
Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

# BOX OF SUNLIGHT
## Riflessione Realistica della Luce nel Rendering di Animazioni 3D

# BOX OF SUNLIGHT
## Realistic Reflection of Light in 3D Animation Rendering

LEONARDO ZETTI

Relatore: *Prof. Stefano Berretti*

Correlatore: *Prof. Michele Ginolfi*

Anno Accademico 2023-2024

# CONTENTS

# LIST OF FIGURES

*Make me a picture of the sun—*
*So I can hang it in my room—*
*And make believe I'm getting warm*
*When others call it "Day"!*

Emily Dickinson

# 1

# INTRODUCTION

The aim of this thesis is to explore a set of physical and mathematical concepts that are foundational to photorealistic computer graphics, to then propose and evaluate an implementation of light reflection simulation techniques, in a program called *BoxOfSunlight*.

Our first objective will be to interpret, in an accessible manner, what it means for digital, 3D art to be *physically based*. There is a lot that goes into the creation of physically based images, and our analysis will only scratch the surface of the problem. More specifically, we will only be concerned with capturing accurate *reflection of light* on virtual materials, as there is a lot to be said on this topic alone.

Also, our discussion will mostly be limited to the sub-field of 3D animation. As a closing theoretical topic, we will apply what we have learned to examine the *Disney "Principled" BRDF*, a model for reflections which has been used in the production of animated movies. The design of the Disney BRDF has been made public by Brent Burley in his 2012 SIGGRAPH talk *Physically Based Shading at Disney*[1] (Burley 2012).

As already suggested, the very last (and more practical) part of the thesis will be composed of various implementation details and tests, associated with the BoxOfSunlight program. BoxOfSunlight was written in C++, using the OpenGL API. Its purpose is to generate believable images of 3D materials by applying the Disney BRDF model in simple virtual scenes, where the source of light is an HDRI environment map of some real-world scenario, generally illuminated by the Sun's natural light.

1 The talk was part of the 2012 SIGGRAPH course *Practical Physically Based Shading in Film and Game Production*, the contents of which can be found at https://blog.selfshadow.com/publications/s2012-shading-course/ (last accessed: 26.03.2025).

*Fig. 1:* An image generated with BoxOfSunlight.

## 1.1 COMPUTER GRAPHICS

Computers have the potential to be powerful tools for artistic expression. One way to create art with a computer is through *computer graphics*. Computer graphics can be defined as

> "The science and art of communicating visually via a computer's display and its interaction devices."
> (Hughes et al. 2014)

Advancements in this field have made computer-generated imagery a pervasive component of our day-to-day lives, ranging from special effects in movies to the *Graphical User Interfaces* (*GUI*s) on our phones.

Thus, the world of computer graphics is a vast one. We should keep in mind that the problems and solutions presented here only refer to the context of "geometry-based 3D graphics", as we'll call them. Our graphics will be "geometry-based" in the sense that we'll first describe the objects we want to draw on the screen with *geometric models*[2] (lines, polygons, polygonal meshes, ...), to then *sample* them for visualization.

We can imagine our process to be as follows.

1. Create a *geometric model* of the object,
2. Describe the *material* it is made of,

2 In the field of computer graphics, the word "model" is used to refer both to *geometric models* and *mathematical models*. A *geometric model* describes an object we want to "take a picture" of (for example, the *Stanford Bunny*). A *mathematical model* describes some computational process that we use to take that picture (for example, the equations we use to tell how much light is reflected by the bunny's surface). (Hughes et al. 2014)

3. Place the object in a *virtual scene* with *light sources*,

4. Use a *virtual camera* to "take a picture" of the object (this is the so-called *sampling* phase).

Our graphics will be 3D, meaning that the geometrical models, other than width and height, will also have *depth*. Our virtual camera will then communicate that depth to the viewer through the use of *perspective*.

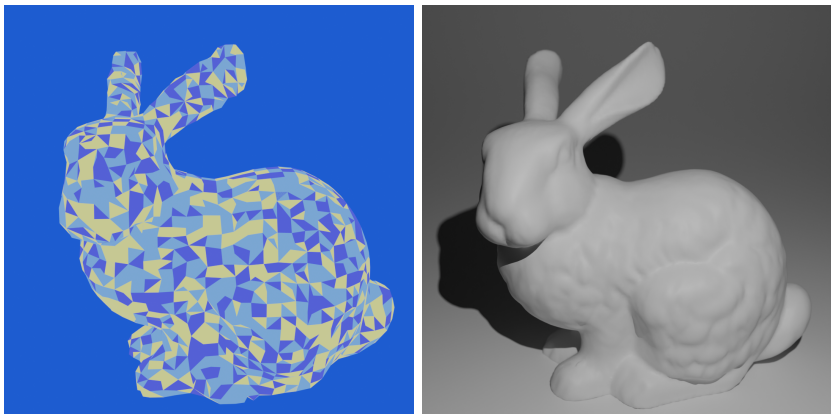


*Fig. 2:* The *Stanford Bunny Model*, drawn on the left as a *triangle mesh* (random color for each triangle), and on the right with realistic lighting. Model courtesy of the Stanford Computer Graphics Laboratory, rendered with Blender `4.3`

To restrict ourselves even further, we'll keep our focus mainly on steps 2. and 4. of the above mentioned process. Our aim will be to derive a single *mathematical model* than can describe as many materials as feasible, in relation to how they *reflect light*. We'll then use this model to record reflection effects with our virtual camera.

The final images will be generated so:

1. Light gets emitted from a light source,

2. Light hits a surface and is reflected in various amounts and directions, depending on the *material* of the surface,

3. Light that's reflected towards the virtual camera gets recorded in the final image.

We'll now give a more rigorous definition of the virtual "picture-taking" step that we're interested in. We'll refer to it as *rendering*.

## 1.2 RENDERING AND PBR

In the book *Physically Based Rendering, From Theory to Implementation* - an important resource for this thesis - *rendering* (or, as it has been historically referred to, *image synthesis*) is defined as

> "The process of converting a description of a three-dimensional scene into an image."
> (Pharr, Jakob, and Humphreys 2023)

Note that a *3D scene* is comprised not only of 3D objects, but also light sources and a camera. Generally speaking, the final image of this process can be generated using any rules we choose. This flexibility enables a wide range of rendering techniques, each of which will communicate a different *message* to the viewer.

This is our final goal in graphics, to *communicate* with viewers through *meaningful images*. As Andrew Glassner put it:

> "The field of *image synthesis*, also called *rendering*, is a field of transformation: it turns the rules of geometry and physics into pictures that mean something to people."
> (Glassner 1995)

If our aim is to create art, as in the case of *animated movies*[3], rendering can be an important vehicle for creative expression. That is because the end result of a rendering process will influence the *emotional responses* elicited in spectators. This interplay of different fields of study - math, physics, art, computer science, psychology, to name a few - is what makes rendering, and computer graphics in general, fascinating disciplines that are worth exploring.

So, in theory, there are no rules on how to *render*[4] a picture. However, what we are often interested in is making our images *physically plausible*, that is, *realistic*. This is why various rendering applications - such as

3 In 3D animated films, each of the frames shown on screen was generated by some advanced rendering software. Achieving high levels of realism demands intense computation: applications used in movie production spend *hours* computing a *single frame*. See for example `https://sciencebehindpixar.org/pipeline/rendering` (`last accessed: 28.03.2025`)

4 The word "render" can be used to express two different concepts. When used as a *verb*, it is the process followed by the computer to produce an image. When used as a *noun*, it refers to the final image produced.

*Fig. 3:* Stylized render from Blender Studio's *Project Gold*. ©Blender Foundation

*RenderMan*, which is used to make Pixar movies - are, we say, written to be *physically-based*.

*Physically Based Rendering* (or *PBR* for short) is a collection of rendering techniques based on physical models of real-world light and materials (Pharr, Jakob, and Humphreys 2023).

Our objective in this thesis will be physical plausibility, so we'll stick with PBR. As an alternative approach, one could, for instance, render *stylized* images to favor *expressiveness* over *realism* (as shown in Figure 3).

## 1.3 HISTORICAL NOTES

In this paragraph, we'll go through a short summary of some key steps that have been taken in the research progress of physically based rendering[5]. We will also see how PBR has been gradually adopted in film production.

Physically based rendering research started to gain traction in the 1980s. An important seminal work was Whitted's 1980 paper, *An Improved Illumination Model for Shaded Display* (Whitted 1980), which has since served as a foundation for modern *ray-traced* computer graphics which feature *global illumination*.

PBR generally makes substantial use of *ray tracing*, and for this reason, in Chapter 3, we'll give a brief description of this technique. For now,

5 The summary is a condensed version of the chapter *A Brief History of Physically Based Rendering*, from (Pharr, Jakob, and Humphreys 2023)

let's just say that ray tracing involves - very surprisingly - tracing rays of light in a 3D scene, and that this is done to determine the colors of objects. As for *global illumination*, a brief introduction to the topic can be found in Appendix A.

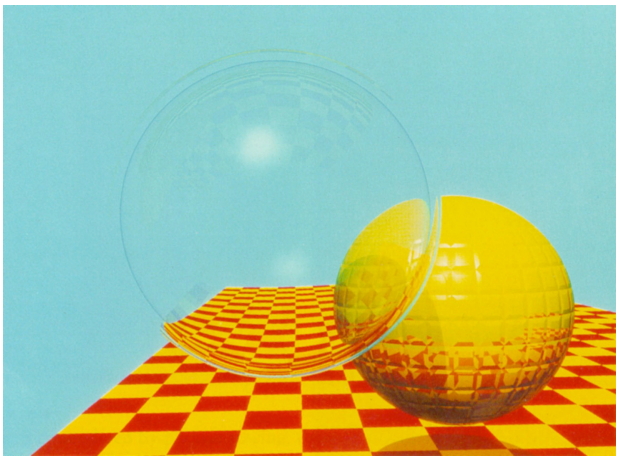


*Fig. 4:* One of the renders from Whitted's 1980 paper. ©ACM

We should mention that an important early contribution to PBR techniques was also given by Cook and Torrance, when they proposed a new reflection model based on *microfacet theory* (Cook and Torrance 1982). The techniques for realistic reflections described in Chapter 3 will be rooted in said theory.

Many other historic steps were necessary in order to make today's PBR techniques a reality. One of these was the work of Kajiya, who in 1986 introduced the *rendering equation* (Kajiya 1986). This equation describes in a concise and elegant way the problem that any realistic rendering application wants to solve. More on the rendering equation can be read in Appendix A.

Kajiya also proposed *path tracing*, an advanced form of ray tracing that properly takes into account global illumination effects, through the use of *Monte Carlo integration*.

Again, this topic is covered better in Appendix A. However, to give an idea of what we mean, let's put it this way. In order to do proper realistic rendering, it's necessary to approximate the values of many (nested) definite integrals. Monte Carlo integration is a technique, based on random numbers, for doing exactly that.

Monte Carlo integration really transformed the field, allowing the creation of images unlike any before. Many important contributions were made to Monte Carlo-based efforts during the years, one of the most important ones being Eric Veach's 1997 PhD thesis. Veach advanced key theoretical foundations of Monte Carlo rendering, and also developed new algorithms that improved its efficiency, like *bidirectional path tracing* (Pharr, Jakob, and Humphreys 2023).

It took some time to incorporate physically based rendering techniques in film production, due to their computational costs. An early example of a movie with Monte Carlo global illumination was the short film *Bunny* (1998), by Blue Sky Studios. Its visual look was substantially different from other films and shorts of the past. Before then, renderers were mostly based on *rasterization*, a rendering technique that's faster than ray tracing, but, in general, less realistic. *Reyes* is an important example of a *rasterization-based* architecture that had been used to generate photo-realistic images (Pharr, Jakob, and Humphreys 2023).

After *Bunny*, another turning point was reached in 2001, when Marcos Fajardo presented at the SIGGRAPH conference an early version of his *Arnold* renderer. *Arnold* was able to render scenes with complex geometry and global illumination in just tens of minutes (Pharr, Jakob, and Humphreys 2023). With the help of Sony Pictures Imageworks, *Arnold* was developed into a production-capable rendering system. It was first used on the movie *Monster House* (2006), and is now available as a commercial product.

In the early 2000s, Pixar's *RenderMan* renderer began transitioning to PBR, by supporting hybrid rasterization and ray tracing algorithms. It also introduced a number of innovative algorithms for computing global illumination solutions in complex scenes (Pharr, Jakob, and Humphreys 2023).

*RenderMan* was recently rewritten to be a physically based ray tracer. It was first employed in its new form for the production of the 2013 movie *Monsters University* (Hery and Villemin 2013).

*Fig. 5: Monsters University*, ©Disney/Pixar 2013

## 1.4 THESIS OUTLINE

The rest of the thesis will be organized as follows.

Chapter 2 will be devoted to the properties and behavior of light, in a physical sense.

Chapter 3 will describe mathematical models commonly used in ray tracing. This also includes techniques for realistic light reflection, such as the *Disney "Principled" BRDF*.

In Chapter 4, an implementation of the previously discussed techniques will be presented, in the form of the small renderer BoxOfSunlight. The focus here will be on the technical choices made during development.

In Chapter 5, the BoxOfSunlight renderer will be evaluated. We'll see how just by changing the values of a few input parameters, we can simulate the looks of a great multitude of materials.

Finally, Chapter 6 will address some of the shortcomings of BoxOfSunlight, and will contain ideas on how it could be extended into a more complete physically-based renderer.

# 2

# PHYSICS OF LIGHT

This chapter establishes the physical foundations of our physically-based rendering techniques. By its end, we'll have acquired important intuitions relating to three topics in the physics of light, namely, the *dual nature of light*, the *electromagnetic spectrum*, and *light-matter interactions*. The very last subject will be a physical interpretation of the *reflection of light*.

## 2.1 WAVE-PARTICLE DUALITY

When creating images, we are effectively using light as a tool to communicate information to the viewers. This raises an apparently trivial question.

What *is* light?

We'll give the physicist's answer. It might seem non-sensical at first, since it's actually made up of *two*, very different, answers. We say in fact that light has a *dual nature*, in the sense that there are two ways, that is, *models*, to define what light is. One represents light as a *wave*, the other as a stream of *particles*. Now the interesting part: light is actually *both* of these at the same time. There are situations in which light behaves just like a smooth, continuous wave, and you couldn't possibly think of it as being *granular*. But then, we also find cases indicating that light *is* actually composed of small, individual "energy packets", and where the wave theory is definitely not an option. In the following sub-sections, we'll dive (although a bit superficially) into a discussion about both models, and extract some facts that will be useful to us[1].

1 Various examples and definitions that follow were taken from *light*. Encyclopaedia Britannica. (Stark 2025)

### 2.1.1 *Wave Nature of Light*

Obviously, light is important for far more reasons than just visual perception and communication. It is a central component to our lives (actually, to *life* in general as we know it), and for this reason we're usually somewhat familiar with the physical concept of it. Frequently, we say that light is an *electromagnetic wave* (or *radiation*). It's easier to visualize what a *wave* is by thinking of *mechanical waves*. For example, we can imagine light as a phenomenon similar to water waves. We say that they are similar since they are both *disturbances* that *propagate* through space. Water waves propagate as physical displacements of water, a *material medium*. By contrast, electromagnetic waves don't require any material substance to propagate.

Using a proper definition, we say that electromagnetic waves are *oscillations* in the *strengths* of *electric* and *magnetic fields*.

There's a lot to unpack here. Most importantly, we haven't talked about *fields* yet. We can think of a field as of a function that associates some physical quantity to every point in space (and time). For example, each point in Earth's atmosphere has a temperature associated with it. This temperature can be expressed as a function of spatial coordinates and time: $T(x, y, z, t)$, where $T$ is the temperature field, $x$, $y$, and $z$ are the spatial coordinates, and $t$ is the time. We say that temperature is a *scalar* field. On the other hand, *electric* and *magnetic fields* are *vector fields*, since they associate *vectors* to points in space and time. We indicate the electric field as $E(x, y, z, t)$ and the magnetic field as $B(x, y, z, t)$ ($E$ and $B$ for short).

We won't get much deeper into the topic of fields. For us, it's sufficient to know that $E$ and $B$ are abstractions, in the following sense. Two electrically charged bodies, just like two magnets, exhibit forces (repellent or attractive) on each other. The forces depend on the characteristics of *both* bodies. Now, If we focus our attention on only *one* of the bodies, between the two electric charges or the two magnets, we can consider it as the source of a *field*, *electric* or *magnetic* respectively, which extends in the surrounding space. From this point of view, the force exerted on the second body of the couple is caused by the *field*. Thinking of fields instead of forces is a more abstract approach, since, putting it simply, we need to consider only one of the two bodies to describe its influence on the space surrounding it (the field it produces). Thus, we can see that $E$ and $B$ are abstractions which allow for more complex reasoning.

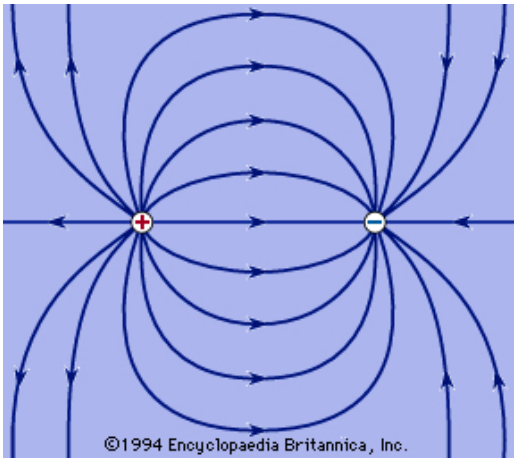



*Fig. 6:* "Electric field lines near equal but opposite charges." Source: *electric field*. Encyclopaedia Britannica.

We know that E and B are needed to properly define light, which is an *oscillation* in the strengths of both fields. The definition of light might still be a bit confusing, though, so our next aim will be to *see* what an electromagnetic wave actually looks like (that is, its *graph*). In order to get there, we should start with a reminder on why we are considering the electric field E and the magnetic field B *together*.

Developments in 19th century physics, which famously culminated in *Maxwell's equations*, proved that E and B are intimately coupled: when one of them changes, the other one does as well. For example, Faraday's well-known *law of induction* describes how a moving magnet can generate electric current[2]. For this reason, we unite E and B in a single concept: the *electromagnetic field*, which describes both electric and magnetic influences produced by bodies in space.

Now, both E and B can be modeled as *time-harmonic fields* (Glassner 1995); that is, they can be described as *traveling harmonic waves*. A harmonic wave can be specified as a sinusoidal function over distance and time, which outputs the *displacement* (in our case, the strengths of the two fields). An example of a harmonic wave traveling in only the $x$ direction is given by

$$y(x,t) = A\cos\left(\frac{2\pi}{\lambda}x - \frac{2\pi}{T}t\right) \tag{2.1}$$

2 From *Faraday's law of induction*. Encyclopaedia Britannica. URL: https://www.britannica.com/science/Faradays-law-of-induction (last accessed: 12.03.2025)

Where:

- $y$ is the vertical displacement at distance $x$ and time $t$,
- $A$ is the maximum displacement of the wave, or *amplitude*,
- $\lambda$, the *wavelength* of the wave, is the spacial distance between successive crests,
- $T$ is the *period*.

If we were standing still, on a point along the wave's path, and were to measure the time for one crest to reach us after the previous, that would get us the *period*.

Another important quantity is the wave's *frequency* $\nu$; it is defined as $\nu = \frac{1}{T}$, and we can think of it as the "speed" at which the wave repeats. It is measured in Hertz (Hz).

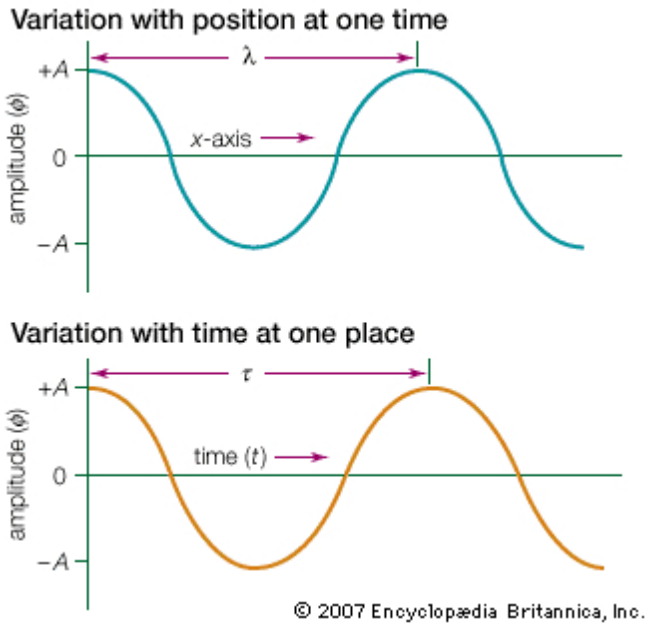



*Fig. 7:* "Snapshots of a harmonic wave can be taken at a fixed time to display the wave's variation with position (top) or at a fixed location to display the wave's variation with time (bottom)." Source: Encyclopaedia Britannica (Stark 2025)

Frequently, instead of writing $\frac{2\pi}{\lambda}$, we substitute it with $k$, the so-called *wave number*. We also use $\omega = \frac{2\pi}{T}$, which is called the *angular frequency*[3]. Moreover, we should mention that, in the more general case, we can add a *phase shift* $\phi$ inside the cosine[4]. The effect of this is to, basically, shift the wave function along the $x$ axis.

With these modifications, (2.1) becomes:

$$y(x,t) = A\cos(kx - \omega t + \phi) \tag{2.2}$$

Putting it all together, we can thus say that an *electromagnetic wave*, that is, *light*, is composed of two harmonic waves, one representing the strength of $E$, the other the strength of $B$. We also add that $E$ and $B$ are always perpendicular to each other, and that the magnitude and direction of propagation of an electromagnetic wave can be calculated with the cross product $E \times B$ (Glassner 1995). We can now see what the graph of an electromagnetic wave looks like by plotting $E$ and $B$ along the $x$ axis (Figure 8). We define their values as

$$\begin{aligned} \vec{E}(x,t) &= E_0\cos(kx - \omega t + \phi)\hat{y} \\ \vec{B}(x,t) &= B_0\cos(kx - \omega t + \phi)\hat{z} \end{aligned} \tag{2.3}$$

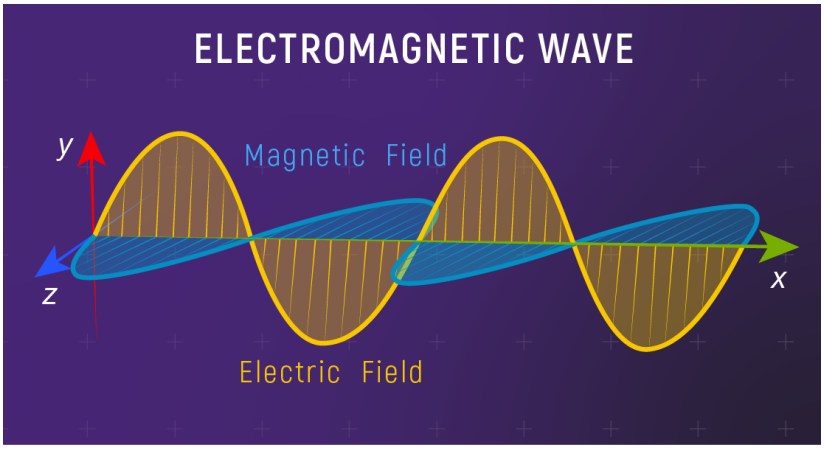



*Fig. 8:* Graph of a light wave. Credits: NASA, ESA, CSA, Leah Hustak (STScI).

To be more precise, what we are considering here is a *linearly polarized* wave, which has the property that the fields oscillate in fixed directions.

3 As done in *The Feynman Lectures on Physics*, Vol. I, Ch. 29 (Feynman, Leighton, and Sands 2011)

4 As done in *The Feynman Lectures on Physics*, Vol. I, Ch. 21 (Feynman, Leighton, and Sands 2011)

Before we move on, it's important that we address a situation that we'll encounter many times. Let's take two or more waves that are overlapping (they're "on top of each other"). Such waves are said to be in *superposition*. The *superposition principle* states that, when two or more waves overlap, they produce a *new* wave, with a net displacement that is equal to the algebraic sum of the individual displacements (Stark 2025). What this means is that the values of a wave function, be it $R(x,t)$, could actually be the result of summing up *many* different harmonic wave functions:

$$\begin{aligned} R(x,t) = {} & A_1 cos(k_1 x - \omega_1 t + \phi_1) + A_2 cos(k_2 x - \omega_2 t + \phi_2) + ... \\ & + A_n cos(k_n x - \omega_n t + \phi_n) \end{aligned}$$

In fact, the electromagnetic waves that we'll talk about should be in general interpreted in this form. In rendering, we'll think of every electromagnetic wave as being composed of (infinitely) many overlapping harmonic waves. As we know, each of these harmonic waves has its own wavelength $\lambda$. Putting it simply, we can thus say that a single ray of light "contains" many wavelengths. We'll find this to be an important fact.

There are good reasons to model light as a wave. A classic demonstration of the wave nature of light is the *double-slit experiment*, first performed by Young in 1801. In this famous experiment, two parallel slits on an opaque surface are equally illuminated by a light source, and the light that passes through the slits is observed on a screen. When the slits are close together, light that strikes the screen creates a pattern of alternating bright and dark bands (Glassner 1995) (see Figure 9).

The easiest way to explain this result is by positing that the light exiting the two slits has a wavelike nature. As we know, we can describe light waves with the two functions $\vec{E}(x,t)$ and $\vec{B}(x,t)$ (from (2.3)), periodic over distance and time. However, to simplify this discussion, let's express light as a single periodic function $A(x,t)$, instead of two (with $A(x,t) = A_0 cos(kx - \omega t + \phi)$).

We start by noting that when waves pass through a slit in a barrier, the slit acts as a source of cylindrical waves (the waves propagate symmetrically beyond it). This is due to a phenomenon called *diffraction* (Glassner 1995).

In the experiment, both slits are at the same distance from the light source, so the waves leaving the two slits have the same *phase* (they're in "sync", so to speak). This means that, at any time $t$, the *same wave* is generated in synchrony at both slits. Now, even though the waves leaving the two slits are the same, the two distances $x_1$, $x_2$ that they travel to

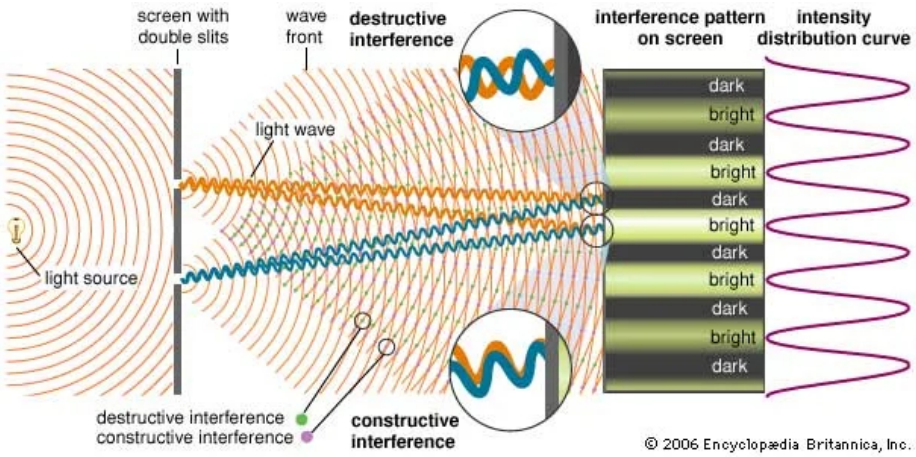



*Fig. 9:* "Young's double-slit experiment." Source: Encyclopaedia Britannica (Stark 2025)

reach a same point on the screen will be, in general, different ($x_1 \neq x_2$). Consequently, their wave functions in that point might differ, that is, $A(x_1) \neq A(x_2)$ (with $t$ fixed). In some points, for example, one wave arrives at its maximum and the other at its minimum, causing their sum to be 0; the waves here leave a dark spot, and we say that they *destructively interfere*. In some other points, both wave functions might arrive at their maximum, creating a bright spot. In this case, they *constructively interfere*. Between these two extremes we get different intensities due to different amounts of *interference* between the two waves.

This is exactly what causes the light-and-dark pattern that we wanted to explain (Glassner 1995).

### 2.1.2 *Particle Nature of Light*

Not all physical phenomena related to light can be explained by thinking of waves. Most famously, the wave model disagrees with the *photoelectric effect*, observed first by Heinrich Hertz in 1887.

Consider an experimental setup which consists in a beam of light shining onto a piece of metal, called the *cathode*. Next to the cathode, we also set up a detection device that can measure the energy of electrons striking it. What happens in this scenario is that, as soon as we shine light on the cathode, the detector starts reporting electrons. Clearly, the

light that strikes the cathode triggers an expulsion of electrons from the metal. For each electron, we find its energy $E$ to be[5]

$$E = h\nu - p \tag{2.4}$$

where $\nu$ is the frequency of the incident wave of light, $p$ is a constant characteristic of the metal, and $h$ is a factor that seems to be constant for all metals and all wavelengths (Glassner 1995). This expulsion of electrons caused by light is what we call *the photoelectric effect*.

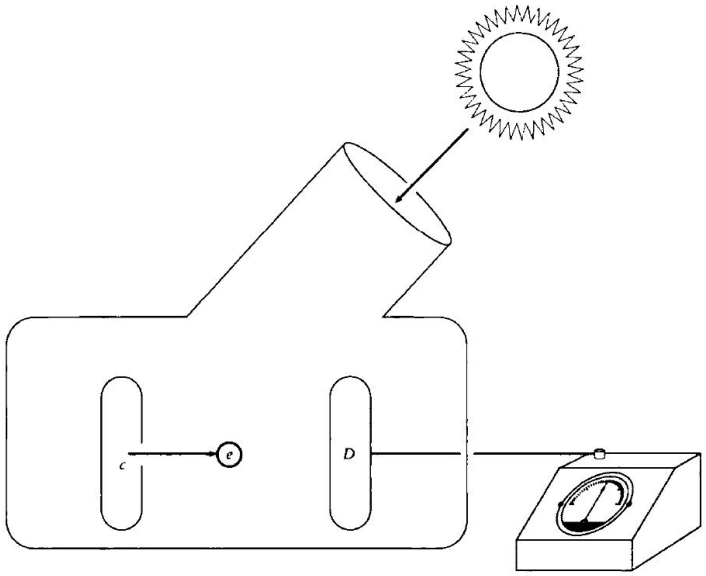



*Fig. 10:* "Apparatus for observing the photoelectric effect. The cathode is *c*, the detector is *D*." (Glassner 1995)

If we experiment with different light "strengths", or, to be more precise, different wave *amplitudes*, strange things start to happen. First of all, the energy $E$ of the electrons is found to be independent of the light beam's amplitude. For example, illuminating the metal with a 40-watt light bulb rather then an otherwise identical 20-watt light bulb causes more electrons to be freed, but their individual energy does not change. According to the wave model, this doesn't make much sense, since a stronger wave should impart more energy to the electrons. Another surprising fact is that even extremely low amplitudes of light free some

5 Note that the energy of the electron $E$ is a result of its *movement* as it's expelled, and is called *kinetic energy*.

electrons. Again, thinking of the light beam as a wave, we would be led to believe the incident energy to be so low that it gets spread *all over* the cathode. Consequently, there wouldn't be any point with enough energy to free an electron[6] (Glassner 1995).

In 1905, Einstein advanced the hypothesis that the energy flowing along the incident beam is quantized into small, individual packets called *photons*. He supposed that to liberate an electron from a surface, there's a minimum amount of energy that's required. A photon that contains this amount of energy transfers it to the electron at the time of collision, resulting in the electron being freed. On the other hand, photons with energies lower than this minimum cannot induce electron emission (Glassner 1995).

