Sheffield Hallam University

Faculty of Science, Technology and Arts

Department of Computing

Project (Technical Computing)

Author	Pablo Bañó Benito
Student ID	C0023130
Year	2020/2021
Supervisor	Luke Melville
Second Marker	Martyn Prigmore
Degree Course	Computer Science for Games
Title of Project	An approach to ... Physically Based Rendering.

Confidentiality Required?

NO ☒
YES ☐

I give permission to make my project report, video and deliverable accessible to staff and students on the Project (Technical Computing) module at Sheffield Hallam University.

NO ☒
YES ☐

Sheffield Hallam University

An approach to ...
Physically Based Rendering

Pablo Bañó Benito

Abstract

Physics-based rendering has been gaining in importance for a few years, until relatively recently, it was used mostly in industries outside of video games, although with the constant advances in graphics card architectures, it is beginning to take hold. This project seeks to develop a 3D scene that has several objects with physics-based rendering, based on the Vulkan graphics API.

This report also seeks to delve a little deeper into the theoretical aspect of this technique. It is an academic report that does not delve too deeply into the development and mathematical foundation, but it does explain a little what the main characteristics are and why the PBR works.

Keywords: Vulkan, Physically Based Rendering, PBR, Reflectance, BRDF

Sheffield Hallam University

Contents Page

Sheffield Hallam University

Chapter I

1. Introduction

1.1. Project Overview

For years, companies that develop videogames seek to create scenes and environments that look and feel like the real world. For this they have been developing and refining graphic techniques behind mathematical and physical models that study, for example, the behavior of light when it bounces off a surface. These mathematical models are not new, they have, in fact, existed for several years, but the technology that was available was not so powerful as to be able to carry it out efficiently.

However, with the powerful machines that are available now, it is possible to implement these mathematical models in videogames to represent scenes that appear realistic. In addition, although it is true that it is used a lot in videogames, it does not apply exclusively to them. For example, the film industry has been using similar techniques for years, with the advantage that they did not need to run in real time, as it does with videogames or interactive media.

This project researches a specific graphic technique: PBR (Physically Based Rendering). With this in mind, it will be developed and researched what the result of these mathematical models is, taking into account various materials and surfaces, such as gold or copper, for example.

1.2. Project Aims

The aim of the project is essentially the development of the basic PBR technique with the Vulkan graphics library. It includes learning how the SwapChains works, render passes, descriptor layout, descriptor pools, etc.

It is not only about implementing the graphical technique, it is also about understanding and researching how to develop correctly with Vulkan, the common pitfalls, solutions, alternative implementations and code structuring.

1.3. Deliverables

The scene of this project will be created using a little demo developed with the Vulkan API, following the Vulkan-Tutorial webpage.

A VisualStudio solution is delivered that includes all code files, both personal and third-party. The final scene has several spheres on the screen, all of them have an assigned material that shows a basic PBR technique, which shows how light acts on materials with more or less roughness.

2. Information Review

2.1. History of Physically Based Rendering

Starting in the 1980s, a number of rendering researchers worked on establishing a solid theoretical basis for rendering, including physical correctness. A 1997 paper from Cornell University Program of Computer Graphics describes the work done at Cornell in this area to that point. The goal of this paper was to develop physically based lighting models that will produce images that are visually and measurably indistinguishable from real-world images.

2.2. Technique of PBR

The technique of Physically Based Rendering seeks to imitate the behavior of light in a virtual world in such a way that it achieves a detail quite similar to what we perceive in the real world. It generally looks more realistic compared to our original lighting algorithms (e.g. Phong, Blinn-Phong).

It is a representation taking into account the physical properties of objects. In the real world, we can see things around us because light rays enter our eye after bouncing off the surface of objects.

One part of these rays is scattered, causing the diffusion effect (diffuse), the other part bounces off the surface at an opposite angle relative to the surface normals, causing a reflection (specular).

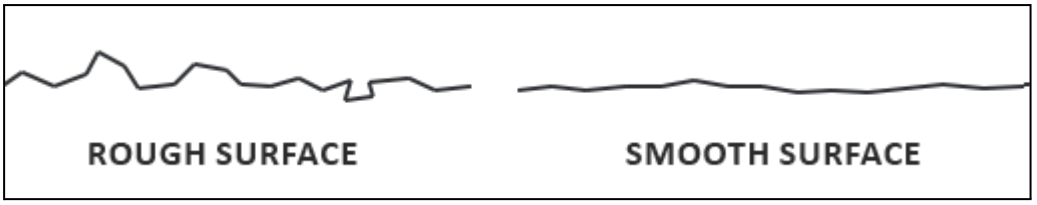


Figure 2.2.1 - Roughness. Image from LearnOpenGL (PBR, The microfacet model).

Metals (conductors) and nonmetals (commonly called dielectrics) reflect and scatter light differently.

- Non-metals are materials that have a pronounced diffuse scattering and weak specularity (plastic, wood, etc.). Due to diffuse scattering, we can see that the red plastic is red. And due to the reflection we can see glitters on the surface.

- Metals (steel, iron, gold, copper, brass, silver, etc.) are materials without diffuse scattering (diffusion is always black), but clearly marked with a colored reflection.

There are techniques such as photogrammetry, which help to discover and encode precise optical properties of materials. Photogrammetry studies the shape, dimensions and position in space of any object using measurements made on one or more photographs of said object.

Chapter II

3. Physically Based Rendering

3.1. Surface Reflections

When the light reflects on a surface, the color that is perceived is, essentially, the ray of light reflected with a specific wavelength. For example, the sand mostly absorbs light in the blue, but reflects the light in the red and green wavelengths.

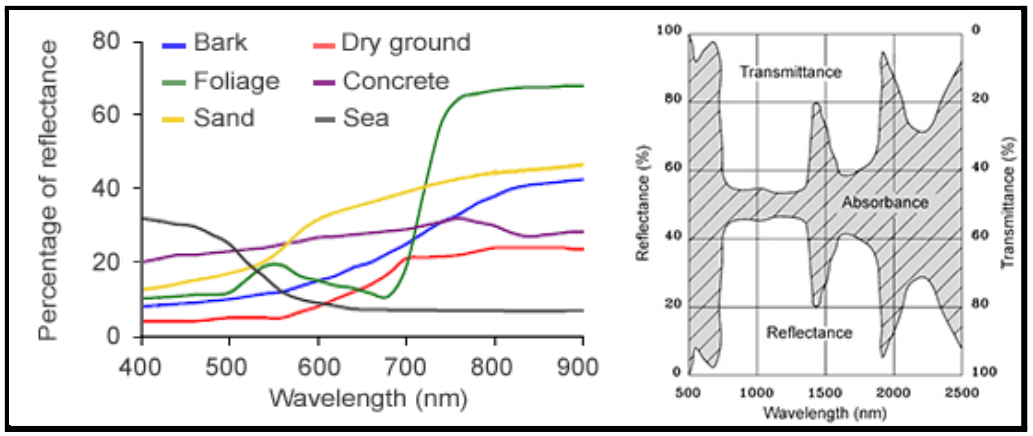


Figure 3.1.1 - Spectrum of reflectance and transmittance of various materials. Images from Luxorion in Astrosurf webpage.

With this in mind it can define what is commonly called "material", and the standard model is described mathematically by a BSDF (Bidirectional Scattering Distribution Function), witch is composed of two other functions: BRDF (Bidirectional Reflectance Distribution Function) and BTDF (Bidirectional Transmittance Function).

3.1.1. Bidirectional Reflectance Distribution Function

Basically, the BRDF states the reflections from a surface. It describes the radiance that the surface/material is leaving in a specific direction toward the viewer.

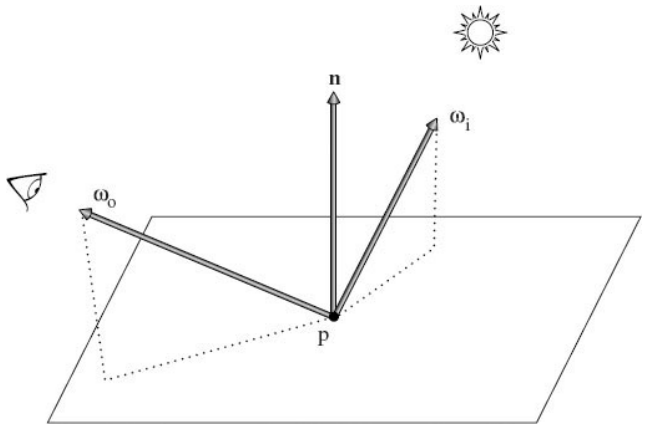


Figure 3.1.1.1 - BRDF is a four dimensional function over pairs of directions ω¡ and ω0 that describes how much incident light along ω¡. Image from the book Physically Based Rendering, 2nd Edition, by Matt Pharr and Greg Humphreys.

