University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

# Physically Based Real-Time Raytracing

Luan Haniel Ferreira Sanches Torres

Advisor: Prof. Dr. Marcel P. Jackowski

São Paulo
November of 2017

# Abstract

Ever since the early days of computer history, image generation have been one of the largest fields of interest for both industry and academia, therefore some of the biggest advances of computer science have been made in computer graphics. While technology has evolved to a point where it's possible to produce images that have almost no difference to real life photos, the generation such graphics still requires a big amount of time, which hints a desire of speeding up the process of creating photorealistic images. The objective here was to develop a renderer that simulate light transport physics in real time, or in other words, generating photorealistic images really fast. To reach those goals, the Raytracing algorithm was implemented using the Vulkan API and Physically Based Rendering (PBR) techniques, which resulted in high quality images being produced in milliseconds.

**Keywords:** PBR, physically based rendering, raytracing, real time rendering, Vulkan

# Contents

# Chapter 1

# Introduction

Rendering is nothing more than the art of making math visible. That phrase tell much about computer graphics as a whole, since it is simple, straightforward and anyone can understand at it's core, but being simple doesn't always means easy.

Following those characteristics very closely is one of the first algorithms in computer graphics, the *Raytracing*. It can be used for photorealistic image generation, since it describes the light behavior in the real world, but is very expensive to calculate, making it the central algorithm of this document.

The objective here is to develop a *Physically Based Real-Time Raytracer*, a renderer capable of generating physically accurate images in real time, while using the raytracing algorithm. Taking a closer look at this title can help in the future:

- Raytracer[1]- that's the algorithm that commands how to get the scene data to an image; it's job is to determine which parts of the scene contribute to each pixel by casting rays that intersect with objects over and over again.
- Real-Time - that means it's generating images really fast, to the point where one could interact with the environment and see the changes take part; in other words, each image is generated in a blink of an eye;
- Physically Based - although the raytracing algorithm simulates the light behavior, things are not so simple in real life: light has energy and interacts differently with different materials, bouncing all over until it reaches the eyes of an observer. Modeling something that considers all the involved variables is almost impossible, therefore techniques that are based on the laws of physics are used to resemble an accurate light behavior.

[1]The central algorithm used is actually in a middle ground between the classic raytracing and what is called *Path Tracer*, that is a variant of raytracing. Both algorithms work by casting rays against the scene geometry, but while the original raytracing algorithm traces rays from the contact point to light sources,

## 1.1 About the technology

Since speed is a core part of this implementation, being able to control and fine tune each part of the program execution is a must. For that reason, a permissive graphics API was needed, and Vulkan, the OpenGL "successor", fulfilled this requirement. This API removes almost all the work from the hands of a hardware driver and put it in the hands of the application programmer, giving almost full control over the graphics card (GPU). That control comes with a much more explicit implementation and increased responsibility for the programmer, requiring a deep knowledge of the problem at hand, the inner workings of a GPU, and computer graphics in general.

Besides the control, Vulkan also provides a common ground between the classic graphics pipeline and the GPU general parallelism, those being everything needed by this renderer, since raytracing is a highly parallelizable task and displaying images on the screen is the best way to evaluate a real time application.

## 1.2 Text structure

This text is divided in two parts describing the background needed to understand the system implementation, followed by the conclusion:

- Part 1: Theory - here some of the involved concepts like the raytracing algorithm, PBR theory, and common computer graphics aspects are explained in more depth.
- Part 2: Implementation - here the most important parts of the implementation are described; it's shown how the code and data is organized, how the base renderer works, and how it all come together in the end.

determining if that point is lit, a Path Tracer cast those secondary rays in random directions recursively, until one of those rays reach a light source.

# Chapter 2

# Theory

Even though computer graphics (CG) is one of the largest fields in computer science, most people are afraid of even getting close to it. A common belief has been created where only the most talented programmers can dare to approach such arcane art and, while there is a lot of math and optimization techniques, the truth is that most concepts are easy to understand.

With that in mind, this chapter is designed to explain some basic concepts used in the renderer while assuming almost no knowledge in CG. That said, even if the reader do have a background in computer graphics, it may be worth taking a look at those topics as they describe how the implementation will proceed afterwards.

## 2.1 Raytracing

When talking about computer graphics, most people think about pretty images on a screen, like in a video game. Usually those images are generated from various types of data going through a set of steps (pipeline) that produce what is seen on the screen.

The way of generating images used by this renderer defines a raytracer pipeline. But before talking about this topic, it's important to have a basic understanding of how graphics are generated in the far most common *graphics pipeline*, as almost all graphics cards are designed to make the best use of that model.

### 2.1.1 Classic Rendering Pipeline

The graphics pipeline works on sets of geometric data, most commonly sets of triangles, submitted to the GPU as vertices, points in space defining position and some other attributes. Inside the pipeline, the data in those vertices can be manipulated in many ways, being repositioned, multiplied and even generating new geometry data. The important part

is that, after those transformations, a process called rasterization takes part. That step produces fragments, things that are basically pixels (tiny little pieces of an image) with some more information. Fragments are then submitted to the (usually) most expensive part of the graphics pipeline, where the final pixel color is calculated.

The rasterization process is important because of the way it works: given the geometry info, the rasterizer determines which pixels that geometry can affect. But more than that, the rasterizer can determine the info residing inside a shape for free, which is a big thing since only the information about vertices is passed to the pipeline.

Having a basic understanding of the graphics pipeline makes it easier to see why raytracing is a slow process.

### 2.1.2 How raytracing works

The raytracing algorithm describes the behavior of light in a intuitive way, so it's useful to have a simple understanding of how light travels through the real world.

Say someone is in a room looking through a window, and outside is a field with a tree and a blue sky. Obviously that person cannot see what is behind that tree, but why? The reason is because there is no light rays coming from behind the tree that reach the observer's eyes. Everything visible from a particular point in a scene is a result of light being *emitted* (like a candle flame) or *reflected* by an object.

Back to the window, the sun is *casting rays* of light that reach the scene objects, like the tree, and some of that light is reflected in the direction of the observer, making the tree visible. All that is possible because light travels in a straight direction, and after hitting something, part of that light is reflected. The color that is seen when the ray reaches the observer is a result of the light object interaction, where part of the light is absorbed.

That's pretty much what the raytracing algorithm does, but there is a catch: when a light ray is casted from a light source, it probably will never reach the observer's eyes. Light is reflected in many directions when it reaches an object, so only a very small part of those rays will contribute to what someone see in a scene.

The solution here is to follow the reverse path: cast rays from the observer to the scene. That's possible because of a pretty useful property about reflections: if the incoming ray is swapped by the outgoing ray with inverse direction, the new outgoing ray will be the original incoming ray with inverse direction. With that, the algorithm just work with rays that are

visible, bypassing all those rays that never reach the observer.

Putting it all together in a simple form of the algorithm, what raytracing does is cast rays from each pixel in the image, intersecting those rays with the scene geometry; if an intersection was found, a color is calculated for that point, and then more rays, the reflected ones, are casted into the scene, now starting from the contact point. This process repeats until a given limit of ray bounces, but as each new ray can contribute to that pixel's color, this limit usually have a big impact on the final image quality.

### 2.1.3 Classic Rendering Pipeline vs Raytracing

Understanding both raytracing and the graphics pipeline makes it easy to see why the latter is faster: while both can operate over the same data and use the same equations to calculate a pixel color, the work of finding that pixel is much more expensive on a raytracer.