Going back to the two phenomena mentioned earlier, we can now see that each photon actually interacts with each electron *independently*. Therefore, if we increase the incident beam's energy, that is, the *number* of photons, we don't increase the energy of the emitted electrons, we just produce *more* of them. On the other hand, even a beam with low energy (less photons) will trigger the emission of some electrons, if the energy carried by individual photons is over a certain threshold.

Einstein's interpretation of the *photoelectric effect* was based on the work of Max Planck, who in 1900 had proposed a mathematical construct to calculate *blackbody radiation* (we'll mention more about *blackbodies* later). Planck had considered electromagnetic energy as *released* and *absorbed* in discrete packets, with the energy of a packet being directly proportional to the *frequency* of the radiation: (Stark 2025)

$$E = h\nu \tag{2.5}$$

Where the constant $h$ has since been called *Planck's constant*. Einstein's contribution, which earned him the Nobel price in 1921[7], was interpreting *light itself* as composed of energy packets, that is, *photons*.

So, the particle model is better suited to explain phenomena such as the *photoelectric effect* and *blackbody radiation*. It is still, however, indisputable that light is also a wave, due to the manifestation of *interference effects* such as the ones present in Young's experiment. But how can both theories be right?

6 More precisely, the energy wouldn't be higher than the constant $p$ from (2.4)

7 See https://www.nobelprize.org/prizes/physics/1921/summary/ (last accessed: 12.03.2025)

*Quantum mechanics* solves this dilemma by stating that light exhibits *wave-particle duality*. Actually, this is also true for *electrons* and other discrete bits of matter; that is, they also possess wave properties such as *wavelength* and *frequency*. For example, in modern versions of the double-slit experiment, electrons have been shown to form the same interference pattern as the one demonstrated originally by Young (Stark 2025).

But how does this work? Briefly put, each particle has a *wave function* associated with it, which should be interpreted *statistically*, as suggested originally by the physicist Max Born. In fact, the *wave function* is required to calculate the *probability* of finding a particle at any point in space (Stark 2025).

We now close our discussion on the dual nature of light. We started with what seemed like a simple question and ended up in the realm of modern physics, which have challenged not only earlier theories in the field, but our own perception of reality. It seems that to answer the question "what is light?" we need embrace a vision of the Universe where concepts that baffle the intellect, like particles being also waves, and waves being also particles, are a reality.

## 2.2 THE ELECTROMAGNETIC SPECTRUM

From here on, we'll alternate freely between the wave and particle models. Let's now consider the first. Changing the wavelength of an electromagnetic radiation, we can span a broad *spectrum*, from very long waves to very short. This spectrum can be divided in *bands*, each with a different name. For the sake of curiosity, we'll provide a short summary of them, based on the NASA Science article series *The Electromagnetic Spectrum* (National Aeronautics and Space Administration, Science Mission Directorate 2010).

### 2.2.1 *From Radio Waves to Gamma Rays*

The longest waves in the spectrum are *radio waves*. Their existence was proved by Hertz in the late 1880s, and their wavelengths range "from the length of a football to larger than our planet". As suggested by the name, they are used to transmit *radio signals*.

At the higher frequency end of the radio spectrum, *microwaves* can be found. They are distinguished from radio waves because of the technolo-

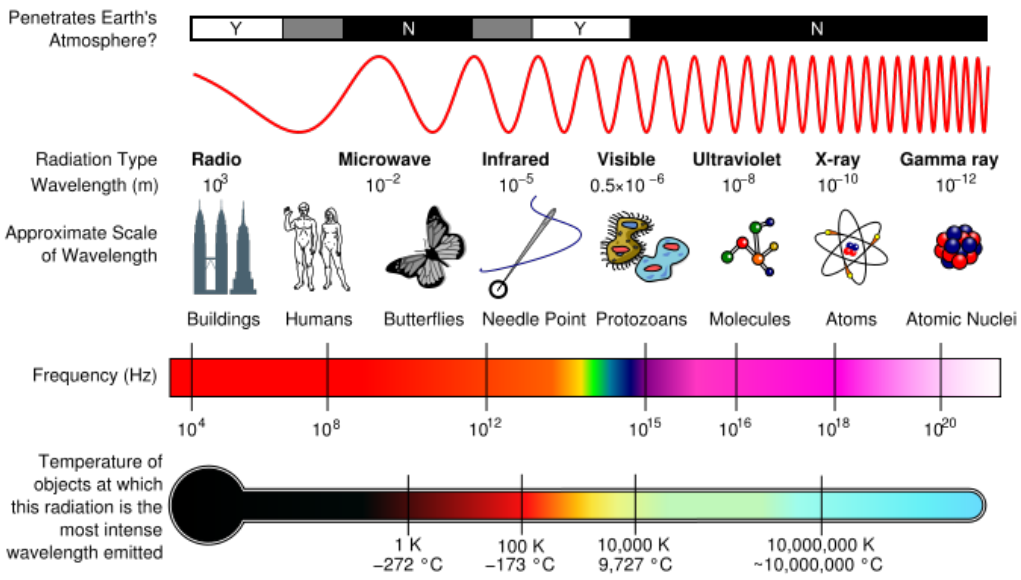



*Fig. 11:* The electromagnetic spectrum. Source: Wikimedia Commons. Licensed under the CC BY-SA 3.0 license.

gies used to access them. They are used by microwave ovens to cook food.

Really hot objects, like fire, emit *visible light*. Other objects, such as humans, are not as hot and only emit only so-called *infrared waves*. Infrared waves are just beyond the visible spectrum of light; although the human eye cannot see them, we do sense them as *heat*.

*Visible light* falls roughly in the wavelengths

$$380 < \lambda < 700$$

measured in *nanometers* (nm).
There is a direct correlation between the wavelength of a visible ray of light and the color that our eyes see in response. Short wavelengths look blueish (the shortest waves look violet), and long wavelengths red. In the middle, there's green.

*Ultraviolet light* (*UV*) has shorter wavelengths than visible light. The Sun is a source of ultraviolet radiations, some of which can be harmful to living organisms. Luckily, the vast majority of these are absorbed by Earth's atmosphere.

*X-rays* possess extremely small wavelengths, between 0.03 and 3 nm (that is smaller than a single atom of many elements). They are used for

medical diagnosis, since bones are dense and absorb more x-rays than skin does, as rays pass through the body.

The smallest wavelengths (and highest energies) are reserved to *gamma rays*. On Earth, they are generated from events such as nuclear explosions and lightning.

We'll now focus a bit more on *visible light*, to explain how we perceive colors and, therefore, how they should be treated inside a computer program.

### 2.2.2 *Spectral Distributions and Color*

When we'll be defining physical quantities to measure light, we'll find it necessary to talk in terms of *spectral distributions*. We'll also refer to these, simply, as *spectra* (*spectrum* for the singular). A *spectral distribution* is a distribution function that gives us the amount of light depending on wavelength (Pharr, Jakob, and Humphreys 2023).

To make an example, let's consider the energy of a ray of light. We know by now that it can be decomposed into discrete packets, each of which is carried by a photon. We are also aware of the fact that, given a light wave's frequency $\nu$, the energy $E$ of its photons is (from (2.5))

$$E = h\nu$$

We've mentioned earlier that a light wave can be considered as the sum of many harmonic waves, each with its own wavelength $\lambda$. A light wave's frequency $\nu$ can be calculated from its wavelength $\lambda$ as

$$\nu = \frac{c}{\lambda} \tag{2.6}$$

where $c$ is the speed of light in a vacuum (Stark 2025). Consequently, a light wave "contains" many frequencies, just like it does with wavelengths. This means that it's composed of many types of photons; that is, photons with different energies. These individual energy values are summed up to obtain the total energy carried by light.

Let's now see how the light's energy could be used to determine its color. To do so, we follow a simplified overview from the book *An Introduction to Ray Tracing* (Glassner 2019).

So, we know that light is composed of photons at many different frequencies (or, equivalently, wavelengths). When these photons reach

our eyes, each of them will be "deciphered" as a *different color* from the visible spectrum (based on the wavelength). What happens next is that these base colors are blended together by the eye. The resulting coloring will be tinted towards the base colors for which there were more photons (of that wavelength). This means that, for example, if most of the light's energy was carried by photons with long wavelengths, it will appear more red (it will be more intense at the corresponding wavelength). On the contrary, if energy came mostly from short-wavelength photons, the light will look more blueish.

So, in our case, we don't really care for the light's *total* energy: it only reveals the light's "total" intensity. What we actually want to know is how much energy there is *per-wavelength* (we're asking ourselves "how red is this light?", or "how blue?"). Thus, we want to be able to *distribute* the total energy between the range of wavelengths that the light ray is made of. This is exactly the purpose of a *spectral distribution function*; in this case, it's called *spectral energy distribution* (or *SED*)[8].

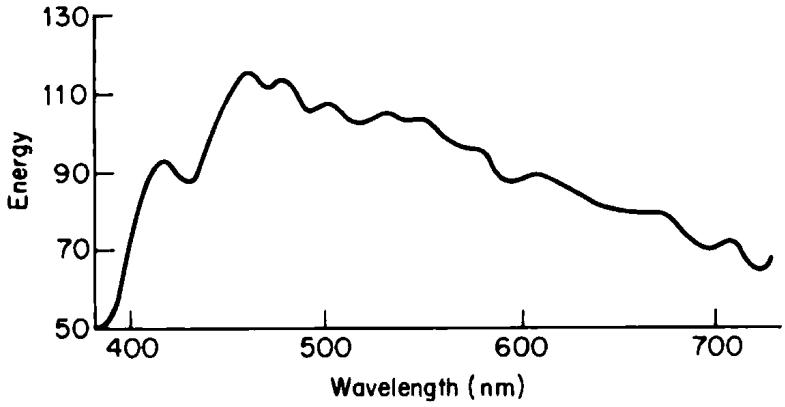



*Fig. 12:* "The spectrum of CIE Standard Illuminant D6500, which approximates sunlight on a cloudy day." (Glassner 2019)

Putting it simply, we thus say that colors are the result of a "blending" operation, which can be predicted through spectral distributions. At this point, we might be led to think that colors *are* spectral distributions. However, we should be careful not to mix up a purely physical concept such as spectra with the *experience* of colors, which is strongly related to human *perception*. In fact, the human visual system actually employs

8 From the NASA IPAC Teacher Archive Research Program (NITARP) wiki, at: https://coolwiki.ipac.caltech.edu/index.php/SED_plots_introduction (last accessed: 12.03.2025)

some tricks that we can exploit for our own rendering purposes. Most notably, we can avoid representing each color as a distribution over all wavelengths.

Since the 1800s, numerous experiments have proven that all colors perceived by the human eye can be represented using three scalar values. This is called the *tristimulus theory*, and it's been confirmed by the study of *cone cells* (Pharr, Jakob, and Humphreys 2023). *Cone cells* are light-sensitive cells on the retina which allow us to see colors. There are three types of cone cells, which we can call *short*, *medium* and *long*, after the wavelengths of light they are most sensitive to. To each of these cone cells, we can associate a *response function*, which describes how strongly one type of cone "responds" to beams of light of different wavelengths. Let's call these functions $k_s(\lambda)$, $k_m(\lambda)$ and $k_l(\lambda)$ (for short, medium and long cones respectively). Since each ray of light with a specific (or "single") wavelength $\lambda$ produces three response values on the retina, one for each type of cone, we can visualize this response as a single point in a 3D space:

$$[k_s(\lambda), k_m(\lambda), k_l(\lambda)]^t$$

We call this space a *color space*[9]. Light rays which don't have a specific wavelength (they have "multiple" wavelengths) produce colors that are just linear combinations of the "simpler" ones; that is, the ones that *did* come from light rays with well-defined wavelengths (so, the colors are still just points in the color space) (Gortler 2012). Note that, since we're just speaking of coordinates in a 3D space, we can easily describe the same color using a different basis (any three linearly independent base colors).

It is well known that colors of computer pixels are the result of blending together red, green and blue (*RGB*), and now we can clearly see why this works. In fact, red, green and blue constitute the basis of the *RGB color space*.

In our implementation of Physically Based Rendering techniques, we'll be using RGB to represent *spectral quantities*; that is, values that vary with wavelength, like the energy of a light ray. Advanced rendering engines, such as *pbrt*, actually employ *spectral rendering*, where full spectra are

9 We should mention that what we're talking about here is called *retinal color*. The *perceived color*, the one we experience and base judgements upon, is actually the result of some non-trivial processing steps done by the human visual system. For example, our brains make us perceive the colors of objects as more or less constant, even under different illuminants. (Gortler 2012)

used for higher precision (Pharr, Jakob, and Humphreys 2023). We'll consider the loss of information caused by RGB color spaces as negligible for visual aspects.

Before we conclude this section, we should also mention that digital images are, in general, not stored using the RGB format. More commonly, computers use the *sRGB color space*, which involves a non-linear transformation from RGB, for the purpose of storing color values more efficiently in the computer's memory (Gortler 2012). This, however, in rendering, will be our concern only at the moment of reading or storing an image from/to memory, while our physically based calculations will be done in the linear, RGB space.

## 2.3 LIGHT-MATTER INTERACTIONS

Now we have a pretty good mental model of light, but we should remember that we also need one for materials. To build material models, we need to talk about *matter*. Matter can either *emit* light, as a light source, or *respond* to incident energy (for example, through *reflection*). We'll consider both cases.

For the moment, however, we should settle on a model to describe matter. We'll be considering the simplest form of it: the atom.

It is well-known that the classical model of the atom includes a center, the *nucleus*, composed by small particles called *protons* (with positive charge $+e$) and *neutrons*. The nucleus is also orbited by another group of much smaller particles, called *electrons* (with negative charge $-e$).

In quantum mechanics, these particles don't have a precise location. Basically, unless a particle is observed, it is *nowhere*; particles are only associated with *probabilities*. For example, instead of individual electron particles, we have a *cloud of electrons*, with the cloud being more dense where electrons are more likely to be found (Glassner 1995) (see Figure 13).

Electrons are very susceptible to external influences, which make them move between different *states* by *absorbing* and *releasing* energy, often in the form of photons (Glassner 1995). The state of an electron at any time is given by a set of four *quantum numbers*. The *principal quantum number* $n$ characterizes the electron's energy, with $n \epsilon \{1, 2, 3, ...\}$. Higher values of

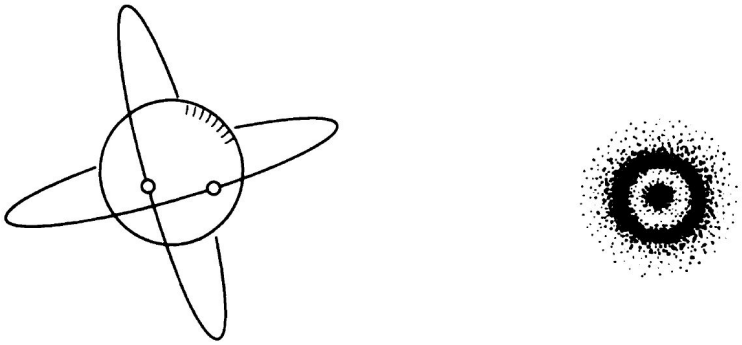


*Fig. 13:* The classical model of the atom (left) compared to a more modern view (right). (Glassner 1995)

$n$ indicate that the electron is more likely to be found far away from the nucleus[10].

The electrons of a neutral atom normally inhabit *ground states*. Apart from these, however, there are many higher-order *excited-state energy levels*. Above, we find the *ionization continuum*, where electrons become disassociated from the atoms and are free to move away (Glassner 1995). We have already seen this in the photoelectric effect, where electrons where expelled from atoms with kinetic energy $E$.

Finally, we can say a couple things about molecules. A molecule is an electrically neutral, stable combination of two or more atoms. The energy transitions for electrons change when they are involved in a bonding process that brings atoms together into molecules. Also, the size (much greater compared to atoms) and translational/vibrational energy of molecules affects what energy is absorbed and emitted by their atoms (Glassner 1995).

### 2.3.1 *Emission*

We can classify the emission of light into two distinct types: *thermal* and *luminescent* (Glassner 1995).

*Thermal emissions* are caused by heat. We can explain this phenomenon as follows. It is known that, when the temperature of an object is above absolute zero, its atoms are moving. As is described by *Maxwell's equations*, the motion (more precisely, the *acceleration*) of atomic particles that hold

10 From *orbital*. Encyclopaedia Britannica. URL: https://www.britannica.com/science/orbital (last accessed: 12.03.2025)

electrical charges causes objects to emit electromagnetic radiation (Pharr, Jakob, and Humphreys 2023). In fact, we've already mentioned that humans emit light at infrared frequencies, and that, to emit light that is visible, an object needs to be much warmer. This is because, to put it simply, the atoms need to move *faster*. An incandescent light bulb is an example of a *thermal radiator*; it contains a small filament which, when heated by the flow of electricity, emits electromagnetic radiation (Pharr, Jakob, and Humphreys 2023).

When talking about thermal emission, it is common to run into the term *blackbody* (actually, we already have while discussing the particle nature of light). Blackbodies are (theoretical) perfect absorbers; they absorb all incident light, without reflecting any of it. They are also perfect emitters; that is, they convert *power* to electromagnetic radiation as efficiently as physically possible[11]. Even though true blackbodies are not physically realizable, the idea of them is still quite useful. This is because *Planck's law* specifies a way to, given a temperature, determine the emission of a blackbody as a function of wavelength. Taking a non-black body emitter, if the shape of its emitted spectral distribution is similar to the one of a blackbody at some temperature, we say that the emitter has the corresponding *color temperature*. Color temperatures over 5000 K are generally described as "cool," while those at 2700–3000K "warm". For example, the CIE standard illuminant A, which represents the average incandescent light, corresponds to a blackbody radiator of about 2856 K (Pharr, Jakob, and Humphreys 2023).

*Luminescent emission* is due to energy stored (perhaps for a very short time) in the material, and is determined primarily by factors different that temperature, although temperature can affect the material (Glassner 1995). In the context of *luminescent emission*, it's important that we mention *phosphors*. Speaking broadly, a *phosphor* is a material that absorbs some form of energy, like an electromagnetic wave or an electron beam, and then emits it as light over some period of time (Glassner 1995). When certain phosphors *luminesce* from *electron excitation*, the process is called *electroluminescence*. This is used in the production of TV screens and computer monitors[12].

Given our model of the atom, we can actually explain luminescence. We already know that electrons tend to move through different states,

11 This makes sense if we think of emission as of the reverse operation of absorption. (Pharr, Jakob, and Humphreys 2023)

12 From *phosphor*. Encyclopaedia Britannica. URL: https://www.britannica.com/science/phosphor (last accessed: 12.03.2025)

absorbing and emitting energy in the process, and that this energy is often in the form of photons. When a photon is absorbed, it generally disappears completely (a photon cannot exist at rest, and cannot transfer only *some* of its energy), and the electron transitions into what we called an *excited-state energy level*. Generally, this can't be kept up for long, so after a while the electron will drop back to its *ground state*, emitting the difference in energy between the two states as a *new* photon. Most times, this happens under $10^{-8}$ seconds. This number is used to divide phosphors into two categories. If the material responds to incident energy by reradiating most of it within $10^{-8}$ seconds, we call it *fluorescent* (this is what happens, for example, in *fluorescent lamps*[13]). If the emission persists longer, we're talking about *phosphorescence* (Glassner 1995).

### 2.3.2 *Response*

We now consider some common events that can be triggered when light strikes matter.

To do so, we'll first move back to a simpler model of light; we won't be thinking of waves or particles, but of *rays*. This is the point of view of *geometrical optics*, where we describe the interaction of light with objects much larger than its wavelength, allowing for this abstract idea of *rays of light* (Pharr, Jakob, and Humphreys 2023).

To predict how matter will respond to light, we can use a property called the *index of refraction* (*IOR*). This is a complex number, where its real part describes how much the matter slows down the speed of light (which, as we know, is equal to c in a vacuum), and its imaginary part determines whether the light is *absorbed* as it propagates in the material (Hoffman 2012). We should mention that by *absorption* we mean the conversion of the energy of light into some other energy form, internal to the atoms. One example for this is *thermal energy*, which derives from the *movement* of atoms[14].

We now consider matter as a *medium* through which light is propagating (just like it does through air, for instance). We can distinguish two types of media: *homogeneous* and *heterogeneous*.

13 From *fluorescent lamp*. Encyclopaedia Britannica. URL: https://www.britannica.com/technology/fluorescent-lamp (last accessed: 12.03.2025)

14 From The IBM article *What is thermal energy?*, at https://www.ibm.com/think/topics/thermal-energy (last accessed: 12.03.2025)

We say that a medium is *homogeneous* if it presents a uniform index of refraction (at the scale of the light's wavelength). Water and glass are examples of this. We also say that they're *transparent* media, since they don't absorb visible light in any significant way. In fact, their refraction indices have a very low imaginary part for visible light wavelengths (Hoffman 2012).

*Fig. 14:* Over large distances, the slight absorptivity of water becomes noticeable. Image by Andreas Schau, from Pixabay.

On the other hand, if the homogeneous medium *does* absorb light from the visible spectrum significantly, what happens is that the light's intensity dies down as it moves farther into the medium. However, the direction of light does not change.

We should note that these properties are always in function of *scale*. For instance, on the scale of many feet of distance, water actually absorbs quite a bit of light, especially red colors (Hoffman 2012) (see Figure 14).

Let's turn now to *heterogeneous media*, which have variations in the index of refraction. If the index of refraction changes slowly and continuously, then the light bends in a curve. If it changes abruptly, the light *scatters*, meaning that is splits into multiple directions. We can think about light encountering microscopic particles, which induce regions in space where the index of refraction becomes different. The type of particle affects the

distribution of the scattered light's directions (Hoffman 2012) (see Figure 15).

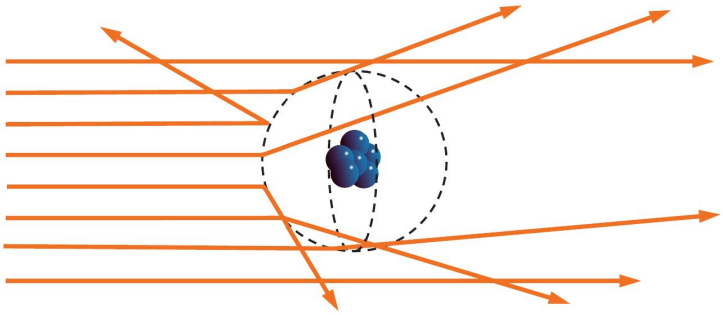


*Fig. 15:* "Particles cause light to scatter in all directions." (Hoffman 2012)

Media that are *translucent* (or *opaque*) contain such a high density of scattering elements that, as a result, light gets scattered in completely random directions.

*Fig. 16:* Example of an opaque medium. (Hoffman 2012)

Most media both scatter and absorb light, at least to some extent.

Now, we want to know what happens when light, propagating through air, hits an object's surface. An analytical solution to the problem can be found if we consider the surface of the object as infinite and perfectly flat (relatively to the light's wavelength). In this case, we get special solutions to Maxwell's equations, called the *Fresnel equations*. These equations tell us that the material causes light to split only in two directions: *reflection* and *refraction* (see Figure 17). They also describe the portions of reflected and refracted light (Hoffman 2012).

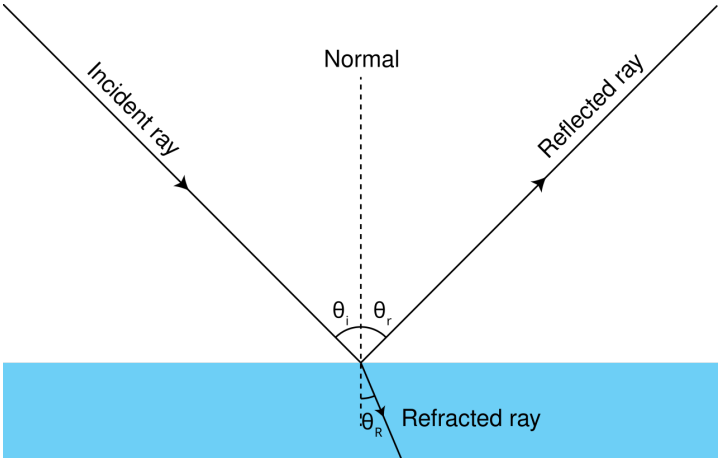



*Fig. 17:* An incident ray gets split into a reflected ray and a refracted ray. Source: Wikimedia Commons.

The angle formed by the reflected ray of light with the surface *normal*[15], that is, the *angle of reflection*, is equal to the angle of the incoming ray. The *angle of refraction* depends on the refractive index, and can be found through *Snell's law*.

### 2.3.3 *Reflection and Subsurface Scattering*

If real-world surfaces were really perfectly flat, computing realistic reflections would be extremely simple, in the sense that we would just reflect the incident light vector. In most cases, however, surfaces present irregularities larger than the light's wavelength, but too small to be seen by the eye. To account for this, we can model the surface as a large collection of smaller optically flat surfaces, which will cause light to be reflected in various directions, and various amounts for each direction. Notice in Figure 18 how these "micro-surfaces" make the overall surface more irregular, that is, more *rough*. This resulting surface *roughness* will cause reflections to be more *blurry*.

Let's now consider the refracted light: what happens to it depends on the type of material. Metals immediately absorb refracted light (more precisely, their free electrons do). As for non-metals (also called *dielectrics* or *insulators*), light behaves the way we would expect, in part getting absorbed and in part scattered. Most times, the refracted light gets scattered

15 The *normal* to a surface at a given point is a vector perpendicular to the surface in that point.


*Fig. 18:* Light gets reflected off a surface, modeled as a collection of smaller, optically flat surfaces. Source: *PBR Theory*, Learn OpenGL tutorial series. Licensed under the CC BY 4.0 license.

so much that it's re-emitted out of the surface, in general, at a different point. This is called *subsurface scattering* (Hoffman 2012).

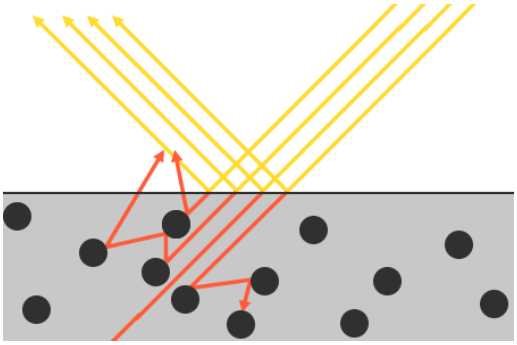


*Fig. 19:* The refracted light scatters until it re-emerges from the surface. Source: *PBR Theory*, Learn OpenGL tutorial series. Licensed under the CC BY 4.0 license.

### 2.3.4 *Reflection, as seen by Electrodynamics*

Here, we'll form a quick intuition on how the phenomenon of reflection fits into the framework of *classical electrodynamics*. This means that we'll pretend again that light is simply a wave. Addressing the reflection of *photons* is a problem that belongs to *quantum electrodynamics* (Feynman, Leighton, and Sands 2011), and would constitute a topic too complex for this thesis.

We'll be going through a very brief summary of the Feynman lectures on electrodynamics (Volume I, Chapters 28-30) (Feynman, Leighton, and Sands 2011).

Remember that Maxwell's equations tell us that an accelerating charge produces an electromagnetic field. To be more precise, we consider the case where a charge is moving nonrelativistically at a very large distance.

We can also have many charges instead of one. If we can make them move together, all in the same way, we can get the resulting field just by summing the effects of the individual charges. This is a consequence of the *superposition principle*, which we have already come across.

Let's be more concrete now, and conceive an experiment. We have two pieces of wire connected to a generator (as shown in Figure 20). The generator makes a *potential difference*, which first pulls electrons from piece A and pushes them into B, to then almost immediately reverse the effect, and make the electrons move from B into A. This way, the charges accelerate up and down in the wires. This is the same as having a single charge (summing the effects of the individual charges) accelerating up and down as though A and B were a *single* wire. When the wire is very short compared to the distance traveled by light in one oscillation period, it is called an *electric dipole oscillator*.

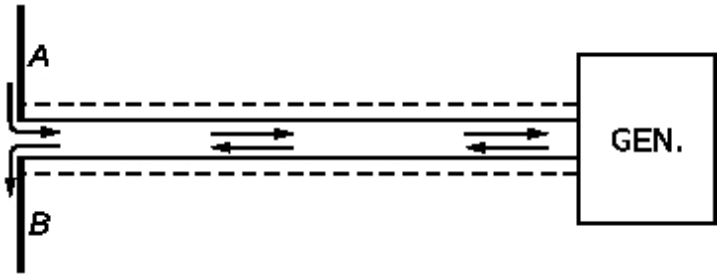



*Fig. 20:* "A high-frequency signal generator drives charges up and down on two wires." (Feynman, Leighton, and Sands 2011)

Our accelerating charge generates an electric field, which we pick up with an identical instrument; that is, another pair of wires just like A and B. We call this a *detector*. The incoming electric field will produce a force which will pull the electrons up on both wires or down on both wires. The resulting signal is then detected by a *rectifier* mounted between A and B, and amplified by an *amplifier*, so that we can perceive it in some form (like sound).

With this setup, we can observe that the field is strongest when the detector is parallel to the generator, and is 0 when they are perpendicular. In fact, what matters when calculating the field is the acceleration of the charge *projected* on the plane *perpendicular* to the *line of sight*.

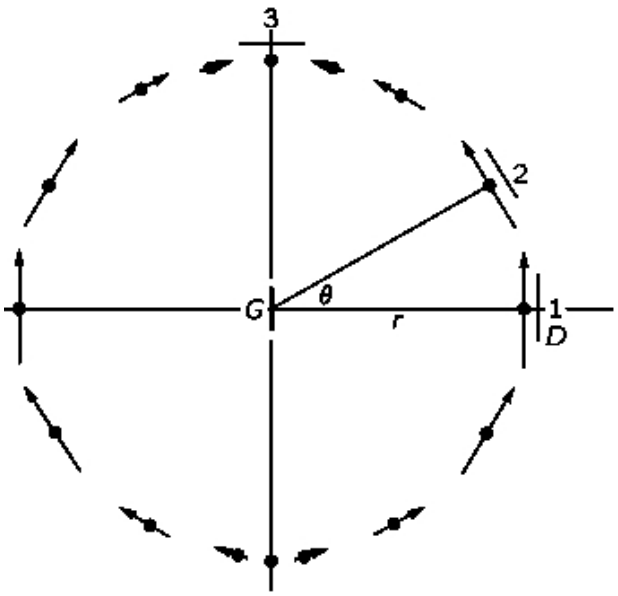



*Fig. 21:* The detector *D* finds a strong field when parallel to the detector *G* at point 1. The field is 0 when the detector is at 3. (Feynman, Leighton, and Sands 2011)

We now consider a situation where there are $n$ oscillators, equally spaced at a distance $d$, all with the same amplitude, and lying on the same line (Figure 22). Their phases are different to the observer, both because of some intrinsic shift in phase from one to the next, and because we are looking at them at different angles, resulting in different distances traveled by the individual waves to reach us.

The intrinsic shift in phase, one to the next, is $\alpha$. If we are observing in a given direction $\theta$ from the normal, there is an additional phase difference contribution $2\pi d\sin\theta/\lambda$, because of the time delay between each successive oscillator. Thus, The sum of the waves becomes

$$\begin{aligned} R =& A[\cos\omega t + \cos(\omega t + \phi) + \cos(\omega t + 2\phi) + ... \\ &+ \cos(\omega t + (n-1)\phi)] \end{aligned} \tag{2.7}$$

where $\phi = \alpha + 2\pi d\sin\theta/\lambda = \alpha + kd\sin\theta$ is the net phase difference between one oscillator and the next one.

If we consider the intensity of the resulting wave (2.7), we notice that we have a maximum when $\phi = 0$ (all the oscillators are in phase). We also get maxima, in general, when $\phi = 2\pi m$, where $m$ is any integer (successive waves are out of phase by a multiple of $2\pi$).