3.1.2. The Cook-Torrance specular BRDF

$$f_{CookTorrance} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

Where:

- ω**o** is the outgoing direction
- ω**i** is the light incoming direction (camera / viewer)
- n is the surface normal
- D is the **Normal Distribution function**
- F is the **Fresnel equation**
- G is the **Geometry function**

3.1.2.1. Normal Distribution function

Physically Based Shading in Film and Game Production was the job that Brent Burley was doing in 2012, at Disney. Brent observed that long-tailed normal distribution functions (NDF) are a good fit for real-world surfaces.

It states that the total value of the projected mi-crofacets areas should be the same as the differential area of the considered surface.

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

```
float D_GGX(float NoH, float roughness) {
    float a = NoH * roughness;
    float k = roughness / (1.0 - NoH * NoH + a * a);
    return k * k * (1.0 / PI);
}
```

Figure 3.1.2.1.1 - Implementation of the NDF function in GLSL. Code extract from GoogleFilament Github.

When a surface is smooth, many microfacets line up around a very small radius, resulting in highly shiny spots on the surface.

Just the opposite happens when the surface is quite rough, since, since the microfacets are aligned in a more or less random way, the light is not concentrated in specific points, but rather it is distributed relatively homogeneously over the entire surface.

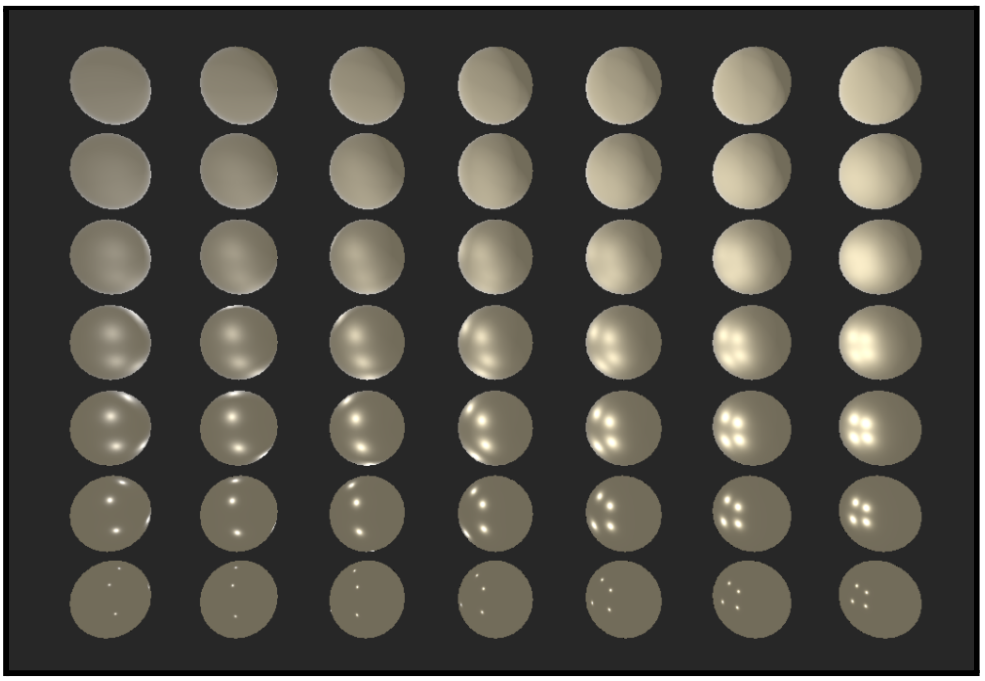


Figure 3.1.2.1.2

In this case, the image shows several spheres in a scene with 4 lights. It is a grid in which the roughness of the spheres increases as it approaches the upper right corner, in the same way that as it is closer to the lower left corner the roughness decreases, as does the intensity of the light.

The image shows some spheres with four points with a lot of brightness, as they are quite smooth surfaces. Each point corresponds to each light that is located in the scene. The spheres that have a more scattered and better distributed light correspond to rougher spheres.

3.1.2.2. Fresnel equation

Is the amount of light that the viewer sees reflected from a surface, and depends on the viewing angle.

The light that is perceived is different depending on the angle in which we are seeing it. The reflections change as we move or pivot around them. Each surface has

a different reflectivity. This can be easily verified by observing wooden or metal surfaces, such as a table or kitchen utensil, while the viewing angle is changed or the object is rotated.

This phenomenon is known as 'Fresnel', and, being based on a mathematical model, the reflectivity of an object can be calculated by applying the corresponding formula:

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

However, the formula does not apply equally to metallic and non-metallic materials, as they have different refractive indices.

At first this can be a problem, but, nevertheless, if the value that the surface 'returns' is previously calculated if it is viewed with an angle of 0 degrees, the value can be interpolated depending on the angle from which it is looking. By doing this, the same formula could already be applied for metallic and non-metallic materials.

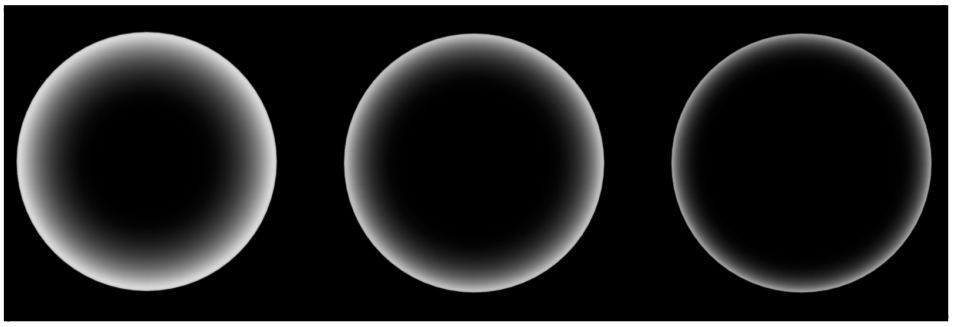


Figure 3.1.2.2.1 - Representation of the fresnel effect. Image from GameBanana.

3.1.2.3. Geometry function

When light is applied to a surface, the rays do not have to bounce just once. This means that it could bounce off itself repeatedly, causing some areas of the surface itself to be shaded.

On a perfectly flat and smooth surface, light rays bounce only once. However, the rougher the surface is, the more likely it is to occur.

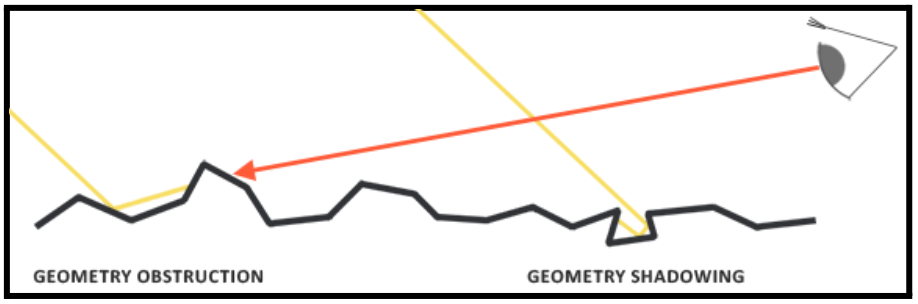


Figure 3.1.2.3.1 - Graphical representation of the geometry function. Image from LeanOpenGL, PBR Theory.

The geometry function is a function of the roughness of the surface, it is used as an approximation.

For this, it must take into account both the directing vector of the light and the direction of the person who is looking at the object (the camera, in this case).

For the correct implementation of the geometry formula, two different methods are taken, the Schlick-GGX function and Smith's method:

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

Schlick-GGX formula

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

Smith's method.

Using both methods, the geometry function determines which areas darken more or less depending on the roughness of the surface being observed.