With the raytracer, it's necessary to make several ray intersection tests for each ray exiting the pixel, finding which piece of geometry (if any) is contributing to the final color, and then which point of that geometry will be used in the calculations. All that only for primary rays; the whole process is repeated for each batch of reflected rays.

With rasterization in the graphics pipeline, the pixels that will be affected are easily determined, as the process runs in a per geometry basis, so there is no cost in finding which points in the scene contribute to the final pixel color.

That said, rasterization is not perfect; many desired visual effects like transparency and shadows are harder to obtain using that pipeline, while coming with little effort when using raytracing. Not only that, but since raytracing works by simulating the light behavior, the resulting image usually have a much higher quality, presenting visual effects like soft shadows, global illumination and reflections without changing one line of the algorithm.

### 2.1.4 Ray intersection tests

When using a raytracer, the main job of the algorithm is to find which part of the scene contribute to a pixel's color. For that, it's necessary to define what is a ray and how it can intersect scene geometry. Notice that some math fundamentals may be necessary to fully understand this section.

A ray $r$ is a line segment that have an origin and a direction, so it's defined by:

$$r := o + td$$

$$o, d \in \mathbf{R}^3, t \in \mathbf{R}$$

Where

- $o$ is the origin; a point in space
- $d$ is the direction; a vector
- $t$ is a scalar used to uniquely represent any point in the ray's line

For the renderer described by this document, the geometry used by the scene is composed of spheres and planes, defined below.

*Spheres* are sets of points that reside in a fixed distance from an origin. For easier calculations, this origin is treated as the vector (0, 0, 0):

$$s := |p| - r = 0$$

$$p \in \mathbf{R}^3, r \in \mathbf{R}$$

Where

- $p$ is a point in the sphere surface
- $|p|$ is the length of vector p; the distance from p to the origin
- $r$ is the sphere radius

The intersection between a ray and a sphere can be calculated by using the ray equation as the input for the sphere equation, which results in a quadratic equation with 0, 1 or 2 real values for t representing where the ray intersects the sphere:

$$t^2(d \cdot d) + 2t(o \cdot d) + (o \cdot o - r) = 0$$

$$\Delta = 4(o \cdot d)^2 - 4(d \cdot d)(o \cdot o - r)$$

$$t = \frac{-(o \cdot d) \pm \sqrt{\Delta}}{(d \cdot d)}$$

Where the $\cdot$ represents the dot product and $\Delta$ determines in how many places the ray intersects the sphere:

- If $\Delta < 0$, then the ray does not intersect the sphere
- If $\Delta = 0$, then the ray intersects the sphere in exactly one point[1]
- If $\Delta > 0$, then the ray intersects the sphere in two points

Some simplifications can be made, since d usually is a unity vector, which means the dot product $d \cdot d$ always equals 1:

$$\Delta = 4(o \cdot d)^2 - 4(o \cdot o - r)$$

$$t = -(o \cdot d) \pm \sqrt{\Delta}$$

*Planes* are sets of vectors generated from a point that are perpendicular to a normal vector; in other words, given an origin for a plane $\pi$ and a normal vector, every point in that plane respects the following equation:

$$\pi := (p - c) \cdot n = 0$$

$$p, c, n \in \mathbf{R}^3$$

Where

- $p$ is a point in the plane
- $c$ is the origin of the plane
- $n$ is the vector normal to the plane

Notice that any point in that plane can be used as the origin.

The intersection for planes follows the same process as with spheres: it's just a matter of using the ray formula inside the plane equation:

$$(o + td - c) \cdot n = 0$$

$$t = \frac{(o - c) \cdot n}{(d \cdot n)}$$

[1]Even though when $\Delta$ equals 0 there is a collision, the system will not consider it as one. The reason is that when the program is calculating the color contribution of a point, the cosine of the angle between the ray and the normal at collision point is used, and in the case where $\Delta$ equals 0 this angle equals $\frac{\pi}{2}$, so it's cosine equals 0 and can be discarded.

Here, if $d \cdot n$ equals 0, then the ray is parallel to the plane and thus there's no intersection[2].

For both spheres and planes, if an intersection results in t<0, then the contact point found is behind the ray and can be discarded.

## 2.2 Physically Based Rendering (PBR)

Even though raytracing can simulate how light moves, there are some neat physical properties that are usually overlooked by the algorithm's most basic form.

When dealing with light in the real world, each interaction between a ray and the environment is modified by a bunch of variables, resulting in very different visuals. Those variables are what define from which *material* an object is made of.

The goal here is to explain the properties of light and materials used by the renderer to produce physically accurate images.

Now, simulating all physical properties of light is near impossible, so a set of techniques was developed in order to make the rendering of physically plausible scenes relatively fast. It works by *basing* it's calculations at the theory behind light interactions, cutting corners in some places for speed, which results in a *Physically Based* Rendering set of techniques, or PBR for short.

Bear in mind that most topics covered by this section barely touches the surface of what is behind PBR, with only the most crucial parts being exposed in order to explain of how the renderer calculates colors. Besides that, it's recommended to follow the topics in the presented order, as later topics use some previously defined concepts.

This section is based on Chapter 7 of Akenine-Möller *et al.* (2008), the introduction to PBR by Hoffman (2015), and the PBR series by de Vries.

### 2.2.1 Light and Color

As said in Section 2.1.2, an observer can only see something if light is reaching it's eyes, be it reflected or emitted from some object. For that reason, the following paragraphs try to explain what light is and how it's related to colors, consequently being a fundamental part of understanding how PBR works.

[2]There is the case where $o$ belongs to the plane, so it's technically a collision, but it's not considered as one. The reason is the same used when discarding ray-sphere collisions where $\Delta$ equals 0.

### Radiometry

While light can be treated as a ray, which is the whole principle behind raytracing, there is something more behind it. Keeping it simple, light is a kind of electromagnetic radiation, meaning it's a bunch of *photons* that move around caring energy.

For the purposes needed by this document, it's not so important to know what photons are as it is to know how they behave, as this item focus in properties of light needed to calculate colors. That said, one aspect that cannot be overlooked is that photons are both particles and waves.

While it's safe to treat photons as only being particles (as is the case when using light as rays), knowing their wavelength (or frequency), a wave property, is needed for future calculations.

Frequency, measured in Hertz, and wavelength, measured in meters, are related by:

$$v = \frac{c}{\lambda}$$

$$\lambda = \frac{c}{v}$$

Where

- $v$ is the wave frequency
- $\lambda$ is the wavelength
- $c$ is the speed of light

The photon frequency is what define if something is visible to human eyes, since it's only possible to visually perceive wavelengths ranging approximately from 380 to 780 nanometers, the so called *visible spectrum*, where lower frequencies waves turns to be red and higher ones blue.

Another aspect of frequency is its direct relationship with the photon energy, the basic unity in radiometry, measured in Joules. That relationship is given by:

$$Q = hv$$

Where

- $Q$ is the photon energy
- $h$ is the Planck's constant

From that, several quantities are derived, but two are more important for the renderer implementation: irradiance and radiance.

Irradiance, measured in watts per square meters, describes how much photon energy is passing through an area per second. That's what is used when calculating how much light is coming in and going out of an object.

Radiance, measured as watts per steradians per square meters, is the aspect of light perceived by the eye, measuring how much illumination there is in a ray (or more specifically, around a solid angle in a given direction), and therefore directly related to the color produced by the renderer.

For the sake of clarity, solid angles, measured by steradians, can be though as being 3D angles: while radians measure 2D angles that can be represented as part of a circumference, a steradian measure 3D region that can be represented as part of a sphere surface. That said, the renderer simplifies this by simulating a very small solid angle that can be treated as a ray.