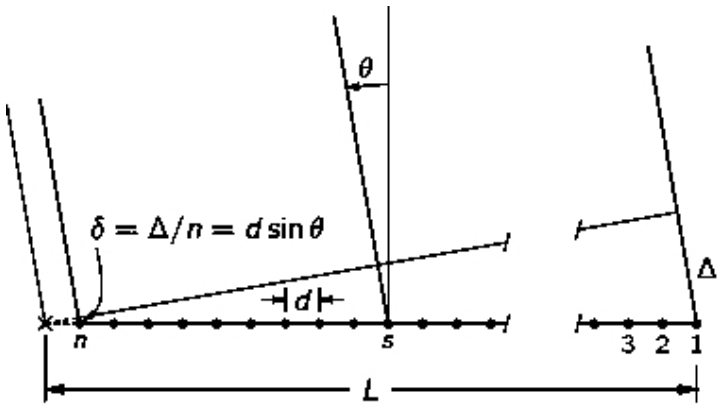



*Fig. 22:* "A linear array of n equal oscillators." (Feynman, Leighton, and Sands 2011)

Now suppose that we have a source of electromagnetic radiation that's far away, practically at infinity, and that its light is coming in at an angle $\theta_{in}$. The corresponding electric field will cause the electrons to move up and down in the wires, and in moving, they will generate *new* waves. This phenomenon is what we call *scattering*. In the case of a material hit by a light wave, its electrons will also start moving just like for the oscillators, generating new waves.

In *diffraction gratings*, instead of wires we have a flat piece of glass with tiny notches in it, each of which represents a source of slightly different scattered waves. Diffraction gratings can be used to split light based on its wavelength (that is, its color). In one of its forms, a diffraction grating is composed of just a plane glass sheet (transparent and colorless) with scratches on it, all equally spaced.

So, we have a light source at infinity, which sends in light to the scratches at an angle $\theta_{in}$. We are interested in the scattered beam at the angle $\theta_{out}$, which is the angle from which we are observing the grating (we called it $\theta$ earlier). The incident light will hit the scratches one after the other, with some time delay, resulting in a phase shift between adjacent scratches. This can be calculated as $\alpha = -2\pi d sin\theta_{in}/\lambda$. Putting it all together, we have:

$$\phi = 2\pi d sin\theta_{out}/\lambda - 2\pi d sin\theta_{in}/\lambda$$

To have maximum intensity, we know that $\phi$ should satisfy $\phi = 2\pi m$:

$$2\pi m = 2\pi d sin\theta_{out}/\lambda - 2\pi d sin\theta_{in}/\lambda$$

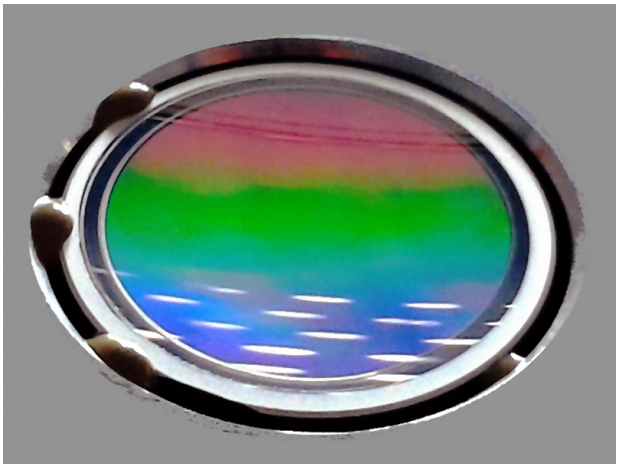


*Fig. 23:* A parabolic reflective diffraction grating. Source: Wikimedia Commons. Licensed under the CC BY-SA 3.0 license.

Multiplying both sides by $\lambda/2\pi$, we get

$$\lambda m = d\sin\theta_{out} - d\sin\theta_{in}$$

Now, if $d$ is less than $\lambda/2$, this equation can have no solution except $m = 0$. In this case, we have

$$\sin\theta_{in} = \sin\theta_{out}$$

Which may mean two things: either $\theta_{out} = \pi - \theta_{in}$ or $\theta_{out} = \theta_{in}$.

- In the first case, it's as if the light goes "right through" the grating.
- In the second, we have *reflection*: the *angle of incidence* is equal to the *angle of scattering*.

We can now end this chapter with a solid, physical interpretation of reflection, which goes as follows. When light hits a surface, it generates motion in the atoms (or electrons, to be more precise) of the object, causing them, in turn, to regenerate *new* electromagnetic waves. These waves will add up, resulting in a *single* wave, produced by the object as a whole. If the spacing of the scatterers is small compared with one wavelength, the reflected wave will be strongest in intensity (only) at the angle $\theta_{out} = \theta_{in}$ (ignoring light that goes "right through"). Thus, we consider the direction of scattering as symmetric to the direction of incident light, with respect to the surface normal.

# 3

# MATHEMATICS FOR RAY TRACING

Whereas the previous chapter was mostly a description of light phenomena in the physical world, here we wish to put together more abstract, mathematical ideas that can let us *imitate* the behaviour of light inside a computer. Our final goal is to simulate light reflections in a way that makes virtual materials feel "real", and the abstraction process that we'll be going through will lead us exactly there.

First of all, it should be noted that practically all nontrivial graphics programs require a geometric foundation (at least to *some* extent), this foundation being made of *vertices*, *polygons*, and similar constructs (Pharr, Jakob, and Humphreys 2023). In our case, we'll be building objects out of two simple primitive shapes: triangles and spheres. At an even lower level, we'll be dealing with *points* and *vectors* in 3D space. Vectors will also, and more importantly, be needed to define *rays of light*.

Once a 3D scene is built, it needs to be shown to the viewer. This is where *ray tracing* will come in. Its job will be to simulate the path taken by light, from the *light sources* to the *virtual camera*, after (potentially) being *reflected* by an object[1]. Light will travel along *rays*, and the reflection calculations will be conducted with the help of a powerful mathematical tool: the *Bidirectional Reflectance Distribution Function* (*BRDF*).

## 3.1 RAY TRACING FUNDAMENTALS

In this section we'll get an idea of how ray tracing works, and what makes it a powerful rendering technique. We'll also go through the fundamental mathematical operations that simple *ray tracers*[2] rely on.

1 Actually, we'll see that in ray tracing we *reverse* this path.
2 A *ray tracer* is a renderer that uses ray tracing.

### 3.1.1 *What is Ray Tracing*

We should point out that ray tracing is only *one* of the various rendering techniques commonly employed in computer graphics. Another especially important method is called *rasterization*, which is at the base, for example, of the widely-adopted *OpenGL* graphics API[3]

In a rasterization-based rendering environment such as OpenGL, 3D objects are represented as collections (also called *meshes*) of triangles, which are then individually projected onto the screen to figure out what pixels are covered by them[4], to finally determine the appropriate colors of those pixels by performing "some" computation. These final computations (which are also referred to as *shading*) typically involve equations that simulate the way that light reflects off of the triangles, based on the materials they're made of (Gortler 2012).

Through the use of *z-buffering*, OpenGL draws on screen only the triangles that are closest to the viewer. The way this works is that OpenGL goes though the list of triangles, one by one, and, while coloring the pixels they cover, it also memorizes a *distance* for them. This distance is relative to the virtual camera, and is memorized in a buffer called the *depth buffer* (or *z-buffer*). Successive triangles are then drawn only if they are closer than what was drawn earlier, and this is checked on a per-pixel basis in the depth buffer. This way we can have triangles drawn on top of each other correctly. We refer to this as *z-buffering* because, in our 3D space, we consider the viewing direction to be along the $z$ axis.

We can thus summarize rasterization in the following form (Algorithm 1). (Gortler 2012)

**Algorithm 1** Rasterization

```
initialize z-buffer
for all triangles
    for all pixels covered by the triangle
        compute color and z
        if z is closer than what is already in the z-buffer
            update the color and z of the pixel
```

A nice property of this algorithm is the fact that each triangle in the scene is "touched" only once (Gortler 2012).

3 See Chapter 4 for more about *OpenGL*.

4 The process of taking a triangle and filling in the pixels it covers is called the "rasterization step". (Gortler 2012)

*Ray tracing* represents an alternative to rasterization, based on ideas that are closer to the physics of light. To put it intuitively, we can think of the (basic) ray tracing algorithm as a result of *inverting* of the two **for all** loops from Algorithm 1: (Gortler 2012)

**Algorithm 2** Ray Tracing

```
for all pixels on the screen
    for all objects seen in this pixel
        if this is the closest object seen at the pixel
            compute color and z
            set the color of the pixel
```

Note that here, instead of "triangles", we're using the term "objects". In fact, in ray tracing, our scene can be made up of any geometrical primitives we want. For instance, we can render smooth spheres without having to "dice them up" into triangles (Gortler 2012). So, when we say "object", we basically mean any geometrical shape.

The objects "seen" by a pixel are found by casting a *ray* in the 3D scene, and checking what geometries are *intersected* by it. The rays are cast from the camera, in the direction of a pixel (later, we'll understand better what this means), and that pixel is given the color of the closest object.

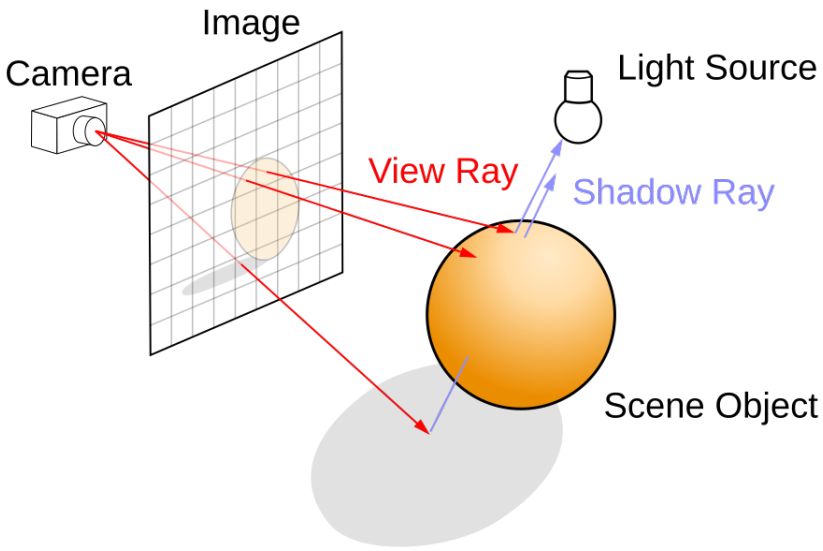



*Fig. 24:* Rays are cast through pixels and given the color of the intersected object. Source: Wikimedia Commons. Licensed under the CC BY-SA 4.0 license.

In general, in ray tracing rays are cast *recursively*. For example, once we've found the closest object hit by a ray, we can cast *another* ray, from the hit point towards a *light source* in the scene (also called a *shadow ray*), as shown in Figure 24. This way, we can determine whether (and how much) light reaches that point, or if it's in shadow (Gortler 2012).

Going back for a moment to the physics of light, we can see that what we're doing here is following light rays *in reverse*. In the real world, light is emitted by a light source, hits an object, gets reflected, and, finally, reaches the camera. We, however, retrace this path backwards so that we're sure to be following rays that have reached the camera (we're not in interested in the rest).

One of the clear computational disadvantages of ray tracing is that, every time we cast a ray into the scene, we have to go through *all* the geometric primitives (all triangles, all spheres...) to check which ones are intersected. In other words, we conduct a *linear search* between all the geometries to find the closest one. To make this process faster, we can use *acceleration data structures*, such as *Bounding Volume Hierarchies* (*BVH*), which turn the search into a *tree traversal process*, reducing its time complexity from $O(n)$ to $O(\log n)$ (Akenine-Möller et al. 2018).

On the other hand, a great advantage of ray tracing is its ease in simulating realistic light effects such as *reflection*, *refraction*, and *smooth shadows*. This is due to the possibility of shooting rays in any direction, from any point in the scene.

### 3.1.2 *Ray and Camera Models*

Up until now, we've been purposely vague about the practical aspects of ray tracing. For example, we've been talking about "virtual cameras" without really knowing what they look like inside a renderer. We'll now see how ray tracers can understand the concept of a camera by expressing it in a mathematical way.

We'll be using the simplest possible model of a camera, that is, a *pinhole camera* (Gortler 2012). In a pinhole camera, we imagine having an opaque surface with a hole on it (the so-called *pinhole*), of the size of a point. Light passes through this point and gets recorded on the camera's *film* (or *sensor*), which is just a plane behind the opaque surface. The image produced by this process is actually *mirrored*, and later needs to be *flipped* if we want to show the scene correctly. We can avoid doing this, though, by modeling the pinhole camera with the film plane *in front* (as

seen in Figure 24), at a distance which we'll call *focal length*[5]. Note that this wouldn't make sense in the physical world, but works just as well mathematically (Gortler 2012).

Another name we'll use to refer to the film plane is *viewport*. The viewport is divided into *pixels*, which, for us, are just equally-spaced points. Each pixel has two integer coordinates, $[x, y]^T$, with $x$ going right in the image and $y$ going down. "Right" and "down" are indicated by the viewport's horizontal axis $\vec{H}$ and vertical axis $\vec{V}$ ($V_u$ and $V_v$ in Figure 25). The top-left pixel has coordinates $[0, 0]^T$. Adjacent pixels are separated by a horizontal shift $\Delta\vec{h}$ or a vertical shift $\Delta\vec{v}$.

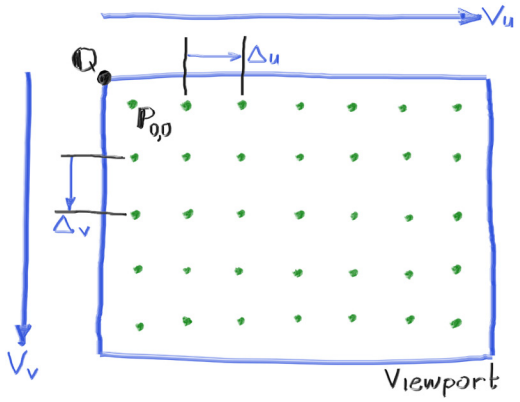



*Fig. 25:* "Viewport and pixel grid." (Shirley, Black, and Hollasch 2024a)

The camera has a *center*, which corresponds to the *pinhole*, and is oriented along a *coordinate frame* formed by its *right*, *forward* and *up* vectors (as shown in Figure 26).

To the camera, we also associate the focal length $d$ (the distance of the viewport from the camera center) and an angle $\theta$ called the *vertical field of view*. These two values, together with the aspect ratio $\alpha$ (width/height), are used to determine the *scale* of the viewport (Shirley, Black, and Hollasch 2024a). In essence, we can say that the vertical field of view $\theta$ determines a *cone of vision*, which grows as we move farther away from the camera[6]. The viewport will constitute a "slice" of this cone, at the distance $d$ (see Figure 27).

5 Following the terminology used by Peter Shirley in the book series *Ray Tracing In One Weekend*. (Shirley, Black, and Hollasch 2024a)

6 It's more of a "rectangular pyramid" than a "cone", actually.

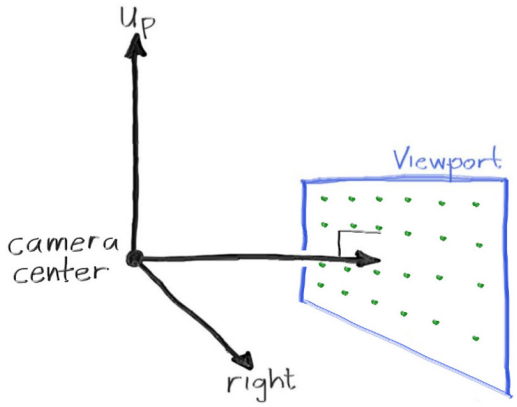



*Fig. 26:* "Camera geometry." (Shirley, Black, and Hollasch 2024a)

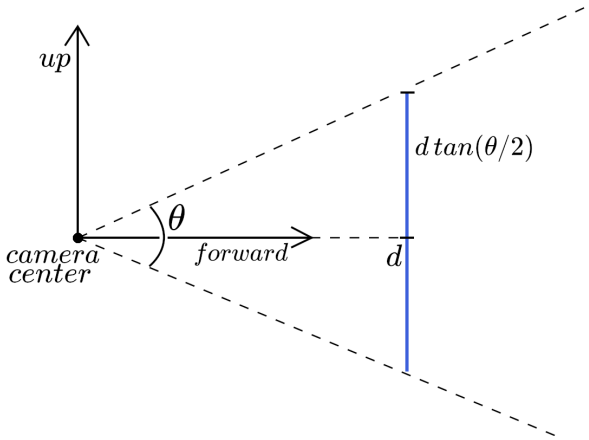



*Fig. 27:* Cone of vision and viewport geometry.

Thus, we can calculate the vertical size of the viewport $||\vec{V}||$ as:

$$||\vec{V}|| = 2d\,\tan(\theta/2)$$

And the viewport width $||\vec{H}||$ as

$$||\vec{H}|| = \alpha||\vec{V}||$$

$\vec{V}$ and $\vec{H}$ will then be

$$\vec{V} = -||\vec{V}||\,\vec{u}$$
$$\vec{H} = \quad ||\vec{H}||\,\vec{r}$$

Where $\vec{u}$ and $\vec{r}$ are, respectively, the *up* and *right* vectors of the camera's coordinate frame.

Finally, $\Delta\vec{u}$ and $\Delta\vec{v}$ can be found by dividing $\vec{V}$ and $\vec{H}$ by the number of pixels, horizontal and vertical respectively.

This is all we need to describe our simple, pinhole camera. At this point, we would like to "take a picture" with it; that is, to color the viewport's pixels. To do so, we need to know what the camera *sees* along each pixel. As already mentioned, we find this out by casting a *ray* through the pixel and into the scene.

We define a ray as a function $\tilde{p}(t)$, which outputs a point[7] along the ray at a distance $t \epsilon \mathbb{R}$ from the ray origin $\tilde{o}$: (Pharr, Jakob, and Humphreys 2023)

$$\tilde{p}(t) = \tilde{o} + t\vec{D} \qquad 0 \leqslant t < +\infty \tag{3.8}$$

Where $\vec{D}$ is the direction of the ray. Equation (3.8) is also called the *parametric form* of a ray.

To "shoot" the ray through a pixel with coordinates $[x, y]^T$, we pick the camera center as its origin point $\tilde{o}$, and calculate its direction $\vec{D}$ as

$$\vec{D} = (\tilde{p}_{00} + x\Delta\vec{h} + y\Delta\vec{v}) - \tilde{o}$$

Where $\tilde{p}_{00}$ is the position in 3D space of the pixel with coordinates $[0, 0]^T$.

As we know, what we do next is find the closest object intersected by the ray, and then give the pixel its color.

### 3.1.3 *Antialiasing*

If we send a single ray through each pixel, that is, we do *point sampling* (Shirley, Black, and Hollasch 2024a), we'll quickly find that our rendered images contain *visual artifacts*, also called *aliasing artifacts*. One example of this are *jagged patterns* (or "stair steps"), which manifest at shape edges (see Figure 28).

*Aliasing artifacts* occur when there is too much visual complexity to fit in a single pixel (Gortler 2012). If we think about it, it's obvious why this happens. We are trying to represent on a fixed, discrete grid of points something that has (potentially) infinite resolution. so we can't expect it to look 100% right. If we zoom in on an image, sooner or later this discretization process will become obvious to the eye.

However, these artifacts can be mitigated by taking *multiple samples* around each pixel, that is, sending multiple rays, and then averaging the

7 We'll be indicating points in 3D space as $\tilde{p}$, with a tilde on top, to distinguish them from vectors, as done by (Gortler 2012). Also, most of the times vectors will be capitalized (e.g. $\vec{V}$) to make the formulas more compatible with the code in Chapter 4.

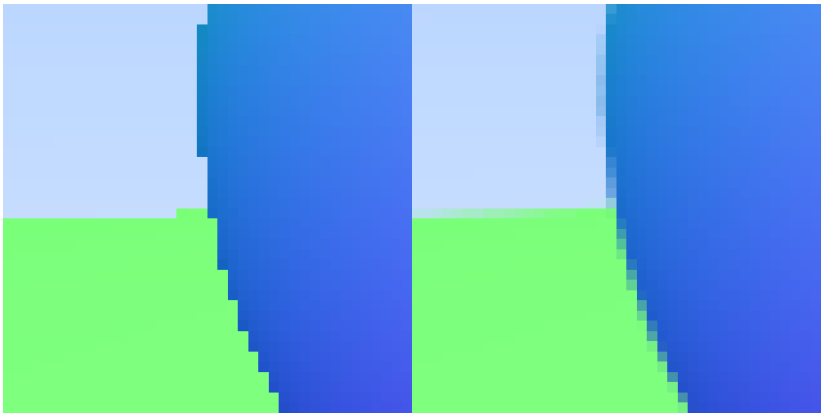


*Fig. 28:* "Before and after antialiasing." (Shirley, Black, and Hollasch 2024a)

colors returned by those rays. In simpler terms, it's as if we *blur together* multiple, smaller pixels. These types of methods are collectively known as *antialiasing* (Gortler 2012).

So, we should take multiple samples per pixel. Each of these samples corresponds to a point around the pixel, with some small offsets $\Delta x$, $\Delta y$ with respect to the pixel coordinates $[x, y]^T$. The ray will be sent through the point on the viewport at $[x + \Delta x, y + \Delta y]^T$.

There are various ways to pick $\Delta x$ and $\Delta y$. For instance, we can generate them as two random numbers, uniformly distributed in the interval $[-1, 1]$. This way, we use a sampling region which extends to the eight neighbouring pixels. We can also apply some sort of filter (like a *tent filter*), to the generated points[8]. This will make them less uniform and more distributed towards the central pixel (see Figure 29), which is what we want, since the central pixel is more important.

### 3.1.4 *Ray-Object Intersections*

In ray tracing, the most fundamental computation is the intersection of a ray with an object in the scene. Our objects will be either spheres or triangles, and here we'll describe intersection methods for both.

Let's consider first the case of spheres. Ray-sphere intersection can actually be made quite simple, as we'll see, by writing the equation of a sphere in vector form (Shirley, Black, and Hollasch 2024a).

8 As done in the *smallpt* renderer. URL: https://www.kevinbeason.com/smallpt/ (last accessed: 19.03.2025)

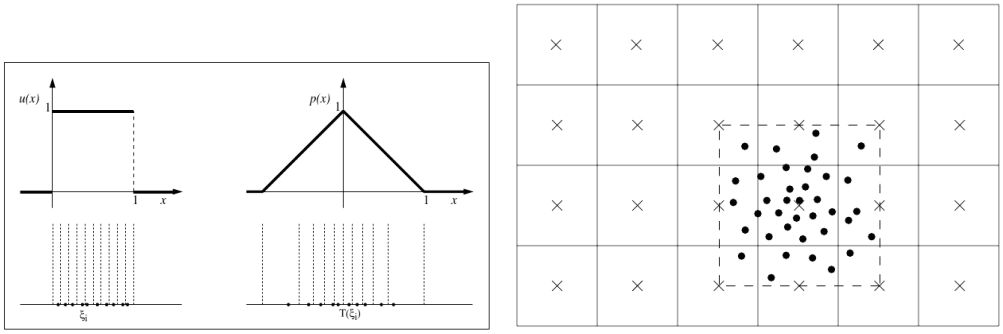



*Fig. 29:* A tent filter is applied to a set of uniform samples to make the points more distributed towards the central pixel. Images from the book *Realistic Ray Tracing*, by Shirley and Morley. (Shirley and Morley 2003)

A sphere is the set of points with the same distance (the sphere's *radius*) from its center. For this reason, we define a sphere by specifying $\tilde{c}$, its center, and $r$, its radius. If the sphere is centered in the origin, the condition we just mentioned can be expressed as

$$x^2 + y^2 + z^2 = r^2$$

For each point $\tilde{p} = [x, y, z]^T$ on the sphere. When the sphere is centered around a point $\tilde{c} = [c_x, c_y, c_z]^T$, the equation becomes:

$$(c_x - x)^2 + (c_y - y)^2 + (c_z - z)^2 = r^2$$

We rewrite this as

$$(\tilde{c} - \tilde{p}) \cdot (\tilde{c} - \tilde{p}) = r^2 \tag{3.9}$$

Remember now (from eq. (3.8)) that a ray is defined as $\tilde{p}(t) = \tilde{o} + t\vec{D}$. Saying "the ray hits the sphere" is the same as saying that there exists a point $\tilde{p}$ which satisfies both eq. (3.8) and eq. (3.9) (it's both on the ray, and on the sphere):

$$(\tilde{c} - \tilde{p}(t)) \cdot (\tilde{c} - \tilde{p}(t)) = r^2$$
$$(\tilde{c} - (\tilde{o} + t\vec{D})) \cdot (\tilde{c} - (\tilde{o} + t\vec{D})) = r^2$$

Next, we solve this for $t$:

$$(-t\vec{D} + (\tilde{c} - \tilde{o})) \cdot (-t\vec{D} + (\tilde{c} - \tilde{o})) = r^2$$
$$t^2\vec{D} \cdot \vec{D} - 2t\vec{D} \cdot (\tilde{c} - \tilde{o}) + (\tilde{c} - \tilde{o}) \cdot (\tilde{c} - \tilde{o}) - r^2 = 0$$

And what we get is a quadratic equation with variable $t$.

- If the equation has two real roots, the ray passes through the sphere. Doing so, it hits two points on the surface, one on the front and one on the back, which correspond to the two solutions for $t$.
- If there's only one real root, the ray is tangent to the sphere.
- If there are no real roots, the ray misses the sphere.

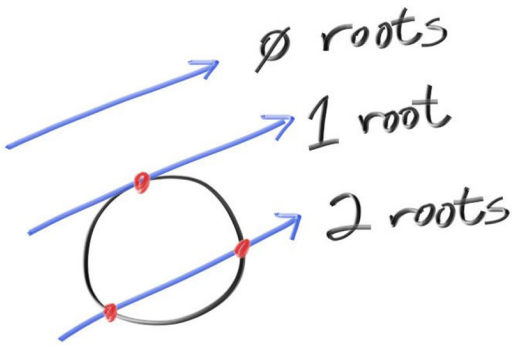



*Fig. 30:* "Ray-sphere intersection results." (Shirley, Black, and Hollasch 2024a)

At the intersection point $\tilde{p}$, the normal $\vec{N}$ of the sphere can be found by computing

$$\vec{N} = \tilde{p} - \tilde{c}$$

And then normalizing $\vec{N}$.

Let's now move on to triangles. First, we'll decide on a standard way to calculate a triangle's normal.

Let $\tilde{v}_0, \tilde{v}_1, \tilde{v}_2$ be the three vertices of the triangle. We can calculate its normal $\vec{N}$ as a cross product between two vectors along the triangle's edges (see Figure 31):

$$\vec{N} = \frac{(\tilde{v}_1 - \tilde{v}_0) \times (\tilde{v}_2 - \tilde{v}_0)}{\|(\tilde{v}_1 - \tilde{v}_0) \times (\tilde{v}_2 - \tilde{v}_0)\|}$$

Where the denominator is there to normalize $\vec{N}$.

A consequence of this choice is that, when looking at the triangle, if $\vec{N}$ is pointing *towards us*, we're seeing the vertices in *anti-clockwise* order, while if it's pointing *away* from us, *clockwise*. To simplify things, we will discard ray-triangle intersections in the second case, considering only triangles

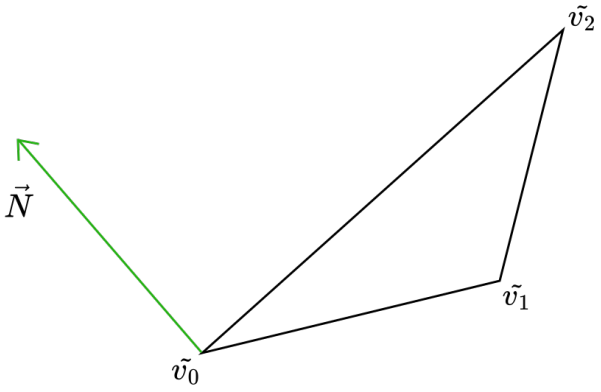



*Fig. 31:* Surface normal of a triangle.

which have their normal pointed towards us[9]. *Triangle meshes* will be following this convention. In fact, triangle meshes are usually defined as collections of *vertices* rather than triangles, the latter of which are built only at the time of *loading* the mesh into the program. When loading a triangle mesh, we'll go through the list of vertices, in their defined order, and join them in groups of three to build triangles. Thus, to predict the orientation of an assembled triangle, we'll just have to follow the clockwise/anticlockwise convention when defining the vertices in the mesh. That is, if we want to show a triangle on screen, we'll order its vertices in the list so that the camera sees them in clockwise order.

We can now describe ray-triangle intersection, which is usually composed of two steps (Gortler 2012). The first consists in calculating the intersection with the triangle's *plane*[10].

The equation of a plane is given by

$$ax + by + cz + d = 0 \tag{3.10}$$

Where $a, b, c, d$ are constants, and $x, y, z$ are the coordinates of a point $\tilde{p}$ lying on the plane. A more intuitive way to express this formula is, again, its vector form (Shirley, Black, and Hollasch 2024b). In fact, we can interpret $a$,$b$, and $c$ as the components of the plane's normal vector $\vec{N}$, that is, $\vec{N} = [a, b, c]^T$. Therefore, we rewrite eq. (3.10) as

$$\vec{N} \cdot \tilde{p} = -d$$

9 This is also called *backface culling*, since we only see the "front face" of the triangle, and cut out the "back face". (Gortler 2012)

10 A triangle lies on exactly one plane.

We can see that we're basically calculating the projection of $\tilde{p}$ on the normal $\vec{N}$, and saying how far along $\vec{N}$ it should be. We already know $\vec{N}$, since it's also the triangle's normal.

As done with spheres, we now impose the condition $\tilde{p}(t) = \tilde{o} + t\vec{D}$, and solve for $t$:

$$\begin{aligned} \vec{N} \cdot (\tilde{o} + t\vec{D}) &= -d \\ \vec{N} \cdot \tilde{o} + t(\vec{N} \cdot \tilde{D}) &= -d \\ t &= -\frac{\vec{N} \cdot \tilde{o} + d}{\vec{N} \cdot \tilde{D}} \end{aligned}$$

Before calculating the division, we check if the denominator is zero. In that case, the ray is parallel to the plane, and there's no intersection.

So, we now use $t$ to get the ray-plane intersection point $\tilde{p} = \tilde{p}(t) = \tilde{o} + t\vec{D}$. The next step is to check whether the point is inside the triangle.

Following our anti-clockwise convention, we want $\tilde{p}$ to be "on the left" of each of the triangle's "edge vectors"[11]. Let's see what this means by taking $\tilde{v}_1 - \tilde{v}_0$ as the first edge vector, and calling it $\vec{A}$ (see Figure 32). Let's also define the vector $\vec{B} = \tilde{p} - \tilde{v}_0$ and compute the cross product $\vec{C} = \vec{A} \times \vec{B}$. Now, compare this with Figure 31. If $\tilde{p}$ (or, to be more precise, $\vec{B}$) is on the *left* from $\vec{A}$, the cross product $\vec{C}$ will be pointing in the *same direction* as the normal $\vec{N}$. Otherwise, the direction of $\vec{C}$ will be *opposite* to that of $\vec{N}$. Thus, $\vec{p}$ is on the left from the edge vector $\vec{A}$ if:

$$\vec{N} \cdot \vec{C} \geqslant 0$$

Where if $\vec{N} \cdot \vec{C} = 0$, $\tilde{p}$ is lying *along* the edge.

When this holds true for each of the triangle's edges, that is, the point $\tilde{p}$ is on the left of each of the edge vectors $\tilde{v}_1 - \tilde{v}_0$, $\tilde{v}_2 - \tilde{v}_1$ and $\tilde{v}_0 - \tilde{v}_2$, we can say that $\tilde{p}$ is inside the triangle. Otherwise, there's no intersection.