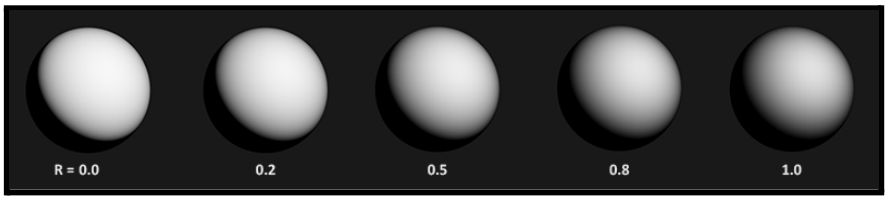


Figure 3.1.2.3.2 - Representation of the Geometry Function varying the roughness. Image from LearnOpenGL, PBR Theory.

3.2. Roughness

The roughness value determines how rough a surface is. The rougher a surface, the more blurred and wider the reflections of light are. The opposite happens with smooth surfaces, since the reflections are clear and well focused. The roughness value ranges from zero to one, with one being a rough surface, and zero being a completely smooth surface.

It is very common to use textures to determine the roughness of a material, and since the value ranges between zero and one, black and white textures are used as a 'height map' that determines the roughness based on color.

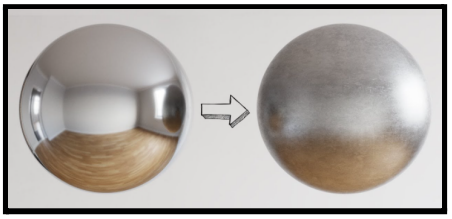


Figure 3.2.1 - Metallic roughness

3.3. Reciprocity and energy conservation

Physically based BRDF have two important qualities:

- **Reciprocity**: it is reciprocal because the directions of incident and outgoing light do not alter the result for all pairs of directions **ωo** and **ωi**.

$$f(\omega_i, \omega_o) = f(\omega_o, \omega_i)$$

  In the real world, light emitters such as a spotlight or a light bulb emit photons that are scattered through objects until they 'hit' the observer's eye. In a computer-created environment, the direction of the light is reversed, and it is actually the camera's directing vector that 'hits' the light-emitting object. This is only mathematically correct if the BSDF is reciprocal, since, if it were not, apart from the fact that a fundamental physical law would be violated, it would also render the BSDF unusable.

- **Principle of energy conservation**: is a key component of a good BRDF. States that "the amount of specular and diffuse reflectance energy is less than the total amount of incident energy." This means that no more radiant energy can be reflected from a point than the energy incident to that point.

Diffuse + Reflection + Refraction <= Incoming Light

If the specular reflection area increases, the brightness also decreases at increasing roughness levels. This is why we see specular reflections more intensely on smooth surfaces.

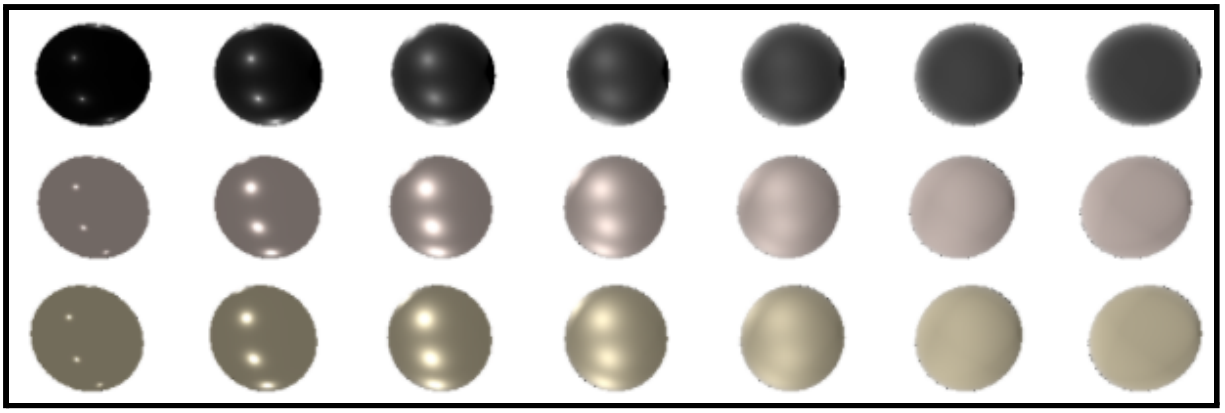


Figure 3.3.1 - Roughness with different materials (e.g. Copper, Gold, etc.). Image taken from the little demo developed with the Vulkan API.

3.4. Image based lighting (IBL)

In real life, light comes from multiple directions and from different sources. It can come directly from a light source or indirectly after bouncing off an object in the environment. This implies that it does not always come from a light-emitting source, such as a light bulb or a bulb, but that almost any object can be treated as if it emitted light. Cubemaps are a great way to represent this. This technique is known as 'diffuse irradiance'.

3.4.1. Diffuse irradiance

When lighting a scene, there are various methods to illuminate objects, either with light-emitting sources (see section 6) or by taking a large source of light, such as a room, as a reference. This technique consists of treating each texel of the cubemap as a light emitting point.

In this way it is achieved that a metallic object with little roughness reflects the environment in which it is located.

Figure 3.4.1.1 - Example of a PBR material with reflections of the environment around the object.

To correctly implement this type of technique, the reflectance equation is widely used:

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

This formula is a function of the direction the light comes from, which, in this case, is anywhere in the scene.

The reflectance equation is also a function of position. It makes sense, since the reflections would change if the center of the environment changed position (or if the position of the rest of the objects within it changed).

It can be a problem, as constantly solving the reflectance equation can be expensive. However, we can pre-calculate the equation from different points and use the result of one or the other depending on where the object is located.

There are several types of IBL, used quite commonly in video game engines:

- **Distant lights:** helps to capture lighting information at 'infinity'. It also provides lighting data for buildings, lawns, etc.

- **Surface reflections:** These are often used to capture information from relatively flat areas, such as water or the floor of a building.

- **Local lights:** they are more accurate than the information provided by distant lights. They are often used to capture a specific area of the scene from a specific point of view. They are very practical for adding reflections to materials locally.

3.4.2. Anisotropy

Anisotropic filtering is a method of improving the quality of a texture on a surface that is viewed from an oblique angle to the projection angle of the texture on a surface.

The anisotropic filter analyzes each pixel in the texture in real time and adds more information (new pixels), so a 16x16 pixel texture can be converted to a 128x128 texture after applying the anisotropic filter.

Chapter III

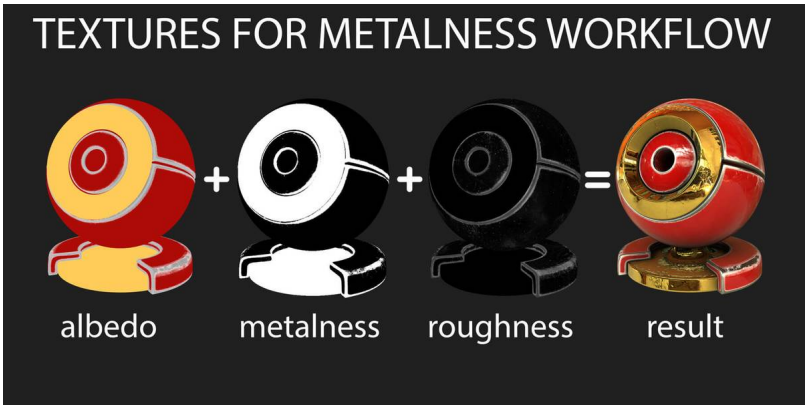
4. Materials

Generally, the objects that are located in a scene have a material attached that determines, roughly, how that object should be rendered. A material has different attributes that are processed differently. For this, several textures are usually used that contain the information that is needed to process all that data.

For example, the 'albedo' texture determines what color the object will have in its different parts, the 'metalness' texture determines whether a material is metallic or not, the 'roughness' texture determines which parts of the object are rougher, etc.

The previous chapter talked about how light behaves differently depending on the surface material. In the case of the 'metalness' texture, the light spectrum only takes values from 0 to 1, with 1 being completely white and 0 being completely black. In this case, the whiter the texture, the more metallic the object to be painted ends up being.

In the case of the 'roughness' texture the same thing happens, except that the whiter, the more roughness is applied.