Notice that irradiance can be derived from radiance since radiance is irradiance per steradian. That is common other radiometric quantities, making radiance the most relevant quantity for calculations. Speaking of radiometric quantities, measuring electromagnetic radiation is exactly the job of *radiometry*.

**Colorimetry**

For calculations, radiance is the ideal quantity to be worked. But the way humans perceive light has to be considered in order understand how colors are represented when generating images.

Colorimetry is the field responsible for describing and evaluating how humans perceive light, or colors to be more specific.

In the real world, every light is composed of a set of photons. The distribution of wavelengths from those photons forms the *spectrum* of a given light, and that basically defines which color is perceived by the eyes, but there is a catch.

The human eye perceives light because of three different types of receptors, each one "reading"a different range of wavelengths. The catch is that light's spectrum is composed by an infinity of wavelengths distributed in a way that (appears to) result in a color, but with that limitation in humans eyes, many different spectra can be perceived as the same color.

For that reason, any visible spectrum can be represented by only three values[3]and, not by coincidence, each pixel in a monitor is represented using three values: red, green and blue. Keep in mind though that each monitor have limitations and different configurations, so it's almost impossible to produce images that will look the same regardless of the hardware.

### 2.2.2 Surfaces

Most interesting things in life happen when there is some sort of interaction, and with light that interaction happens when a ray hits an object's *surface*. This section is the core part of PBR, explaining what happens when light hits something and how that is used by the renderer when dealing with all sorts of materials.

#### Microfacet theory

Every object is made of some material that has characteristic physical properties. One very common set of properties to most people defines how each object feels when touched, e.g. how fast something changes its temperature, or how slippery it is. Those characteristics are perceived from the object's *surface*, the place of contact when light reaches an object.

Just for a second, imagine a metal knife with a unpolished wooden handle sitting on a desk. When light hits that knife, the metal part gets some bright spots reflecting the light's shape. Looking at the wooden part, it's possible to tell the wood color and some details like the wood grains, but no bright spots can be seen. One of the reasons for that behavior is that the wood's surface is *rougher* than metal's surface, or equivalently the metal's surface is *smoother* than wood's surface.

Taking a closer look at both wood and metal surfaces, it's possible to see something like tiny patches (flat surfaces) connected to each other. The difference is that in the wood those patches are messy, having a big variation in their orientation when compared to neighbor patches. Looking at the metal, the patches are more behaved, with little variations in neighboring orientations.

To see how that affects the visual appearance of a material, imagine that each of those patches is a mirror. The difference in alignment then is visible when light reaches the surface, since it's reflected based on the patches orientation: when the patches are aligned, more of the reflected light will follow the same path, resulting in a *mirror*-like behavior, and when

[3]These three values are hypothetic, proposed by the CIE (*Commission Internationale d'Eclairage*) in order to represent any visible spectrum. In real life, when using three specific spectra (light rays) with different intensities, sometimes it's not possible to represent a color, as the combination of intensities resulting in that color can include negative values, meaning that the light rays with such values would be added to the desired color instead of being added with the other light rays.

the patches are not so aligned, light is *scattered* all around.

This idea is exactly what the *Microfacet Theory* describes, receiving that name because each of those patches is actually in a microscopic scale, bigger than visible wavelength, but still smaller than a pixel. The patches are treated like perfect mirrors and receive the name microfacets, being responsible for how light will behave based on the surface's roughness.

If the reader still finds the concept of a surface's roughness confusing, it may help to think of some materials or take a look around, observing that when things feel rougher on the touch, they are usually not shiny.

Although treating a surface like a mirror on the microscopic level can describe how reflections work in different materials, when light hits an object it's actually divided in two parts: the immediately reflected part, called *specular* reflection, and the refracted part, that needs some explanation.

As said in the previous section, light transports energy, a quantity that according to the laws of physics is always conserved, implying that light leaving a surface can never have more energy than light coming to that surface. Naturally, as light is divided when reaching a surface, it's energy is divided between the reflected and refracted parts.

When light is refracted, it starts to bounce around under the surface, having part of its energy absorbed (being transformed in heat or some other kind of energy). The thing is that the non-absorbed part of light keeps bouncing, and since it has less energy, its color starts to change. Eventually some of the bouncing rays *may* find their way out of the surface, resulting in the called *diffuse* light color of that material. That behavior of light when bouncing under a surface and then find its way out is called *Subsurface Scattering*, and is responsible for the characteristic color of most materials.

The result from light-surface interaction then comes from the combination of the specular and diffuse components turning in some color output, or in terms that will help soon, outgoing radiance.

For rendering purposes, since most part of diffuse light is scattered in a region smaller than the pixel being calculated, it is possible to simplify the subsurface scattering by treating all the outgoing rays as leaving the surface from a single point.

### BRDFs

Although light-surface interaction results in colors, an observer can perceive a scene in completely different ways just by moving the light sources. With that, it's safe to assume that the way a surface outputs radiance (i.e. how it reflects light) depends on where both the observer and light sources are located in relation to the object being observed. That dependency is actually a function called *Bidirectional Reflectance Distribution Function*, or *BRDF* for short.

BRDFs are what make PBR shine. They work by taking two inputs: the direction of light coming to a point in the surface and the direction from that point to the observer. Their job is to tell how much radiance should go in the viewer direction based on how much irradiance is coming from the light source. The resulting formula used, that assumes no area lights in the scene, is given by:

$$f(l, v) = \frac{L(v)}{E \max(0, \cos\theta)}$$

$$l, v \in \mathbf{R}^3, \theta \in [0, \pi]$$

Where

- $l$ is the direction to the light
- $v$ is the direction to the observer
- $L(v)$ is the outgoing radiance in the observer's direction
- $E$ is the incoming irradiance of the light source in a plane perpendicular to $l$
- $\theta$ is the angle between $l$ and the surface's normal at the contact point

With that, it's easy to see how the BRDF can be used to calculate the outgoing radiance of a surface in a scene:

$$L(v) = f(l, v) E \max(0, \cos\theta)$$

There are many different BRDFs used in computer graphics, but the one chosen for this implementation is called *Cook-Torrance BRDF*, as it is one of the most used in real-time rendering applications.

The Cook-Torrance BRDF consists of two parts, one for dealing with the diffuse component and the other for the specular component. The first, dealing with light that is refracted, uses the *Lambertian reflectance* function multiplied by a coefficient determining how much of incoming light goes to the diffuse component. The second, dealing with reflected light, uses

the Cook-Torrance specular BRDF, also multiplied by a coefficient, this time representing the reflected amount of light.

Putting both together results in:

$$f(l, v) = f_{Lambertian}(l, v)k_d + f_{Cook-Torrance}(l, v)k_s$$

Where

- $k_s$ is the coefficient for specular contribution
- $k_d$ is the coefficient for diffuse contribution

The Lambertian reflectance function distributes radiance equally in a hemisphere, and when used by the renderer, it gets the albedo (natural color) of a surface to calculate the diffuse (illuminated color) component as follows:

$$f_{Lambertian}(l, v) = \frac{c}{\pi}$$

Where

- $c$ is the albedo color

If the reader have some experience with computer graphics, that equally distributed reflection may resemble something like the Blinn-Phong model, but the key here is the division by $pi$, used to guarantee the energy conservation in future calculations.