That's it: we can now calculate the closest intersection with an object, be it a sphere or a triangle, for any ray sent into the scene. At this point, the problem we're left with is establishing the color of the object's surface, which derives from the reflection of some (previously emitted) light. As the scene's light source, the simplest option is using a *point light*. A point light source is located at a single point in the scene, and radiates light in all directions equally. For now, this is the light source we'll be working with.

11 This is described by (Gortler 2012) as the three "sub" triangles $\Delta(\tilde{v}_0\tilde{v}_1\tilde{p})$, $\Delta(\tilde{v}_0\tilde{p}\tilde{v}_2)$, $\Delta(\tilde{p}\tilde{v}_1\tilde{v}_2)$ being all "clockwise".

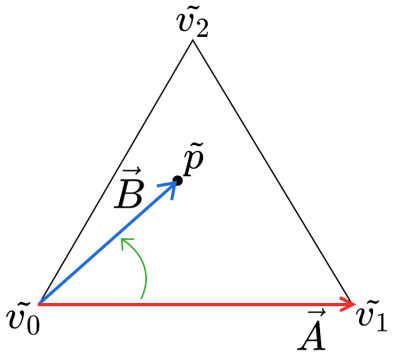



*Fig. 32:* $\tilde{p}$ is "to the left" of $\vec{A}$.

We'll see that the orientation of the *light vector* $\vec{L}$, which is directed towards the (point) light from the point of intersection $\tilde{p}$, is a key factor for reflection, along with the surface normal $\vec{N}$ and the *view vector* $\vec{V}$, the latter of which points from $\tilde{p}$ towards the viewer (that is, the camera center) (Gortler 2012). The directions of these vectors, together with the *material parameters*, will determine the amount and color of light reflected by the intersected surface towards the camera's pixel.

We should be careful not to rush ahead, though: haven't we overlooked something? If we think about it, we have no clue as to what it means for an "amount" of light to be reflected towards a camera's pixel. In fact, we still need to properly choose a way of putting *light* into *numbers*. In the previous chapter, all we said was that "amounts" of light are usually specified through *spectral distributions*, and then went on to prove that, for our purposes, we can use RGB triplets instead of full spectra. In general, we know that the three RGB values tell us "how much" some light is red, green or blue, and are used to mix these three base colors. An "amount" of light to which we applied this idea was the light's *energy*. The issue here is that energy, as a *unit of measurement* for light, is too vague for us: it can't really express how much light travels along *rays*. To solve this important issue, we'll briefly go back to physics, and build a series of abstractions which will culminate in an adequate unit of measurement called *radiance*.

## 3.2 RADIOMETRY

We saw how ray tracing thinks of light as a ray. Recall now that we already encountered this model in Chapter 2, when talking about macroscopic

descriptions of light-matter interactions (section 2.3.2). We called it the point of view of *geometrical optics*, where light interacts with objects much larger than its wavelength.

We now resort again to geometrical optics to describe quantities for the measurement of light, also called *radiometric quantities*.

*Radiometry* can be defined as

> "The study of the propagation of electromagnetic radiation in an environment."
> (Pharr, Jakob, and Humphreys 2023)

Geometrical optics and radiometry form the basis of many PBR algorithms, as they offer adequate models for the description of light and light scattering.

We'll follow the process provided by (Pharr, Jakob, and Humphreys 2023) to abstractly derive some basic radiometric quantities, by taking limits over time, area, and directions. We should always keep in mind that these quantities are, in general, wavelength dependent.

### 3.2.1 *Energy*

We already know that the energy of light is carried by photons[12], each of which is at a particular *frequency* $\nu$. Recall that the energy of a photon, which we'll indicate as $Q$ here, is

$$Q = h\nu$$

Where $h$ is Planck's constant ($h \approx 6.626 \times 10^{-34}\,m^2\,kg\,s^{-1}$).

### 3.2.2 *Radiant Flux*

*Radiant flux*, also known as *power*, is the total amount of energy passing through a surface, or region of space, per unit time:

$$\Phi = \lim_{\Delta t \to 0} \frac{\Delta Q}{\Delta t} = \frac{dQ}{dt}$$

It is measured in watts (W).

12 Basically, all the radiometric quantities that we'll see are just different ways of measuring photons.

So, when talking about flux, we have an area, crossed by photons during infinitesimally small amounts of time. One might make the observation that, because photons are discrete quanta, it doesn't really makes sense to take limits that go to zero for differential time. However, if we assume the number of photons to be enormous with respect to the measurements we are interested in, this detail is not problematic.

### 3.2.3 *Irradiance and Radiant Exitance*

Let's say that we have photons passing through a finite area $A$. We now define the *average* density of power over this area as

$$E = \Phi/A \qquad [W/m^2]$$

and call it *irradiance* if the photons are *arriving* at the surface, and *radiant exitance* (indicated as $M$) if they're *leaving* it.

To be more precise, we should, again, take a limit. This time, it's over the differential area at a point $\tilde{p}$ on $A$:

$$E(\tilde{p}) = \lim_{\Delta A \to 0} \frac{\Delta\Phi(\tilde{p})}{\Delta A} = \frac{d\Phi(\tilde{p})}{dA}$$

This gives us a proper definition of irradiance/radiant exitance.

We can use irradiance to understand *Lambert's law*:

> "The amount of light energy arriving at a surface is proportional to the cosine of the angle between the light direction and the surface normal."
> (Pharr, Jakob, and Humphreys 2023)

We do so as follows. Consider a light source with area $A$ and a flux $\Phi$, illuminating a surface. If the light is directly on top of the surface (see Figure 33), the area of the surface $A_1$ which receives light will be equal to $A$, and the irradiance

$$E_1 = \frac{\Phi}{A}$$

If, however, the light is at an angle $\theta$ to the surface, the area on the surface which receives light will be *larger* than $A$. If $A$ is (infinitesimally) small, the area receiving flux, which we call $A_2$ now, will be approximately $A/\cos\theta$, and the irradiance

$$E_2 = \frac{\Phi\cos\theta}{A}$$

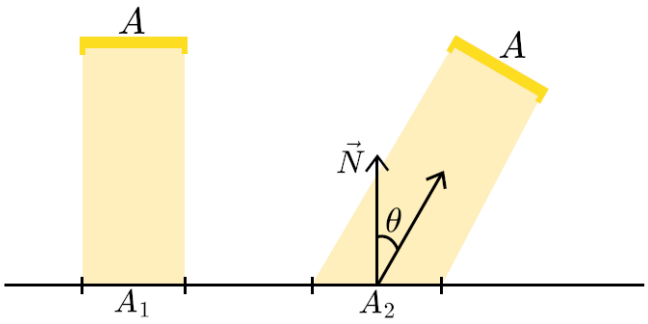



*Fig. 33:* Visualization of Lambert's law.

### 3.2.4 *Radiance*

*Radiance* measures irradiance with respect to a *solid angle* $\omega$, which contains the directions of light incident to the surface at the point $\tilde{p}$. More precisely, we take $\omega$ to be infinitesimal, getting $d\omega$. Since $d\omega$ is infinitesimal, it corresponds, approximately, to a single direction $\vec{\omega}$ of incoming light (see Figure 34). The radiance $L$ is therefore defined as

$$L(\tilde{p}, \vec{\omega}) = \lim_{\Delta\omega \to 0} \frac{\Delta E_\omega(\tilde{p})}{\Delta\omega} = \frac{dE_\omega(\tilde{p})}{d\omega} \qquad [\frac{W}{m^2\,sr}] \tag{3.11}$$

In eq. (3.11), $E_\omega$ denotes irradiance at the surface when it's perpendicular to $\vec{\omega}$. We can calculate $E_\omega$ by applying Lambert's law; that is, simply by dividing $E$ for a $\cos\theta$ factor.

Between the radiometric quantities we've seen, radiance is the most fundamental. Knowing the radiance, we can calculate the irradiance $E$, the radiant flux $\Phi$ and the energy $Q$ by taking *integrals*, instead of *limits*.

It can be proven that radiance remains *constant* along *rays* through empty space. This clearly makes it a perfect quantity to *measure light* in *ray tracing*.

Given a point $\tilde{p}$ on a surface, we'll use radiance to define both *incoming* and *outgoing* (that is, *reflected*) *light* at $\tilde{p}$. To express this distinction, we'll write the *incoming radiance* as $L_i(\tilde{p}, \vec{\omega})$, and the *outgoing radiance* as $L_o(\tilde{p}, \vec{\omega})$.

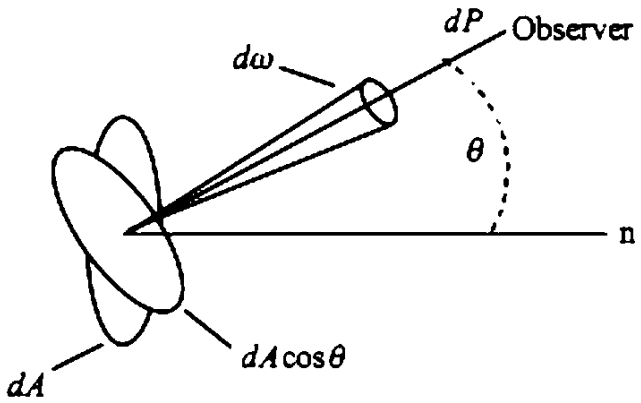



*Fig. 34:* In this figure, a surface of area dA emits (or reflects) some radiance into the solid angle $d\omega$, corresponding to the direction of an observer. The same figure can be used also to depict *incoming* radiance to the surface, if instead of the observer we have a light source. Source: *Planetary Photometry (Tatum and Fairbairn)*, LibreTexts Physics. Licensed under the CC BY-NC 4.0 license.

## 3.3 BIDIRECTIONAL REFLECTANCE DISTRIBUTION FUNCTION

We've finally arrived at a point where we can actually discuss models for the reflection of light, called *Bidirectional Reflectance Distribution Functions* (*BRDF*). We'll start by deriving the definition of a generic BRDF from radiometric quantities.

### 3.3.1 *Definition*

Let's reformulate the problem of reflection, now in more precise terms. We want to compute the amount of outgoing radiance $L_o$, leaving the point $\tilde{p}$ of a surface in the direction $\vec{\omega}_o$ toward the viewer, as a result of some incident radiance $L_i$ coming in from the direction $\vec{\omega}_i$ (see Figure 35).

We could do this easily if we had a function, call it $f_r$, which defines the ratio between $L_o$ and $L_i$. Actually, for reasons that will become clear later, it would be better if $L_o$ were expressed in differential terms, that is, if we used $dL_o$.

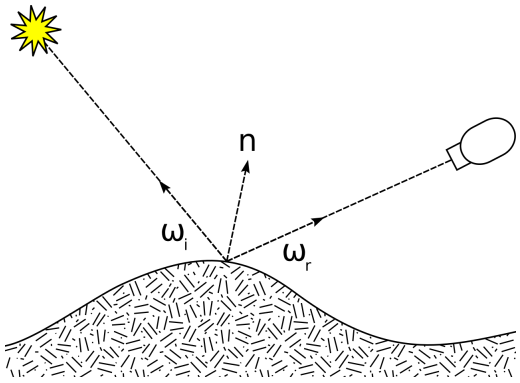



*Fig. 35:* The vectors used by a BRDF ($\omega_r$ is the outgoing direction). Source: Wikimedia Commons. Licensed under the CC BY-SA 3.0 license.

To derive this function, we start by rewriting (3.11) as

$$dE(\tilde{p}, \vec{\omega}_i) = L_i(\tilde{p}, \vec{\omega}_i) cos\theta_i \, d\omega_i$$

Where $\theta_i$ is the angle between the incident direction and the surface normal, also called *angle of incidence*[13].

Remember that $d\omega_i$ is just the direction $\vec{\omega}_i$ interpreted as a *differential* (that is, infinitely small) *cone of directions*.

The incoming irradiance $dE$ induces some differential amount of reflected radiance in the direction $\vec{\omega}_o$, proportional to the irradiance:

$$dL_o(\tilde{p}, \vec{\omega}_o) \propto dE(\tilde{p}, \vec{\omega}_i) \tag{3.12}$$

The reason their values are proportional lies in the *linearity assumption*. In fact, in geometrical optics, we assume the following fact to be true.

> "The combined effect of two inputs to an optical system is always equal to the sum of the effects of each of the inputs individually." (Pharr, Jakob, and Humphreys 2023)

This is a reasonable assumption. In fact, for most real materials, the following observation can be made. If all of the light is coming in from a single small cone of directions, and we double the width of this incoming cone, the amount of flux reflected along a fixed outgoing cone, and thus the amount of radiance reflected along a fixed outgoing direction, will roughly double as well (Gortler 2012). In our case, the size of the incoming

13 $cos\theta_i$ is simply the cosine factor from Lambert's law made explicit.

cone is given by $d\omega_i$, and so, if $d\omega_i$ doubles, both $dE$ *and* $dL_o$ will double as well (hence, they are proportional to each other with respect to the size of $d\omega_i$).

The BRDF function $f_r$ is then defined as the constant of proportionality in (3.12): (Pharr, Jakob, and Humphreys 2023)

$$f_r(\tilde{p}, \vec{\omega}_o, \vec{\omega}_i) = \frac{dL_o(\tilde{p}, \vec{\omega}_o)}{dE(\tilde{p}, \vec{\omega}_i)} = \frac{dL_o(\tilde{p}, \vec{\omega}_o)}{L_i(\tilde{p}, \vec{\omega}_i) cos\theta_i d\omega_i} \tag{3.13}$$

To describe how the reflection of light should take place on surfaces, we'll be defining BRDF functions. We'll then base our calculations, other than on the parameters $\tilde{p}, \vec{\omega}_o, \vec{\omega}_i$, also on material properties, which will effectively "tweak" the behaviour of the BRDF.

We should mention that, in PBR, BRDFs can't just be defined arbitrarily. To make a BRDF physically based, we should make sure that it obeys the following principles: (Pharr, Jakob, and Humphreys 2023)

- *Helmholtz reciprocity*: the incoming and outgoing directions can be swapped without affecting the value of $f_r$. That is, for all pairs of directions $\vec{\omega}_i$ and $\vec{\omega}_o$, $f_r(\tilde{p}, \vec{\omega}_i, \vec{\omega}_o) = f_r(\tilde{p}, \vec{\omega}_o, \vec{\omega}_i)$.
- *Energy conservation*: the surface can't reflect more energy than it receives.

### 3.3.2 *The Reflection Equation*

Even though we have the BRDF $f_r$, we still haven't explicitly said how it should be used. Let's rewrite eq. (3.13) as

$$dL_o(\tilde{p}, \vec{\omega}_o) = f_r(\tilde{p}, \vec{\omega}_o, \vec{\omega}_i) L_i(\tilde{p}, \vec{\omega}_i) cos\theta_i d\omega_i$$

If we integrate this equation over all directions above $\tilde{p}$, we can now actually calculate $L_o$:

$$L_o(\tilde{p}, \vec{\omega}_o) = \int_{H^2(\vec{N})} f_r(\tilde{p}, \vec{\omega}_o, \vec{\omega}_i)\, L_i(\tilde{p}, \vec{\omega}_i)\, cos\theta_i\, d\omega_i \tag{3.14}$$

Where $H^2(\vec{N})$ is the hemisphere centered around the normal at $\tilde{p}$, assuming that the normal is pointing outwards from the surface. This is called the *reflection equation*[14], and, during rendering, our objective will

14 The *reflection equation* is a special case of the *rendering equation*. (Hoffman 2012)

be to solve it for each point $\tilde{p}$ seen by the camera. For now, we'll only present the solution in the case of a single point light, which is very straightforward.

Let's summarize briefly the situation. In order to determine the color of a pixel, a ray has been sent out in the scene, which intersected an object at the point $\tilde{p}$. The pixel will be colored based on the reflected radiance $L_o$ coming out from $\tilde{p}$, in the direction $\vec{V}$ towards the camera. Remember that radiance, like any other light-measuring quantity, is spectral (it varies with wavelength). Also remember that we can express it as an RGB triplet. To calculate the reflected light, that is, $L_o(\tilde{p}, \vec{V})$, we multiply the BRDF $f_r$ by all the incoming radiance $L_i$ (this is what happens inside the integral). The BRDF is a spectral quantity as well, so its value is also an RGB triplet. Hence, to multiply $f_r$ by $L_i$ we'll do a component-wise vector multiplication. As for the incoming radiance $L_i$, there's only one place where it could originate from: the point light. The point light is in the direction $\vec{L}$ with respect to $\tilde{p}$, so we compute:

$$L_o(\tilde{p}, \vec{V}) = f_r(\tilde{p}, \vec{V}, \vec{L}) L_i(\tilde{p}, \vec{L}) \cos\theta_i \tag{3.15}$$

And we've calculated the color $L_o(\tilde{p}, \vec{V})$ of the pixel.

Note that $\cos\theta_i$ can be computed as the dot product $\vec{N} \cdot \vec{L}$. To avoid negative values, we clamp it to 0. This way, we only consider light if it comes from above the surface, and ignore it otherwise.

### 3.3.3 *Diffuse and Specular Terms*

In Chapter 2, we learned about the behaviour of light when it encounters a planar boundary. We found out that, as explained by the *Fresnel equations*, light gets split into the two directions of *reflection* and *refraction*. We also defined *subsurface scattering* as the phenomenon where refracted light gets scattered so much that it's eventually re-emitted out of the surface.

Even though subsurface scattering is not the same as reflection, a BRDF model can partly simulate it. In fact, BRDFs are usually divided into two separate parts, called *terms* (or *lobes*). One of them describes proper reflection and is called the *specular term*, while the other, the *diffuse term*, partially models subsurface scattering[15]. We say "partially" because it only accounts for subsurface-scattered light that is re-emitted from (approximately) the same point where it entered (Hoffman 2012). An

15 However, we call this diffuse "reflection" regardless.

example is shown in Figure 36, where the BRDF terms are based on the *Phong lighting model*.

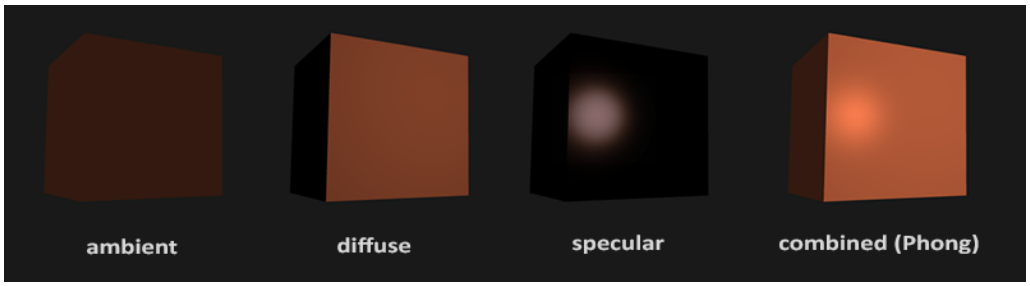



*Fig. 36:* The Phong lighting model. Other than diffuse and specular lighting, constant *ambient lighting* is also added to model reflection. Source: *Basic Lighting*, Learn OpenGL tutorial series. Licensed under the CC BY 4.0 license.

If we need to simulate proper subsurface light transport, meaning light that exits the surface also at points that are relatively distant from its entry, we should use a *Bidirectional Scattering Surface Reflectance Distribution Function* (*BSSRDF*) instead of a BRDF (Pharr, Jakob, and Humphreys 2023).

The *diffuse term* of a BRDF is usually quite simple. A common example is the *Lambertian model*, where the BRDF is just a constant value: (Hoffman 2012)

$$f_{Lambert}(\vec{L}, \vec{V}) = \frac{\rho}{\pi} \tag{3.16}$$

Here $\rho$ is an RGB triplet indicating the fraction of light which is diffusely reflected, and is typically referred to as the *diffuse color*. In fact, during the scattering process, light is also partially absorbed by the material, depending on wavelength. Thus, only some wavelengths will be re-emitted, and the diffuse response will be tinted towards a certain color (the diffuse color) (Burley 2012). The diffuse color $\rho$ corresponds to what typically people think of as the *surface color* (Hoffman 2012).

What equation (3.16) means is that the outgoing radiance will be constant for all outgoing directions, that is, it's independent from $\vec{V}$ (see Figure 37). Incoming directions $\vec{L}$ will influence Lambertian reflection only through the $\cos\theta_i$ term in the reflection equation (3.14). Note that in (3.16) we've omitted the parameter $\tilde{p}$ from the BRDF and made it implicit, as we'll be doing from now on.

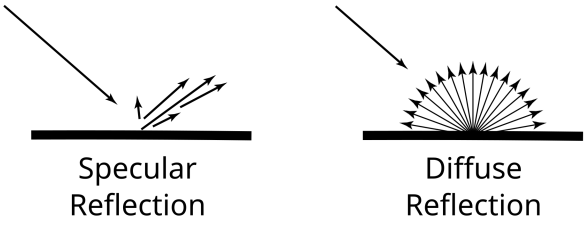



*Fig. 37:* Geometries of specular and diffuse reflection. Source: Wikimedia Commons. Licensed under the CC BY 3.0 license.

The *specular term* of a BRDF is, usually, more complex, as it's often based on *microfacet theory*.

## 3.4 MICROFACET THEORY

*Microfacet theory* allows us to generalize the idea of reflection to non-optically flat surfaces. We know from Chapter 2 that most surfaces in the real world present irregularities which are larger than visible light wavelengths, but also too small to be seen by the eye (see section 2.3.3). Microfacet theory takes this into account by modeling a surface as a collection of many *microfacets*, each of which is optically flat[16](Hoffman 2012). Most physically plausible models, even if they're not specifically described in microfacet form, can still be interpreted through the point of view of microfacet theory (Burley 2012).

So, the specular term of a BRDF applies the idea of microfacets. Each microfacet reflects light the way we know, in a single outgoing direction depending on the orientation of the microfacet's normal, which we'll call $\vec{M}$. Since, given a light direction $\vec{L}$, we evaluate the BRDF in a single view direction $\vec{V}$, only the microfacets that happen to be angled just right between $\vec{L}$ and $\vec{V}$ can contribute to reflectance. More precisely, their surface normal $\vec{M}$ needs to be oriented exactly halfway between $\vec{L}$ and $\vec{V}$. The vector halfway between $\vec{L}$ and $\vec{V}$ actually has a specific name: the *half-vector* (Hoffman 2012). We denote it as $\vec{H}$, and calculate it as follows.

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

16 We can think of the microfacets as tiny mirror reflectors, with random orientations. (Hughes et al. 2014)

Even in the case of microfacets with $\vec{M} = \vec{H}$, it's not certain that they will contribute to reflection. This is because some microfacets might be blocked by others from the direction $\vec{L}$ (*shadowing*), or from the direction $\vec{V}$ (*masking*), or from both (see Figure 38). This light is assumed to be lost.

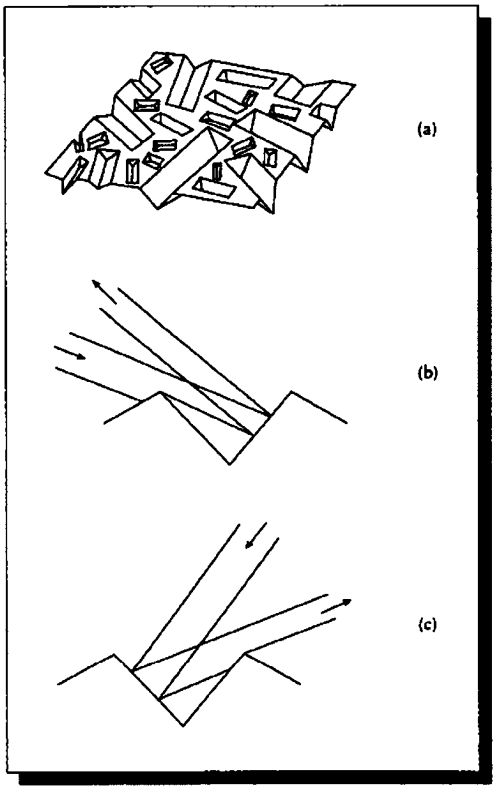



*Fig. 38:* (a) A surface's microfacets. (b) Shadowing. (c) Masking. (Glassner 1995)

The specular term of the BRDF can therefore be derived in the following form. (Hoffman 2012)

$$f_{\mu facet} = \frac{F(\vec{L}, \vec{H})G(\vec{L}, \vec{V}, \vec{H})D(\vec{H})}{4(\vec{N} \cdot \vec{L})(\vec{N} \cdot \vec{V})}$$

Where

- $D(\vec{H})$ is the microfacet *normal distribution function* evaluated at the half-vector $\vec{H}$. It expresses the concentration of microfacets which are oriented so that they *could* reflect light from $\vec{L}$ into $\vec{V}$.
- $G(\vec{L}, \vec{V}, \vec{H})$ is the *shadow-masking function*. It tells us the percentage of microfacets with $\vec{M} = \vec{H}$ that are not *shadowed* or *masked*. Its product

with $D(\vec{H})$ gives, as a result, the concentration of *active microfacets*, which contribute to reflection.

- $F(\vec{L}, \vec{H})$ is the *Fresnel reflectance* of the active microfacets. In other terms, it's the amount of incoming light that's reflected from each of the microfacets that are active.
- The denominator $4(\vec{N} \cdot \vec{L})(\vec{N} \cdot \vec{V})$ is a correction factor, which is there because of quantities being transformed between the local space of the microfacets and that of the overall macrosurface. To make sure we don't divide by zero, other than clamping the dot products to avoid negative results, we also add a very small positive value to the denominator.

We now present a more thorough explanation of the three functions $D$, $G$ and $F$, based on Naty Hoffman's course notes *Physics and Math of Shading*, made for the 2012 SIGGRAPH course *Practical Physically Based Shading in Film and Game Production* (Hoffman 2012).

### 3.4.1 *Fresnel Reflectance*

Let's repeat the meaning and purpose of the Fresnel equations, one last time. The Fresnel equations tell us the portions of reflected and refracted light at a planar boundary (that is, an optically flat surface) based on the incoming angle of light and the refractive index of the material. In our case, the incoming angle is between $\vec{L}$ and $\vec{H}$, since we are only concerned with active microfacets (that is, with normal $\vec{M} = \vec{H}$). The Fresnel reflectance function $F(\vec{L}, \vec{H})$ is based on the Fresnel equations, and computes the amount of light that's reflected.

Since the refractive index may vary over the visible spectrum, the Fresnel reflectance should be defined as a spectral quantity (so, for us, an RGB triplet). However, sometimes the spectral nature of reflectance is ignored, and a single real number is used instead. This is done, for example, in the Disney "Principled" BRDF (Burley 2012). Fresnel reflectance has to lie in the $[0, 1]$ range, since a surface cannot reflect less than 0% or more than 100% of the incoming light.

Specifying a refractive index is often inconvenient, especially for artists. For this reason, it's customary to simplify the Fresnel equations into a more intuitive form. This can be done thanks to some common patterns observed in the behaviours of ordinary real-world materials.

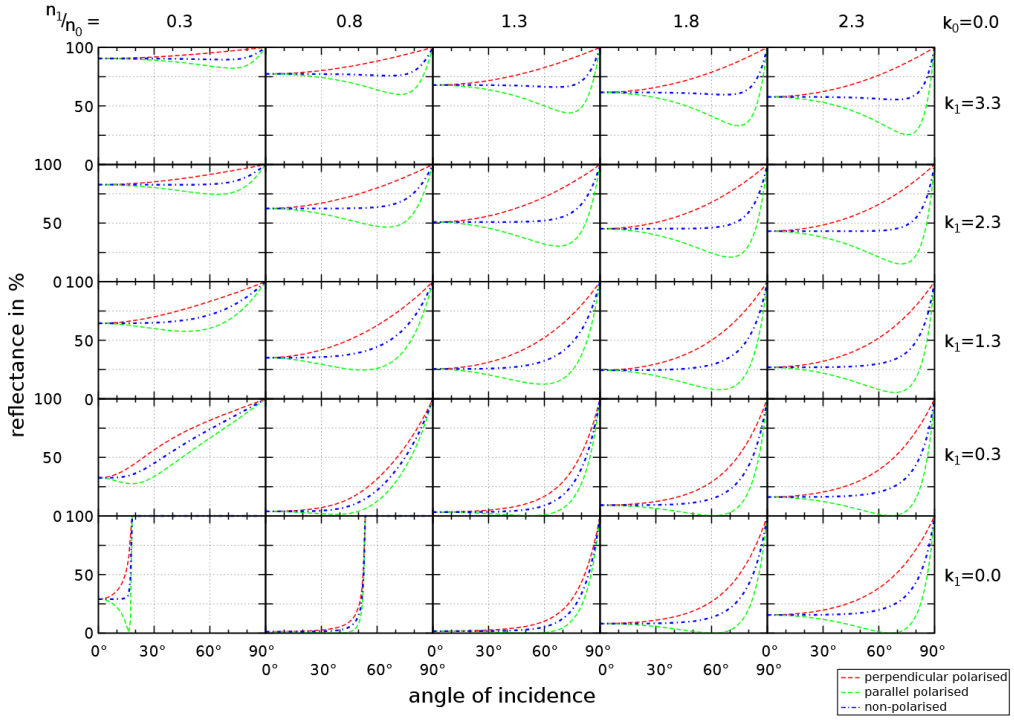



*Fig. 39:* "Fresnel reflection coefficients for a boundary surface between air and a variable material in dependence of the complex refractive index and the angle of incidence." Source: Wikimedia Commons. Licensed under the CC BY-SA 3.0 license.

First of all, for many materials, the reflectance is almost constant for incoming angles between $0°$ and (about) $45°$. The reflectance then goes on to change more significantly (typically increasing) between $45°$ and about $75°$. Finally, between $75°$ and $90°$, reflectance always goes rapidly to 1 ($[1,1,1]^T$ if viewed as an RGB triplet), making surfaces perfectly reflective at grazing angles (see Figures 39 and 40).

So, since the Fresnel reflectance stays close to its value at $0°$ over most of the range, we can think of this value as being *characteristic* for the *material*. We denote it as $F_0$, and, in this thesis, we'll refer to it as the *base reflectance*.

A widely used approximation of Fresnel reflectance computes it as a function of the base reflectance $F_0$ instead of the refractive index:

$$F_{\text{Schlick}}(F_0, \vec{L}, \vec{H}) = F_0 + (1 - F_0)(1 - (\vec{L} \cdot \vec{H}))^5 \tag{3.17}$$

This called *Schlick's approximation*.

Basically, in (3.17), we're interpolating between the base reflectance $F_0$ and the maximum reflectance 1, following the smooth function $(1 - (\vec{L} \cdot \vec{H}))^5$, which goes from 0 to 1 as $\vec{L}$ becomes perpendicular to $\vec{H}$ (and thus, the light becomes parallel to the surface).

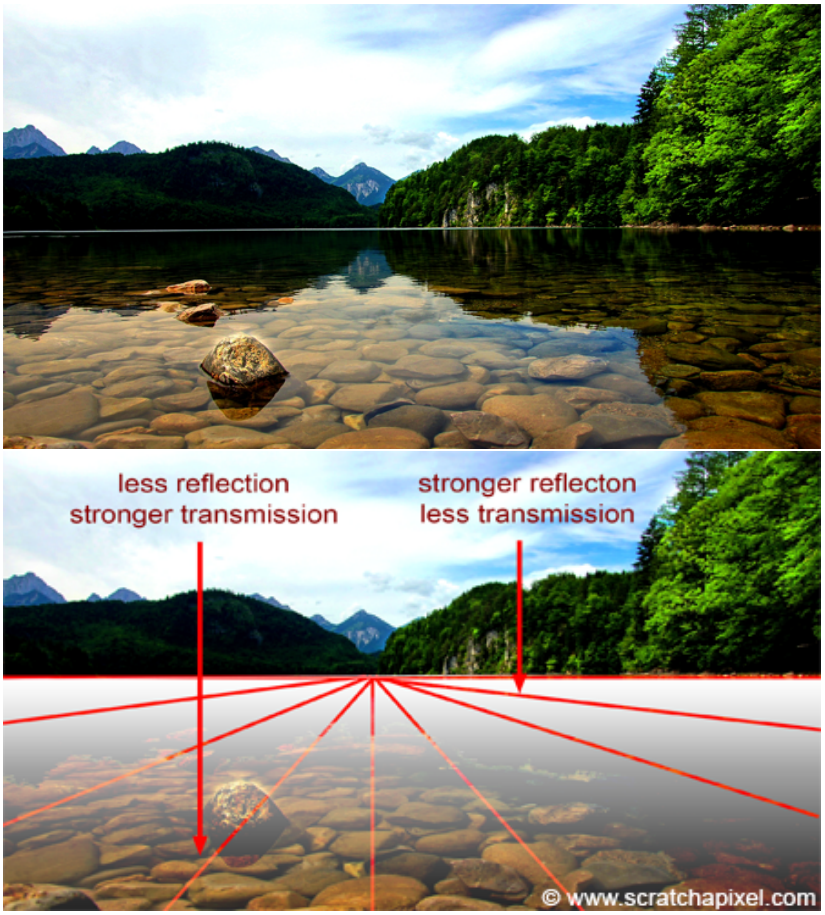



*Fig. 40:* "The ratio of reflected light increases as the angle between the view direction and the surface normal increases." Source: *Introduction to Shading (Reflection, Refraction and Fresnel)*, Scratchapixel. Licensed under the CC BY-NC-ND 4.0 license.