5. Light casters

In a 3D environment, there must be objects that emit light so that those illuminated by it can be rendered in the scene. This chapter exposes three different types of light, although it does not delve too deeply into the implementation of each of them, it does define how they work and what characteristics they have.

Throughout the execution of the demo, the attributes of the lights involved in the scene can change. Can change the direction of a Directional light, the position of a Point light, or both in the case of a Spot light, so they have to be calculated and modified appropriately.

5.1. Directional lights

The main purpose of directional lights is to recreate important light sources for the outdoor environment, e.g. the sun or the moon (as considered from Earth). Directional lights are 'infinitely' far away and only need one direction to be processed.

The intention of this type of lights is to give what we could call 'ambient light'. All objects in the Scene are illuminated equally as the light from a Directional light does not diminish.

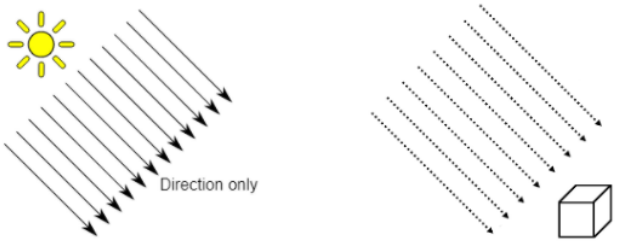


Figure 5.1.1 - Interaction between a directional light and a surface. Image from Unity3D and Qt3D documentation.

5.2. Point lights

A PointLight is characterized by illuminating a scene in 360°, as if it were a light bulb. This is why we will only take into account the position you are in, and not your address. Just as a light bulb loses strength as a function of distance, Point lights are implemented with a parameter called 'attenuation', which also increases or decreases depending on the distance at which the object is from the light source.

Attenuation is calculated with the following formula:

$$F_{att} = \frac{1.0}{K_c + K_l * d + K_q * d^2}$$

Where:

- ***Kc*** is the **constant** term.
- ***Kl*** is the **linear** term.
- ***Kq*** is the **quadratic** term.
- ***d*** is the distance from fragment to the light source.

Kc is a variable that is responsible for 'ensuring' that the denominator is never less than one, since, if it were less, there would be some areas in which the intensity of the light would increase even though the object was very far from it.

Kl is multiplied by the distance and reduces the intensity of the light linearly.

Kq is multiplied by the square of the distance, and sets the decrease in light intensity.

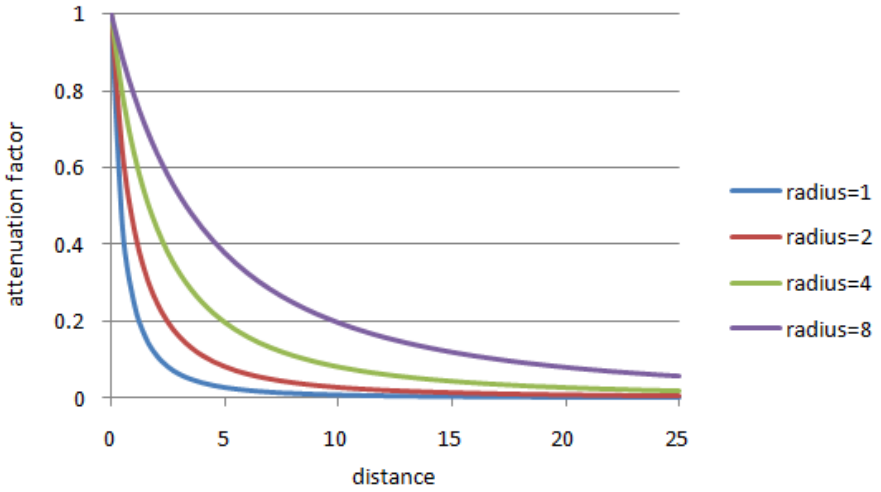


Figure 5.2.1 - The graph shows the effect such an attenuation has over a distance of 25. Image from ImDoingItWrong web by Tom Madams.

5.3. Spot lights

A Spotlight is a type of light that is located in a specific place in the scene, and, in addition, it also consists of a direction. It is a mix between a Directional light and a Point light, but it has the peculiarity of consisting of two interior 'cones' whose function is to determine the attenuation within the light source itself.

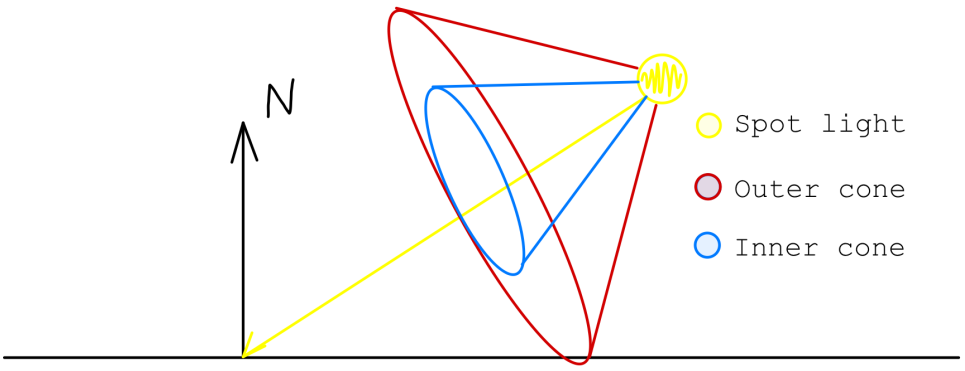


Figure 5.3.1: Small sketch of a SpotLight made by hand.

In the image, the outer cone is shown in red, that is where, depending on the attenuation parameter and the angle between the inner and outer cone, the light will gradually soften as

it approaches the extremes. The higher the angle, the more it will gradually soften. It is not usually a common practice to allow the value to change depending on what the player wants, which is why the normal thing ends up being to interpolate values so that you do not have to do calculations constantly.

The inner cone, shown in blue, can also be changed at a higher or lower angle to change the softness of the light.

The intensity is calculated using the following formula:

$$I = \frac{\theta - \gamma}{\epsilon}$$

Where:

- **θ** is the inner cone.
- **γ** is the outer cone.
- **ε** is the cosine difference between the inner and outer cone.

Figure 5.3.2 - Image from Age of Armour, Advanced Spotlight for DAZ Studio.

5.4. Approximations of lighting

5.4.1. Phong / Blinn-Phong models

The Phong model establishes the way light is reflected off a surface as a combination of diffuse and specular. It is not a perfect model, since it does not apply the principle of conservation of energy or reciprocity (see section 4.2.2.3). In addition, it presents a problem when establishing the specular reflections, since, when the angle of incidence exceeds 90 degrees, the result of applying this model is a vector that is located below the surface that is being illuminated.

Fortunately, in 1977, Jim Blinn proposed a modification to the Phong model. The Blinn-Phong model, instead of using the reflection vector, as the Phong model does, uses a unit vector that is located between the view vector and the light direction vector. This means that when the scene observer's director vector is aligned with the reflection vector, it will also align with the vector perpendicular to the surface.

The 'halfway vector' is easy to calculate using the following formula:

$$\bar{H} = \frac{\bar{L} + \bar{V}}{||\bar{L} + \bar{V}||}$$

Where:

- **L** is the direction vector of the light
- **V** is the vector between the viewer and the surface.

```
vec3 lightDir   = normalize(lightPos - FragPos);
vec3 viewDir    = normalize(viewPos - FragPos);
vec3 halfwayDir = normalize(lightDir + viewDir);
```

```
LightsBuffer lights_buffer{};
lights_buffer.number_lights_.x_ = 0; // Directional
lights_buffer.number_lights_.y_ = 1; // Point
lights_buffer.number_lights_.z_ = 0; // Spots

lights_buffer.directional_light.direction_ = mathmorra::Vector3(0.0f, 0.0f, -1.0f);
lights_buffer.directional_light.position_ = mathmorra::Vector3(3.0f, 0.0f, 3.0f);
lights_buffer.directional_light.intensity_ = 30.0f;

lights_buffer.directional_light.ambient_ = mathmorra::Vector3(1.0f, 1.0f, 1.0f);
lights_buffer.directional_light.diffuse_ = mathmorra::Vector3(1.0f, 1.0f, 1.0f);
lights_buffer.directional_light.specular_ = mathmorra::Vector3(1.0f, 1.0f, 1.0f);

lights_buffer.directional_light.cutOff_ = 0.41f;
lights_buffer.directional_light.outerCutOff_ = 0.12f;

lights_buffer.directional_light.constant_ = 1.0f;
lights_buffer.directional_light.linear_ = 0.7f;
lights_buffer.directional_light.quadratic_ = 1.8f;
```

Figure 6.4.1 - Tiny implementation of the lights buffer in Vulkan. Code extract from the little demo developed for this project.