The Cook-Torrance specular BRDF is where things get really interesting. That function is defined by:

$$f_{Cook-Torrance}(l, v) = \frac{F(l, h)G(l, v, h)D(h)}{4(n \cdot l)(n \cdot v)}$$

$$n, h \in \mathbf{R}^3$$

Where

- $h$ is the half vector; it's the vector that is halfway between $l$ and $v$
- $n$ is the normal vector of the surface
- $F(l, h)$ is the Fresnel Reflectance Equation
- $G(l, v, h)$ is the Geometry Function
- $D(h)$ is the Normal Distribution Function

Each of those functions deserve a dedicated explanation as they are basically used as presented inside the renderer, but before that, here is how the Cook-Torrance BRDF looks like:

$$f(l, v) = \frac{c}{\pi}k_d + \frac{F(l, h)G(l, v, n)D(h)}{4(n \cdot l)(n \cdot v)}k_s$$

**Fresnel Reflectance Equation**

The Fresnel Reflectance Equation describes the ratio between reflected and refracted light exiting the surface depending on both incoming light direction and surface normal. The effects of that equation can easily be seen in the real world, for example, with water: when looking down at a lake, it's possible to see through the water, but when observing it from some angle, like looking near the opposite lake shore, the water acts like a mirror.

With that equation, each material has a characteristic behavior depending on the incident light angle, but all of them rapidly start to act like mirrors for angles greater than ($\approx$) 70°. For angles lower than that there is not much variation, so the amount of light reflected at the incident angle of 0° is used for calculations, being represented by $F_0$.

Besides that, while describing the behavior of each kind of material escapes the scope of this document, it's important to say that usually materials are divided into two groups[4]: metals and dielectrics.

Metals are conductors: they are highly reflective, usually with $F_0 > 0.5$ in all spectra, have characteristic specular colors, and have no diffuse color, as they absorb all the refracted light. Iron, gold, and silver are some examples of materials that fall into this category.

Dielectrics are non-conductors: they are not very reflective, usually with $F_0 \approx 0.04$, have dark, non-colored (gray scale) specular colors, and have diffuse color, coming from the subsurface scattering. Plastic, chalk, and rubber are some examples that fall into this category.

Since dielectrics do not have much variation in the $F_0$, with almost all of them having $F_0 \in [0.02, 0.2]$ and more commonly with $F_0 \in [0.04, 0.045]$, it's normal to use a fixed value of $F_0 = 0.04$ for all materials in that group. That, combined with metals not having diffuse colors, form a fundamental part of what is called the *Metallic Workflow*, commonly used in many modern renderers.

The most common formula used in rendering when dealing with the Fresnel Reflectance Equation is the Schilick Approximation. It gives fairly accurate values while being cheap to calculate, making it ideal for this implementation. The formula is given by:

$$F_{Schilick}(F_0, l, n) = F_0 + (1 - F_0)(1 - (l \cdot n))$$

It's easy to see that the amount of reflection goes up as the angle between the light and surface normal gets closer to 90°, at which point the equation converges to 1, i. e. all

[4]There is third group for semiconductors, but as they have somewhat weird behaviors and are rarely used in rendering, it's common to not deal with them at all

incoming light is reflected.

### Normal Distribution Function

The Normal Distribution Function, or NDF for short, is responsible for describing the distribution of specular light exiting a surface in relation to the half vector. To understand what that means, it's easier to know how reflections are calculated.

When a reflected vector resulting from a ray reaching a surface is calculated, it's done so by using the surface normal at that point with the following formula:

$$r = n(l \cdot n) - 2l$$

That function results in the reflected vector $r$ but, although not so costly, it's possible to make it better by using one simple relation: when the view vector is aligned with the reflected vector, the normal vector is aligned with the half vector. With that, calculations depending on both reflected and view vectors can generally be replaced by equations using both normal and half vectors. That is better because calculating the half vector is cheaper than calculating the reflected one, plus the half vector is used as input by the NDF.

This substitution is not perfect, as the difference between the normal and half vectors does not escalates linearly in relation to the difference between the reflected and view vectors, but it's good enough for rendering purposes.

Using that substitution, the NDF works with basis in the microfacet theory: only microfacets that have their normals aligned with a given half vector will contribute to the reflected color. What the NDF returns then is the concentration of how many microfacets are oriented in that direction, determining the appearance of highlights in a surface.

There are several NDFs used in computer graphics, each one defining some sort of shapes for highlights, but the one used here is the Trowbridge-Reitz GGX. This function produces a smooth, fast, and long falloff in the highlight starting at its center. It's definition is given by:

$$D_{Trowbridge-ReitzGGX}(n, h, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2}$$
$$\alpha \in \mathbf{R}$$

Where

- $\alpha$ is the roughness of the surface

When $\alpha$ is low, it represents a smooth surface, and the equation produces a sharp bright highlight that shows a high concentration of light rays. As $\alpha$ increases, representing a rougher

surface, that highlight gets bigger, and with energy conservation taking part, it also gets less bright.

#### Geometry Function

The Geometry Function is responsible for describing how many microfacets actually contribute to reflected color in the observer's direction, as depending on the surface roughness, some energy of incoming rays may be lost.

Looking closely at the surface, no material has its microfacets perfectly aligned, meaning that any surface has imperfections in geometry that can cause loss of energy. Common cases of that loss are caused by small gaps that capture light or slopes overshadowing microfacets that could contribute to the reflection according to a NDF.

There are some functions to describe that effect, but the one chosen has been shown to be both physically realistic and mathematically valid, being known as the Smith Function:

$$G_{Smith}(l, v, n, \alpha) = G_{SchlickGGX}(l, n, \alpha)G_{SchlickGGX}(v, n, \alpha)$$

$$G_{SchlickGGX}(l, n, \alpha) = \frac{l \cdot n}{(l \cdot n)(1 - g(\alpha)) + g(\alpha)}$$

$$g(\alpha) = \frac{(\alpha + 1)^2}{8}$$

Where

- $G_{SchlickGGX}$ is the geometry function approximation known as Schlick-GGX. That is used for evaluating the geometry contribution in a single direction based on the surface normal
- $g$ is a transformation of the roughness that depends on the implementation intent

By increasing the roughness, more energy is lost due to geometry imperfections. Besides, both view and incoming light directions have to be accounted by the function, as light may be obstructed both when reaching and leaving the surface.

### 2.2.3 The Rendering Equation

In computer graphics, the process of obtaining the final color of a pixel is described by the Rendering Equation, responsible for telling how much radiance is leaving a surface given how much radiance is coming to it. Several forms of that equation were used along the years, but to describe how things work, the following will be used:

$$L_o(p, v) = L_e(p, v) + \int_\Omega f(l, v) * L_o(r(p, l), -l)cos\theta d\omega$$

Where

- $L_o(p, v)$ is the outgoing radiance in direction $v$ coming from point $p$
- $L_e(p, v)$ is the radiance emitted by the surface in direction $v$ coming from point $p$
- $\Omega$ is the hemisphere perpendicular to the surface at $p$
- $f(l, v)$ is the BRDF of the surface
- $r(p, l)$ is a point obtained from casting a ray with origin $p$ and direction $l$
- $L_o(r(p, l), -l)$ is the outgoing radiance from some point $r(p, l)$ in $-l$ direction
- $\theta$ is the angle between $n$ and $l$
- $d\omega$ are small parts of the $\Omega$ hemisphere, meaning they are small solid angles

The $*$ represents a per component multiplication, as the quantities worked by the equation are represented as 3D vectors that can be tough of as being RGB colors.

