can retrieve all lights which it will be affected by in the fragment shader. Clustered shading offers an improvement over tiled shading in that it has support for transparent objects and the light lists generated are generally shorter than tiled shading as the light volume culling is more accurate (Akenine-Möller, et al., 2018). Like tiled shading clustered shading can too be used in either forward rendering architectures or deferred shading architectures.

6. Conclusion / Future Work

As it currently stands, I am very happy with how my work has progress and the rendering engine I have created. I achieved the goals I had set out during the week 8 presentation to have a 3D Vulkan rendering engine that uses a PBR shading model and uses a tiled deferred shading system. The engine's performance is quite good, and I aim to improve this as I discover new optimisation techniques that I can implement.

For the upcoming semester my primary focus will be as following:

Step 1: Implement global illumination techniques.

To support global illumination, I will be implementing a hybrid rendering system utilising ray tracing for shadows, reflections and global illumination while using rasterization for drawing triangles and performing local illumination as is currently done.

Step 2: Support transparent and translucent materials

Currently, I only support opaque materials, I will need to research the correct approach for this. There are techniques that allow transparent and translucent material to be rendered in one pass without the need to do a pre-ordering of the objects according to depth and this will be my preferred approach.

Step 3: Efficient scene representation and asset management

Currently, I have a basic scene and asset management system in place that has worked for basic scenes. This will need to be improved upon to manage memory issues that will arise when I try to render complex scenes with many different objects. One approach for better scene management that I will explore is render graphs, which allow efficient automation of rendering a scene.

Step 4: Object culling/acceleration algorithms

Currently, I render all objects in a scene and leave it to the rasterization stage to cull triangles outside of the viewport. This will soon become a bottleneck so I will need to implement an efficient object culling system so that unnecessary objects are not submitted to the GPU for rendering.

Step 5: Tackle Stretch Goals

The main stretch goal I intend to implement is an abstraction layer over Vulkan and DirectX12 so that both are available as an option to use to render a scene. The Diligent Engine (Diligent Graphics, 2021) has an open-source solution for this so I aim to take inspiration from this project to guide my implementation.

7. References

Akenine-Möller, T. et al., 2018. Clustered Shading. In: *Real-Time Rendering 4th Edition.* s.l.:A K Peters/CRC Press, pp. 899 - 900.

Akenine-Möller, T. et al., 2018. Deferred Shading. In: *Real-Time Rendering 4th Edition.* s.l.:A K Peters/CRC Press, pp. 883 - 887.

Akenine-Möller, T. et al., 2018. Environment Lighting. In: *Real-Time Rendering 4th Edition.* s.l.:A K Peters/CRC Press, p. 391.

Akenine-Möller, T. et al., 2018. Microfacet Theory. In: *Real-Time Rendering 4th Edition.* s.l.:A K Peters/CRC Press, p. 331.

Akenine-Möller, T. et al., 2018. Physics of Light. In: *Real-Time Rendering 4th Edition.* s.l.:A K Peters/CRC Press, pp. 293 - 306.

Akenine-Möller, T. et al., 2018. Tiled Shading. In: *Real-Time Rendering 4th Edition.* s.l.:A K Peters/CRC Press, p. 892.

Barré-Brisebois, C. & @ZigguratVertigo, 2019. *Are We Done With Ray Tracing? State-of-the-Art and Challenges in Game Ray Tracing.* s.l.:SEED – Electronic Arts.

Burley, B., 2012. *Physically-Based Shading at Disney,* s.l.: Walt Disney Animation Studios.

CodingLabs.net, n.d. *Physically Based Rendering - Cook–Torrance.* [Online]
Available at: http://www.codinglabs.net/article_physically_based_rendering_cook_torrance.aspx
[Accessed 12 January 2021].

Diligent Graphics, 2021. *Diligent Engine.* [Online]
Available at: https://github.com/DiligentGraphics/DiligentEngine
[Accessed 19 January 2021].