5.5. Occlusion

Occlusion has become a very popular technique in recent years. The film industry has used this technique for years, but it has generated a lot of interest in the video game community for real-time rendering.

5.5.1. Ambient occlusion

This technique consists of pre-calculating points of light on a surface and checking which are 'obstructed' or blocked by other objects, giving rise to a slightly more shaded area. The visibility function will return zero if the corresponding surface is occluded and one if it is not. In this way, efficient rendering is achieved without constantly performing runtime calculations.

In a more or less static scene, it is a very common technique to pre-calculate surfaces illuminated with direct lights that do not change their direction or position in space.

However, it presents a slight problem, which is that objects that are dynamic are not affected by ambient occlusion generated by static objects, since dynamic objects do not contain an ambient occlusion component of their own.

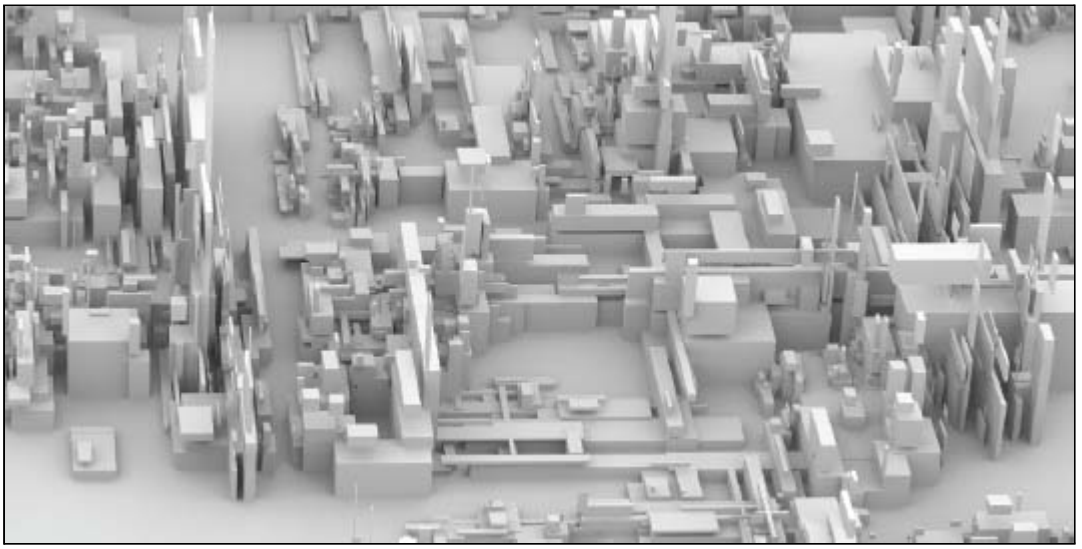


Figure 5.5.1.1 - Example of precomputed ambient occlusion. Image from the book Physically Based Rendering, 2nd Edition, by Matt Pharr and Greg Humphreys.

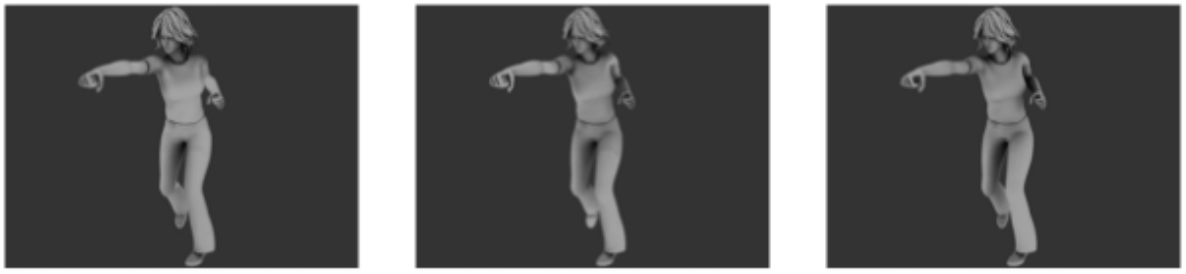
5.5.2. Dynamic occlusion

Applying occlusion to animated objects, such as a character in a video game, is considerably more complicated than pre-calculating static objects. However, a study carried out by Janne Kontkanen and Timo Aila at the University of Helsinki, in 2006, called 'Ambient Occlusion for Animated Characters', shows a

very interesting approach on how occlusion should be applied for animated characters.

This approach is based on the parameterization of the environmental occlusion as a linear combination of parameters, such as inclination angles that exist, for example, between the character's arm and his torso.

It is automatically computed given a reference to the character's poses and their pre-calculated ambient occlusion values. The objective is to establish a 'linear mapping' of arbitrary values from a certain pose.



A B C

Figure 5.5.2.1 – Example of 'dynamic' ambient occlusion. The image shows a comparison of the average ambient occlusion model (A), the technique developed by Kontkanen and Aila (B) against the ground truth (C). Image from 'Ambient Occlusion for Animated Characters'.

Chapter IV

6. Design

The design of the project at the time of development in Vulkan has been structured by classes, there are several classes with different and well-differentiated functionalities.

To explain it better, it will go from the smallest classes to the largest:

- **Camera**: fundamental, it takes care of the basic movement (sideways, up and down), and the calculation of the corresponding matrices of view, projection, etc.

- **GameObject**: the most basic class of all, it is like a container for data and components. It contains information about the geometry, the references to its uniform buffer, the position, rotation, scale, etc.

- **Light**: contains all the data that a light needs to be processed in a shader. Stores its intensity, direction, position, type (see section 7), color of ambient, diffuse and specular, etc.

- **Input**: this class stores all mouse and keyboard input that is received since the demo is run. It consists of a buffer of keys that we alternate depending on whether a key has been pressed, held down, released, etc. It is thanks to this that other classes, such as the camera, can work properly.

- **Scene**: the scene class has a vector of GameObjects. In essence, it contains all the objects that are going to be

painted on screen and takes care of updating them when appropriate.

- **AppManager**: it is a container class for data containers, forgive the redundancy. It is a static class (singleton), which we can access from any part of the execution. Very practical to be able to use the input buffer, the ResourceManager and others.

- **ResourceManager**: it is in charge of the initialization of all the internal buffers, both those of vertices and those of indices of all geometries. In addition, it also stores the primitives (cube, sphere, and 3D models).

- **Window**: by far the largest class of all. Initially, it was going to be in charge of the initialization of the window and its corresponding properties. In the end, it was more like the container for all of Vulkan's tools and features. Initialize the window and Vulkan. In addition, it also runs the game loop.

7. Development and implementation

7.1. Programming Language

The reason for choosing C++, and not C, for example, is because it is a very versatile programming language, object-oriented which makes it much easier to organize code into different classes, depending each with its own specific function. Also with a memory control that comes in handy when working with graphical APIs like Vulkan.

In addition, the performance it offers us is very important, since when dealing with graphical techniques in real time, it must have an optimal performance for it.

Another reason is that it is a multiplatform language, which would allow me (if it were the case) to test the small demo on another operating system other than Windows, such as Linux or Mac.

And last but not least, C ++ is a widely used language, since most programs and applications have even a few fragments written in C ++.

7.2. Third-party libraries

tinyObjLoader: The reason for choosing "tinyObj" is important because it is a small header, as well as powerful. Since it allows us to load millions of polygons with a relatively moderate loading time. Another reason is, essentially, because I had already worked with this header, therefore working with it has been more familiar and comfortable to me than using, for example, Assimp (Open Asset Import Library).

GLFW: it is a library for OpenGL, Vulkan and OpenGL ES. It is multi-platform and Open Source that provides an API for creating windows, contexts and surfaces, receiving inputs and events.

It has support for keyboard and mouse, gamepad and time. Also has bindings for other languages maintained by the community like Node.js, Java, Perl, Delphi, etc.

GLM: 'OpenGL Mathematics' is a mathematics header for graphics software based on Shading Language GLSL. Provides functions, classes and types like scalar, vector, matrix, quaternions, etc.

