Technical Report CS-2019-10

RAYTERM:
A Ray-Tracing Rendering Engine for XTerm-like Terminals

Saejin Mahlau-Heinert

Submitted to the Faculty of
The Department of Computer Science

Project Director: Dr. Janyl Jumadinova

Second Reader: Dr. Gregory Kapfhammer

Allegheny College
2019

I hereby recognize and pledge to fulfill my responsibilities as defined in the Honor Code, and to maintain the integrity of both myself and the college community as a whole.

_____

Saejin Mahlau-Heinert

SAEJIN MAHLAU-HEINERT. RAYTERM**:**
A Ray-Tracing Rendering Engine for XTerm-like Terminals.
(Under the direction of Dr. Janyl Jumadinova.)

ABSTRACT

Over the many years of innovation in the field of computer graphics, advances in rendering have led to massive increases in the fidelity of engaging, satisfying, and realistic computer visualizations. RAYTERM is a new and unique entry into the ranks of rendering engines, and makes its own contributions to the field of computer graphics. While harkening back to the retro aesthetics of the seventies and eighties, RAYTERM embraces new advances in computing power to bring fully ray-traced visuals to an old screen – the terminal. Using Unicode block characters to simulate pixels and a ray-tracer written in C++, RAYTERM renders a fully three-dimensional (3D) scene, complete with global illumination and physically-based materials. RAYTERM can be used as an engine for terminal-based 3D tools, visualizations, games, and more; it is fully open-source and ready for integration into other projects.

Contents

List of Figures

Chapter 1

Introduction

In the early days of computing, real-time rendering engines that powered games like "Doom" or "NetHack" had to run on extremely underpowered hardware and render on low-resolution screens. The dream of real-time, photorealistic graphics was far, far away. However, even then the simple, blocky graphics, easily recognizable shapes, and maze-like environments were fantastic entertainment. Today the retro-style of low-resolution graphics, pixel art, and 8-bit color abounds in the gaming space. RAYTERM is one part of enabling that retro-aesthetic to grow into a new and unique style that, while similar to the old classics, can be more engaging and real than they ever were.

This thesis document describes a system that creates a moving image on a terminal screen. This system is called RAYTERM; it performs real-time updating of the displayed image, thus generating a window into a three-dimensional (3D) world. RAYTERM is available for a wide variety of other programs to use as a rendering engine. With the current generation of powerful computers, able to execute billions or even trillions of calculations per second, it is finally possible to do real-time, close-to-photorealistic rendering. While the goal of RAYTERM is not high-resolution photorealism, some approximation is achieved.

In this introduction to RAYTERM, we will first cover the motivations behind creating this system in Section 1.1. Then, we will give an overview of the project in Section 1.2. Section 1.3 expands on the overview to provide some mathematical grounding for some of the algorithms used in RAYTERM. In Section 1.4 we cover the main development tools used in this project, such as the implementation language. Finally, Section 1.5 outlines the rest of this thesis document, introducing the major topics of discussion for the remaining chapters.

1.1 Motivation

I have always been fascinated with rendering algorithms, especially ray-tracing. This fascination fueled an interest in game engine development, 3D modeling, and other computer graphics topics. The idea for RAYTERM came out of a disappointment – I was working on a project that made a virtual space seem larger than it was when the user was wearing

a virtual reality headset. Sadly, someone had already developed a toolkit that did exactly what I wanted to achieve.

Through my search for a replacement project, I discovered that, for me, one of the biggest draws to an idea was the ability for that idea to affect other people. Whether it was something that many others could use or an idea that people could contribute to, I wanted to make something other people saw. That is where the draw of open-source software came from for me, and why RAYTERM is open-source. Additionally, I discovered that there are many niches in software – places where there just is not a lot of people making things. Computer graphics in a terminal emulator (see Section 1.2.2) is one of those corners. The only real application many people saw for this niche was the rendering of images to the terminal – in fact, some of the initial inspiration for RAYTERM came from TerminalImageViewer [16], a program to accomplish that very task.

I saw more possibilities, however. There is no need to keep the terminal in a by-gone era, stuck with only colored characters and still graphics. The terminal is, after all, only a window into some text. Why does this text have to be stationary? Why does it need to be text at all? The answer, of course, is that it does not. There is a huge amount of potential in a 3D rendering engine that outputs to the terminal – there are so many applications, from rendering 3D model files to playing video games to visualizing music. All that these programs need is an infrastructure around the terminal for 3D rendering. RAYTERM aims to create that infrastructure through a completely novel combination of ray-traced 3D rendering and Unicode-based image composition.