We've just encountered the first parameter, $F_0$, which will be used to define a material. To model realistic materials, the values assigned to $F_0$ should be coherent with our intents. For instance, metals have much higher values of $F_0$ than dielectrics. In fact, we know that metals have no sub-surface scattering, so most of the incident light will be re-emitted by the specular component of the BRDF. Aside from metals (and

also gemstones and crystals) pretty much any material that can be seen outside of labs has between 2% and 5% base reflectance.

### 3.4.2 *Normal Distribution Function*

The statistical distribution of microfacet orientations is defined via the microfacet normal distribution function $D$. Most microfacets usually point in the same direction as the overall surface normal $\vec{N}$, making $D(\vec{N})$ the highest value in the distribution. Choosing a distribution function for $D$, we determine the size, brightness, and shape of specular highlights on the surface. Many times, the distribution functions are somewhat Gaussian-like, with some kind of *roughness* (or *variance*) parameter, resulting in circular highlights of various sizes. However, anisotropic functions are also used, so that the highlights are less symmetric (see Figure 41).

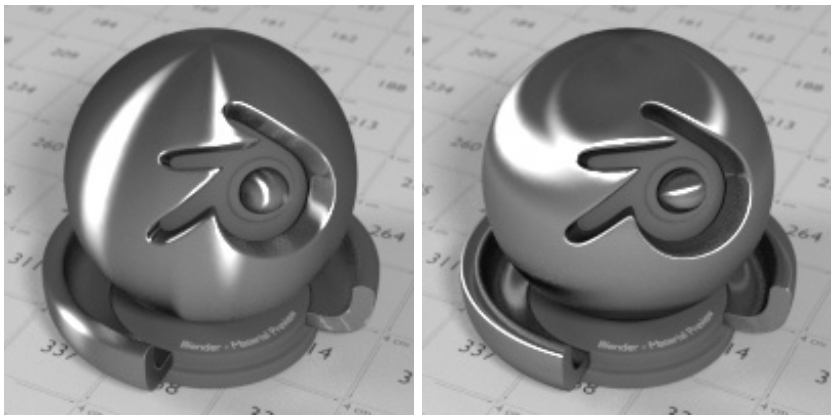


*Fig. 41:* Anisotropic specular highlights at 0° (left) and 90° (right). Source: *Anisotropic BSDF*, Blender 3.1 Manual. Licensed under the CC-BY-SA 4.0 license.

In general, the values of $D$ are not restricted to lie between 0 and 1, although they must be non-negative.

### 3.4.3 *Shadow-Masking Function*

The shadow-masking function $G(\vec{L}, \vec{V}, \vec{H})$ represents the probability that microfacets with normal $\vec{H}$ will be visible from both the light direction $\vec{L}$ and the view direction $\vec{V}$. Sometimes, it's also referred to as the *geometric attenuation term* (as done by (Burley 2012)).

Since G represents a probability, its values are scalar and constrained to lie between 0 and 1.

This function typically does not introduce any new parameters to the BRDF; it either has no parameters, or uses the roughness parameter(s) of D.

The shadowing-masking function is essential for BRDF energy conservation: without it, the BRDF could reflect arbitrarily more light energy than it receives.

## 3.5 THE DISNEY "PRINCIPLED" BRDF

We'll now study the math behind the Disney "Principled" BRDF, as was described by Burley in 2012 (Burley 2012). This physically-based BRDF was developed at Disney Animation Studios, and was first deployed in production for the movie *Wreck-it Ralph*[17]. The development process of Burley and his team consisted in comparing analytical BRDFs with real-world light reflection data, previously measured by the Mitsubishi Electric Research Laboratories (MERL) and collected in the *MERL BRDF Database*[18].


*Fig. 42:* "Production still from Wreck-It Ralph." (Burley 2012)

The output of the Disney BRDF is the result of blending between different reflection modes. Other than the *diffuse* and *specular* lobes, there's

17 In *Wreck-it Ralph*, the Disney "Principled" BRDF was used for virtually any material other than hair (which is a bit more tricky to render). (Burley 2012)

18 The *MERL BRDF Database* can be found at https://www.merl.com/research/downloads/BRDF/ (last accessed: 20.03.2025)

also a *clearcoat* lobe. The clearcoat lobe can be used to form an additional layer of reflection on top of the base material, giving it a "glossy" look.

### 3.5.1 *Parameters*

The Disney BRDF is actually *not* physically correct; that is, not completely. It is a "principled" model, rather than a strictly physical one, meaning that it's physically *plausible*, but not *exact*. This is because it was essential for the model to be "art directable", even at the cost of sacrificing some physical rules (like *energy conservation*). After all, in animation, reflection models are meant to be tools for artists, so they must allow proper creative expression.

During the development of the BRDF, one of the requirements was to keep the number of parameters as low as possible. This resulted in eleven of them, the first one being a vector that defines the surface color, while the others are scalars that go from 0 to 1 (in their "plausible range"). They are described as follows.

- *baseColor*: this is simply the surface color.
- *subsurface*: when different from zero, the BRDF uses an approximation of subsurface scattering, which alters the shape of the diffuse reflection.
- *metallic*: this is the "metallic-ness" of the material (0 = dielectric, 1 = metallic). It blends linearly blend between the two reflection models, where the metallic model has no diffuse component and its reflections are tinted towards the base color, while the dielectric model has a diffuse component and achromatic reflections.
- *specular*: the "incident specular" amount. It corresponds to what we earlier called the *base reflectance* $F_0$, and is used in lieu of an explicit index of refraction.
- *specularTint*: this is a concession for artistic control that tints the incident specular towards the base color. The "grazing specular", that is, $F_{90}$, is still achromatic.
- *roughness*: the surface roughness, which influences both diffuse and specular response.
- *anisotropic*: the degree of anisotropy for specular highlights (0 = isotropic, 1 = maximally anisotropic).

- *sheen*: this is an additional grazing specular component, primarily intended for cloth.
- *sheenTint*: it tells how much the sheen should be tinted towards the base color.
- *clearcoat*: this activates the clearcoat lobe and regulates its strength.
- *clearcoatGloss*: controls the clearcoat's glossiness (0 gives a "satin" appearance, while 1 a "gloss" appearance).

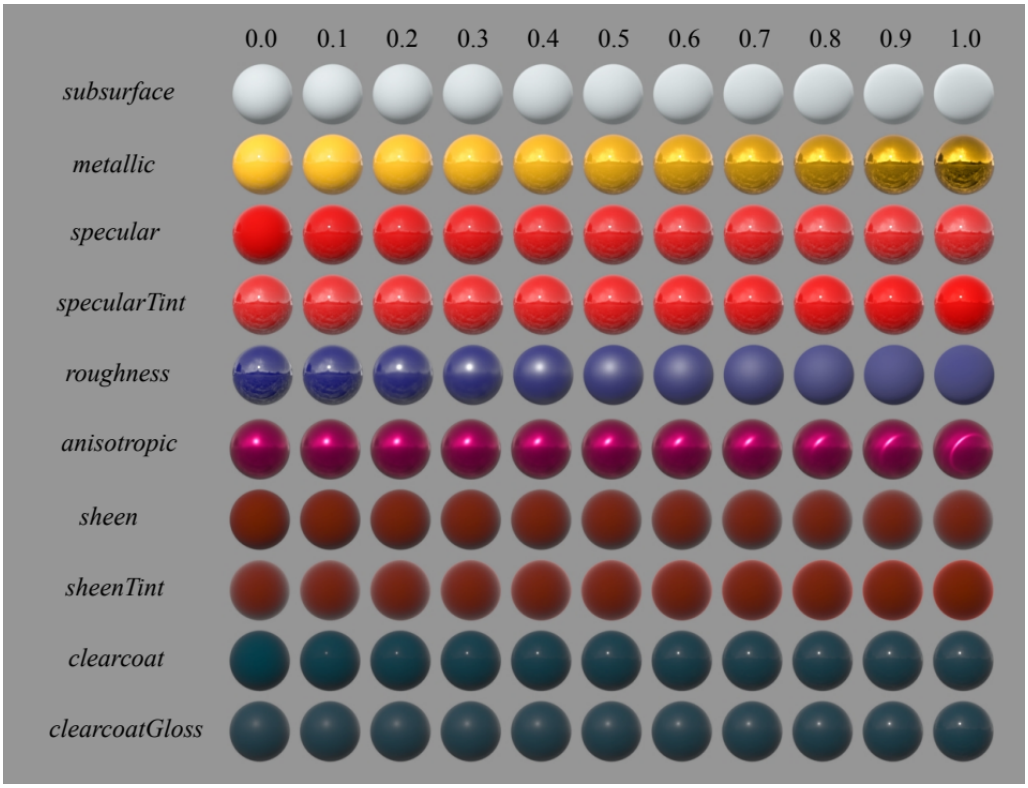



*Fig. 43:* "Examples of the effect of our BRDF parameters. Each parameter is varied across the row from zero to one with the other parameters held constant." (Burley 2012)

Let's now analyze better why these parameters are needed and how they are used.

### 3.5.2 *Diffuse Term*

Here and in the next sub-sections, we'll express the BRDF function as

$$f(\vec{L}, \vec{V}) = \texttt{diffuse} + \frac{D(\theta_H)F(\theta_D)G(\theta_L, \theta_V)}{4\cos\theta_L \cos\theta_V}$$

Where we need to decide on a $\texttt{diffuse}$ term. $\theta_L$ and $\theta_V$ are the angles formed by the $\vec{L}$ and $\vec{V}$ vectors with the surface normal, $\theta_H$ is the angle between the normal and the half vector $\vec{H}$, and $\theta_D$ is the "difference" angle between $\vec{L}$ and the half vector (or, symmetrically, between $\vec{V}$ and $\vec{H}$).

Earlier, we introduced the *Lambertian diffuse model* (see section 3.3.3). This model assumes that refracted light gets scattered enough inside a surface to lose all its directionality when being re-emitted. For this reason, light at a surface point is re-emitted in all directions uniformly.

The Lambertian model is widely used in rendering, as it is a good approximation for various types of surfaces. However, very few materials exhibit *true* Lambertian response. It can be observed that many materials show a drop in *grazing retroreflection* (that is, retroreflection at grazing angles), while many others show a peak (Burley 2012). *Retroreflection* is a type of reflection phenomena where the material scatters light primarily back along the incident direction (Pharr, Jakob, and Humphreys 2023).

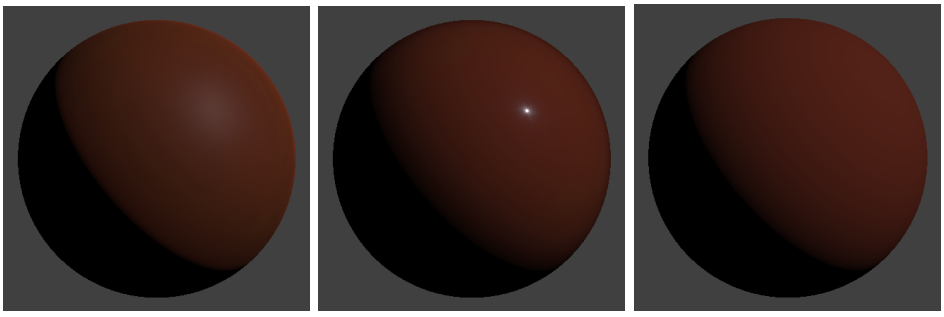


*Fig. 44:* "Point light response of *red-plastic*, *specular-red-plastic*, and *Lambert diffuse*." (Burley 2012)

Notice in Figure 44 that, at *edges*, the *smooth surface* (middle) tends to be in *shadow*, while the *rough surface* (left) exhibits *reflection peaks*. Grazing shadows for smooth surfaces are actually a product of the Fresnel equations, since, at grazing angles, all light is reflected and none is reserved to diffuse reflection (making the diffuse contribution zero).

On the other hand, the reflective peaks at the edges of rough surfaces aren't usually considered by diffuse models.

This is why at Disney it was decided to develop a new, empirically based model for diffuse retroreflection. The model transitions from the classic diffuse Fresnel shadow for smooth surfaces into an added highlight for rough surfaces, as the *roughness* parameter grows in value.

The base Disney diffuse model is defined as

$$f_d = \frac{baseColor}{\pi}(1 + (F_{D90} - 1)(1 - \cos\theta_L)^5)(1 + (F_{D90} - 1)(1 - \cos\theta_V)^5) \tag{3.18}$$

Where

$$F_{D90} = 0.5 + 2\,roughness\cos^2\theta_D \tag{3.19}$$

In (3.18), we can see that a Fresnel factor multiplies the base Lambertian model. There is not much information about this formula, other than the fact that it was derived from the Fresnel factor[19] $(1 - F(\theta_L))(1 - F(\theta_D))$ using Schlick's approximation (3.17).

In (3.19) the grazing retroreflection response is modified to go to a specific value determined from roughness, rather than zero. According to Burley,

> "This produces a diffuse Fresnel shadow that reduces the incident diffuse reflectance by 0.5 at grazing angles for smooth surfaces and increases the response by up to 2.5 for rough surfaces."
> (Burley 2012)

The base Disney diffuse (3.18) is blended together with another model, which is a rough approximation of subsurface scattering at small scattering path lengths, inspired by the *Hanrahan-Krueger subsurface BRDF*.

### 3.5.3 *Specular D*

In the Disney BRDF, the normal distribution function D follows the *Generalized-Trowbridge-Reitz distribution* (*GTR*):

$$D_{GTR}(\theta_H) = c/(\alpha^2\cos^2\theta_H + \sin^2\theta_H)^\gamma$$

Where c is a scaling constant and $\alpha$ is a *roughness* parameter, derived as $\alpha = roughness^2$. This squaring operation was found to produce

19 Where refraction due to the Fresnel law is considered twice, once on the way in and once on the way out of the surface.

a more *perceptually linear* change in roughness. The GTR model is a generalization of the *Trowbridge-Reitz distribution* (*TR*), where $\gamma = 2$. This is also equivalent to the so-called *GGX distribution* (Burley 2012).

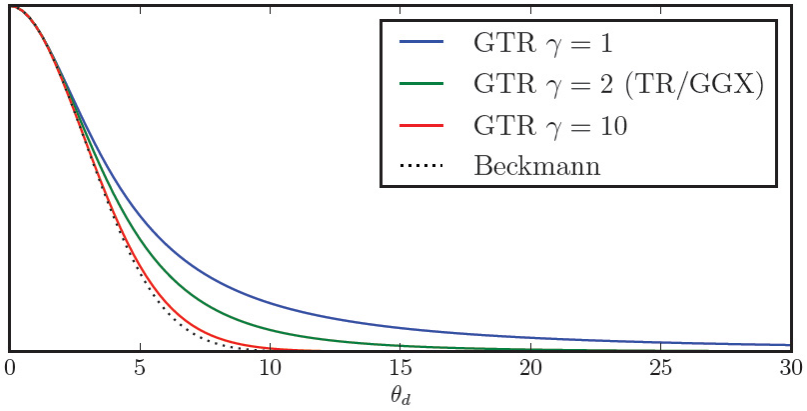



*Fig. 45:* "GTR distribution curves vs $\theta_h$ for various $\gamma$ values." (Burley 2012)

Both the primary specular and the clearcoat lobe use GTR, with exponents $\gamma = 2$ and $\gamma = 1$ respectively. Here are the complete function definitions:

$$D_{GTR_1}(\theta_H) = \frac{\alpha^2 - 1}{\pi \log \alpha^2} \frac{1}{(1 + (\alpha^2 - 1)\cos^2\theta_H)}$$
$$D_{GTR_2}(\theta_H) = \frac{\alpha^2}{\pi} \frac{1}{(1 + (\alpha^2 - 1)\cos^2\theta_H)^2}$$

We can also derive *anisotropic forms*. In the case of $D_{GTR_2}$, this is

$$D_{GTR_{2aniso}}(\theta_H) = \frac{1}{\pi} \frac{1}{\alpha_x \alpha_y} \frac{1}{(\sin^2\theta_H(\cos^2\phi_H/\alpha_x^2 + \sin^2\phi_H/\alpha_y^2) + \cos^2\theta_H)^2}$$

Where $\phi \epsilon [0, 2\pi]$ is the *azimuthal angle*[20], and the two roughness parameters $\alpha_x$ and $\alpha_y$ are calculated from $\alpha$ as follows.

$$\begin{aligned} aspect &= \sqrt{1 - 0.9 anisotropic} \\ \alpha_x &= roughness^2/aspect \\ \alpha_y &= roughness^2 \cdot aspect \end{aligned}$$

The 0.9 factor is used to limit the aspect ratio to 10 : 1.

20 The *azimuthal angle* is the angle of rotation of $\vec{H}$ around $\vec{N}$.

The primary specular lobe may be anisotropic, while the clearcoat lobe is always isotropic.

The Disney BRDF doesn't describe a material with an explicit index of refraction, and instead uses the base reflectance $F_0$ (seen in section 3.4.1), which corresponds to the *specular* parameter. This parameter is remapped linearly to the range $[0.0, 0.08]$, which corresponds to values in the range $[1.0, 1.8]$ for the IOR's real part. Most common materials can be found in this range. However, the *specular* parameter can also be pushed beyond 1 to reach higher IORs.

The clearcoat layer uses a fixed IOR of 1.5, representative of *polyurethane*. Artists can scale its overall strength using the *clearcoat* parameter.

### 3.5.4 *Specular F*

For the Fresnel reflectance $F$, the Schlick approximation was deemed sufficient:

$$F_{Schlick} = F_0 + (1 - F_0)(1 - \cos\theta_D)^5$$

$F_0$ is achromatic for dielectric and tinted for metals. The response always becomes achromatic at grazing incidence, as all light is reflected.

### 3.5.5 *Specular G*

In microfacet models, sometimes we can apply a derivation method to get a shadowing function G directly from the microfacet distribution, D. This process is called *Smith G derivation*, and was also used in the case of the Disney BRDF, where the G term derives from the GGX distribution.

Before being passed to G, the *roughness* parameter is scaled from $[0, 1]$ to $[0.5, 1]$. This is done because it has been observed that small roughness values result in exaggerated reflection gains.

Remember from earlier that the roughness is squared. This is done after the scaling, resulting in a roughness parameter $\alpha_g$ for G that is $\alpha_g = (0.5 + roughness/2)^2$.

For the clearcoat specular layer, the GGX G is used simply with a fixed roughness of 0.25.

### 3.5.6 *Sheen*

The MERL BRDF database contains measures indicating that materials made of *fabric* exhibit *tinted specular response* at *grazing angles*. This makes sense, since cloth often has *transmissive fibers* which can pick up the material color also near object silhouettes. Another interesting observation is that, for a fixed *roughness* value, fabric seems to have a stronger Fresnel peak than other kinds of materials. The qualities we just mentioned are collectively referred to as the *sheen* of the fabric.

Hence, another lobe was added to the Disney BRDF to show extra grazing reflectance for fabric. Since this extra reflectance is very Fresnel-like, it is modeled with the Schlick-Fresnel "shape" $\mathrm{sheen} \cdot (1 - \cos\theta_D)^5$. The sheen can also be tinted towards the base color with the use of the *sheenTint* parameter.

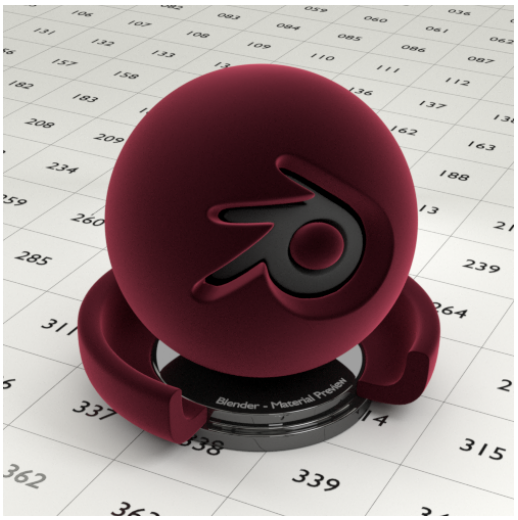


*Fig. 46:* Example of a *sheen* effect in Blender. Source: *Sheen BSDF*, Blender 4.4 Manual. Licensed under the CC-BY-SA 4.0 license.

## 3.6 A FINAL NOTE

Our overview of the Disney "Principled" BRDF is now complete. This point also marks the end of the theoretical part of this thesis.

All in all, we should feel quite satisfied. Starting with only a few base mathematical notions, we have assembled a rich toolbox for the imitation of real-world light. Next, our tools will be put to the test inside an actual

program, and we'll be able to witness ray tracing and the Disney BRDF at work.

We will later comment on the qualities and limitations of these techniques. However, we should acknowledge right away that they could be made more complete. Most importantly, the version of ray tracing presented here is quite "bare-bones". Better and more believable visuals can be achieved through *path tracing*, an advanced form of ray tracing which allows for *global illumination* effects. A short introduction to global illumination and path tracing can be found in Appendix A.

# 4

# THE BOXOFSUNLIGHT RENDERER

This chapter discusses the implementation of *BoxOfSunlight*[1], a 3D rendering application written in C++ with the aid of the *OpenGL* graphics API (version `4.6`)[2]. BoxOfSunlight is a simple ray tracer which takes as input an arbitrary amount of spheres and/or triangles, as well as a light source, to then compute the reflection of light on these geometries by applying the Disney "Principled" BRDF. The light source can be either a *point light* or an *HDRI environment map*, the latter of which is wrapped on the faces of an infinitely large surrounding cube (this is also called a *cubemap*).

## 4.1 THE OPENGL API

BoxOfSunlight is written in C++ in the sense that, at startup, it executes C++ code to prepare all that's needed for rendering. Basically, this consists in creating a window, initializing OpenGL, and setting up all the data required for ray tracing. The actual ray tracing code can be found inside a *compute shader*, a small program written in the *OpenGL Shading Language* (*GLSL*)[3] that integrates into the OpenGL rendering pipeline. To understand the inner workings of BoxOfSunlight, it is therefore necessary to know at least a little about OpenGL.

In this section, we offer a simplified "lightning tour" of the OpenGL API, to present all the concepts needed for our purposes. Before we get into that, though, a clarification should be made as to why OpenGL was chosen to be our program's "backbone".

1 The name "Box of Sunlight" is a reference to Tom DiCillo's 1996 movie *Box of Moonlight*.
2 The complete source code of the project can be found at `https://github.com/leonardozt/BoxOfSunlight`
3 GLSL is a C-like programming language specifically tailored for graphics.

### 4.1.1 *Why OpenGL*

The material that follows is based on the official OpenGL Wiki[4]. It should also be mentioned that, for writing this whole chapter, the online *Learn OpenGL* tutorial series has also been used extensively as a resource[5].

First of all, let's explain a little better what OpenGL actually *is*.

*OpenGL* stands for *Open Graphics Library*. It is an industry-standard API for drawing hardware-accelerated 2D and 3D graphics using the GPU. It is maintained by the Khronos Group, and implemented by graphics card vendors.

Basically, OpenGL consists in a *specification*; that is, a list of *functions*, along with *descriptions* which specify what should be the result/output of each function. It is then up to the developers to implement OpenGL on a given platform, like an operating system, and to make sure the requirements given by the specification are satisfied. In all the three major desktop platforms (Linux, macOS, and Windows), OpenGL more or less comes with the system. To develop BoxOfSunlight, Windows was used.

The main advantage of writing an OpenGL-tailored renderer, instead of, say, one written in plain C++, is that OpenGL allows us to leverage the power of a computer's GPU. GPUs are much faster than CPUs at calculations which involve intrinsically parallel problems (colloquially called "embarrassingly parallel"), such as the ones found in computer graphics[6].

At the beginning of Chapter 3, we said that OpenGL is *rasterization-based*. This might be causing some confusion now, given the fact that BoxOfSunlight is a *ray tracer*. How can it be that a renderer, which at its core employs a rasterization-based graphics library, is actually a ray tracer? Well, we'll see that this "transition" from rasterization to ray tracing is made possible by *compute shaders*. These small programs are reserved a special spot in the OpenGL graphics pipeline, making them independent from other rasterization-based processes that might also be going on. Obviously, doing ray-traced rendering with OpenGL is not the optimal solution, and would soon prove unsustainable for non-trivial ray

4 The OpenGL Wiki can be read at `https://www.khronos.org/opengl/wiki` (`last accessed: 28.03.2025`)

5 Learn OpenGL website: `https://learnopengl.com/` (`last accessed: 28.03.2025`)

6 Think, for instance, about ray tracing. Rays travel in a scene completely independently from each other, and hence their intersection and reflection calculations can be parallelized without much effort.

tracers. In our case, however, the functionalities offered by OpenGL are more than enough.

Still, choosing OpenGL might seem odd. There are, in fact, other libraries which allow for GPU-accelerated graphics, such as NVIDIA's *CUDA*[7], or *Vulkan*[8]. Both of these libraries offer actual GPU-based ray tracing. For BoxOfSunlight, the choice came down to OpenGL because of familiarity with it from prior use, in the development of other rasterization-based graphics applications.

### 4.1.2 *Base Concepts*

We'll now introduce some fundamental OpenGL terminology. We'll only be going through the "ABC": for a more advanced description, refer to the OpenGL Wiki (previously linked).

At its core, OpenGL employs a *state machine*, which keeps track of all the OpenGL *state variables*. The state variables define how OpenGL should currently operate, and can be set through OpenGL's functions. The state of OpenGL is commonly referred to as the *OpenGL context*.

An application can create multiple OpenGL contexts, and each context can represent a separate viewable surface, like a window. In BoxOfSunlight, there is a single OpenGL context associated with a window, which is created with the *GLFW* library[9].

Each context has its own set of *OpenGL objects*. An OpenGL object is a collection of options representing a *subset* of OpenGL's *state*.

Now, all of this is used in the *OpenGL rendering pipeline*, the sequence of steps taken by OpenGL to draw graphics on screen. The OpenGL rendering pipeline is lengthy and complicated. All we need to know is that it involves taking vertices as inputs and connecting them into triangles, to then project these triangles from the 3D space onto the 2D screen, and finally color the pixels covered by the projected triangles. This is, in fact, how we said rasterization works in Chapter 3. Our simple summary is hiding many steps taken by OpenGL to make this rasterization-based process actually work. These steps (or *stages*) are independent and or-

7 See `https://developer.nvidia.com/cuda-toolkit` (`last accessed: 28.03.2025`)

8 *Vulkan* was introduced by the Khronos group as a successor to OpenGL. In general, it allows developers to have more low-level control.

9 *GLFW* is a library written in C which, among other things, can be used to create windows and handle user input for OpenGL.

dered in a *pipeline*, where the output of a stage is the input of the next one.

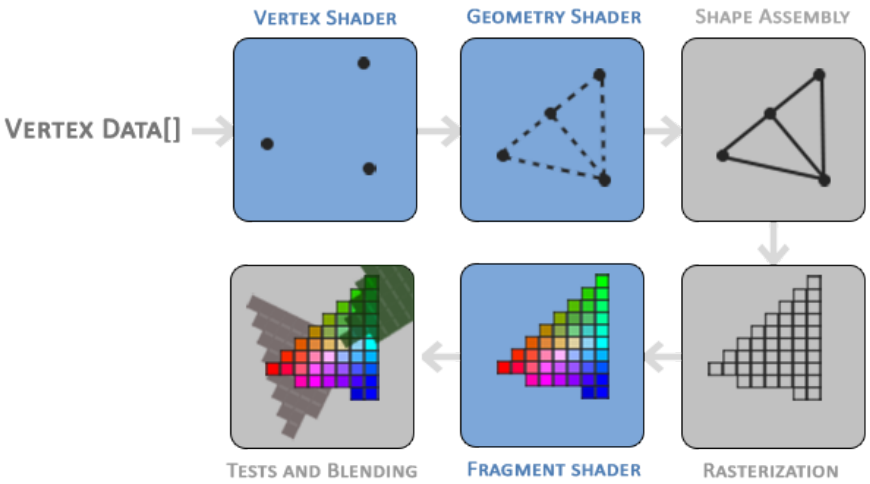



*Fig. 47:* Simplified view of the OpenGL rendering pipeline. The blue boxes are programmable stages. Source: *Hello Triangle*, Learn OpenGL tutorial series. Licensed under the CC BY 4.0 license.

Some stages in the OpenGL rendering pipeline are *programmable* (that is, we can write their code). The two main programmable stages are controlled by a *vertex shader* and a *fragment shader*. A *shader* is a user-defined program, written in GLSL. Speaking very broadly, we can say that the main purpose of a *vertex shader* is to take the 3D coordinates of vertices and apply transformations on them (like scales, rotations and translations) to turn them into *different* 3D coordinates. As for the *fragment shader*, its job usually is to do *shading*; that is, it determines the final colors of pixels based on light calculations. These calculations are, actually, pretty much the same as the ones done in BoxOfSunlight (the evaluation of a BRDF). For this reason, it wasn't strictly necessary for us to rely on ray tracing rather than rasterization. Nonetheless, ray tracing was deemed as the more "natural" choice for our program, since, as seen in Chapter 3, it is more directly inspired by the physics of light. Also, building BoxOfSunlight around ray tracing gives us the potential to, perhaps one day, extend it into a *path tracer* to simulate proper *global illumination* (see Appendix A).

Remember that OpenGL employs a state machine, composed of *objects*. A so-called *program object* is an example of an OpenGL object. We create

it by "attaching" to it the compiled source code of one or more shaders (for example, a vertex shader and a fragment shader). The program can then be *activated* to be used for rendering.

### 4.1.3 *Compute Shaders*

Another type of shader, which for us is of particular importance, is represented by *compute shaders*. A compute shader is a general-purpose shader, used to compute arbitrary information.

Compute shaders work very differently from vertex/fragment shaders. Whereas vertex shaders operate on *vertices*, and fragment shader on *fragments*[10], the data elements of a compute shader are much more abstract, and their meaning is left up to interpretation.

In fact, we say that compute shaders operate in an abstract "space" called a *compute space*. Again, it is up to each compute shader to decide what the space means. The compute space is divided into *work groups*, which are distributed in the three dimensions of the space ("X", "Y" and "Z") (see Figure 48). Each work group executes all the instructions written in the compute shader, potentially in parallel with other groups.

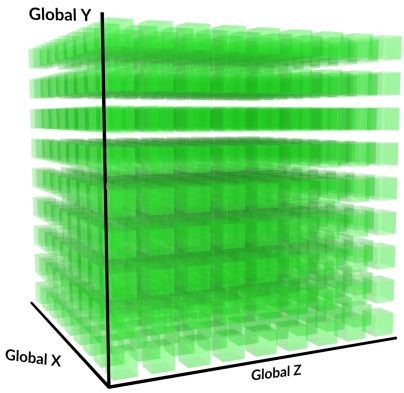



*Fig. 48:* The *compute space*. Each cube represents a work group. Source: *Compute Shaders*, Learn OpenGL tutorial series. Licensed under the CC BY 4.0 license.