Stb: it is a single-file header that provides types, image loading, decoding from file, decoding ogg vorbis for audio, memory management like malloc/free leak-checking. The name STB are just the initials of the author: Sean T. Barrett.

7.3. Developing the demo

For the development of the project, I chose to use Vulkan as it is a graphical API that is growing due to its high performance. I relied mainly on the Vulkan tutorial page 'https://vulkan-tutorial.com/', which explains decisions are the steps to be followed from scratch.

It was a long part considering that it is a low level API and it is very extensive and complicated to understand. Starting, as is usually the case, by drawing a triangle on the screen having previously initialized Vulkan.

The steps to follow to be able to paint a triangle can be quite a few and very rough. Starting with having to initialize the 'Instance', 'Surface', 'Physical / logical devices', 'SwapChain', etc.

7.3.1. Buffers

They are a fundamental part of development with graphical APIs, since when the CPU and GPU work in a more or less synchronized way, they need to communicate in some way, so the buffers give us exactly that function. They are data containers that contain a multitude of information, for example, data about the objects that are going to be rendered on the screen.

- Vertex buffers: they are data structures that contain information about which vertices are going to be painted and in what order (e.g. a cube has 8 vertices and 36 indices).

- Uniform buffers: they contain the data that will be passed to the graphics card so that it can process them at the shader level (e.g. delta time, buffers of the lights, information of the objects, such as the rotation, the scale and the position).

- Command buffers: these are the instructions that Vulkan will execute to paint the objects on the screen. As they are not function calls, we save painting information in a buffer for Vulkan to execute later. To do this, in addition, we will use the 'command pools', which are responsible for managing memory to store the buffers that we use to paint.

7.3.2. Command lists and Swap chain

The SwapChain is a queue of images waiting to be rendered on the screen. As there is no default 'FrameBuffer' in Vulkan, we have to explicitly create it. The general purpose of the SwapChain is to render the images you have stored and synchronize them with the refresh rate of the screen. The ultimate goal is to grab an image from the queue, render it, and return it to the end of the queue.

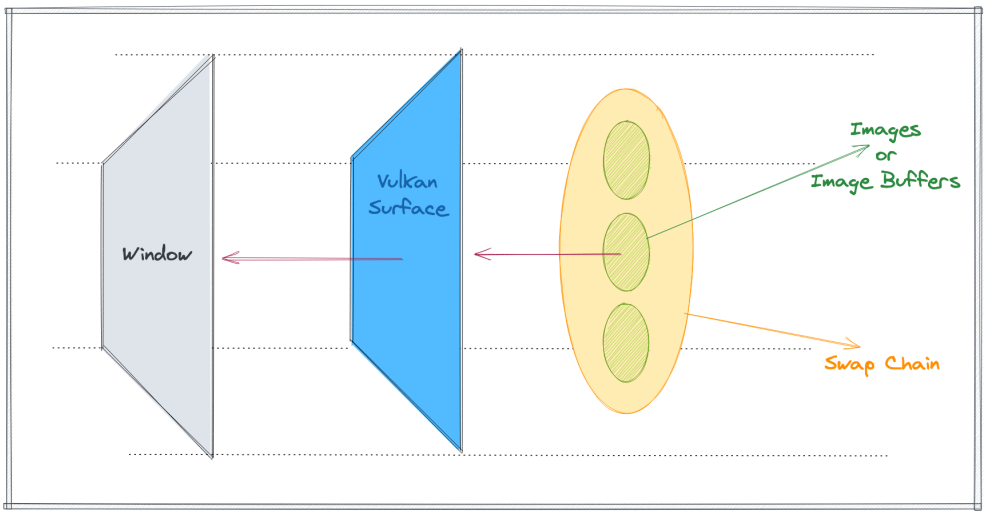


Figure 7.3.2.1 - Graphical representation of the SwapChains in Vulkan. Image from 'https://willofindie.com/' by Subroto Biswas.

Although not all graphics cards are capable of rendering images on the screen, either because they are designed for managing servers or for any other reason, so we have to check beforehand that the device we are working on is compatible.

7.3.3. Error codes

At the time of initializing any application, several errors may occur, either because the device with which you are working is not compatible with the graphical API, because it does not have enough memory or for multiple reasons, it is highly recommended to have all these errors controlled and give information about it.

In this case, I used the 'AppManager' class (see section 2), which controlled what errors could occur after initializing each Vulkan feature and wrote in the console what error it was and why.

```
// Error codes
typedef enum ErrorCode{
    kErrorCode_Ok                           = 0,
    kErrorCode_Window                       = -10,
    kErrorCode_Vulkan                       = -20,
    kErrorCode_PhysicalDeviceUnsupported    = -30,
    kErrorCode_PhysicalDevice               = -40,
    kErrorCode_WindowSurface                = -50,
    kErrorCode_SwapChain                    = -60,
    kErrorCode_LogicalDevice                = -70,
    kErrorCode_ImageViews                   = -80,
    kErrorCode_File                         = -90,
    kErrorCode_Shader                       = -100,
    kErrorCode_Pipeline                     = -110,
    kErrorCode_RenderPass                   = -120,
    kErrorCode_Framebuffers                 = -130,
    kErrorCode_CommandPool                  = -140,
    kErrorCode_CommandBuffers               = -150,
    kErrorCode_Semaphores                   = -160,
    kErrorCode_SubmitDraw                   = -170,
    kErrorCode_SyncObjects                  = -180,
    kErrorCode_VertexBuffer                 = -190,
    kErrorCode_Memory                       = -200,
    kErrorCode_DescriptorLayout             = -210,
    kErrorCode_NullContext                  = -220,
    kErrorCode_Textures                     = -230,
    kErrorCode_OBJ                          = -240,


    kErrorCode_Unknown                      = -500,

};
```

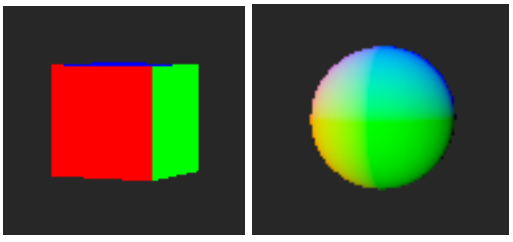
Figure 7.3.3.1 - Code extract from the demo. Small list of possible execution errors.

```
    // Initialization
    APPMANAGER.PrintErrors(CreateInstance());
    APPMANAGER.PrintErrors(CreateSurface());
    APPMANAGER.PrintErrors(CheckPhysicalDevice());
    APPMANAGER.PrintErrors(CreateLogicalDevice());
    APPMANAGER.PrintErrors(CreateSwapChain());
    APPMANAGER.PrintErrors(CreateImageViews());



    // Pipeline
    APPMANAGER.PrintErrors(CreateRenderPass());
    APPMANAGER.PrintErrors(CreateDescriptorSetLayout());
    APPMANAGER.PrintErrors(CreateGraphicsPipeline());
    APPMANAGER.PrintErrors(CreateCommandPool());
    APPMANAGER.PrintErrors(CreateDepthResources());
    APPMANAGER.PrintErrors(CreateFrameBuffers());
    APPMANAGER.PrintErrors(CreateTextures());
    APPMANAGER.PrintErrors(CreateTextureImageView());
    APPMANAGER.PrintErrors(CreateTextureSampler());
#ifdef LOAD_OBJ
    APPMANAGER.PrintErrors(LoadOBJ());
#endif
    APPMANAGER.resource_manager_.data_->CreateInternalVertexBuffers();
    APPMANAGER.resource_manager_.data_->CreateInternalIndexBuffers();
    APPMANAGER.PrintErrors(CreateUniformsBuffers());
    APPMANAGER.PrintErrors(CreateDescriptorPool());
    APPMANAGER.PrintErrors(CreateDescriptorSets());
    APPMANAGER.PrintErrors(CreateCommandBuffers());
    APPMANAGER.PrintErrors(CreateSyncObjects());
```

Figure 7.3.3.2 - Code extract from the demo. Thread of execution that checks if some initialization function returns an error.