This function is explained as the sum of outgoing light of a surface. It combines light emitted with light reflected, the later being calculated by a integral that runs through all possible $l$ in the $\Omega$ hemisphere above $p$, i.e. the reflected light is a combination of light incoming from all directions.

**The Reflectance Equation**

While the Rendering Equation describes how color is calculated for a point in a surface, the scenes worked by the renderer usually do not have materials that emit light, so the equation can be simplified. Besides that, casting rays outside the point is a heavy task computationally, so real-time renderers usually use the Reflectance Equation, a simpler, more restricted case of the Rendering Equation is used and is defined by:

$$L_o(p, v) = \int_{\Omega} f(l, v) * L_i(p, l) cos\theta d\omega$$

Where

- $L_i(p, l)$ is the radiance coming to $p$ from direction $l$

This function dispenses the need of casting rays to find the incoming radiance, but since this renderer uses raytracing as the core algorithm, the calculation of $r(p, l)$ happens as part of the process, just not so heavily as the Rendering Equation proposes.

One of the best things about that equation, and PBR for that matter, is that it yields physically accurate results regardless of the scene lighting, helping both artists and programmers in many ways. For example, it eliminates the often needed rework when reusing props and make every scene coherent and visually plausible, even for stylized work as seen in cartoon movies.

## 2.3 Technical aspects

Before getting to the implementation, some useful terms and concepts must be explained in order to justify some decisions made in the rendering engine. This section then aims to show part of the inner workings of what is involved with computer graphics.

### 2.3.1 Vulkan

From the beginnings of computer graphics, many rendering techniques were developed, but what helped the advance of the field was the advance of hardware, i.e dedicated graphics cards, or GPUs.

As said in section 2.1, images are produced from data going through a pipeline defined by several steps, each one producing input for the next. That concept is not new to the industry, as long ago production lines were invented, but this model is really useful when dealing with repetitive tasks over a predefined set of inputs, being perfect for computer graphics and therefore being perfect for GPUs.

In the early days that pipeline was totally fixed, with only some variables being able to change, and the inner workings of the GPU were defined by the manufacturers and their drivers. Time passed and some parts of the classic rendering pipeline became more flexible, allowing programmers to have better control over what was going on the graphics card. One problem though was that much of the GPUs potential was being wasted.

Since each pixel is totally independent of other pixels calculations, the hardware developed for graphics cards are highly parallelized, using the *Single Instruction Multiple Data* (SIMD) model to proceed through work. That model allows to much more things than only pixel computations, being useful to many areas, so a new way to use the graphics card has emerged: the *General GPU programming*, or GPGPU for short.

While many applications were improved by this model, programmers were still lacking control over some useful GPU properties. More granular control, combined with an easy integration between GPGPU and the classic rendering pipeline is what Vulkan provides.

By having a very thin layer of drivers between the hardware and the programmer, Vulkan allows almost full control over the GPU. However that control comes with increased responsibility: whoever uses it must deal with many constraints previously defined by the less permissive drivers. That, combined with a *very* explicit and verbose API, makes the usage of Vulkan be recommended only to programmers that really need that control over the graphics card, which is often the case in real-time applications like games.

### 2.3.2 Memory

One aspect that hardware advances brought to programmers was the lack of necessity of memory management, as nowadays memory is a cheap resource.

While most programmers may never have to worry about how much memory there is left or how allocations work, mainly because modern programming languages can take care of that, when dealing with high performance applications it's crucial to understand how memory behaves, as programs are ultimately made to run in some hardware, making the use of clever algorithms alone not sufficient to speed things up.

A well known talk given on the memory subject was given by Acton (2014), where some important properties used by this implementation were explained. Between those properties the most important is the location and access of data in memory, both in the GPU, called *Device Local Memory*, and in the RAM, called *Host Memory*.

Every time a computation takes place inside a processing unity (PU), be it the GPU or the CPU, data has to be transfered to the chip. The problem is that data transferring takes time, sometimes much more time than the required by the computation. To speed things up, the PU fetches a whole chunk of memory around the requested address, as it's common to subsequent requests be in nearby addresses.

When dealing with graphics cards another problem arises, as not only data localization is important, but also all the data of an application comes from the RAM, in the CPU side. Transferring that data is usually much slower than any operation that will take place in the GPU, so it's preferred to pass whole chunks of memory at once.

### 2.3.3 Image representation

In order to produce images, one has to know how they are represented and displayed by a computer. This final part aims to explain some properties used when translating the formulas to a 2D grid of pixels and why things are not so straightforward as they may seem.

The previous sections explained how radiance is the calculation product that represents color, and it's correct to assume that when some one doubles the energy being irradiated, the luminosity should be doubled. The thing is that the human eye is tricky, as with doubling the values does not result in a *perceptual* doubled luminosity, i.e. the image does not appear to have doubled the brightness.

As explained in Chapter 5 of Akenine-Möller *et al.* (2008), when calculating light with the rendering formulas presented here, while things escalate linearly in the equations, the human perception escalates in a inverse exponential way, so it's easier to detect changes in darker parts of a scene than it is in bright ones.

Older CRT monitors used to have a power law relationship that closely resembled the inverse of the way that eye perceives luminosity: when doubling the input voltage, the perceived luminosity seemed to be doubled. Nowadays, that relationship is simulated by monitors to keep compatibility, and because of this behavior, a way to translate values from the linear space used in calculations to the eye perception space is needed. That mapping is called *Gamma Correction*.

The CRT power law curve follows a curve that is closely represented by $x^{\gamma}$, a exponential function. Because of that, gamma correction applied to the linear radiance following the inverse exponential curve $x^{\frac{1}{\gamma}}$. Those calculations take place in a space where radiance is normalized to a $[0, 1]$ range and the $\gamma$ exponent is what gives name to the function.

When using modern monitors, it's usual to make calculations with $\gamma = 2.2$, as that is a good approximation in most cases. The problem is that several variables, both inside and outside the hardware, interfere with how light is perceived, e.g. not only each monitor have a particular configuration but the room where someone uses that computer have interfering illumination. For that reason, both monitors and applications usually have options to adjust the gamma value and screen and brightness.

Besides correcting the color space to produce visually appealing images, there is another problem that arises from image representation: monitors work with RGB values in the $[0, 1]$ range, but that limit does not exist when using the rendering equations, therefore a lot of information is lost. The solution is to make calculations freely, using a $[0, \infty]$ range and then translating it back to the $[0, 1]$ range.

That more permissive range is called *High Dynamic Range*, or HDR for short, and is extensively used in computer graphics. The technique used to translate from that to the range used by monitors, called *Low Dynamic Range* (LDR), is called *Tone Mapping*.

The effects of using HDR and tone mapping can easily be seen in the real life, being similar to when the eye adjusts based on how bright a place is: when someone is in a dark place and then turns some light on, everything appears very bright; the eye then begins to get used to the new configuration, and everything seems normal. The moment that the light is turned on represents the image in HDR format and the eye adjusting represents the tone mapping.

There are several tone mapping functions to chose from, but the one used here is the more standard Reinhard Tone Mapping, defined by:

$$L_{Reinhard} = \frac{L_{HDR}}{1_v + L_{HDR}}$$

Where

- $L_{HDR}$ is the radiance in the HDR range
- $1_v$ is a RGB vector where all components equals 1

It's easy to see that any value expressed in HDR range will be translated to LDR range by that function. The catch is that this method tends to make the image darker, so it's not the best option when talking about quality, but it is fast and simple, justifying the usage.