1.2 Overview

RAYTERM uses the recursive ray-tracing algorithm, simulating the path of light through the scene – this is described in more detail in Section 1.2.1. Many images are generated, or rendered, each second; once an image is rendered the Tickit library [9] is used to display it in a terminal. Multiple images are displayed in quick succession, creating the illusion of movement. More details on this process are given in Section 1.2.2. Rendered images are displayed in two different modes; the first uses single half-character pixels, and the second more complex Unicode block characters. The mechanics of these image composition methods are discussed in Section 1.2.3. Finally, Figure 1.1 is a still image of the final terminal output.

1.2.1 Rendering Engines and Ray-Tracing

A rendering engine is an algorithm that takes a scene – a description of some collection of objects – and generates an image or visual representation of that scene. There are many different algorithms that accomplish this goal. RAYTERM utilizes a variant of the recursive ray-tracing algorithm first pioneered by Turner Whitted in his ground-breaking paper

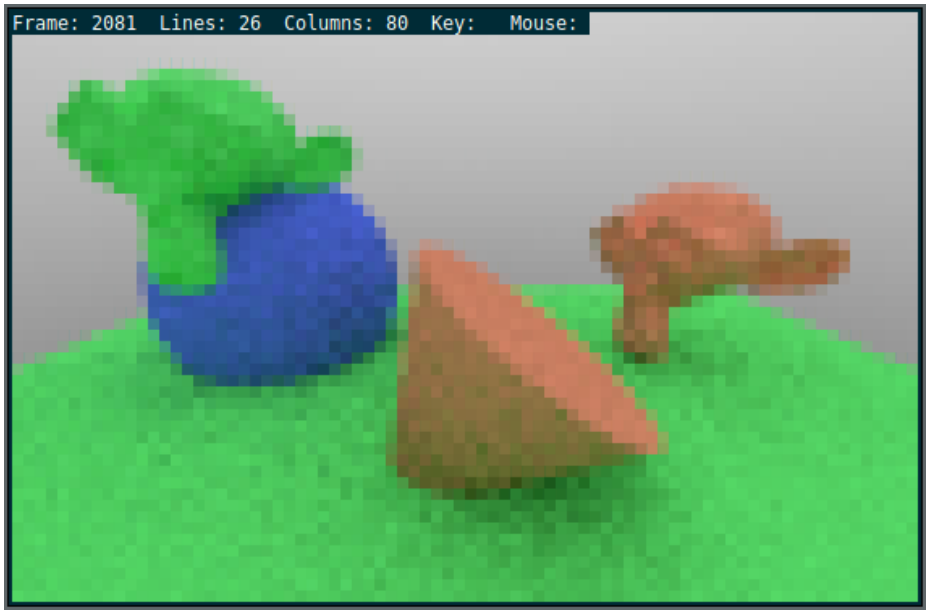


Figure 1.1: Example of terminal output

"An Improved Illumination Model for Shaded Display" [51]. Ray-tracing was one of the first algorithms developed in the field of computer graphics, and although there have been some improvements since then, the idea behind the algorithm has maintained its original simplicity.

Ray-tracing has been used as the rendering algorithm of choice for photorealistic scenes because with only a few modifications to Whitted's original algorithm, it can generate fantastic images. In the past, however, render times have been so slow that it was impossible to generate images fast enough for real-time use. For example, Figure 1.2 is a render created by the POV-Ray engine [5]. POV-Ray can take between a few minutes to several hours to complete one single image depending on the processing power involved. However, there have been a few recent innovations that change this, such as NVIDIA's RTX hardware acceleration. Section 1.4.2 provides more details on these advances and how RAYTERM takes advantage of them.

Ray-tracing revolves around the idea of *rays*, a mathematical construct that represents an infinite line emanating in some direction from a starting point in three-dimensional space. In RAYTERM rays are used to simulate the path that light takes as it travels around the scene. When a ray intersects with an object in the scene various interactions take place that simulate how light may travel under different conditions. It is worth noting that a base assumption is made in ray-tracing when using straight rays as described here: light follows a straight line without changes. This is only the case in reality when light travels through a vacuum with no gravitational bodies; thus, basic ray-tracing is not exactly photorealistic and does not model phenomena like atmospheric scattering. Additional mathematics and systems must be used to bend, attenuate, or scatter a ray to accurately render transparent volumes – this is called volumetric ray-tracing. Volumetric ray-tracing is not currently

Figure 1.2: POV-Ray render created by Gilles Tran [47]

supported in RAYTERM, but is a fascinating area for future work.