The size of the compute space can be decided at runtime. This dictates the number of work groups that OpenGL uses in each dimension, and allows us to perform also two-dimensional or one-dimensional compute

10 A fragment, in OpenGL, is all the data needed to render a single pixel.

operations, other than 3D ones, by setting the sizes of the dimensions we don't need to 1.

Work groups are called "groups" because they may be subdivided further into multiple compute shader invocations, the number of which is called the *local size* of the work group. When running BoxOfSunlight, this feature won't be used, meaning that the local sizes of work groups will always be kept to 1.

What we'll be doing to render an image is *dispatch* (that is, execute) one *work group* for each *pixel*. We'll be using only the first two dimensions of the compute space (X and Y). Given a work group with $w_x$ and $w_y$ as its first two compute-space coordinates, we'll associate to it the pixel with viewport coordinates $[w_x, w_y]^T$ (see Chapter 3 section 3.1.2).

So, for each pixel, the compute shader code will be run once. Each compute shader invocation will figure out which pixel it must compute by checking one of the compute shader's built-in input variables[11], called `gl_WorkGroupID`. This 3-component vector contains the XYZ coordinates of the workgroup that the compute shader invocation belongs to, so its first two components will also be the coordinates of the pixel.

Compute shaders can receive user input through *uniform variables* (also used by vertex and fragment shaders). These variables are global and shared by all work groups.

For example, in BoxOfSunlight, the scene's *point light*[12] is passed as a uniform variable, which is declared in the compute shader as:

```
uniform PointLight pLight;
```

Where `PointLight` is a previously defined GLSL struct:

```
struct PointLight {
   vec3 position; // position of light in 3D space
   vec3 emission; // emitted radiance
};
```

Another way of passing input to a compute shader is through *buffer objects*. A buffer object is an OpenGL object used to store a block of data on the GPU's memory. In particular, we're interested in *Shader Storage Buffer Objects* (*SSBO*), since they allow us to pass *variable amounts* of data

11 Built-in input variables are handled by OpenGL, and have different values for each compute shader invocation.

12 We first mentioned *point lights* at the end of section 3.1.4 of the previous chapter.

to a compute shader. In fact, the amount of data passed by an SSBO is determined at *runtime*.

To give an example, we can see how BoxOfSunlight uses an SSBO to receive spheres as input. In the compute shader, spheres are represented as `Sphere` structs:

```
struct Sphere {
   vec4 center;
   float radius;
};
```

The reason we memorize the sphere's center as a 4-component vector, rather than a 3D one, stems from low-level details related to the memory layouts of SSBOs[13]. Putting it simply, 3-component vectors don't work well with SSBOs, so we're obliged to turn them into `vec4`s, and simply ignore the fourth component.

The SSBO `spheresBuf`, used to pass spheres to the compute shader, is defined as:

```
layout(std430, binding = 1) readonly buffer spheresBuf {
   Sphere spheres[];
};
```

Ignore the `layout` part (again, a low-level detail): all that matters is that the `spheresBuf` SSBO contains an array of spheres (called `spheres`) without a predetermined size. In fact, the size of `spheres` will be determined at runtime, based on how many spheres are passed to the shader in the moment.

### 4.1.4 *Textures and Images*

A useful type of uniform variable is a *sampler*, which is used to access *textures*. A texture is an OpenGL object that contains one or more *images*. During rendering, we'll be reading image data from two types of textures: *2D textures* and *cubemaps*, which are accessed through the two GLSL sampler types `sampler2D` and `samplerCube`, respectively.

A 2D texture will contain a 2D image, and will be used to do *texture mapping* on spheres and triangles. Texture mapping is an important technique in computer graphics. To put intuitively, it consists in "gluing"

13 More on memory layouts can be found in the OpenGL Wiki, at https://www.khronos.org/opengl/wiki/Interface_Block_(GLSL) (last accessed: 28.03.2025)

part of a texture image onto a geometric primitive (see Figure 49). This allows us to make the texture-mapped objects more visually complex, without having to change their base geometry (Gortler 2012). A point on a 3D object, call it $\tilde{q}$, will possess some two-dimensional *texture coordinates* $[u, v]^T$ (also called *UV coordinates*). The texture coordinates $[u, v]^T$ will correspond[14] to some pixel coordinates $[p_x, p_y]^T$, which will indicate what pixel of the 2D image needs to be mapped on $\tilde{q}$.

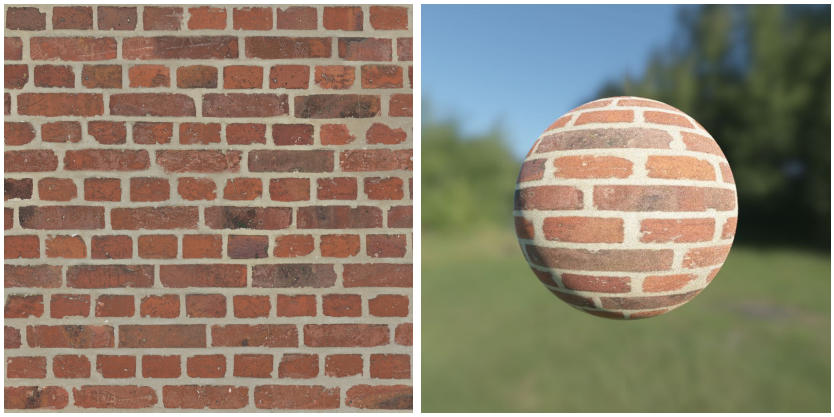


*Fig. 49:* A brick texture (left) mapped on a sphere (right). Texture from ambientCG (*Bricks 085*).

On the other hand, *cubemap textures* will be composed of 6 images, each of which will represent the face of a cube. Imagining that we are at the origin of the virtual scene's 3D space, and that we're surrounded by this cube, OpenGL allows us to use a *vector*, instead of texture coordinates, to get the color of the cubemap in the *direction* pointed by that vector (see Figure 50).

Beside texture samplers, we can also use *image* variables to read and write information directly from/to *images*. These variables are also uniform, and there are many types of them. The type used for 2D image reads/writes is `image2D`.

By using images directly, we take on the responsibility of managing by ourselves what would be managed by OpenGL with *textures* (for example, write operations on images are not automatically coherent). This, however, also allows us to write to individual pixels, which is something that we

14 Note that texture and pixel coordinates are not the same: texture coordinates are *independent* from the number of pixels in the image. They are, in general, continuous and belong to the $[0, 1]$ interval. (Gortler 2012)

couldn't do with texture samplers (or other mechanisms that involve textures).

Writing to a uniform `image2D` variable is a common method for storing a compute shader's output, and is the one we'll be using.

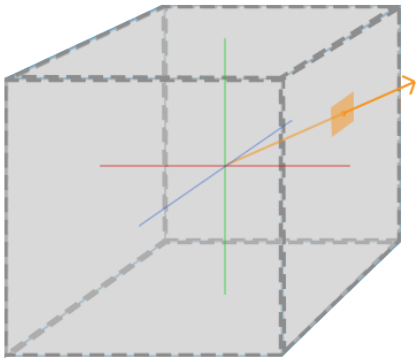


*Fig. 50:* Cubemap sampling. Source: *Cubemaps*, Learn OpenGL tutorial series. Licensed under the CC BY 4.0 license.

## 4.2 PROJECT STRUCTURE

Now we should be able to (mostly) understand how BoxOfSunlight works. Note that we won't be getting into *every* detail, but only focus on the main logic, both from the C++ and the GLSL side.

### 4.2.1 *C++ Side*

As a reference for C++ mechanisms and best practices, the classic book *The C++ Programming Language*, by Bjarne Stroustrup, was used.

The C++ side of BoxOfSunlight was mainly inspired in its structure by the open-source project *GLSL-PathTracer*[15].

All of the classes and structs in the C++ project[16] are grouped in a common namespace, called `BOSL` (for "Box Of Sun-Light"). Instances of these classes are then created and used in a specific order inside the main file (*Main.cpp*). We'll now see how this is done.

15 From GitHub, at https://github.com/knightcrawler25/GLSL-PathTracer (last accessed: 28.03.2025)

16 See https://github.com/leonardozt/BoxOfSunlight/tree/master/src

Before we create any object, the GLFW library needs to be initialized. After that, a `BOSL::Window` object is instantiated so that it handles an on-screen window (created with GLFW). The window has an OpenGL context associated with it, which we'll use for rendering.

After we've initialized OpenGL as well, we can start putting together our virtual scene's data. This is done inside a `BOSL::Scene`-type object. For example, a `BOSL::Scene` object contains a `std::vector` of spheres, where each sphere is a C++ struct defined in the same exact way as in the compute shader (see section 4.1.3). By altering the data members of the `BOSL::Scene` object, we can also specify a few rendering options, such as whether BoxOfSunlight should use a point light or a cubemap as the 3D scene's light source (this is simply done through boolean variables).

Another (perhaps more interesting) example of data members contained in a `BOSL::Scene` instance is represented by `BOSL::Texture` objects. `BOSL::Texture` belongs to a group of classes in the project meant for *wrapping OpenGL objects*, in that their instances contain OpenGL objects and handle all the API calls needed to manage them, effectively hiding them from view. `BOSL::Texture`-type objects are wrappers for OpenGL texture objects. To provide an additional example, `BOSL::Program`-type objects are used in BoxOfSunlight to wrap OpenGL program objects. Using wrapper classes for OpenGL objects is quite useful and makes the code more readable, but it also entails some additional work. This is especially true when we're trying to follow C++ principles like *Resource Acquisition Is Initialization* (*RAII*). For more on the topic see the OpenGL Wiki, at the entry *Common Mistakes*[17].

Going back to the project's main file, what's done next is the creation of a `BOSL::Renderer`-type object. `BOSL::Renderer` objects compile and link the GLSL shaders of the project into *two* distinct OpenGL program objects. One program object contains the compiled *compute shader* code, while the other is made up of a *vertex shader* and a *fragment shader*. We'll soon see the reason for this. `BOSL::Renderer` objects are assigned a `BOSL::Scene` through their constructor method. After having compiled the shaders, they pass to them all of the scene's data.

Finally, in the main file, the `BOSL::Renderer` object is used to render multiple frames, by repeatedly calling its `render()` function, and the `BOSL::Window` object is updated each time (through its `update()` function)

17 URL: https://www.khronos.org/opengl/wiki/Common_Mistakes (look for the section called *The Object Oriented Language Problem*) (`last accessed: 28.03.2025`)

to show the new frames on screen. Also, during the whole setup and rendering process we've just seen, errors are handled with C++ exceptions. More specifically, the `BoxOfSunlightError` class is used, which is defined as an extension of the `std::runtime_error` class, from the C++ standard library.

### 4.2.2 *GLSL Side*

As we just mentioned, there are, in total, three GLSL shaders: a compute shader, a vertex shader and a fragment shader[18]. Also, we said that there are two OpenGL program objects, one of which contains only the compute shader, while the other is made up of the vertex and fragment shaders.

We use these two program objects in the following way. In a first "render pass", we activate the compute shader's program object, and run it. This way, the compute shader produces an image through ray tracing. In the second render pass, we activate the *other* program object, whose vertex and fragment shaders are tasked with showing this image on screen[19]. To be clear, this is done through *rasterization*.

In fact, to display the image on screen we use a simple trick, which consists in mapping it as a *texture* on a *quadrilateral* (made up of two triangles) that covers the *entire screen*. This is, pretty much, all that happens inside the vertex and fragment shaders. So, to summarize, the output image is first *produced* with *ray tracing*, but then *shown* to the user with *rasterization*.

We have established that a compute shader invocation should compute the color of a single pixel (section 4.1.3). Remember now our discussion on *antialiasing*, from Chapter 3 (section 3.1.3). BoxOfSunlight uses a simple version of an antialiasing technique called *temporal antialiasing*[20]. Basically, what BoxOfSunlight does is it averages samples from *multiple frames*. Each frame (that is, image) is generated by taking only one sample per pixel, from a random point around that pixel. This is done in the way suggested in Chapter 3, including the application of a *tent filter* to the randomly-sampled points. The compute shader receives in input

18 See https://github.com/leonardozt/BoxOfSunlight/tree/master/shaders (note that the compute shader is divided into multiple files)

19 More precisely, the shaders memorize the image in the GLFW window's framebuffer, and then GLFW is asked to show it on screen.

20 See for example NVIDIA's *TXAA* technique: https://docs.nvidia.com/gameworks/content/gameworkslibrary/postworks/product.html (last accessed: 28.03.2025)

an `image2D` uniform variable, which contains the running average of previously generated frames, calculated on a per-pixel basis. To store a new pixel color, the compute shader adds it to the running average, and memorizes the result in another `image2D` variable, which serves as the compute shader's output.

The compute shader casts a ray into the scene and verifies which objects are intersected by applying the intersection formulas from Chapter 3. The closest intersection is found through a *linear search*, meaning that we check each object for intersections with the ray, to then only keep the nearest one. If the ray doesn't intersect any object, either we define its radiance as $[0, 0, 0]^T$ (the color black), or we get a radiance value by sampling a cubemap (if there is one). In fact, as already mentioned, BoxOfSunlight can optionally use cubemaps as sources of illumination, that is, *radiance*[21]. Remember that the cubemap surrounds the 3D scene, meaning that it constitutes its *background*. For this reason, if a ray misses the scene's geometries, the cubemap should be sampled directly (in the direction of the ray). Since cubemaps are light sources, we'll see that they are sampled also during light reflection calculations.

*Fig. 51:* Faces of a cubemap. Built from the HDRI *Meadow 2*, by Sergej Majboroda on Poly Haven.

21 In computer graphics, this technique is commonly called *image-based lighting*.

As shown in Figure 51, the faces of a cubemap are usually photographs of some sort of real-world environment. Frequently, the environment is illuminated by the Sun, hence the name "Box of Sunlight".

A cubemap can be used as a proper source of *radiance* only if it's composed of *HDR images* (in a moment, we'll see what these are).

Also, recall that cubemap sampling is based on the assumption that we're looking at the cubemap from the 3D scene's origin. We'll pretend that this is always the case, which makes sense if we think of the cubemap's faces as being *infinitely far away* from us.

Finally, note that our cubemaps can be interpreted conceptually as simple approximations of *global illumination* (Appendix A). In fact, they depict light that, after being emitted by the Sun, has already been reflected/scattered (by the sky, by the ground, etc.). For this reason, the colors of 3D objects will be tinted towards those of the environment, creating a very natural "fit" between the geometries and the surrounding scene.

Having found the closest ray-object intersection, we get to the heart of the program: solving the *reflection equation*. Being this such an important part of BoxOfSunlight, there's an entire section of this chapter that's dedicated to it (4.3). For now, however, let's delay its discussion, and assume that the reflection equation has been solved. This means that we have the outgoing radiance $L_o$ (`Lo` in the code), which is leaving the surface and going towards the pixel.

The value of the outgoing radiance `Lo`, a 3-component floating-point vector, is given directly to the output pixel. In fact, the output image of the compute shader is memorized in a "free" floating-point format, rather than standard RGB, meaning that the pixel values are not constrained to lie in the $[0, 1]$ range. This $[0, 1]$ range, employed by RGB, is also called *Low Dynamic Range* (*LDR*).

To be more precise, what we do is use 32-bit floating points, which allow us to store a really wide range of color values. The result of this is called a *High Dynamic Range* image (*HDR*)[22]. For us, HDR images always memorize *radiance* values.

Remember that the compute shader's output image is passed to the vertex and fragment shaders for displaying. To show the HDR image on screen, the fragment shader computes a transformation from HDR to LDR; that is, it does *tone mapping*. In our case, we use an exponential tone

22 For more on HDR images, see https://learnopengl.com/Advanced-Lighting/HDR (last accessed: 28.03.2025)

mapping operator, as done in the Learn OpenGL tutorial series (in the above-linked article):

```
// Convert the pixel color from HDR to LDR
vec3 mapped = vec3(1.0) - exp(-hdrColor * exposure);
```

Where `exposure` is a uniform variable defined in the fragment shader. As suggested by the name, its value can be set to make the output image more or less bright (like in photography).

## 4.3 SOLVING THE REFLECTION EQUATION

We'll now take a look at how the compute shader finds (or approximates) a solution to the reflection equation, which we defined in Chapter 3 as

$$L_o(\tilde{p}, \vec{\omega}_o) = \int_{H^2(\vec{N})} f_r(\tilde{p}, \vec{\omega}_o, \vec{\omega}_i)\, L_i(\tilde{p}, \vec{\omega}_i)\, cos\theta_i\, d\omega_i$$

Keep in mind, again, that we'll be ignoring various (uninteresting) implementation details.

### 4.3.1 *Point Light*

First of all, the compute shader calculates the *view direction* (indicated as `V` in the code), which is oriented from the ray-object intersection point `p` towards the camera. This direction is used to evaluate the Disney BRDF. The other important parameter of a BRDF, as we know, is the *light direction* (`L`). In the case of a point light source, a BRDF needs to be evaluated for only one pair of `L`, `V` directions (see section 3.3.2). So, to compute the outgoing radiance `Lo`, all we need to do is:

```
// sample point light
vec3 L = normalize(pLight.position - p);
vec3 Li = pLight.emission; // incident radiance
float NdotL = max(dot(N, L), 0.0);

vec3 Lo = disneyBRDF(L, V, N, T, B, material) * Li * NdotL;
```

This corresponds to equation (3.15), from Chapter 3:

$$L_o(\tilde{p}, \vec{V}) = f_r(\tilde{p}, \vec{V}, \vec{L}) L_i(\tilde{p}, \vec{L}) cos\theta_i$$

In the above code snippet, `disneyBRDF` is a function, defined elsewhere in the compute shader, which provides an implementation of the Disney

"Principled" BRDF. The original implementation of the Disney BRDF, by Burley et al., is open source, and can be found on GitHub[23]. Our version, `disneyBRDF`, is pretty much identical to it. Inside the function's body, all the calculations described in section 3.5 can be found.

Looking at the `disneyBRDF` function call, we notice that, other than `L` and `V`, it also takes the parameters:

- `N`, the normal vector,
- `T`, which, as we'll see, is the surface *tangent*,
- `B`, which is the surface *bitangent*,
- `material`, the object's material.

The normal vector `N` is, in the simplest case, the normal of the geometric primitive (triangle or sphere) at `p`. However, to give objects a more detailed look, BoxOfSunlight also supports *normal mapping* (or *bump mapping*). In normal mapping, we map normal vectors on surface points with the aid of a 2D texture, also called a *normal map*. Each of the pixels in the 2D texture has some RGB value, which is interpreted as the *orientation* of a normal vector (Gortler 2012) (see Figure 52). To find the normal `N` at a point `p` on an object, we use the texture coordinates of `p` to sample the normal map. This allows us to make surfaces look less smooth.

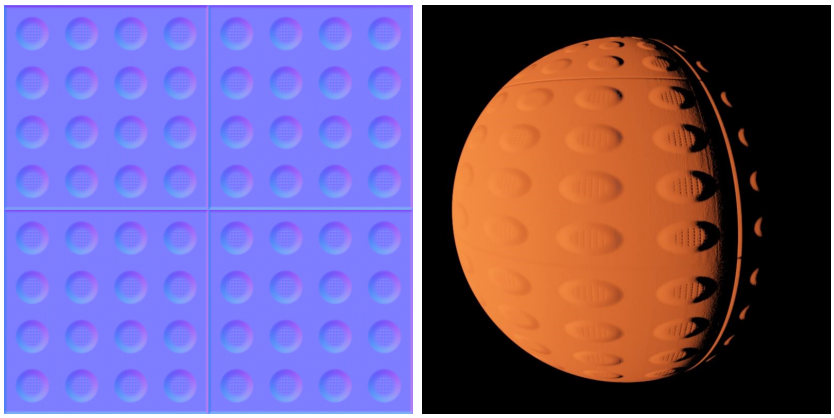


*Fig. 52:* A normal map (left) mapped on an otherwise smooth sphere (right). Notice the non-uniform shadows and highlights.

Mapping normal vectors on a surface is not as straightforward as, say, mapping colors. This is because, to assign a normal $\vec{N}$ to a point $\tilde{p}$, only

23 At: https://github.com/wdas/brdf (last accessed: 28.03.2025)

sampling the normal map is actually not enough. In fact, the normal vector found in the texture is expressed in terms of the surface's *tangent space*. This tangent space has a coordinate frame formed by a *tangent vector* $\vec{T}$, a *bitangent vector* $\vec{B}$ and the *surface normal*[24] The surface normal is the *actual*, *base normal* of the *geometry* at $\tilde{p}$. To avoid confusion, we'll call it the *geometric normal*, and indicate it as $\vec{G}$[25]. The orientations of the three vectors $\vec{T}$, $\vec{B}$, $\vec{G}$ depend on local surface properties, as shown in Figure 53. For more details, refer to section 4.4.

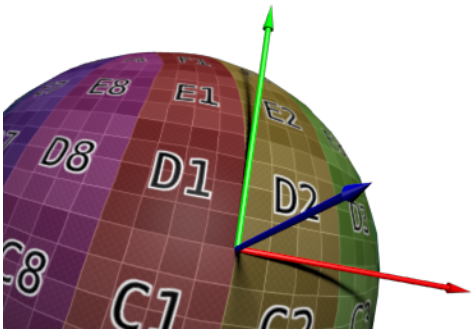



*Fig. 53:* Tangent space for a point on a sphere. The red vector is $\vec{T}$, the green $\vec{B}$, and the blue $\vec{G}$. Source: *Normal Mapping*, OpenGL Tutorial. Licensed under the CC BY-NC-ND 3.0 FR license.

So, the $x, y, z$ components of a normal vector sampled from a normal map, call it $\vec{N}'$, are relative to the $\vec{T}$, $\vec{B}$, $\vec{G}$ axes. That is to say, $\vec{N}' = [x, y, z]^T$ is our normal vector expressed in *tangent space coordinates*. These can be transformed into *world coordinates* (the ones used in the scene) simply by computing:

$$\vec{N} = [\vec{T}, \vec{B}, \vec{G}]\, \vec{N}'$$

Where $\vec{T}$, $\vec{B}$ and $\vec{G}$ are used as the columns of a matrix, called the *Tangent-Bitangent-Normal matrix* (*TBN*), and the multiplication between the TBN matrix (which is just a change of basis matrix) and $\vec{N}'$ computes the correctly expressed normal $\vec{N}$.

Returning to the parameters of `disneyBRDF`, we now understand the meaning of `T` and `B` (the tangent and bitangent vectors). The tangent and

24 Our definition of the tangent space is based on the one by (Hughes et al. 2014), as well as on the Learn OpenGL article on normal mapping, at `https://learnopengl.com/Advanced-Lighting/Normal-Mapping` (`last accessed: 28.03.2025`)

25 To be clear, until now we've been calling this $\vec{N}$. Now, however, $\vec{N}$ will take on a different meaning: it's the normal *sampled* from the *normal map*.

bitangent directions are used by the BRDF to orient *anisotropic highlights* in the specular term.

The last parameter of the Disney BRDF is `material`, the material that the surface is made of. This is an instance of the `Material` struct, which is defined as we would expect; that is, with the parameters from section 3.5.1:

```
struct Material {
   vec3 baseColor;
   float subsurface;
   float metallic;
   float specular;
   float specularTint;
   float roughness;
   float anisotropic;
   float sheen;
   float sheenTint;
   float clearcoat;
   float clearcoatGloss;
};
```

`material`, which is a uniform variable, is also passed to the compute shader as input, just like we saw with the point light or the spheres.

### 4.3.2 *Cubemap*

We still haven't discussed how the reflection equation is solved in the case of cubemaps. This scenario is slightly more complex, as there is actually no analytical solution. What we need to do here is find a good-enough approximation of the integral from the reflection equation. In BoxOfSunlight, this is achieved through (trivial) Monte Carlo sampling: (see Appendix A)

```
// sample cubemap
vec3 Lo = vec3(0);
for (int s = 0; s < hemisphereSamples; s++){
  float u1 = randomFloat(); // uniformly generated random number
  float u2 = randomFloat();
  vec3 L = mat3(T,B,G) * uniformSampleHemisphere(u1, u2);
  float NdotL = max(dot(N, L), 0.0);
  // the texture() function is used to sample the cubemap
  vec3 Li = texture(cubemap, L);
```

```
  Lo += disneyBRDF(L, V, N, T, B, material) * Li * NdotL;
}
Lo *= (2*PI)/(hemisphereSamples);
```

In the code, the light direction `L` is chosen randomly from the hemisphere centered around the geometric normal `G`. To be more precise, we do *uniform sampling* in the *tangent space*. As with normal mapping, the uniformly sampled direction is then multiplied by the TBN matrix to be expressed in world coordinates. Various light directions `L` are used, resulting in various samples (their number is chosen by setting the uniform variable `hemisphereSamples`). The samples are then averaged together by the Monte Carlo estimator:

$$\langle L_o(\vec{\omega}_o)^n \rangle = \frac{2\pi}{n} \sum_{i=1}^{n} f(\vec{\omega}_o, \vec{\omega}_i) L_i(\vec{\omega}_i) \cos\theta_i$$

From Appendix A.

## 4.4 ADDITIONAL NOTES

The compute shader needs a source of random numbers, both for antialiasing and for Monte Carlo sampling. This is somewhat problematic, since GLSL doesn't offer any built-in random number generator. BoxOfSunlight solves the issue by implementing the *xorshift pseudorandom number generator* (Marsaglia 2003) inside the compute shader, which is then used to calculate random floating point values in the range $[0, 1)$.

Earlier, we brought up the problem of *texture mapping*, and said that each point on a 3D object has some texture coordinates $[u, v]^T$ associated with it. How to compute these depends on the geometric primitives that the object is made of. In the case of a triangle, we define UV coordinates for each of its three vertices, which are then *interpolated* to find the UV coordinates of the points *inside* the triangle. For spheres, calculating texture coordinates is a bit more tricky. After all, we're trying to "glue" a flat, 2D image on a shape that is *round*. To do this, we need to think about *longitude* and *latitude*, as explained in Appendix B.

While talking about *normal mapping*, we also introduced *tangent spaces*. Again, the way to compute these varies between triangles and spheres.

Each point of a triangle lies on the same tangent space. To put it briefly, the tangent vector $\vec{T}$ of a triangle is defined as the direction in which the $v$ coordinate (from $[u, v]^T$) remains constant, while the direction

where $u$ is constant corresponds to the bitangent $\vec{B}$ (Hughes et al. 2014). If the triangle is made up of the vertices $\tilde{v}_0$, $\tilde{v}_1$, $\tilde{v}_2$, which possess the UV coordinates $[u_0, v_0]^T$, $[u_1, v_1]^T$, $[u_2, v_2]^T$, the tangent vector $\vec{T}$ can be calculated as follows. (Hughes et al. 2014)

$$\vec{T} = S(\Delta v_2 \, \vec{e}_1 - \Delta v_1 \, \vec{e}_2)$$

Where:

$$\Delta v_i = v_i - v_0$$
$$\vec{e}_i = \tilde{v}_i - \tilde{v}_0$$

And $S$ is a scalar factor. Having calculated the tangent vector $\vec{T}$, and knowing the triangle normal $\vec{G}$, the bitangent $\vec{B}$ can simply be computed as $\vec{G} \times \vec{T}$.

Tangent spaces, like UV coordinates, are also a bit more complicated for spheres than for triangles. A sphere's tangent space is described in Appendix B.

# 5

# EVALUATION

Despite all the baffling physics, the endless mathematics, and the convoluted coding, all we really want to do in animation rendering is creating some good-looking images. We now have a program that, supposedly, is able to perform this task. All that's left to do is run it, and judge if and how well it satisfies our needs.

*Fig. 54:* Three different types of materials, rendered by BoxOfSunlight. From left to right, the materials demonstrate the effects of diffuse, specular and glossy reflection (glossy reflection is due to *clear-coating*).

Luckily, evaluating a renderer is much easier than building it, especially in our case. At the end of the day, all we want from our images is to "look right" to the general viewer. In other words, we want any viewer, even (and especially) those that don't know anything about reflection models, to immediately recognize a rough surface as rough, a metallic surface as metallic, and so on and so forth. Consequently, in a sense, there is no need to explain the qualities and flaws our renderer, as they should be self-evident. If the images look fine, the renderer works, if the images look unpleasant (or bizarre), it doesn't.

However, we have also learned that models for the reflection of light can be quite nuanced, as in the case of the Disney BRDF. There are many

details, in fact, that serve to better bridge the gap of realism. These details matter as well: they influence the viewers' perception of materials, only at a less obvious, and more subconscious level.

The nuances of the Disney BRDF *do* require considerable knowledge to be evaluated, and should be checked with care. To the extent possible, that is what we'll be doing in this chapter: we will examine images produced by BoxOfSunlight, and assess whether the peculiarities of the BRDF are correctly expressed in them.

Other than that, two crucial elements in our discussion will be *image quality* and *runtime performance*. Actually, we will start with these.

## 5.1 IMAGE QUALITY

In this section, we mostly ignore the contents of the Disney BRDF, and instead comment on the *sharpness* of output images, as well as on the *noise* in our approximation of the reflection equation.

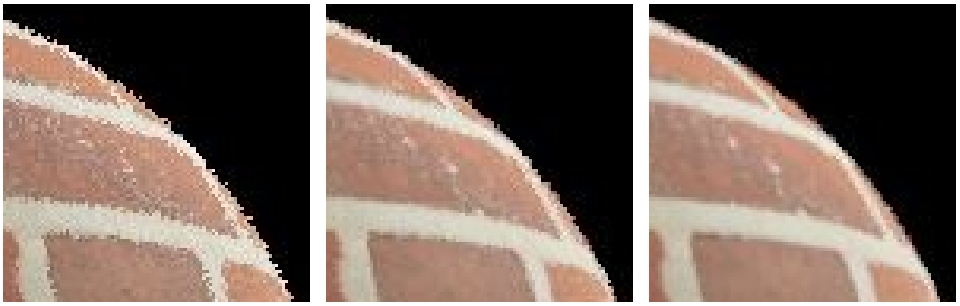
### 5.1.1 *Antialiasing*

*Fig. 55:* Test scene for antialiasing.

We first check whether antialiasing works properly. Recall that BoxOfSunlight employs a simple version of *temporal antialiasing*, which consists in keeping a running average of all frames. The effect of this is that, at first, the image on-screen is somewhat subject to aliasing artifacts. These artifacts are then mitigated with time, as more and more frames are averaged together.

Taking a sphere illuminated by a point light, we map a texture on it and let the program render only a fixed amount of frames, on a 400x400 pixel image (our scene can be observed in Figure 55). We render a single frame,

then 4, then 16, and each time take a screen capture of the resulting image. Remember that, for each frame, only one sample is taken per-pixel. This means that the three images are computed with 1, 4, and 16 total samples per-pixel, respectively. We notice that taking only 1 sample produces very grainy edges, and also visible artifacts in the texture's colors. Both of these effects are quickly attenuated, however, by the successive samples (see Figure 56).



*(a)* 1 sample per-pixel *(b)* 4 samples per-pixel *(c)* 16 samples per-pixel

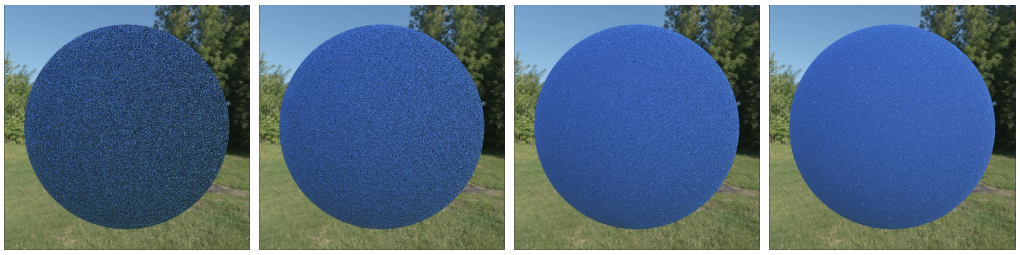
*Fig. 56:* Results of antialiasing with various amounts of per-pixel samples.