7.3.4. Rendering

With all of this initialized, the first step was to draw a triangle. It is usually an initial step, since, if the triangle is painted on the screen, everything is going as expected, so from there making the jump to 3D is not particularly complicated. The step immediately after painting the triangle was to draw a cube or a sphere on the screen:



7.3.4.1. Development issues

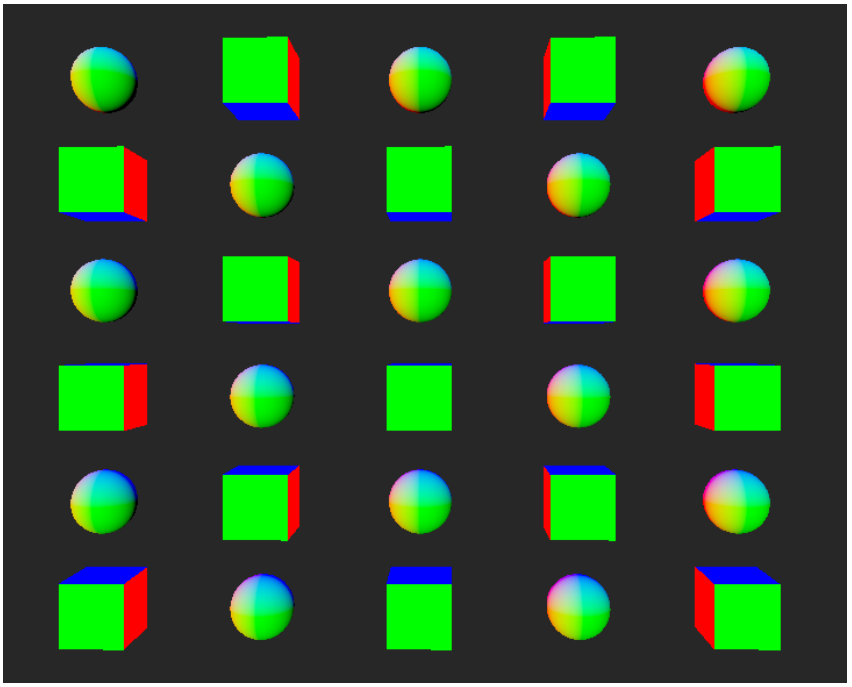
This is where things started to get 'complicated', since the tutorial I was following until now focused on painting a single thing on the screen (e.g one cube, one triangle, one sphere ...). As I had structured the code so far, getting to paint several things at the same time was quite complicated, which took me several days of work.

The issue here really is that there were hundreds of lines of code to get to paint only one object on the screen, and wanting to add more led to editing almost every part of the code and structuring it differently.

However, forcing me to rewrite much of the code was a positive point as well, as it helped me better understand how the API works.

To add several objects on the screen, it was necessary to change the number of DescriptorPools, the number of commands that are stored in the CommandBuffers to bind the vertices and indexes of each object, and, in addition, to updating the UniformBuffers every frame for each object.

```
// Uniform buffers : 1 for scene and 1 per object (scene and local matrices)
descriptorPoolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
descriptorPoolSizes[0].descriptorCount = 1 + static_cast<uint32_t>(game_objects_.size());

// Combined image samples : 1 per mesh texture
descriptorPoolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
descriptorPoolSizes[1].descriptorCount = static_cast<uint32_t>(game_objects_.size());

// Create the global descriptor pool
VkDescriptorPoolCreateInfo descriptorPoolCI = {};
descriptorPoolCI.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
descriptorPoolCI.poolSizeCount = static_cast<uint32_t>(descriptorPoolSizes.size());
descriptorPoolCI.pPoolSizes = descriptorPoolSizes.data();
// Max. number of descriptor sets that can be allocated from this pool (one per object)
descriptorPoolCI.maxSets = static_cast<uint32_t>(game_objects_.size());
```



From here, having several objects rendered on the screen, I wanted to add a geometry 'component' for each object in the scene to have its own geometry, such as a sphere or a cube. As a result of this, another problem arose, and that is that the CommandBuffers section had to be restructured again so that, when binding the vertices and indices, it took into account that not all objects would have the same geometry.

```
switch (go_to_draw.kind){
case GameObject::kGeometry_Cube: {
    VkBuffer vertexBuffers[] = { APPMANAGER.resource_manager_.data_->vertex_buffer[1] };
    VkDeviceSize offsets[] = { 0 };
    vkCmdBindVertexBuffers(command_buffers[i], 0, 1, vertexBuffers, offsets);
    vkCmdBindIndexBuffer(command_buffers[i],
        APPMANAGER.resource_manager_.data_->index_buffer[1], 0, VK_INDEX_TYPE_UINT32);
}
break;
case GameObject::kGeometry_Sphere: {
    auto test = APPMANAGER.resource_manager_.data_;
    VkBuffer vertexBuffers[] = { APPMANAGER.resource_manager_.data_->vertex_buffer[0] };
    VkDeviceSize offsets[] = { 0 };
    vkCmdBindVertexBuffers(command_buffers[i], 0, 1, vertexBuffers, offsets);
    vkCmdBindIndexBuffer(command_buffers[i],
        APPMANAGER.resource_manager_.data_->index_buffer[0], 0, VK_INDEX_TYPE_UINT32);
}
break;
default:
break;
}
```

7.3.4.2. Implementation of PBR

When adding a material with PBR, having already several objects on the screen, we have to create new buffers that store information that the graphic needs to process, such as a small buffer that stores how metallic the material is and what is the level of roughness , the lights buffer or the UBO (uniform buffer object). This is necessary, since when we are working at the shader level, and, above all, when implementing the PBR, at least one light is needed to be able to correctly visualize the result.

From here, we need to add functionalities to the PBR shaders, since the Fresnel implementation **(F)**, the Normal Distribution Function **(D)** and the Geometric Shadowing Function **(G)** are done there.

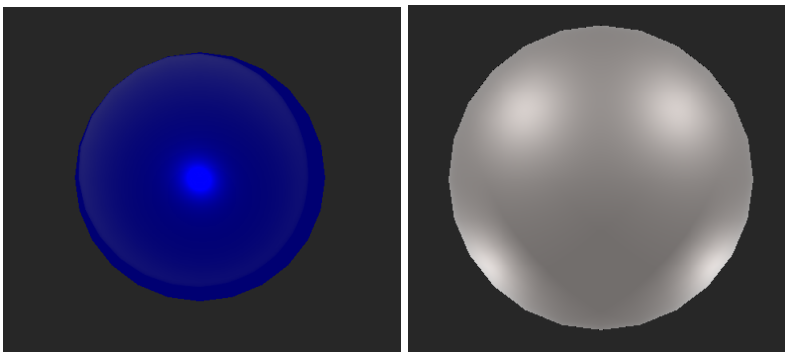
The implementation is done in the fragment shader, which works at the pixel level. The order of execution in this case is **(D)**, **(G)** and **(F)**, multiplied by each other and later by each of the lights applied to the object:

```
float rroughness = max(0.05, roughness);
// D = Normal distribution (Distribution of the microfacets)
float D = D_GGX(dotNH, roughness);
// G = Geometric shadowing term (Microfacets shadowing)
float G = G_SchlicksmithGGX(dotNL, dotNV, roughness);
// F = Fresnel factor (Reflectance depending on angle of incidence)
vec3 F = F_Schlick(dotNV, metallic);

vec3 spec = D * F * G / (4.0 * dotNL * dotNV);

color += spec * dotNL * lightColor;
```

Figure 7.3.4.2.1 - Code extract from the demo. BRDF calculation implemented in GLSL.

```
// Normal Distribution function --------------------------------------
float D_GGX(float dotNH, float roughness)
{
    float alpha = roughness * roughness;
    float alpha2 = alpha * alpha;
    float denom = dotNH * dotNH * (alpha2 - 1.0) + 1.0;
    return (alpha2)/(PI * denom*denom);
}

// Geometric Shadowing function --------------------------------------
float G_SchlicksmithGGX(float dotNL, float dotNV, float roughness)
{
    float r = (roughness + 1.0);
    float k = (r*r) / 8.0;
    float GL = dotNL / (dotNL * (1.0 - k) + k);
    float GV = dotNV / (dotNV * (1.0 - k) + k);
    return GL * GV;
}

// Fresnel function ----------------------------------------------------
vec3 F_Schlick(float cosTheta, float metallic)
{
    vec3 F0 = mix(vec3(0.04), materialcolor(), metallic); // * material.specular
    vec3 F = F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
    return F;
}
```

Figure 7.3.4.2.2 - Code extract from the demo. Implementation of Fresnel, Normal Distribution function and Geometric shadowing function in GLSL (see section 5.2.2).