The formal basis for ray-tracing is the rendering equation, articulated by James Kajiya in 1986 [22]. The rendering equation models the outgoing light at some point in some direction given all incoming light, a bidirectional reflectance distribution function (BDRF), and a normal (a perpendicular direction) for the surface at the point modeled. If solved for every point in the scene, the rendering equation could generate a completely photorealistic image – this would, however, require massive amounts of computation. This is because the rendering equation contains an integral over all incoming light which must be solved through numerical analysis. Ray-tracing algorithms that do this are known as *path-tracers* and are the most photorealistic rendering algorithms invented.

RAYTERM does not use a full path-tracer – such algorithms are still many times too slow for most real-time rendering situations; however, a Monte-Carlo method that can approximate a full path-tracer is used. In RAYTERM, the rendering equation is solved by sampling the incoming light at each point with rays. This is known as recursive ray-tracing since it starts with a single ray that "recurses" when it encounters a surface. The eventual goal of the recursive ray-tracing algorithm is to create a tree of rays for each *fragment* to render. A fragment is either a pixel or some sub-pixel – many systems will use many fragments per pixel to get better anti-aliasing and more accurate results. Each ray contains some color that represents the color of the light in that ray. When a ray hits an object, it is *scattered* by the *material* of the object, generating a new ray. This ray's trajectory is modified randomly based on probability and the specific material hit; for instance, if the material is reflective

then the ray will be biased towards the reflection direction.

The base of each generated tree is an *eye-ray* – a ray with its origin at the fragment location on the camera plane. The eye-ray's color is the color that will be rendered for that fragment. As the eye-ray projects forward, away from the eye, it is tested for intersection with all objects in the scene. When an intersection happens and the generated rays scattered, the eventual color values of the scattered rays are combined to produce the color of the original eye-ray. This is done recursively to fully render the scene.

1.2.2 Terminal Output using Tickit

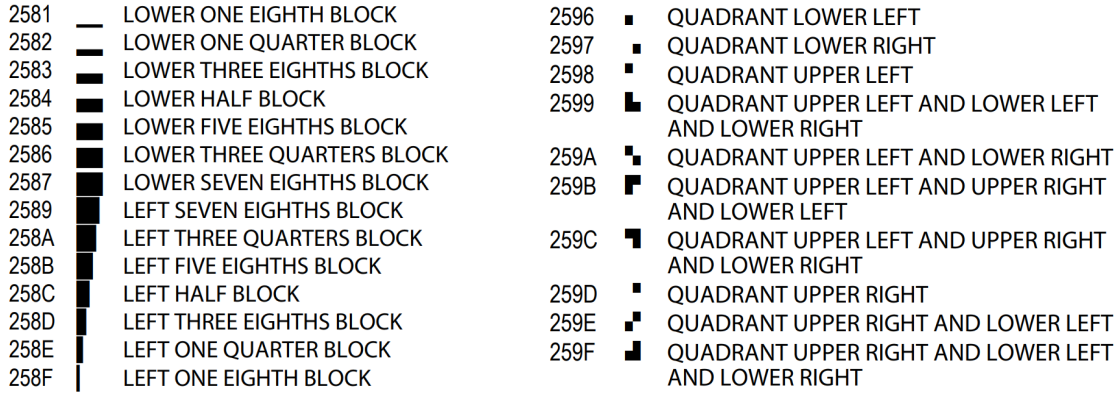
Output in RAYTERM is shown in a terminal window. A terminal is a specific kind of interface for a computer that uses text-based commands and feedback, as opposed to a Graphical User Interface (GUI). It descends from the old days of mainframe computers, where the actual computer was not under a desk – instead, users would connect over a network to the main computer, using an actual terminal machine. Today, the terminals most people use are actually "terminal emulators", because they simply emulate these old mainframe terminal machines. These terminal emulators are often simply programs that run in an existing graphical desktop environment, providing a text-based interface to the operating system.