# Chapter 3

# Implementation

Implementing a renderer is not an easy task, and having to implement one with the responsibility that Vulkan requires from the programmer only makes it a little harder. Following what was described in Part 1, here is shown how the most important parts of the renderer were implemented. It's worth noting that the full implementation won't be included in this document as Vulkan requires the code to be very explicit (the full implementation of the renderer contains thousands of lines), so it's not viable and probably of little interest to most readers.

The implementation is divided in two parts: the base renderer, containing all the Vulkan and C++ parts that get things to the GPU; and the raytracer, containing the GLSL code used in computations and how the data is handled there. After that both parts are put together to produce the final renderer.

## 3.1 Base Renderer

This part of the implementation is responsible for the vast majority of code, showing why the usage of Vulkan should be considered only when really needed, as both maintenance and overall understanding[1]are made harder by the API.

It's called Base Renderer because it composes the backbone of where the raytracing algorithm will run, being responsible for maintaining the main rendering loop and flow of data between host (CPU) and device (GPU).

### 3.1.1 Choosing a GPU

Vulkan requires the programmer to be very explicit when chosing something in exchange of being very descriptive about what there is to chose from. After starting a Vulkan instance

[1]When one understands the ways of Vulkan and have a background in computer graphics, the code is actually very descriptive and straightforward, with nothing being hided from the programmer.

(vulkan application), one of the first jobs to be performed is the choice of which physical device, or devices, will be used by that instance.

By looking at the description and capabilities of each available device, the programmer knows which one will best suit the job. For this renderer, it's preferred a discrete GPU (dedicated graphics card, as those are usually faster), and required that the device can perform three operations: graphics computations, for using the usual graphics pipeline; compute shaders, for calculating the image each frame; and present support, offering capability to show something on the screen.

All those required operations are defined by the presence of *Queues*, interfaces that allow the recording of *commands* to be submitted to the device, meaning that the chosen device must have support to graphics, compute and present queues[2].

In order to actually use the chosen physical device, a *logical device* must be created. That logical unity is first an interface that defines which aspects of the physical device should be enabled, and second a handle used by Vulkan to manipulate GPU resources. The physical device is only used when creating queues and the logical device, therefore logical device will be referred as Device from now on.

For this implementation, there is no need for any special device feature (geometry and tessellation shaders for example), so the only real usage of device is as an interface for the GPU. That said, when creating the device handle, it's required to activate an extension enabling screen presentation. Extensions are parts of Vulkan that are not required supported by GPUS vendors, but are often supplied, as is the case of presentation support.

The reason that presentation support is not part of the core Vulkan is that in order to someone present an image on the screen, OS specific functions must be called, breaking the multi-platform purpose of the API. For having a renderer that supports screen drawing while still being cross platform, the GLFW library was used.

GLFW provides an easy to use OS independent Window System Integration, i.e. it create a window and let the programmer use it's surface for drawing, regardless of the underlying operating system. For Vulkan that means an extension must be enabled when the device is created, and a VkSurfaceKHR, or surface for short, is acquired using a GLFW function.

[2]Currently Vulkan does not support present queues separated from graphics queues, so if a device supports a graphics queue it also contains a present queue. Vulkan requires any device that supports it to have at least one compute queue (not necessarily dedicated), but since the API can be used to GPGPU alone, graphics queue is not required.

### 3.1.2 A brief Vulkan breakdown

Before delving deeper into the renderer implementation, it's useful to understand how Vulkan operates. For that, some core API concepts will be explained, not only as they are essential to the data workflow, but also as they gently introduce how to proceed through the code.

All work that the GPU executes comes from the CPU when queues are submitted in a certain point of the code. But queues alone are not useful, since they are just an interface responsible for taking the earlier mentioned commands to the GPU.

Commands are instructions that the programmer wants to execute in the GPU, like setting up constants or performing draw calls, but they are not so simple, since each command must be submitted to a specific type of queue using *command buffers*. More than that, commands are not just created like a common C++ variable, as they must be also located on the GPU. For that reason, they are recorded into command buffers that are created using a Command Pool.

A Command Pool is like an allocator located on the GPU, but with specific capacity and designed to work with a specific queue family. Many Vulkan structures use similar kinds of allocation pools, as they are simple and provide large control over which resources will be used, allowing the driver implementation to optimize usage.

Just with those constructs it's already possible to see that Vulkan is composed by many small and simple parts that come together in order to do what the programmer needs, but for that to work properly, the implementation must know beforehand how that data will be used in detail, ideally creating the whole framework before any computation begins.

All those concepts alongside with many more will be used by the next sections, having the code workflow looking something like this at this point:

1. Create a Vulkan Instance
2. Create a GLFW window
3. Create a surface
4. Select a physical device
5. Create the graphics, compute and present queues
6. Create a logical device

Notice that there is nothing related to commands so far. Thats because commands are specific to some parts of the implementation, more precisely the ones where the device do something useful, and so far the only interaction with the GPU was to know it's properties and features, i.e. only the most basic to using Vulkan was done.

While describing all the used Vulkan constructs could be useful to someone, it's not so important in order to understand how the renderer was implemented. The main goal of this section was to introduce how the API operates and distributes the workload, that way it's easier to follow the next sections without caring too much about specific stuff.

### 3.1.3 Graphics Pipeline

Having created a Device, it's now possible to make the real interesting stuff with Vulkan, and the first used by the renderer is to create a classic rendering pipeline.

The classic rendering pipeline have two main programmable stages: the vertex shader and the fragment shader. If the reader does not know what shaders are, just think of them as programs that run in the GPU, because that's pretty much what they are.

As said in Chapter 2, data comes to the pipeline in the form of vertices, those are submitted to the vertex shader, where their positions are transformed to *Normalized Device Coordinates*, or NDC for short, a vector space where visible vertices stay in the $[(-1,-1,-1),(1,1,1)]$ range.

From that, the rasterizer takes part and then gives the produced fragments to the fragment shader, where color is calculated and written to an image.

In Vulkan the classic rendering pipeline is created by the vkCreateGraphicsPipelines function, taking, among others, a VkGraphicsPipelineCreateInfo as parameter. Now those can seem daunting at first, but everything in Vulkan operates with structures, and that structure just defines every aspect of this pipeline, i.e. the programmer configures all the pipeline steps, describing which shaders to use, how to perform fixed pipeline parts, how input will come and many more.

From all properties that the graphics pipeline used by this renderer uses, the only worth some dedicated explanation is the *render pass*. That property defines how *attachments*, i.e. images and buffers, should be treated by the GPU. For this renderer, the only operation that the graphics pipeline performs is read a texel from a texture and output its value, so the render pass just describes the output image format.

Notice that the render pass just *describes* how an attachment should be treated, it does not link the attachment itself. This job is performed by a *framebuffer*, another structure that define which attachments should be used. That framebuffer is linked to the pipeline later, when the command buffers are being filled, and then executed when this buffer is submitted to the graphics queue.

While this pipeline is highly optimized for rendering, it does not fit the purposes of this renderer, as it uses raytracing to generate the images. That said, this pipeline acts as a middle ground between those images and what is presented on the screen. It works by sending a single quad (rectangle) composed of two triangles to the GPU. Each vertex of that quad contains a UV coordinate used to sample from the raytraced image.

UV coordinates are 2D vectors with its components ranging in $[0, 1]$. Inside the fragment shader they are used to sample (get texels) from some input image, i.e. the attachment defined by the framebuffer, in this case the image produced by the raytracer.