With this correctly implemented, we obtain a basic PBR with a result that depends on the number of lights that affect the object. In this case, the image on the left shows a sphere that is being pointed at by a single light (Fresnel effect can also be seen around it), while the image on the right shows a sphere that is focused by four lights from different places.

7.3.5. VisualStudio and project structure

During development, using Visual Studio as a tool to debug the code has been very useful, as well as allowing me to structure the project more comfortably and handle the GIT tool, since it has a built-in terminal. The project consists of several folders:

- **bin**: folder containing the executable
- **build**: the folder will be created when we build the Visual Studio solution with the provided script. Contains VS solution and temporary files.
- **deps**: folder containing all additional dependencies (see section 7.2)
- **include**: contains all the headers of the created classes (see section 6)
- **src**: contains the main code files
- **tools**: folder containing the compilation tool 'GeNie'
- **compile_vs2019**: Visual Studio solution build script.

The process to compile is very simple, all you have to do is run the compilation script "compile_vs2019", this will generate the Visual Studio solution inside the 'build' folder (this folder will not exist until we compile previously).

Once inside the Visual Studio project, two projects will appear: 'mathmorra' and Physically Based Rendering, 'mathmorra' is a mathematical library developed by

Adrián Garvía Arrogante and by me a few years ago. The math library links to the PBR project, which is the only one that needs to be compiled to run.

7.4. Performance and testing

Unfortunately, due to the current situation with COVID-19, it has been impossible to perform performance and execution tests on other computers, so I have only been able to test it on my own computer.

The main characteristics of the computed in which the project has been tested are:

Model	MSI GF65 Thin 9SEXR
Memory	DDR4-2666, 16 GB
Processor	Intel(R) Core(TM) i7-9750H @ 2.60GHz
Graphics	Nvidia RTX 2060 6GB GDDR6

The project may not work properly on other computers, mainly because the graphics card is not the same. As the project has been developed taking into account a single graphics card, there may be data that is aligned in memory in buffers with a certain minimum size, so if you try to run with a computer with a graphics card that has a smaller buffer, it is possible that there are execution problems.

7.5. Dependencies

To get the demo to work properly, Visual Studio Community 2019 has been used, since it provides tools to be able to debug step by step and follow the execution of the program in order to locate any possible errors.

Apart from that, there are tools for debugging graphical APIs, such as RenderDoc, which provides a frame-by-frame breakdown of what is happening in the 3D scene.

7.6. Alternatives

The possibility of using the DirectX12 graphical API for the development of the demo was studied. It is, in turn, a collection of APIs for the development of applications and multimedia tasks. It is widely used for the development of video games on platforms such as Windows or Xbox.

It is, in a way, quite similar to Vulkan; but, nevertheless, since the latter is a more recent graphical API, it was decided to use it.

In addition, the idea of using a commercial graphics engine, such as Unity or Unreal Engine, was also considered. Of these two, Unreal Engine is more powerful when it comes to graphics techniques. Although in both cases you can opt for different graphic APIs, either Direct3D12, Vulkan and OpenGL.

If this option had been chosen, by not having to develop the demo from scratch, it would have given more accurate results when showing the PBR technique. In addition, it would have allowed the performance analysis in a simpler way, since both engines provide tools for this.

8. Critical evaluation

8.1. Introduction

This chapter analyzes the development of the project, ideas that ended up being discarded, analysis of results and the overall performance of the project. In addition, it also includes a small section on future work, possible project improvements, and common mistakes when developing with Vulkan.

8.2. Performance review

The idea of the project in the beginning was to develop a 3D scene with different geometries that would move and react to the music at runtime. The scene would have various geometries with PBR materials. However, a short time later I decided to focus more on digging a little deeper into this technique, so in the end I completely discarded that idea.

In the end, I opted to create a 3D scene in which there were geometries that would capture the physics-based rendering, so I used the Vulkan graphics API for this. With all this, it took me about four months to develop the practical part of the project, counting on learning Vulkan and understanding the PBR.

However, having not used Vulkan before, the base of the project is quite crude, having to modify many parts of the code to be able to add some new features, such as Normal Mapping or IBL. Although it did end up fulfilling the main objective, it got a bit cumbersome over time. The positive part of this is that, in the future, the organization of the code will be much better, allowing to modularize everything

so that it is not so impractical and clumsy to have to add more new features and functions.

8.3. Post-mortem report

During the development of the little demo, some ideas were discarded like implementing 3D Audio with SoLoud library, multi-thread design using px_sched, User interface using Dear ImGui.

In the end, they are tools that would not have contributed excessively to the project, since the main idea was to implement graphic techniques. With this in mind, adding audio or a user interface would not have been very interesting. The idea of implementing audio and a graphical interface arose from the idea of wanting to make a small 3D graphics engine. However, as the project developed, I chose to focus more on understanding and deepening Vulkan and how it works. Having previously worked with audio and user interfaces, it would not have been particularly difficult to introduce it into the project.

8.4. Future work

Having worked a little with Vulkan and having developed a first prototype, it would be interesting, in a way, to carry out the same project again but with an organization of someone who has already worked with tools like Vulkan. An interesting idea would be to implement a small 3D engine that is capable of creating and destroying objects on the screen, at the same time that it can change the material at runtime.

Another interesting idea would be to perform a performance analysis based on the number of objects on the screen, in order to draw conclusions based on data and try to optimize the techniques used to achieve greater efficiency.

9. Conclusion

The use of graphic techniques that seek to simulate a realistic environment is becoming more and more common. In the case of video games, there are graphics engines such as Unreal that are often used in the film industry to create 3D environments without the need to recreate the same environment on a recording set. The implementation of the PBR is an example of how to achieve that characteristic realism that surrounds modern video games.

Obviously there are many more techniques that have been developed and refined over the years, such as RayTracing, for example.

When it comes to evaluating the work, the most complex thing to develop has undoubtedly been the work environment with Vulkan as a graphical API. As the graphical techniques are based on mathematical models, it is not particularly difficult to implement them, the underlying problem is to be able to do it within an environment that can vary depending on which graphical API is being used, such as OpenGL, DirectX or Vulkan.

It is also true that when researching these types of techniques, the fact that the mathematical process behind is not always understood can be complicated. It depends on the depth to which you want to go. Demonstrating a mathematical process is not the same as simply using a formula. In a formula you can understand what variables it takes into account and what operations are performed, but that does not mean that you know how this mathematical model has been shown to work.

10. References

10.1. Vulkan

- Khronos Group. (n.d.). Vulkan Tutorial. Vulkan-Tutorial.
    - https://vulkan-tutorial.com/

The Vulkan tutorial web page will help to improve my knowledge and research on the graphical API.

- Singh, P. (2016). Learning Vulkan (1st ed., Vol. 1). Packt Publishing.
- K. (2017). Introduction to Computer Graphics and the Vulkan API (1st ed., Vol. 1). Van Haren Publishing.

These two books will present an overview of the Vulkan API and its distinct features compared to its predecessor, OpenGL API. By using this book, it is intended to learn the fundamentals of Vulkan, understand the Vulkan application and get started with the Vulkan programming model.

10.2. PBR (Physically Based Rendering)

- Google, Guy, R., & Agopian, M. (2018, August 3). Filament. GitHub.
    - https://github.com/google/filament

Extract some examples of PBR in shader and their respective calculations.

- de Vries, J. (2014, June). LearnOpenGL - Theory. LearnOpenGL.
    - https://learnopengl.com/PBR/Theory

This web page will help me better understand the physics behind PBR techniques, the theory behind this technology, and some practical examples showing the potential of this type of rendering.

- Pharr, M., & Humphreys, G. (2010). Physically Based Rendering (2nd ed.). Elsevier Gezondheidszorg.
    - https://learning.oreilly.com/library/view/physically-based-rendering/9780123750792/OEBPS/9780123785800_cover.htm

Understanding how PBR works and provides a more accurate representation of materials and how they interact with light when compared to traditional real-time models. Extract some examples of PBR in shader and their respective calculations.

- Haniel Ferreira, L. (2017, November). Physically Based Real-Time Raytracing.
    - https://linux.ime.usp.br/~luantorres/mac0499/monography.pdf

Investigate and understand the interaction between the Vulkan graphics API and PBR techniques.