### 5.1.2 *Convergence of Monte Carlo Estimator*

When the 3D scene's light source is a cubemap, we use a *uniform Monte Carlo estimator* (Appendix A) to approximate the amount of light that reaches a surface point, and, consequently, also the amount of light that's reflected. Monte Carlo estimators converge to the right answer with a rate of $O(n^{-1/2})$, where $n$ is the number of samples (Pharr, Jakob, and Humphreys 2023). Therefore, to reduce the error by 2, we need 4 times more samples than those used before. We now visualize this property with a couple of quick tests, as described below.

First of all, we should clarify that the "samples" we are talking about here are are not the same as in the previous section. What we are referring to now are samples in the *hemisphere* above a *surface point*, needed to approximately solve the reflection equation (see section 4.3.2). To avoid confusion, we momentarily "turn off" antialiasing, by rendering a single frame and then halting the program (the effect of this is that we take only one per-pixel sample). Meanwhile, we vary the amount of hemisphere samples taken by the Monte Carlo estimator, and verify visually the convergence property mentioned earlier.

Our first test is conducted on a *diffusely reflecting* sphere. We start with $n = 1$ hemisphere samples, and gradually multiply $n$ by 4 until we get to 4096 samples. By doing so, it looks like the error in the Monte Carlo estimate is, indeed, reduced each time by half (Figure 57). In the images (a)-(d), there is a clear, positive progression in denoising. In (e)-(g), on the other hand, the denoising effect is not so obvious. Actually, it might even seem that the image is getting worse, as a great number of small, isolated white dots start to appear. In truth, this is supposed to happen: those white dots gradually make the sphere brighter, as it should be. In fact, the sphere is positioned so that it faces the Sun (that is, the pixels of the cubemap that represent it), and thus we expect it to be quite bright.

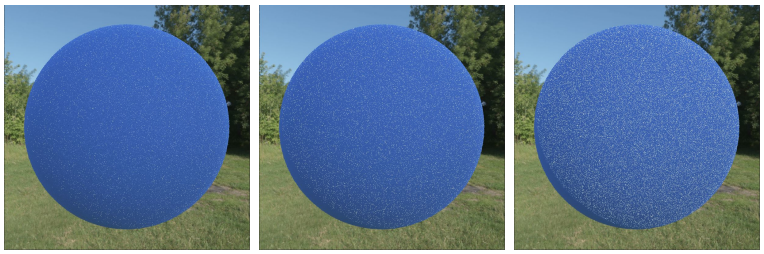


*(a)* 1 hemisphere sample

*(b)* 4 hemisphere samples

*(c)* 16 hemisphere samples

*(d)* 64 hemisphere samples



*(e)* 256 hemisphere samples

*(f)* 1024 hemisphere samples

*(g)* 4096 hemisphere samples

*Fig. 57:* The error in the Monte Carlo estimate is gradually reduced in half, by taking 4 times the amount of samples.

Our final estimate, at 4096 samples, is still far off from the correct solution, as the illumination is considerably noisy and uneven across the sphere. A better result can be achieved by blurring the cubemap, so that we effectively lower its resolution, and make it much easier to sample for incoming radiance. Applying a Gaussian blur to a cubemap before loading it into the scene brings significant benefits to BoxOfSunlight's image quality. Notice, for instance, the drastic amount of noise removed on the diffusely reflective sphere, in Figure 58. Among the images in this

chapter, the vast majority will be generated with cubemap blurring. The blurring operation will not be performed by BoxOfSunlight. It will be, instead, done beforehand, on the cubemap's HDR image files[1].

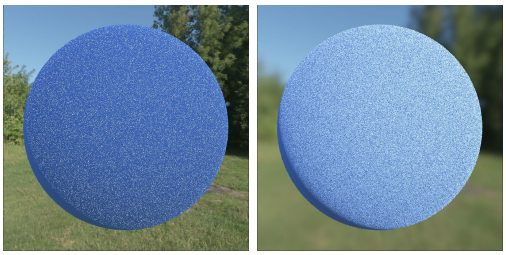


*Fig. 58:* For both images, the same Monte Carlo estimator (at 4096 samples) was used to compute the reflected light. In the image to the right, Gaussian blur was also applied to the scene's cubemap.

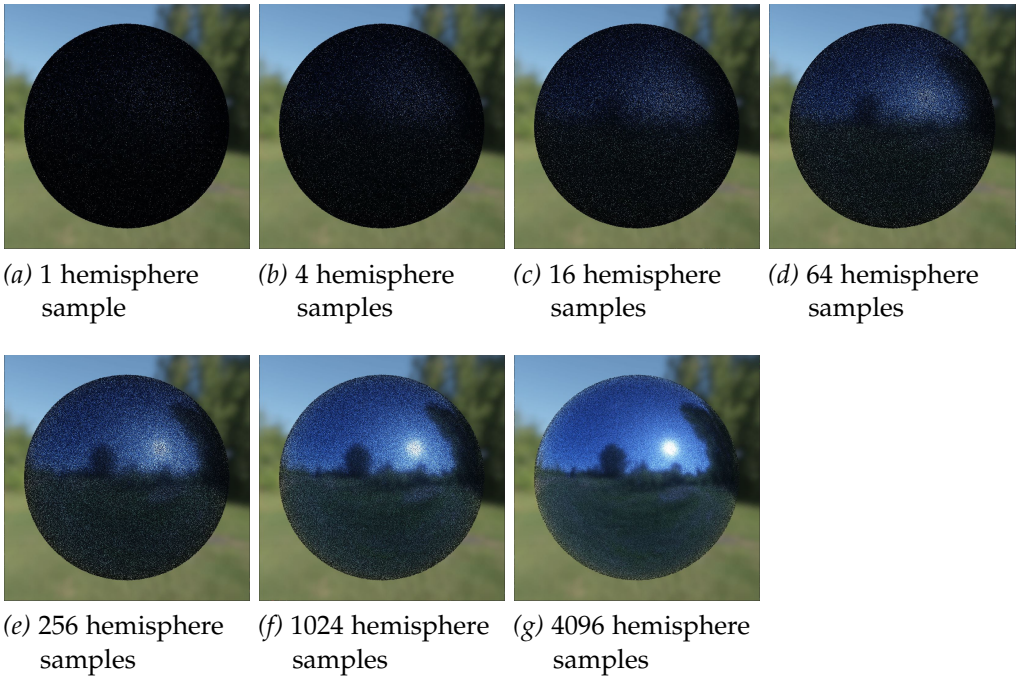
To form a more complete view on the Monte Carlo estimator's performance, we repeat the same experiment on a *specularly reflective* sphere, instead of a diffuse one (Figure 59). We pick our surface to be metallic, with a *roughness* value of 0.1.

It is immediately apparent that, for specular surfaces, our estimate is much worse: at the start, the sphere is completely dark, and we can't detect even the slightest reflection. It is only much later (that is, after many other samples) that the specular sphere becomes recognizable. The main reason for this is that our samples are generated *uniformly* across the hemisphere. That is, for a surface point $\tilde{p}$, the Monte Carlo estimator looks for the incoming light from *every direction* above $\tilde{p}$, with the *same probability*. In the case of specular surfaces, doing this doesn't make sense: only a *few* of those directions actually matter. More specifically, the light reflected from $\tilde{p}$ along the view vector $\vec{V}$ mostly comes from the direction $\vec{L}$ of *perfect specular reflection*[2]. Hence, a better approach would be to generate samples concentrated around said direction. This idea is at the base of *importance sampling*, a technique commonly applied to reduce the noise in *any* type of reflection, not only specular (Pharr, Jakob, and Humphreys 2023).

If we were to pick 0 as the *roughness* value of the metallic surface (which would thus become *perfectly specular*), results would get even worse. As we'll discuss better later, this difficulty in rendering specular reflections on smooth surfaces is one of the biggest flaws of our program.

1 Simply by modifying the images in GIMP.

2 That is, the direction $\vec{L}$ that is symmetric to $\vec{V}$ with respect to the surface normal.

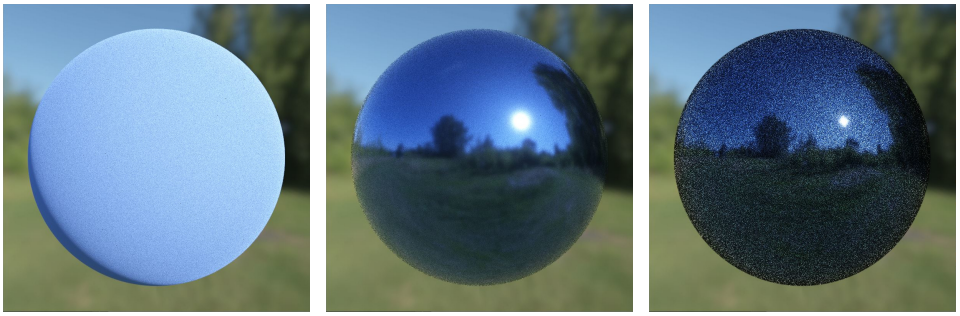


*(a)* 1 hemisphere sample
*(b)* 4 hemisphere samples
*(c)* 16 hemisphere samples
*(d)* 64 hemisphere samples
*(e)* 256 hemisphere samples
*(f)* 1024 hemisphere samples
*(g)* 4096 hemisphere samples

*Fig. 59:* Monte Carlo estimates of light reflected from a metallic surface ($\mathtt{roughness} = 0.1$). The scene's cubemap was blurred beforehand.

On a brighter note, we should mention that antialiasing and Monte Carlo sampling work quite well together, at least in BoxOfSunlight. This is because our antialiasing mechanism, that is, the running average of multiple frames, *helps* the Monte Carlo estimator in reducing noise. We can, in fact, get very low noise even with (relatively) few hemisphere samples, since those samples are taken per-frame. For instance, if the Monte Carlo estimator takes 1024 per-frame hemisphere samples, and we let the program run for 50 frames, it's as if (although not exactly the same) the Monte Carlo estimator is actually taking $50 \cdot 1024$ hemisphere samples.

In Figure 60, this is demonstrated for the diffuse and specular spheres, from the previous two experiments (images (a) and (b)). As an additional example, we also show a perfectly specular sphere (image (c)). Notice that spheres (a) and (b) finally become smooth and natural-looking. In image (c), on the other hand, noise is still lingering on the sphere's surface. Whenever we will render perfectly smooth surfaces, noise will be inevitable, even if we average together many frames.



*(a)* Diffuse sphere

*(b)* Specular sphere (roughness $= 0.1$)

*(c)* Specular sphere (roughness $= 0$)

*Fig. 60:* End result of applying antialiasing and Monte Carlo sampling together (with 1024 hemisphere samples). The program was left to render (and average together) 50 frames.

## 5.2 RUNTIME PERFORMANCE

When Monte Carlo sampling is employed, it is done at the cost of execution time. This is because, to achieve decent results, the Monte Carlo sampler needs to take a high number of samples (preferably, even more than a *thousand*). Thus, it would be proper for us to perform some measures, as to find out exactly *how much difference* in time is brought on by our Monte Carlo technique. We conduct the tests on an *AMD Radeon RX Vega 8* integrated GPU.

Let's take, again, a single sphere. Any material will do, as material parameters don't affect frame rate. We first measure the speed at which this sphere is rendered, on a 800x800-pixel image, when illuminated simply by a point light[3]. Note that the amount of pixels covered by the sphere greatly influences performance. In our case, we make sure that the sphere occupies most of the image, so that most pixels will have reflection calculations associated with them. This is still not the *worst* case, since the sphere doesn't cover *all* pixels (but that's, in general, unlikely to happen).

We now start a timer and, every 500 frames, divide the number 500 by the elapsed time. If the time is expressed in seconds, this gets us the average *FPS* (*Frames Per Second*) for the last 500 frames, which we can then print on console. We let the program run for a while and monitor the FPS

3 The 3D scene is the same as in section 5.1.1, only without the texture.

values. We observe that the program seems to be running consistently at around 40 FPS.

Next, we replace the point light with a cubemap. When taking a single sample in the hemispheres above surface points, performance becomes only slightly worse, as the program runs at 35 FPS[4]. Outside of this test, however, we will never be sampling hemispheres only once, as this would leave too much noise in the output image. Hence, we try to gradually add more hemisphere samples.

At 256 samples, we run into an issue. A driver timeout is detected by the Windows *Time Detection and Recovery* (*TDR*) tool, and the program crashes. Basically, the compute shader invocations are taking too long to run. TDR notices this, and, in response, resets the GPU, to prevent the entire system from getting frozen[5]. To solve the problem, we divide the output image into smaller "chunks" (groups of pixels), and render only one chunk at a time. For 256 hemisphere samples, dividing the image into 200x200-pixel pieces does the job.

Recall that, in section 5.1.2, we produced images with up to 4096 hemisphere samples. We did not mention it, but those images were actually generated in chunks. When taking 4096 samples, it was necessary to divide an 800x800-pixel image into chunks of 20x20 pixels, and the rendering time for each complete frame was of 35 seconds.

So, to summarize, rendering a sphere illuminated by a cubemap, on a 800x800-pixel image with 4096 Monte Carlo samples, takes 35 seconds per-frame. With a point light, since we were rendering 40 frames per-second, rendering a single frame took about 1/40th of a second. This is a big difference, but it is also acceptable, given how much more life and realism is added by the cubemap.

## 5.3 BEHAVIOR OF THE DISNEY BRDF

If we had to draw a conclusion from the previous sections, it would go as follows. The images produced by our renderer are not perfect, but, with enough samples (and time), they should be "good enough". In other words, while images might come out as noisy (as in the case of specular surfaces) they should also, hopefully, offer at least *some* visual cues for

4 The reason for the slight drop in performance might be the texture sampling operation, done on the cubemap to find the incoming radiance.

5 For more on *TDR*, see https://learn.microsoft.com/en-us/windows-hardware/drivers/display/timeout-detection-and-recovery (last accessed: 31.03.2025)

distinguishing between materials (e.g., between diffuse and specular ones). That is, if the reflection model works correctly.

We therefore move on, now reassured, to finally observe the effects of the Disney "Principled" BRDF on some virtual materials.

### 5.3.1 *Diffuse Lobe*

Remember the parameters used by the Disney BRDF:

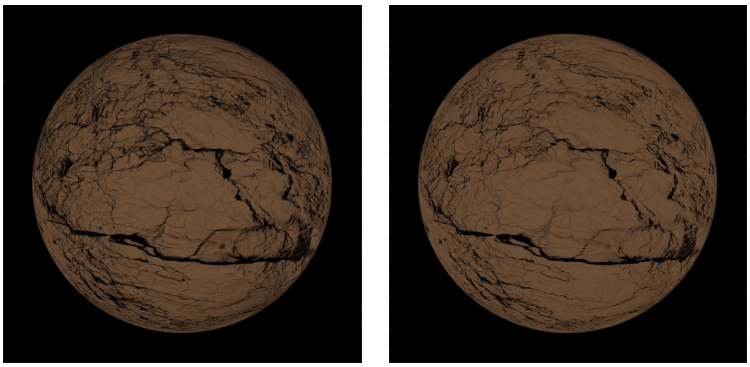
- *baseColor*
- *subsurface*
- *metallic*
- *specular*
- *specularTint*
- *roughness*
- *anisotropic*
- *sheen*
- *sheenTint*
- *clearcoat*
- *clearcoatGloss*

Out of all these parameters, only two determine the behavior of the diffuse lobe: *roughness* and *subsurface*. The first one, as we know, influences *grazing retroreflection*: at edges, it makes rough surfaces (i.e., surfaces with high *roughness* values) exhibit *highlights* instead of *shadows*.

To test this phenomenon, we take another sphere and equip it with a normal map. This will make reflective highlights and shadows more evident. We illuminate this sphere, again, with a point light, which emits a radiance value of $[3, 3, 3]^\mathsf{T}$.

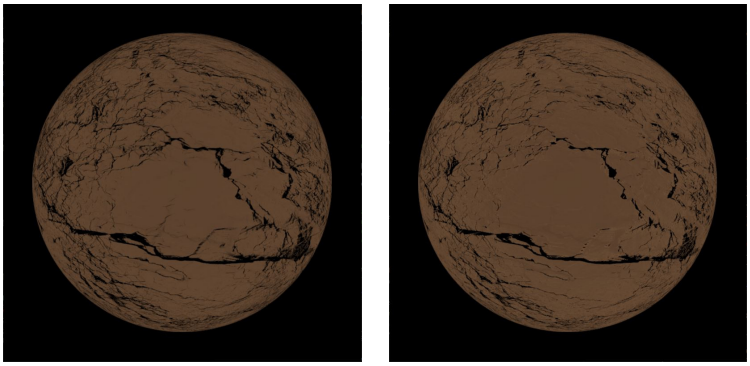
An important question is where to locate the light, as different lighting angles produce different outcomes. For instance, if we place the light at the same point as in the antialiasing test, shifts in roughness are barely visible: a surface with $\mathtt{rougness} = 0$ is pretty much identical to one with $\mathtt{rougness} = 1$. After various tries, we find that the best option is to position the point light *in front* of the sphere, and *far away*. The sphere is centered at the origin, and has a radius of 1, while the point light is located at $[0, 0, 10]^\mathsf{T}$. The camera is looking down along the negative $z$ direction. It makes sense that this is the best configuration for our

experiment, since we are studying changes in *retroreflection*.



*(a)* roughness $= 0$, subsurface $= 0$

*(b)* roughness $= 1$, subsurface $= 0$



*(c)* roughness $= 0$, subsurface $= 1$

*(d)* roughness $= 1$, subsurface $= 1$

*Fig. 61:* Diffuse retroreflection for various *roughness* and *subsurface* values.

In images (a) and (b) of Figure 61, although we cannot see actual *highlights*, we notice that the rough sphere (the one with roughness $= 1$) is *less dark* around its edges than the smooth one (with roughness $= 0$). Not only that, but the sphere also becomes lighter at points on the *inside*, as many shadows are lost. This might be because some portions of the sphere's surface are located on the sides of the normal map's "bumps", causing them to have an orientation similar to that of the sphere's edges. Hence, just like the edges, these surface portions also lose some shadow.

In Figure 61, we also show the effect of the *subsurface* parameter. Turning up its value causes interesting alterations too: many shadows disap-

pear, and the color becomes more uniform across the surface. This is, in fact, how the Disney BRDF approximates *subsurface scattering*.

The softening of shadows is something that does indeed happen in subsurface-scattered materials, since, in them, light can diffuse across shadow boundaries (Hughes et al. 2014). To give an example, *marble* is a material that exhibits subsurface scattering, and one of its distinctive visual features is, indeed, its softness. However, marble would not look the way it does without another important trait, that being *translucency*. Unfortunately, materials modeled with the Disney BRDF cannot look translucent, because light doesn't really pass *through* them. The lack of this visual quality makes the approximation we just mentioned an incomplete substitute to real subsurface scattering. As indicated by Burley, it should only be used for giving a subsurface appearance to distant objects, as well as objects where the average scattering path length is small (Burley 2012).

### 5.3.2 *Specular Lobe*

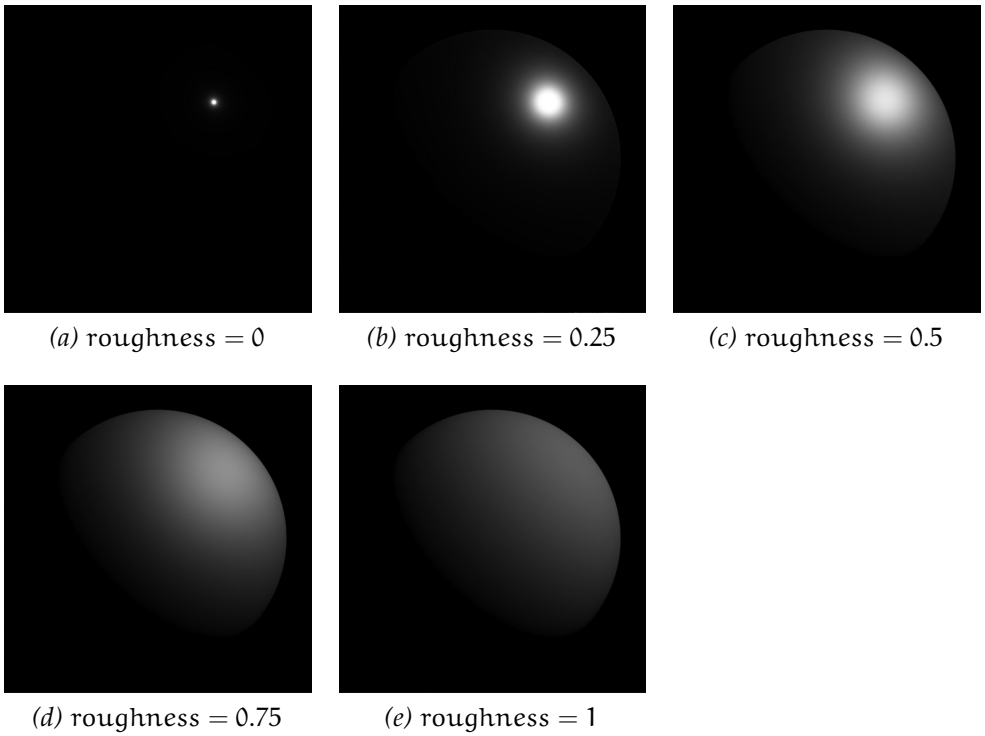
The *roughness* parameter plays an important role also in the specular lobe of the Disney BRDF, where it influences the size and brightness of *specular highlights*.

We know from microfacet theory that the shape of a specular highlight on a surface is determined by the normal distribution function $D$, and that, in the Disney BRDF, $D$ follows the *GTR distribution*. As it's common to do, the *variance* of the distribution function is regulated through the *roughness* parameter of the BRDF. As the value of *roughness* increases from 0 to 1, the specular highlights get more spread out, and there's less of a peak in reflection. What this means is that reflections become more *blurry*.

In Figure 62, we verify the link between the surface roughness and the distribution of highlights, for a dielectric material illuminated by a point light. To make the highlights easier to see, we raise the point light's radiance to $[10, 10, 10]^T$, and set the value of the `exposure` variable in the fragment shader[6] to 100. We can notice that, for $\texttt{roughness} = 0$, the highlight becomes nothing but a small dot, while at $\texttt{roughness} = 1$, it assumes a diffuse-like appearance. Note that these highlights are normally

6 We indicated the purpose of `exposure` parameter at the end of section 4.2.2, from the previous chapter. Up to this point, to produce images, we have been using `exposure` values of either 1 (with the point light) or 3 (with the cubemap).

added *on top* of the surface's diffuse response, seen in the previous section.



*(a)* roughness $= 0$ *(b)* roughness $= 0.25$ *(c)* roughness $= 0.5$

*(d)* roughness $= 0.75$ *(e)* roughness $= 1$

*Fig. 62:* Specular highlights for various *roughness* values.

Specular highlights on smooth surfaces, that is, surfaces with low *roughness* values, only really show their potential when combined with image-based lighting. See for instance Figure 63, where a smooth sphere is illuminated by a cubemap: here, we can clearly observe *mirror-like* specular highlights at edges. The reason why the highlights are located at the edges lies in the Fresnel equations, which, as we are well aware, tell us that light is reflected more strongly at grazing angles. The feeling that we get out of strong, mirror-like specular highlights, is that of *plastic*.

To showcase Fresnel reflectance a little better, we devise another test. We take a 10-by-10 plane (made up of two triangles), and position it horizontally, in a scene illuminated by a cubemap. This time, we set its *roughness* value to 0, and first observe it from a 45° angle (Figure 64).

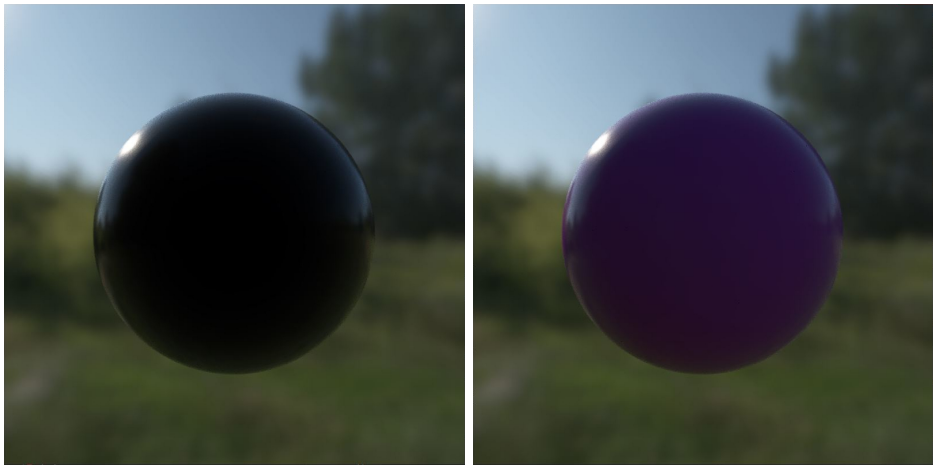


*Fig. 63:* A smooth sphere ($\texttt{roughness} = 0.15$) exhibits mirror-like specular highlights at grazing angles. On the left, only the specular lobe is calculated. On the right, the diffuse lobe is added as well.

*Fig. 64:* Smooth plane seen from a 45° angle.

We later move the camera down, so that the plane is seen from a grazing angle. As we would expect, the surface assumes a remarkably different look, almost like that of a mirror (Figure 65).

Specular reflection effects in dielectrics are quite subtle, and instead become much more significant in the case of *metals* (surfaces with *metallic* set to 1). This is because metals reflect all incoming light.

In the Disney model, the specular highlights on metals are also *tinted* towards the base color. If the base color is bright, the tint makes reflections much more perceptible. Hence, with metals, we can visualize

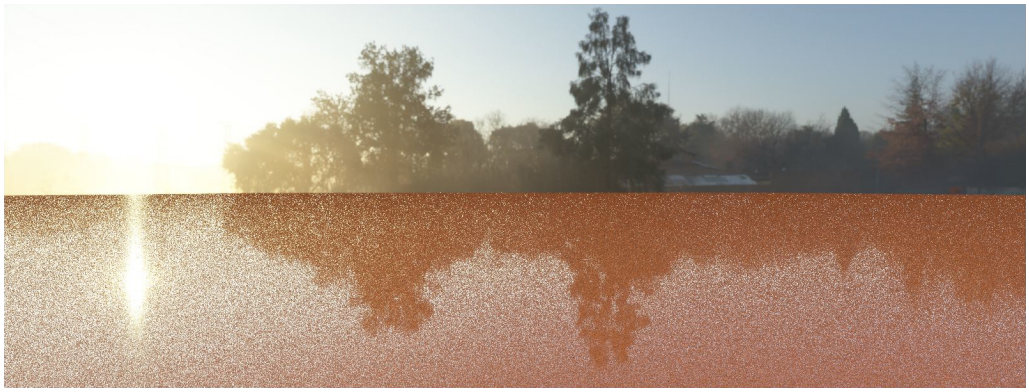


*Fig. 65:* Smooth plane seen from a grazing angle.

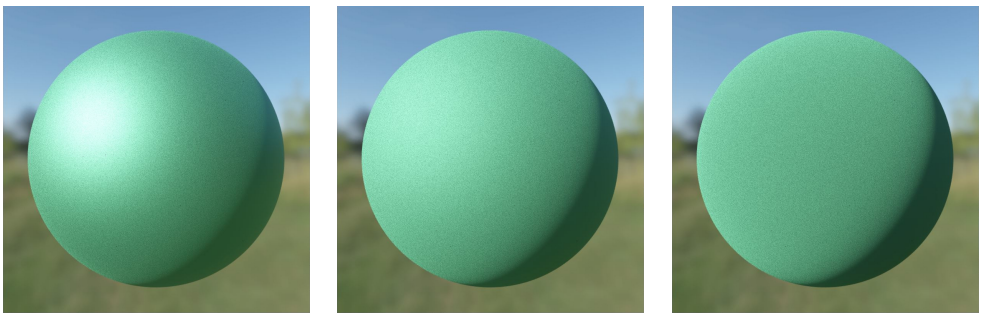
the correlation between the *roughness* parameter and the blurriness of reflections even better, as shown in Figure 66.

The specular highlights we have been observing so far had the characteristic of being *circular*. This is because the GTR distribution is Gaussian-like, that is, symmetric across the surface. This symmetry can optionally be broken through the use of the *anisotropic* parameter. Figure 67 depicts anisotropic highlights on metallic spheres, located around a point light. The direction of a highlight on a sphere is determined by the *tangent vector* (see Appendix B).

We have left out two parameters of the specular lobe: *specular* and *specularTint*. Unfortunately, when generating images with BoxOfSunlight, they don't influence the end result, at least not in a perceptible way. It might be that the form of tone mapping that we employ[7] is not appropriate for specular highlights. In fact, Burley mentions that, due to their extreme values, much of the chromatic information of specular highlights can be lost when not mapped correctly (Burley 2012). In Burley's notes it is also said that, at Disney, it was necessary to develop a new tone mapping operator to solve this issue.

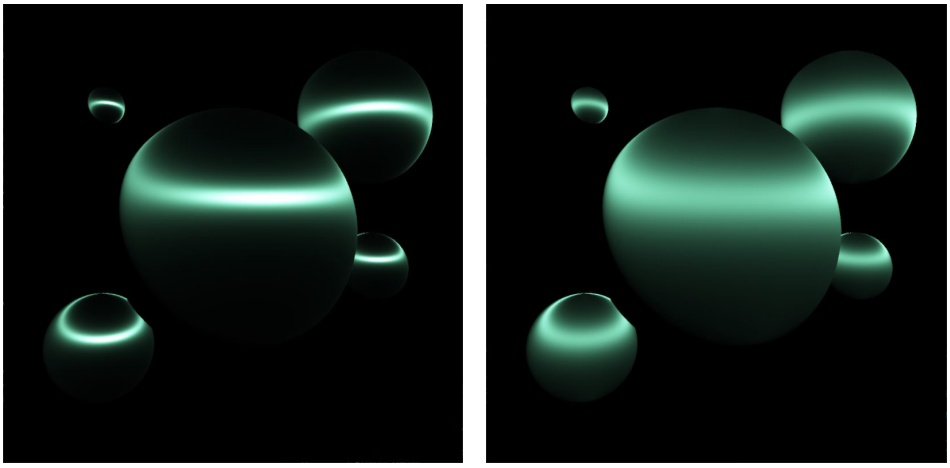
7 Recall that we apply *exponential tone mapping*, regulated by the `exposure` parameter (section 4.2.2).

*(a)* roughness $= 0$ *(b)* roughness $= 0.2$ *(c)* roughness $= 0.4$



*(d)* roughness $= 0.6$ *(e)* roughness $= 0.8$ *(f)* roughness $= 1$

*Fig. 66:* Metallic surface with various *roughness* values.



*(a)* roughness $= 0.3$ *(b)* roughness $= 0.8$

*Fig. 67:* Anisotropic highlights on metallic spheres.

### 5.3.3 *Sheen and Clear-Coating*

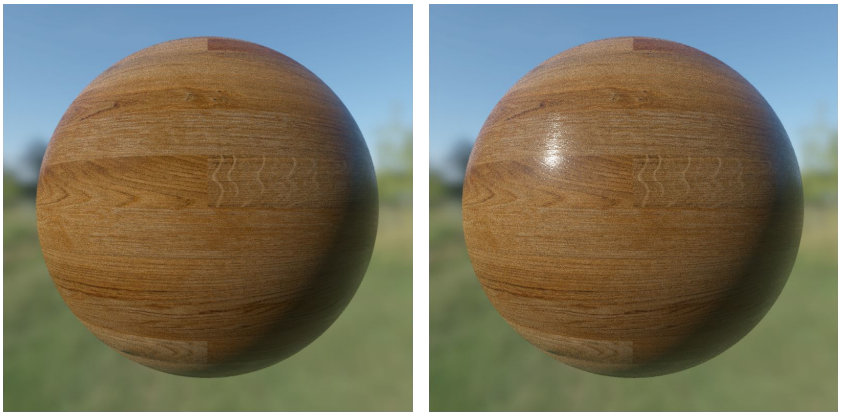
In Chapter 3 we learned that in real-world *fabric* grazing reflectance is intensified, and that the extra reflectance is often *tinted* towards the fabric's base color (section 3.5.6). Inside the Disney BRDF, these properties are regulated with the *sheen* and *sheenTint* parameters. Their effect is demonstrated in Figure 68. Notice the increase in reflectance at the top-left edge of the sphere, caused by setting $\texttt{sheen} = 1$ in image (b). Also, in image (c), observe the red tint induced by the *sheenTint* parameter.