Foley, T., 2015. *Next-Generation Graphics APIs: Similarities and Differences.* s.l., NVIDIA Corporation.

Jeremiah, 2015. *Forward vs Deferred vs Forward+ Rendering with DirectX 11.* [Online]
Available at: https://www.3dgep.com/forward-plus/#Forward
[Accessed 19 January 2021].

Karis, B., 2013. *Graphic Rants - Specular BRDF Reference.* [Online]
Available at: http://graphicrants.blogspot.com/2013/08/specular-brdf-reference.html
[Accessed 14 January 2021].

Karis, B., 2013. *Real Shading in Unreal Engine 4.* s.l., Epic Games.

Lagarde, S. & Rousiers, C. d., 2014. *Moving Frostbite to Physically Based Rendering 3.0.* s.l., SIGGRAPH.

Learn OpenGL, 2016. *Learn OpenGL - Deferred Shading.* [Online]
Available at: https://learnopengl.com/Advanced-Lighting/Deferred-Shading
[Accessed 15 January 2021].

Learn OpenGL, 2016. *Learn OpenGL - Diffuse irradiance.* [Online]
Available at: https://learnopengl.com/PBR/IBL/Diffuse-irradiance
[Accessed 14 January 2021].

Learn OpenGL, 2016. *Learn OpenGL - Specular IBL.* [Online]
Available at: https://learnopengl.com/PBR/IBL/Specular-IBL
[Accessed 14 January 2021].

Learn OpenGL, 2016. *LearnOpenGL - Theory.* [Online]
Available at: https://learnopengl.com/PBR/Theory
[Accessed 5 January 2021].

Learn OpenGL, 2016. *Lighting.* [Online]
Available at: https://learnopengl.com/PBR/Lighting
[Accessed 14 January 2021].

LunarG, Inc, 2019. *Create a Vulkan Instance.* [Online]
Available at: https://vulkan.lunarg.com/doc/view/1.2.135.0/windows/tutorial/html/01-init_instance.html
[Accessed 15 January 2021].

McAuley, S., 2019. *Advances in Rendering, Graphics Research and Video Game Production.* s.l., Ubisoft.

Neverender, 2010. *What is deferred rendering?.* [Online]
Available at: https://gamedev.stackexchange.com/questions/74/what-is-deferred-rendering
[Accessed 16 Janaury 2021].

The Khronos Group, 2021. *Vulkan Overiew.* [Online]
Available at: https://www.khronos.org/vulkan/
[Accessed 15 January 2021].

Unity Technologies, 2019. *Setting up the Rendering Pipeline and Lighting in Unity.* [Online]
Available at: https://docs.unity3d.com/Manual/BestPracticeLightingPipelines.html
[Accessed 19 January 2021].

Unity Technologies, n.d. *Physically Based Sky.* [Online]
Available at: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@7.1/manual/Override-Physically-Based-Sky.html
[Accessed 14 January 2021].

Vulkan Guide, 2020. *Executing Vulkan Commands.* [Online]
Available at: https://vkguide.dev/docs/chapter-1/vulkan_command_flow/
[Accessed 14 January 2021].

Vulkan Guide, 2020. *Rendering Loop.* [Online]
Available at: https://vkguide.dev/docs/chapter-1/vulkan_mainloop/
[Accessed 14 January 2021].

Vulkan Guide, 2020. *Vulkan initialization.* [Online]
Available at: https://vkguide.dev/docs/chapter-1/vulkan_init_flow/
[Accessed 14 January 2021].

Vulkan Tutorial, n.d. *Instance - Vulkan Tutorial.* [Online]
Available at: https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Instance
[Accessed 5 January 2021].

Wihlidal, G., 2019. *Halcyon: Rapid Innovation using Modern Graphics.* s.l.:SEED - Electronic Arts.

Wikipedia, n.d. *Light - Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Light
[Accessed 5 January 2021].