Once an image is generated, RAYTERM must somehow display that image in a terminal with no surrounding prompt or other formatting – as a simple *character field*. To do this, we use a two-step approach: first, we represent the image using Unicode characters (see Section 1.2.3), then we print those characters to the terminal using the Tickit (Terminal Interface Construction Kit) library [9]. This library abstracts implementation details of the terminal to enable full-window 24-bit color character output. In RAYTERM, the terminal window is divided into two "panels", a main panel for the actual image output – updated 30 times or more per second – and an info panel to render information such as frames per second and logging text. Additionally, extra terminal output such as the command prompt or other character output is disabled.

The main problems that Tickit solves are ones which need to use terminal escape codes – short character sequences that tell a terminal emulator to change some specific data or use a different display method. These escape codes tell the terminal how to display text by changing the formatting, positioning, or color. The library also disables character "echoing", the display of characters that are typed by the user. Because of this dependency on Tickit, RAYTERM can be used only with terminals that support it – generally, any XTerm-like terminal will work, as long as it supports `terminfo` [9]. Additionally, full RGB direct color support is required for accurate color information; depending on Tickit's status, XTerm may be the only compatible terminal emulator.

1.2.3 Image Composition using Unicode Characters

Unicode is a character standard that allows anyone to reference many thousands of characters to compose text, no matter the environment around the text [7]. Some critical characters that RAYTERM uses are known as the *block characters* – they are characters `U+2580` – `U+259F`, a subset of which are shown in Figure 1.3. The core of image composition using Unicode is an algorithm coloring the characters and the background – RAYTERM uses this algorithm on every character of the output to both determine the character to display, and the foreground and background colors for that character. The foreground colors the character itself, whereas the background provides a relief color. This allows each *character pixel* to represent a hard gradient.



(a) Block Elements

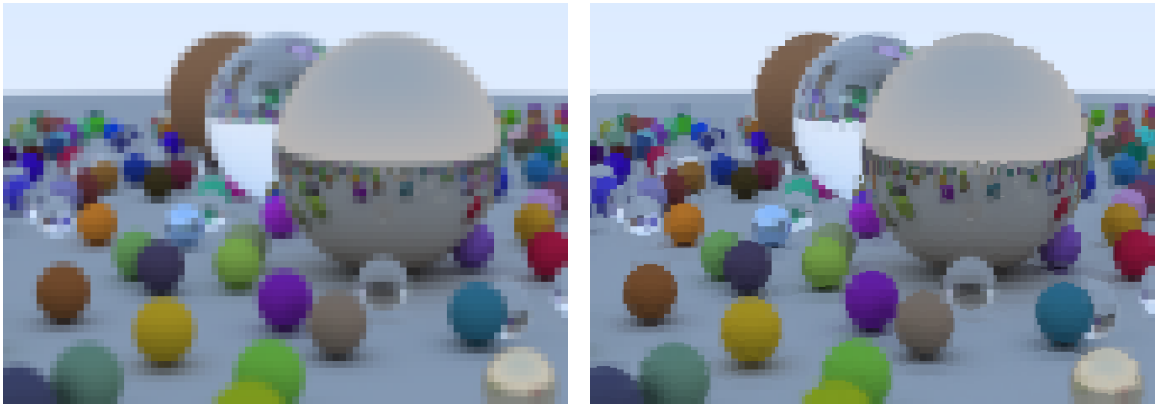
(b) Quadrant Elements

Figure 1.3: Subset of `U+2580` – `U+259F` [7]

There are two main image modes possible for use in RAYTERM. The first is pure *pixel mode*, in which the Unicode "half-block" symbol (`U+2584`) is used; this is the current default. Since monospaced character output (such as in a terminal) is twice as tall as it is wide, the half-block can split a single character into two square pixels that are colored differently: the upper pixel with the background color of the character, and the lower with the character or foreground color of the pixel. This means that a typical 85 by 30 character terminal results in a screen space of 85 by 60 pixels. This mode also dramatically reduces the number of ray-tracing computations needed, since only one ray per pixel per sample is required.

The second image mode is considerably more complicated and slower, as it uses significantly more rays per character in order to determine what Unicode block character most fits the desired output. On the other hand, it allows a much higher perceived resolution, since the characters used have smaller footprints of down to an eighth of a character in width or length. The differences between these two modes are highlighted in Figure 1.4a and 1.4b. It can be seen that the first image mode, *pixel mode*, shows a rather fuzzy definition of the main large sphere. However, in the second image mode, *character mode*, the sphere and the reflections seen in it are more defined. The performance impact of both modes is

another factor that is assessed when making implementation decisions for RAYTERM; it informed how much actual definition was added by the second rendering mode. The current implementation does not fully implement character mode; instead, this is considered a well-defined area for future work.