*(a)* $\texttt{sheen} = 0$, $\texttt{sheenTint} = 0$ *(b)* $\texttt{sheen} = 1$, $\texttt{sheenTint} = 0$ *(c)* $\texttt{sheen} = 1$, $\texttt{sheenTint} = 1$

*Fig. 68: Sheen* effect on fabric. (A normal map was also used.)

Interestingly, in the original implementation of the Disney BRDF, the extra reflectance caused by sheen is added to the diffuse lobe. Perhaps, this is done to prevent the use of sheen for metals (which ignore the diffuse term).

As opposed to sheen, *clear-coating* is modeled on a new, distinct lobe, which is added on top of the diffuse and specular responses. With this new layer we can simulate the effect of applying a *polyurethane* finish on the material. To give an example of why this is useful, in the real world, polyurethane is often used as a protective coating for wood. The look of glossy clear-coating can be observed in Figure 69.



*(a)* clearcoat $= 0$ *(b)* clearcoat $= 1$

*Fig. 69:* Demonstration of the clearcoat layer's effect. For the image to the right, the clearcoatGloss parameter is also set to 1, to give the clear-coating a glossy appearance (rather than a "satin" one).

## 5.4 MINOR IMPROVEMENTS

There are a couple simple additions that we can make to BoxOfSunlight, to slightly enhance its rendering abilities. Here, we leave some short notes on them.

Recall that we use textures both for getting the *colors* as well as the *normals* of surface points[8]. More variety can be added to surfaces by exploiting the texture mapping mechanism even further. In fact, if we think about it, it's clear that *material parameters* could also be sampled from textures, so that each point on a surface reacts differently to light. For instance, in physically-based rendering, it is common to get *roughness* and *metallic* values from two texture images, called *roughness map* and *metallic map* respectively. Figure 70 presents an example where, with this method, we are able to recreate the look of rusty metal.

There is also another (more obvious) way in which we can make our images more interesting. That is, instead of depicting only a few spheres and triangles, we can render some more complex *3D models*. In fact, the main reason why BoxOfSunlight takes *triangles* as inputs is 3D model

8 We called textures which contain normals *normal maps*. Textures that contain *colors*, on the other hand, are commonly called *albedo maps*.

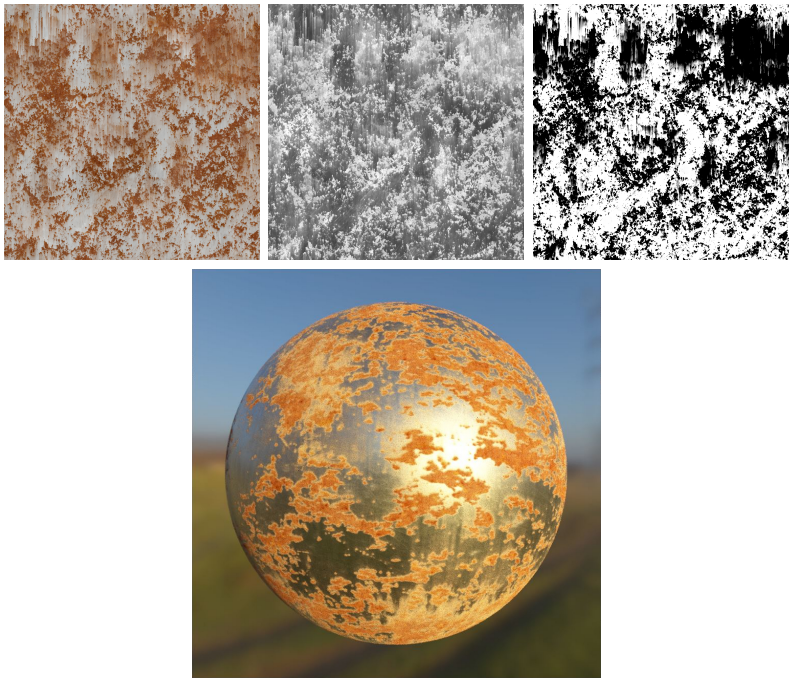


*Fig. 70:* An albedo map, a roughness map and a metallic map (top), mapped on a sphere (bottom). Textures from ambientCG.

support. Recall that 3D models are often nothing more than collections of many small triangles (in fact, we have also used the term *triangle mesh* to indicate them). Consequently, we can easily integrate 3D models into our rendering pipeline by dividing them into their base triangles, and then passing these to the compute shader.

BoxOfSunlight accepts 3D models memorized in the *Wavefront OBJ* file format. The models are loaded into the program with the help of the *tinyobjloader* library[9]. In Figure 71, we show how BoxOfSunlight is able to render a simple model, made up of 2752 triangles.

It takes about 10 minutes to produce Figure 71 (which is 800x800 pixels large) if we render it for 20 frames, with 4096 Monte Carlo samples in each frame.

As we move on to bigger meshes, made up of many more triangles, time becomes much more of an issue. When we get to the tens of thousands, applying Monte Carlo sampling on the triangles becomes too costly, so we are forced to use a point light.

9 See https://github.com/tinyobjloader/tinyobjloader (last accessed: 31.03.2025)

*Fig. 71:* 3D model of a carved wooden elephant, rendered with BoxOfSunlight. Model by Greg Zaal, on Poly Haven.

As our very last experiment, we find that, even with a point light, rendering a 3D model made up of about 12000 triangles takes 1 minute and 30 seconds per-frame.

# 6

# CONCLUSIONS

After five long chapters, we finally reach the end of our discussion. We have exhausted all the topics that we set out to cover, leaving nothing new to be said. The only thing we can do now is look back, and acknowledge the long and winding road that has led us to this point.

## 6.1 SUMMARY OF CHAPTERS

We started off, in Chapter 1, with an introduction to the world of physically-based rendering. During the years, the PBR approach has received much attention from computer graphics researchers, and thus, as we have seen, it has a rich history. We also laid a ground plan for the rest of the thesis, establishing that we would be learning methods to realistically simulate the reflection of light on 3D objects.

To fulfill our purpose, the first step we needed to take was to form a more solid understanding both of the physical concept of *reflection*, as well as that of *light* itself. We did this in Chapter 2, while also learning about many additional ideas from the fields of electrodynamics, quantum mechanics, and optics. While it wasn't strictly necessary to cover the physics of light as extensively as we did, it is hoped that this made us acquire a versatile mindset, as well as confidence, and that it made the rest of the thesis easier to navigate.

A highly fascinating aspect of physically-based rendering is how its models transform the rules of physics into elegant, abstract formulas and algorithms, to be fed to a computer. We explored some of the math behind PBR in Chapter 3, and learned how it allows us to transition from physics to rendering. The final "piece" of the "puzzle" was the *Disney BRDF*, a model which has been used in the past to bring PBR (in particular, physically-based *reflections*) to *animation*. This was the tool that we needed to generate our own realistic reflections of light.

In Chapter 4, we found that we were ready to put our theoretical knowledge into practice. We designed a simple physically-based renderer, or, more specifically, a *ray tracer* application, that could compute light-reflection effects on virtual materials, and called it *BoxOfSunlight*.

Finally, in Chapter 5, we evaluated BoxOfSunlight. Although there were many flaws, we were also able to generate some interesting images, mostly as a result of employing *cubemaps* as rough approximations of *global illumination* in real-world environments.

This brings us to the present, final chapter. What still remains to be done, is to provide better comments on BoxOfSunlight's performance. Most importantly, we should address its flaws, and propose some solutions.

## 6.2 IMPROVING BOXOFSUNLIGHT

When running BoxOfSunlight, there is one main issue that frequently comes up, and that is *noise* on *specular reflections*. We explained in Chapter 5 that this is an inevitable consequence our Monte Carlo technique, which simply takes *uniform* samples across the hemisphere of incoming light directions. Note that this problem only applies to scenes with cubemaps, and is not present in the case of point lights (because, with a point light, there is only *one* direction to sample). But we also know that specular reflections only really show their potential when illuminated by cubemaps, because that is when they acquire a distinct mirror-like look.

As already mentioned, a solution to the noise issue would be to implement *importance sampling* (rather than uniform sampling), so that the samples are taken "more intelligently" by the Monte Carlo estimator. What this means is that, in practice, we would use a different estimator for each type of material, which takes samples concentrated only around *some* of the incoming light directions: that is, those than can have a real impact on reflection. In the case of specular surfaces, we should concentrate our samples around the direction of perfect specular reflection.

Remaining on the topic of specular reflections, in Chapter 4 we said that we were unable to notice any changes produced by the *specular* and *specularTint* parameters of the Disney BRDF. The reason for this is not perfectly clear, but, most likely, has to do with the way we do *tone mapping*. This might also be why, in diffuse retroreflection, we did not notice highlights at sphere edges (section 5.3.1). To try and solve these

issues, it would therefore be a good idea to study the concept of tone mapping in more depth.

Another problem that we have mentioned various times is that, when we cast a ray into a scene, our search for intersected objects is extremely inefficient. In fact, it is a *linear search*, with time complexity $O(n)$ (where $n$ is the number of geometric primitives in the scene).

A better approach is that of *bounding volume hierarchies* (Shirley, Black, and Hollasch 2024b). To give an idea of how they work, these hierarchies are usually nothing more than tree data structures, where the leaves are geometric primitives (triangles, spheres) while the internal nodes are *enclosing volumes*. For example, the parent node of a sphere could be a "box" (a rectangular parallelepiped) that contains that sphere. This box could then be grouped with another box that is close-by (which also contains some geometry), and they could be enclosed together in a third, bigger box, which would become their parent node. If we enclose enough pairs of boxes together, we get to the point where there is a single box that recursively contains all the geometries in the scene: this is the root node of the tree. At the moment of searching for ray-object intersections, we would traverse the three by checking for intersections between the ray and the *boxes*. First we would see if the ray intersects the biggest enclosing box, then, if it does, we would do the same for the two smaller boxes contained inside it, and so on. At some point, we would (perhaps) get to some leaves, and find the ray-object intersections we were looking for.

If the hierarchy is built properly, the search for intersected objects becomes *logarithmic* (time complexity $O(\log n)$) (Shirley, Black, and Hollasch 2024b).

For us, however, implementing BVHs would not be so straightforward. In fact, BVHs are built and traversed *recursively*, but GLSL does not support recursion. So, to use BVHs inside our compute shader, we would need to account for this in some way, like writing an iterative version of the algorithm with only a low, fixed number of hierarchy levels. Also, BVHs should be built by the C++ side of the program, to then be passed to the compute shader in some form.

As a final critique, we should talk about the fact that BoxOfSunlight does not compute real *global illumination* (Appendix A). In particular, objects don't influence the colors of each other, as would happen, instead, in the case of *path tracing*. In path tracing, rays of light are reflected multiple times between the 3D objects of a scene, so that the objects

exchange energy in the form of reflected light. This is also a natural way to compute *shadows*, cast from one object onto the other (when the shadow-casting object is blocking some light from reaching the other object). An important improvement to BoxOfSunlight could therefore be to upgrade it, from a simple ray tracer, to a fully-fledged path tracer.

## 6.3 FINAL THOUGHTS

Before ending this thesis entirely, we might ask ourselves its ultimate meaning. Since the start, we focused on a very specific objective, that being the recreation of lifelike light reflections, and therefore lifelike materials, inside a computer program, and we set out to do this in the same way as a major animation studio, such as Disney, had done it. Ultimately, we mostly succeeded in our purpose.

However, while writing this thesis, another important realization occurred. By doing what we have done, we have implicitly proved that *anyone*, armed with only first principles from physics, mathematics, and computer science, can potentially understand and replicate PBR techniques, adding their own original twist to them. In BoxOfSunlight, although the Disney BRDF dictated the rules of light reflection, we personalized the entire ray-tracing infrastructure for our needs.

Hence, to conclude, we echo our opening words. Computers really *do* have the potential to be powerful tools for artistic expression, as virtually *anyone* can recreate even the most realistic graphics, from the big screen, on their own machines.

# A

## GLOBAL ILLUMINATION

A more advanced form of ray tracing is the *path tracing* light transport algorithm[1]. Path tracing was introduced in 1986 by Kajiya, and serves as a *Monte Carlo* solution to the *rendering equation*, which was also described first in Kajiya's paper (Kajiya 1986). The rendering equation is, effectively, the equation that every realistic rendering application is trying to solve. As we'll see, this is not an easy task, as proper (that is, realistic) solutions need to take into account *global illumination*.

In *global illumination* algorithms, the colors of surfaces are influenced not only by light sources, but also by other *objects*. This derives from the fact that, in the real world, light is reflected many times. If we think about it, this makes perfect sense. After reflection, new light is emitted by a surface, and this light will be incident to other surfaces in the scene, to then be reflected again. To put it simply, we can say that these multiple reflections cause light to "bounce" between objects many times. At each reflection, some portion of light is absorbed, making bounces less visually significant as their number grows. However, the first few ones might influence significantly the colors of surfaces. For instance, if we have a bright light shining on a red object, the reflected light could very well induce a reddish tint on nearby objects in the scene (Pharr, Jakob, and Humphreys 2023).

In the following sections, we'll see how the global illumination problem can be expressed through the rendering equation. We'll also mention a couple concepts related to Monte Carlo techniques, to understand how the rendering equation can be (approximately) solved through their use.

1 Used, for example, in the *pbrt* renderer. (Pharr, Jakob, and Humphreys 2023)

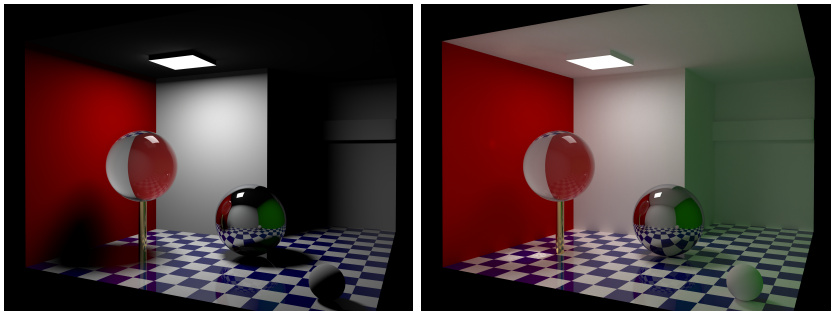


*Fig. 72:* Direct lighting (left) compared with global illumination (right). Source: Wikimedia Commons. Licensed under the CC BY-SA 4.0 license.

## A.1 THE RENDERING EQUATION

In Chapter 3, we've seen a special case of the rendering equation called the *reflection equation*: (section 3.3.2)

$$L_o(\tilde{p}, \vec{\omega_o}) = \int_{H^2(\vec{N})} f_r(\tilde{p}, \vec{\omega_o}, \vec{\omega_i}) L_i(\tilde{p}, \vec{\omega_i}) \cos\theta_i \, d\omega_i$$

Where $f_r$ is the bidirectional reflectance distribution function (BRDF). If we want to simulate proper light transport, this equation is not enough, as it doesn't take into account other ways that light could "move around" in a scene. For example, we know from Chapter 2 (section 2.3.2) that light doesn't only get *reflected*, but also *refracted*; in the latter case, we also say the light is *transmitted*. Transmission can be modeled through a surface's *Bidirectional Transmittance Distribution Function* (*BTDF*) $f_t$, which we can define in a manner similar to the BRDF $f_r$ (Pharr, Jakob, and Humphreys 2023). For convenience, the BRDF and the BTDF can be considered *together* in a function $f$, called the *Bidirectional Scattering Distribution Function* (*BSDF*). This function considers general "scattering" of light, which could include also subsurface scattering (that is, the *BSSDF*, mentioned in section 3.3.3).

The *rendering equation* (also called *light transport equation*) is related to the BSDF $f$. In fact, it uses $f$ to express the distribution of radiance in a scene (Pharr, Jakob, and Humphreys 2023).

The rendering equation is based on the principle of *energy balance*:

> "The difference between the amount of energy going out of a system and the amount of energy coming in must also be

> equal to the difference between energy emitted and energy absorbed." (Pharr, Jakob, and Humphreys 2023)

By simply enforcing this principle at a surface, we get:

$$L_o(\tilde{p}, \vec{\omega_o}) = L_e(\tilde{p}, \vec{\omega_o}) + \int_{H^2(\vec{N})} f(\tilde{p}, \vec{\omega_o}, \vec{\omega_i}) L_i(\tilde{p}, \vec{\omega_i}) \cos\theta_i d\omega_i \quad (A.20)$$

Where $L_e$ is the radiance emitted by the surface. This is integral equation is what we call *the rendering equation*.

## A.2 MONTE CARLO INTEGRATION

The rendering equation (A.20) is, in general, impossible to solve analytically. One of the reasons for this lies in the integral over the hemisphere $H^2$. In fact, it requires us to sum the (weighted) incoming radiance contributions coming in from *all* (that is, *infinite*) directions above the surface point $\tilde{p}$.

However, even though we can't solve it in an analytical sense, *approximate solutions* to the rendering equation can be found. In particular, we can estimate the value of the integral by considering only the contributions of a *few*, *random* directions above $\tilde{p}$. This idea of relying on randomness is at the base of *Monte Carlo estimators*.

Monte Carlo estimators can be built to approximate the values of arbitrary integrals (Pharr, Jakob, and Humphreys 2023). Let's say that we want to compute

$$F = \int_a^b f(x) dx \quad (A.21)$$

And we have a supply of independent random variables $X_i \epsilon [a, b]$. Then, if the random variables are drawn from a probability distribution function $p(x)$, a Monte Carlo estimator of (A.21) can be defined as

$$\langle F^n \rangle = \frac{1}{n} \sum_{i=1}^{n} \frac{f(X_i)}{p(X_i)}$$

It can be proven that the estimator $\langle F^n \rangle$ converges to the right answer, that is, the value of (A.21), with a convergence rate that's proportional to its variance $\sigma[\langle F^n \rangle]$ (Pharr, Jakob, and Humphreys 2023). Because the variance corresponds to the squared error, the error of a Monte Carlo estimator goes down at a rate of $O(n^{-1/2})$, for a number of samples $n$.

The main reason why Monte Carlo integration is preferred to other numerical methods, such as the *quadrature techniques*, is that its convergence rate is independent of the dimension of the integral. This makes it the only practical numerical integration algorithm for high-dimensional integrals, such as the ones found in path tracing (Pharr, Jakob, and Humphreys 2023).

Path tracing follows the path of light through multiple "bounces", resulting in nested integrals over hemispheres. Instead of solving these nested integrals directly, it's more convenient to transform them into non-nested ones, defined over a high-dimensional "path space" whose points are the geometric paths of light (Gortler 2012). This is the idea behind path tracing.

We end this Appendix by mentioning a simple sampling method for Monte Carlo integration over a hemisphere, taken from (Pharr, Jakob, and Humphreys 2023).

In the most general case of hemisphere sampling, we generate random directions[2] $\vec{\omega}$ over the hemisphere $H^2$ that are *uniformly distributed*. Mathematically, this means that their probability distribution function is constant:

$$p(\vec{\omega}) = c$$

To derive the constant c, we can exploit the fact that probability density functions must integrate to 1 over their domain:

$$\int_{H^2} p(\vec{\omega}) d\omega = 1 \implies c \int_{H^2} d\omega = 1 \implies c = \frac{1}{2\pi}$$

Hence, we have $p(\vec{\omega}) = \frac{1}{2\pi}$. If we express $\vec{\omega}$ in spherical coordinates $\theta \epsilon [0, \pi/2]$ and $\phi \epsilon [0, 2\pi]$, we get

$$p(\theta, \phi) = \frac{\sin\theta}{2\pi}$$

This is due to the fact that $d\omega = \sin\theta \, d\theta \, d\phi$, and therefore:

$$p(\theta, \phi) \, d\theta \, d\phi = p(\vec{\omega}) \, d\omega$$
$$p(\theta, \phi) = \sin\theta \, p(\vec{\omega})$$

Next, we find the marginal probability distributions $p(\theta)$ and $p(\phi)$:

$$p(\theta) = \int_0^{2\pi} p(\theta, \phi) d\phi = \int_0^{2\pi} \frac{\sin\theta}{2\pi} d\phi = \sin\theta$$
$$p(\phi|\theta) = \frac{p(\theta, \phi)}{p(\theta)} = \frac{1}{2\pi}$$

2 Remember that $\vec{\omega}$ corresponds to an infinitesimal solid angle $d\omega$

Finally, we use the *inversion method*. This method maps uniform samples from $[0, 1)$ to a given 1D probability distribution by inverting the distribution's cumulative distribution function (CDF). Writing the CDFs for $\theta$ and $\phi$ we get:

$$P(\theta) = \int_0^\theta \sin\theta' d\theta' = 1 - \cos\theta$$
$$P(\phi) = \int_0^\phi \frac{1}{2\pi} d\phi' = \frac{\phi}{2\pi}$$

Then we consider $P(\theta)$ and $P(\phi)$ as two uniformly distributed numbers $\xi_1$, $\xi_2$, and solve for $\theta$ and $\phi$:

$$\xi_1 = 1 - \cos\theta \implies \theta = \cos^{-1}(1 - \xi_1)$$
$$\xi_2 = \frac{\phi}{2\pi} \implies \phi = 2\pi\xi_2$$

Since $\xi_1$ is uniformly distributed, we can replace it with $1 - \xi_1$:

$$\theta = \cos^{-1}\xi_1$$

We can now get our sampling direction by converting $\theta$ and $\phi$ to cartesian coordinates:

$$x = \sin\theta\cos\phi = \cos(2\pi\xi_2)\sqrt{1-\xi_1^2}$$
$$y = \sin\theta\sin\phi = \sin(2\pi\xi_2)\sqrt{1-\xi_1^2}$$
$$z = \cos\theta = \xi_1$$

Putting all of this into GLSL code, we get:

```
vec3 uniformSampleHemisphere(float u1, float u2) {
   float cosTheta = u1;
   float sinTheta = sqrt(1.0 - u1*u1);
   float phi = 2.0 * PI * u2;
   return vec3(sinTheta*cos(phi), sinTheta*sin(phi), cosTheta);
}
```

Where `u1` and `u2` are two uniformly generated random numbers in $[0, 1)$ (that is, $\xi_1$ and $\xi_2$).

The `uniformSampleHemisphere` function can then be used to generate random directions for the Monte Carlo estimator:

$$\langle L_o(\vec{\omega}_o)^n \rangle = L_e(\vec{\omega}_o) + \frac{2\pi}{n}\sum_{i=1}^{n} f(\vec{\omega}_o, \vec{\omega}_i) L_i(\vec{\omega}_i)\cos\theta_i$$

Which estimates (A.20) ($\tilde{p}$ is made implicit).

# B

# TEXTURE MAPPING ON SPHERES

This brief appendix explains how *texture mapping* can be applied to spheres. It contains both a method for calculating *UV coordinates* as well as one for computing *tangent spaces*.

## B.1 UV COORDINATES

For spheres, *UV coordinates* (or *texture coordinates*) are usually based on some form of longitude and latitude, or, equivalently, *spherical coordinates*. These are indicated as $[\theta, \phi]^T$. To calculate the spherical coordinates of a point $\tilde{p}$ on a sphere, we go through the following process[1].

First of all, given the sphere's center $\tilde{c}$, we compute the normal vector at $\tilde{p}$ as

$$\vec{N} = \frac{\tilde{p} - \tilde{c}}{\|\tilde{p} - \tilde{c}\|}$$

Then, we use $\theta \epsilon [0, \pi]$ to express the angle between $\vec{N}$ and the $y$-axis, and $\phi \epsilon [0, 2\pi]$ for the angle between the projection of $\vec{N}$ on the $xz$-plane and the $x$-axis (see Figure 73). We can also say that $\phi$ indicates the *rotation* of $\vec{N}$ around the $y$-axis[2]. Consequently, we can easily express the cartesian coordinates of $\vec{N}$ in terms of the spherical coordinates $[\theta, \phi]^T$:

$$\begin{aligned} x &= \sin\theta \cos\phi \\ y &= \cos\theta \\ z &= -\sin\theta \sin\phi \end{aligned}$$

1 Derived from (Shirley, Black, and Hollasch 2024b).

2 Note that this is not how spherical coordinates are commonly used: here we swap the roles of the $y$ and $z$ axes, because this makes more sense in rendering, where the view direction is along $z$.

Likewise, we can move from the cartesian coordinates $[x, y, z]^T$ to the spherical coordinates $[\theta, \phi]^T$ by calculating:

$$\theta = \arccos y$$
$$\phi = \arctan(-\frac{z}{x})$$

Now that we have $\theta$ and $\phi$, we map their values to the $[0, 1]$ interval, which is what we use to express UV coordinates:

$$u = \frac{\phi}{2\pi}$$
$$v = 1 - \frac{\theta}{\pi}$$

This way, $u$ will grow in the direction of increasing $\phi$, and $v$ in the direction of decreasing $\theta$.

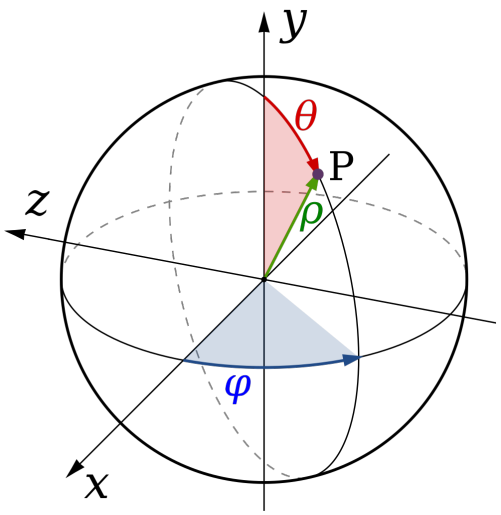



*Fig. 73:* Spherical coordinates for rendering. Source: Wikimedia Commons. (The image was slightly modified.)

## B.2 TANGENT SPACE

To do *normal mapping* at a point $\tilde{p}$ on a sphere, we need to compute the sphere's *tangent space* at that point. The tangent space's frame is made up of the *tangent vector* $\vec{T}$, the *bitangent vector* $\vec{B}$, and the *sphere normal* $\vec{N}$. These vectors are relative to the point $\tilde{p}$ (they vary at each point of the

sphere). In rendering, we want the tangent vector $\vec{T}$ to look "horizontal" (remember that the viewing direction is along the $z$-axis).

Let's suppose that the sphere is centered at the origin of the 3D space[3]. Just like we did with the normal $\vec{N}$, we can parametrize a point $\tilde{p}$ on the sphere as

$$\tilde{p}(\theta, \phi) = r[sin\theta\, cos\phi, cos\theta, -sin\theta\, sin\phi]^T$$

Where $r$ is the sphere's radius.

Now look again at Figure 73, and note that, by varying only $\phi$, we make $\tilde{p}$ move "horizontally". Therefore, we can calculate the tangent vector $\vec{T}$ by taking the derivate of $\tilde{p}(\theta, \phi)$ with respect to $\phi$: (Hughes et al. 2014)

$$\vec{T} = \frac{d\tilde{p}(\theta, \phi)}{d\phi} = -r[sin\theta\, sin\phi,\ 0,\ sin\theta\, cos\phi]^T$$

And then normalizing.

Now that we have both the normal $\vec{N}$ and the tangent $\vec{T}$ of the sphere at $\tilde{p}$, the bitangent vector $\vec{B}$ can simply be computed as

$$\vec{B} = \vec{N} \times \vec{T}$$

3 What we prove here also applies to spheres with centers different from the origin.

# BIBLIOGRAPHY


Akenine-Möller, Tomas et al. (2018). *Real-Time Rendering, Fourth Edition. Online chapter: Real-Time Ray Tracing, version 1.4*. URL: https://www.realtimerendering.com/Real-Time_Rendering_4th-Real-Time_Ray_Tracing.pdf. (accessed: 19.03.2025).

Burley, Brent (2012). *Physically-based shading at Disney*. URL: https://blog.selfshadow.com/publications/s2012-shading-course/burley/s2012_pbs_disney_brdf_notes_v3.pdf. (accessed: 22.02.2025).

Cook, R. L. and K. E. Torrance (Jan. 1982). "A Reflectance Model for Computer Graphics". In: *ACM Trans. Graph.* 1.1, pp. 7–24. ISSN: 0730-0301. DOI: 10.1145/357290.357293. URL: https://doi.org/10.1145/357290.357293.

Feynman, Richard P., Robert B. Leighton, and Matthew L. Sands (2011). *The Feynman lectures on physics*. New Millennium. New York, NY: Basic Books.

Glassner, Andrew S. (1995). *Principles of Digital Image Synthesis*. San Francisco: Morgan Kaufmann.

– (2019). *An Introduction to Ray Tracing*. URL: https://www.realtimerendering.com/raytracing/An-Introduction-to-Ray-Tracing-The-Morgan-Kaufmann-Series-in-Computer-Graphics-.pdf. (accessed: 12.03.2025).

Gortler, Steven J. (2012). *Foundations of 3D Computer Graphics*. Cambridge, MA: MIT Press.

Hery, Christophe and Ryusuke Villemin (2013). *Physically Based Lighting at Pixar*. URL: https://graphics.pixar.com/library/PhysicallyBasedLighting/paper.pdf. (accessed: 23.02.2025).

Hoffman, Naty (2012). *Background: Physics and Math of Shading*. URL: https://blog.selfshadow.com/publications/s2012-shading-course/hoffman/s2012_pbs_physics_math_notes.pdf. (accessed: 12.03.2025).

Hughes, John F. et al. (2014). *Computer graphics: principles and practice*. 3rd edition. Upper Saddle River, NJ: Addison-Wesley.

Kajiya, James T. (Aug. 1986). "The rendering equation". In: *SIGGRAPH Comput. Graph.* 20.4, pp. 143–150. ISSN: 0097-8930. DOI: 10.1145/15886.15902. URL: https://doi.org/10.1145/15886.15902.

Marsaglia, George (2003). "Xorshift RNGs". In: *Journal of Statistical Software* 8.14, pp. 1–6. DOI: 10.18637/jss.v008.i14. URL: https://www.jstatsoft.org/index.php/jss/article/view/v008i14.

National Aeronautics and Space Administration, Science Mission Directorate (2010). *The Electromagnetic Spectrum*. URL: https://science.nasa.gov/ems. (accessed: 12.03.2025).

Pharr, Matt, Wenzel Jakob, and Greg Humphreys (2023). *Physically based rendering: From theory to implementation*. 4th edition. Cambridge, MA: MIT Press.

Shirley, Peter, Trevor David Black, and Steve Hollasch (Aug. 2024a). *Ray Tracing in One Weekend*. URL: https://raytracing.github.io/books/RayTracingInOneWeekend.html. (accessed: 19.03.2025).

– (Aug. 2024b). *Ray Tracing: The Next Week*. URL: https://raytracing.github.io/books/RayTracingTheNextWeek.html. (accessed: 25.03.2025).

Shirley, Peter and Raymond K. Morley (2003). *Realistic ray tracing*. 2nd edition. Natick, Mass.: AK Peters.

Stark, Glenn (Feb. 2025). *light. Encyclopedia Britannica*. URL: https://www.britannica.com/science/light. (accessed: 11.03.2025).

Whitted, Turner (June 1980). "An improved illumination model for shaded display". In: *Commun. ACM* 23.6, pp. 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882. URL: https://doi.org/10.1145/358876.358882.