But presenting that image is not as straightforward, as inside the graphics pipeline only vertices in the valid NDC range get to the fragment shader, so in order to raster the whole generated image, the quad sent to the GPU is defined by:

```
1 struct Vertex
2 {
3     glm::vec3 position;
4     glm::vec2 uv;
5     float _padding;
6 };
7
8 const std::vector<Vertex> quad =
9 {
10     {{-1.0f, -1.0f, 1.0f}, {0.0f, 0.0f}},
11     {{-1.0f,  1.0f, 1.0f}, {0.0f, 1.0f}},
12     {{ 1.0f,  1.0f, 1.0f}, {1.0f, 1.0f}},
13     {{ 1.0f, -1.0f, 1.0f}, {1.0f, 0.0f}},
14 };
```

Since that quad is static for all the program's duration, it's possible to eliminate the need of sending it to the GPU at all. For that, the same vector describing the quad's vertices is defined inside the vertex shader, then the vertex to be used is defined by a build in variable called gl_VertexIndex, indicating which vertex is being shaded.

### 3.1.4 Compute Pipeline

The raytracing algorithm runs in the GPU with a *Compute Shader*, using the GPGPU part of Vulkan to produce the final raytraced image used by the fragment shader.

Like with the graphics pipeline, the compute pipeline is created by Vulkan with a function and primarily a createInfo structure, but it's much simpler that the graphics one.

All the hard work is made by the compute shader, so the only real work of the pipeline is to describe how data is organized, then data about the geometry inside a scene is passed to the GPU, and finally a dispatch command is passed to the compute queue telling the compute shader to be executed.

All the work on the host side then is resumed in configuring the pipeline beforehand and setting the scene variables at execution time.

As data is sent to the GPU as a chunk of bytes, for the configuration part, Vulkan requires *descriptions* of the variables used inside a shader so it knows where in the memory to find each information. That kind of configuration is also needed in the graphics pipeline, but as the only variable sent to the device by it is the image produced by the compute pipeline, the explanation will be given for the compute shader only.

Each kind of resource inside the compute shader have a specific type and is associated with a *binding*. A binding can be tough of as being a constant memory pointer that points to the start of a big array of bytes. That array has no idea of what it holds, so a description of the data inside it must be provided. That description is provided by *DescriptorSets*, structures that associate types of resources to *locations* inside a binding. Locations are what the shader uses to find where in the binding a resource can be found.

The data passed to the GPU by this pipeline must be well defined and organized in order to be used by the compute shader, therefore the following organization was used:

- Array of spheres
- Array of planes
- Array of light sources
- Array of materials
- View and projection matrices

Those arrays first reside into host memory, where they are calculated and maybe updated, then passed to the compute pipeline via command buffers or memory transfer depending

on the amount of data changed. Of course, in order to that be available in device memory, a descriptor set defining those arrays is created and then associated with the pipeline at creation time.

### 3.1.5 Swap Chain

As said previously, Vulkan does not require the hardware supporting it to provide a way to present images on the screen, but when the intent is to show something on the screen, a structure called Swap Chain is used.

It works by taking a *screen surface* (not to be confused with object's surfaces used by the raytracer) from the Operating System, creating several images compatible with that surface. The graphics pipeline then *acquire* one of those images and then draws to it, submitting it to the present queue when the drawing finishes.

There are some limitations that must be followed when creating the swap chain, as it depends on the hardware being used. Those limitations are defined and should be checked when choosing the physical device, but the most relevant is the presentation mode.

The way that image acquisition and presentation works follows the selected (if available) presentation mode. In general, for any of supported modes, images created by the swap chain lies on a queue, then when the graphics pipeline requests an image, the swap chain verifies if there is an image on that queue.

After using an image, the application MUST return that image to the swap chain by submitting to the present queue. The image on the screen then is updated following the presentation mode, varying from immediately showing the newest image to waiting for the screen refresh to get the next image on a specific queue. If a hardware supports presentation, the only mode that Vulkan requires to be present no matter what is the queue one.

## 3.2 Raytracer

When the compute shader begins executing, the algorithm is pretty straight forward, following what was described in Chapter 2. There are however some implementation details to be accounted for. If the reader comes from an object oriented programming mindset, some things may seem nonsense, as there are no objects nor classes inside a shader.

First, as briefly explained in the compute pipeline section, all geometry data comes to the shader basically as arrays of predefined data. Those arrays are actually arrays of structs,

as structs are allowed in shader code.

For both spheres and planes, those structures contain geometric information like position and radius for spheres and position and normal for planes. Not only that, but a index to a PBR structure array is also present. That latter array contains the material information and can be shared between many objects.

Light sources are all punctual, meaning they are just a point in the scene and the light rays directions are calculated per intersection point. Those intersections, the main part of the algorithm, are calculated from each pixel as described below, using one ray per pixel:

1. Calculate the camera looking direction (main direction)
2. For each pixel, add a offset vector to the main direction, representing that pixel's ray
3. With the pixel's ray, test it against ALL the geometry in both spheres and planes arrays, storing the closest intersection point, normal and index to the material array
4. Calculate the light direction for the intersection point
5. Perform the lightning calculations based on the stored information
6. Save that color, calculate a new direction for the bounce ray and repeat the process, accumulating the resulting colors with the original one.

Notice that all the intersection code must be present in the shader code, as common object oriented classes do not exist. That is not a problem, and even common to C programmers, but may be something strange for someone that has never used that approach.

Also, inside the geometry data, only *Plain Old Data* (PoD) is expected, like it is in a plain C struct (in fact, one could treat incoming data from the Host as a bunch of C structs). Any pointer type is useless for the shader, as not only it would refer to random memory but also there is no such concept in GLSL. That said, indexing is used as an alternative, but it's error prone and hard to debug.

If the reader is wondering why someone would like to use pointers inside a shader, for the case of raytracing there is a pretty simple answer: acceleration structures. When calculating ray intersections, it helps a lot to be able to reduce the number of tested geometry, increasing performance and then opening opportunities to better graphics.

Recursion is not supported in shader code. That means it gets really tricky to trace scattering secondary rays. The solution used is to trace only one ray and trace it until it reaches nothing or pass the ray bounce depth. That can be done inside a nested loop, simulating

many scattered rays, but the path tracing way works by only following the path of a single ray.

Lastly, GPUs (and also CPUs, but not so intensely) hate branching. That means the usage of least amount of ifs is a core performance requisite. More often than not it's better to calculate two values and use only one instead of using an if to calculate only one value. GPUs are very good at parallelizing things, also very good at performing calculations, but are terrible when dealing with branching.

## 3.3 Wrapping it up

The main rendering loop of is where everything comes together, as data produced by the raytracing finally gets to the screen. The following code gives a general idea of how that loop works:

```
1 void draw()
2 {
3     setSceneData();
4     drawUsingComputePipeline();
5
6     prepareTargetImage();
7     drawUsingGraphicsPipeline();
8     showImageOnTheScreen();
9 }
```

The code is not exactly how things proceed because there is a crucial aspect to be taken care of: synchronization. When the host code call functions to submit work to the device, that code returns immediately, so execution continues to loop and send commands. More than that, after those commands are submitted to the GPU, the device may (and probably will) execute them in parallel, so the fragment shader may try to read memory that has not been written or the compute shader may try to write memory that is being read.

### 3.3.1 Synchronization

In Vulkan, all kinds of synchronization are responsibility of the programmer, being one big source of visual bugs. They are made by using Semaphores, Fences, and Pipeline Barriers, structures that are signaled in some point of code and then liberates other parts to proceed.