(a) Pixel Mode Image Output (b) Character Mode Image Output

Figure 1.4: Examples of Image Mode Output

1.3 Background

The background information for the algorithms and techniques used to implement RAYTERM is discussed in this section. We introduce basic vector mathematics in Section 1.3.1. In Section 1.3.2 we detail most of the specific mathematical algorithms involved. In RAYTERM, many of these algorithms are implemented behind the scene in APIs like OptiX and CUDA.

1.3.1 Vector Mathematics

Vector mathematics is a way to represent operations on n-dimensional constructions. The main unit of computation is a vector – essentially, a list of n numbers. A vector is denoted with an arrow over its name: $\vec{v} = (-2, 3, 1)$ is a vector. A specific value in a vector is referenced by either its zero-based index, $\vec{v}_0$ is the first value in $\vec{v}$ (note the arrow is above v only, and not the index), or its dimension – $\vec{v}_x$ is the same value. The dimensions are by convention x, y, z, and w. In RAYTERM however, we generally work only with vectors where $n = 3$ (only in x, y, and z), and each vector represents either a direction or a position in three-dimensional space. Vectors are related to both multivariable calculus and linear algebra – an expansion on this basic explanation can be found in virtually any textbook on these topics.

1.3.1.1 Vector Operations

Each set of all possible n-dimensional vectors are closed under the basic arithmetic operations. Addition, subtraction, along with some specialized versions of multiplication, are all defined. Addition and subtraction are calculated on a component-wise basis – for instance, in three dimensions addition and subtraction can be calculated using Formula 1.1 and Formula 1.2.

$$\vec{u} + \vec{v} = (\vec{u}_x + \vec{v}_x, \vec{u}_y + \vec{v}_y, \vec{u}_z + \vec{v}_z) \tag{1.1}$$

$$\vec{u} - \vec{v} = (\vec{u}_x - \vec{v}_x, \vec{u}_y - \vec{v}_y, \vec{u}_z - \vec{v}_z) \tag{1.2}$$

There are three other operations, each based on numerical multiplication. The first is simple scalar multiplication. A scalar in this context is just a regular number, such as 2. Scalar multiplication is defined by Formula 1.3, and is also a closed operation for n-dimensional vectors.

$$k\vec{v} = (k\vec{v}_x, k\vec{v}_y, k\vec{v}_z) \tag{1.3}$$

Then we have the cross product. This is a specialized multiplication operation only applicable when $n = 3$ that results in a vector that is perpendicular to both of its operands. The cross product of $\vec{u}$ and $\vec{v}$ is denoted as $\vec{u} \times \vec{v}$, and can be calculated with Formula 1.4.

$$\vec{u} \times \vec{v} = (\vec{u}_y\vec{v}_z - \vec{u}_z\vec{v}_y, \vec{u}_z\vec{v}_x - \vec{u}_x\vec{v}_z, \vec{u}_x\vec{v}_y - \vec{u}_y\vec{v}_x) \tag{1.4}$$

Finally, the dot product is a third multiplication operation that is applicable to any n-dimensional vector. However, the result of the dot product operation is a simple scalar, not a new vector. The dot product only makes semantic sense when talking about direction vectors: it gives a sense of how much two vectors are "aligned" with each other. It is often used to determine if two vectors are at right angles with each other, since in that case the dot product would be 0. The dot product of $\vec{u}$ and $\vec{v}$ is denoted as $\vec{u} \bullet \vec{v}$, and can be found using Formula 1.5.

$$\vec{u} \bullet \vec{v} = \vec{u}_x\vec{v}_x + \vec{u}_y\vec{v}_y + \vec{u}_z\vec{v}_z \tag{1.5}$$

1.3.1.2 Vector Equations

With the operations described above, we can write equations using vectors. This greatly simplifies the description of the mathematics in Section 1.3.2, since without vector mathematics every operation would need to be notated in cartesian coordinates. It should be

noted that exponentiation for vectors is calculated on a component-wise basis – for example, $(1,\vec{4},2)$ squared is $(1,\vec{16},4)$. This is by convention in graphics programming since it provides an easy description of such an operation. This is not, however, always the case in other mathematical applications of vectors.