Semaphores are used to synchronize execution of queues. When submitting a queue to the GPU it's possible to specify two semaphores, one to be signaled when the queue execution is finished and one that the queue must wait to be signaled before it can start.

There is a synchronization by semaphores happening when the swap chain is used, as the graphics pipeline must finish drawing before that image can be presented. Similarly, the graphics pipeline has to wait until the swap chain finishes presenting an image if there are no available images when fetching the next render target. Another pair of semaphores could be used between graphics and compute queues submission, but the synchronization between those two can be faster by using another method.

Fences work in a similar way, but are used to synchronize execution between the Host and the Device. when submitting a queue, an optional fence can be send. That fence is signaled as soon as the queue finishes executing. The application wait for fences to be signaled, halting the program execution until then or until a provided timeout has been reached. The Host cannot signal any fences, but it is responsible for cleaning the signaled fences, returning them to the default state.

Since the raytracing computation is a low process, after the submission of the compute queue the program usually return to that same point before the computation is over. Therefore a good use of fences is to prevent raytracing from being calculated when a previous calculation is not over yet.

Pipeline Barriers are the trickiest concept of synchronization in Vulkan. They are use for many purposes, some being:

- Image layout transition
- Memory/Image ownership transfer
- Global/Local/Image memory access barriers

Those barriers are used via commands submitted to queues, and follow way to many rules to be described here. Suffice to know that they are at the core of most bugs while being essential to producing robust and fast applications.

There are a few places to use pipeline barriers, since the renderer produces images in the Compute Queue and consume them in the Graphics Queue. If those queues are not the same in the device, the image has to pass through a ownership transfer operation. Besides that, given that the image must be ready before being read, and the shader must not be reading it in order to start writing, two image memory access must be present.

# Chapter 4

# Conclusion

This chapter describes the end of development and relevant information about the project, containing some personal opinions about the whole process.

## 4.1 Results

Unfortunately there is not much to show from the actual implementation, as the code by the time that document was written is halted with some bugs. The project is available in the repository linked in the site.

That said, in earlier tests of the renderer, without the PBR activated and with only one ray bouncing from surfaces, it was possible to run examples with six planes, three spheres and moving point light sources in a stable 22 frames per second using a GeForce 940m with an Intel Core i5-5200U.

The reason for the recent bugs is due to an attempt of implementing both acceleration structures and meshes into the renderer. Not only it did not work, as it wasted a big part of development time, with a bonus of trashing the working code.

All that could be avoided if source control was used earlier in the project, but since almost all the development was made in a offline setup, there were no point in using one.

## 4.2 Future work

After correcting the actual bugs, many improvements can be made on the renderer, but two most immediate ones are: acceleration structures inside the GPU and raytracing with meshes, as they were being developed already.

As mentioned in the raytracing implementation part, shaders do not have a concept of

pointers. One of the reasons behind that is the heavily parallel pipeline and optimizations on the hardware, caching the memory in chunks used by a selected number of cores. That makes the implementation of acceleration structures, specially the ones that change over time, very difficult in GPUs.

Supposing that GPUs had pointers and those acceleration structures could be constructed by a shader, there still a problem that the memory written by a core in the device is not immediately available to all the other cores, so a lot of synchronization would be needed to make that work properly, but that level of control over the execution inside the GPU also do not exists.

As a last alternative, with the hypothetical GPU with pointers, one could argue that the acceleration structure can be constructed on the CPU side and then be transfered to the GPU side, but doing so would invalidate all the pointers, as they contain addresses relative to Host memory, which would point to random memory in the device.

The solution is to go back to the first hard implementation, using indices. Still, updating a acceleration structure in the GPU is even more difficult than using one, so even if it can be implemented, this implementation would probably stick to structures for static geometry only.

The second improvement, raytracing with meshes, is much more simple to solve, as the difficulty lies on the amount of computations needed.

Meshes are composed of triangles, and ray-triangle intersection is easy to calculate. The problem is that differently from the classic rendering pipeline, in raytracing all scene objects must be accessible at once, then transformations must be made before the effective passage of the raytracing pipeline, causing even more delay.

Except for that, implementing raytracing to meshes per se is not much difficult, the problem is that there are many triangles per mesh, so there would be many more ray intersection tests. As a consequence, unless a acceleration structure is used, it's not viable to implement support to that kind of geometry.

Besides that, comparisons with the same code running with other APIs would be nice to evaluate how much the use of Vulkan can contribute with performance.

## 4.3 Personal appreciation

This section contains opinions from the author after working to build this renderer, therefore the small part that is left of this text have a much more personal tone.

### 4.3.1 Difficulties

The most evident difficulty of this project was the scope, not from a code perspective, but from the amount of knowledge needed in order to produce something.

It's possible to divide this work in three distinct and huge parts: Vulkan, PBR and Ray-tracing.

From the Vulkan part, I was naive when first approaching the API. When they say it's a tool for experts, they are not kidding, not even a little. The estimated time to understand how to use it was very lower than it would take, as to this day I still do not fully understand how it works.

It requires deep knowledge in computer graphics, meaning anyone using it for real should know every step of the pipeline by heart, as well as knowing computer architecture: how memory works, how both CPU and GPU works, and how graphics cards are organized, among other things.

I did not have a deep knowledge in any of those topics, but now at the end of the project, it's safe to say I know a little more of how everything goes together, but by no means that makes me an expert. That said, I'm very happy to know that there is so much more places where I can improve, and Vulkan really is an amazing API once you stop to understand how it works.

From the PBR part, it was almost all new. I Had some knowledge on computer graphics and shading techniques that helped a lot when transitioning to the PBR model. That said, for several times I just didn't understand what I was reading, nothing seamed to make any sense and frustration was a common feeling throughout the development.

The thing is that besides finding it extremely hard at first, things started to take place and little by little all made sense. Things that I saw in movies or video-games started to make a were more transparent now, and a bunch of physical concepts that I had never understood started to have a intuitive meaning.

Still, as said in Chapter 2, I only barely scratched the surface of what is possible with PBR, and I'm glad to know that so many people are doing researches in that area.

Least but not least, there is the raytracing part. The algorithm per se is straightforward, so the real difficulty came from making it run in real time. That's pretty much it, and while there is not much to say, this is as hard as PBR, if not more. One proof of that is the amount of researches being made in the topic since the beginning of computer graphics, still going nowadays.

### 4.3.2 Relevant courses

Throughout the development there where two college subjects that really helped:

- MAC0420 - Introduction to Computer Graphics, teaching me all the basic concepts needed to go deeper in computer graphics, while also providing useful information in raytracing and some more advanced concepts
- MAC0422 - Operating Systems, teaching me about how memory is handled by the system and some more stuff on computer architecture. Besides that, it was where I learned a big part of what I know about parallel and concurrent computing

# References

**Acton(2014)** M. Acton. Data-oriented design and c++, 2014. URL https://www.youtube.com/watch?v=rX0ItVEVjHc. Cited in page 20

**Akenine-Möller** *et al.***(2008)** T. Akenine-Möller, E. Haines e N. Hoffman. *Real-Time Rendering, Third Edition*. CRC Press. ISBN 9781439865293. URL https://books.google.com.br/books?id=V1k1V9Ra1FoC. Cited in page 8, 21

**de Vries()** J. de Vries. Pbr series from learn opengl. URL https://learnopengl.com/#!PBR/Theory. Cited in page 8

**Hoffman(2015)** N. Hoffman. Introduction to "physically based shading in theory and practice", 2015. URL https://www.youtube.com/watch?v=j-A0mwsJRmk. Cited in page 8