When thinking about the meaning and algorithms behind equations, it is extremely helpful to keep two main facts in mind. First, *crossing* two vectors results in a vector that is at a right angle to each; that is, it gives a normal to the plane in which both original vectors lie. We should note here that according to only this definition, there are actually two possible answers for each set of operands. However, only one of these answers is correct; to determine the right one, the "right-hand rule" is used. To use the right-hand rule, point your right hand's index finger in the direction of the first vector, and your middle finger in the direction of the second. Your thumb will point in the direction of the resulting cross product. Note that this causes the order of operands to matter – a different order will produce the opposite direction.

$$\vec{u} \bullet \vec{v} = |\vec{u}| \cdot |\vec{v}| \cdot cos(\theta) \tag{1.6}$$

Second, *dotting* two vectors multiplies their lengths while keeping in mind their direction – two vectors perfectly parrallel to each other would have a dot product that was simply their lengths multiplied. However, two vectors perfectly perpendicular to each other have a dot product of zero. There is actually a second definition of the dot product (Formula 1.6) that can be a little easier to understand with this context. In this formula, θ is the angle between the two operands, and $|\vec{w}|$ is the length of $\vec{w}$.

1.3.2 Ray-Tracing Mathematics

The mathematical basis for ray-tracing is heavily dependent on understanding vector mathematics – Section 1.3.1 is a short introduction. The basic mathematical construct used in ray-tracing is called a *ray*. Rays are defined by two vectors: an origin point, referred to as $\vec{R_{origin}}$ for some ray R, and a direction, referenced as $\vec{R_{direction}}$. These two vectors together represent a line of infinite length that starts at the origin and projects along the direction. An important addition to the concept of rays is a point along a ray – this can be defined by using a third variable, t, to represent how far along the ray the point is located. Therefore, Formula 1.7 can be used to get the coordinates of such a point in three-dimensional space. Assuming $\vec{R_{direction}}$ is a unit vector, this is the *parametric form* of a ray.

$$\vec{point} = \vec{R_{origin}} + t\vec{R_{direction}} \tag{1.7}$$

With this concept, algorithms can be described to model the interaction between rays and other surfaces. The algorithms discussed are run millions of times per second in a highly parallelized environment. Parallelization is a technique for running two or more algorithms

simultaneously; therefore even minor optimizations are extremely important since they reduce the overall time to render. All of the ray-tracing mathematics described here are synthesized from Peter Shirley's excellent "Ray Tracing in One Weekend" [44] and the Morgan Kaufmann textbook "Physically Based Rendering: From Theory to Implementation" [37]. Some smaller sections also benefit from ideas in Jean-Colas Prunier's "Scratchapixel" [41].

Scenes that can be ray-traced must be a collection of surfaces that are mathematically intersectable with a ray. Any surface that can be defined by an implicit surface definition function, in the form of Function 1.8, is intersectable with a ray [37]. In Function 1.8 and for the rest of this section, $\vec{p}$ is a vector representing a point in three-dimensional space. The implicit surface definition function must have the property that if and only if $f(\vec{p})$ is 0, then $\vec{p}$ is on the defined surface. The point of intersection between a ray and a surface defined in this way can be found by solving Equation 1.9 for t and then using Formula 1.7 to calculate the coordinates of that point on the ray. We essentially ask, "What points along this ray are on the defined surface?" by passing in all possible points on the ray, parameterized by t.

$$f(\vec{p}) = 0 \tag{1.8}$$

$$f(\vec{R_{origin}} + t\vec{R_{direction}}) = 0 \tag{1.9}$$

In RAYTERM, only triangles are supported, for simplicity of implementation. However, in the research prototype we describe in Section 3.1.1 disks and spheres were the only supported surfaces due to the simplicity of their parametric equations. In fact, a sphere is the simplest three-dimensional object to calculate ray intersection with, and therefore was the first intersection test implemented in `rayterm-cpu`. During extremely early feasibility testing, some of the math described in this section was even implemented in the Go programming language, instead of through existing APIs or in C++ as in the research prototype.

1.3.2.1 Ray-Sphere Intersection

The surface definition function of a sphere is Function 1.10, with S representing the sphere. The intersection point between a ray and a sphere is given by solving Equation 1.11 for t and then using Formula 1.7. Both of these equations are directly adapted from "Ray Tracing in One Weekend" [44]. Notice that Equation 1.11 is quadratic, and the number (and values) of the roots give us the t we want to use. The smallest positive root corresponds to the point on the ray which first intersects the sphere. If there are no real roots, then the ray does not intersect the sphere.

$$f(\vec{p}) = (\vec{p} - \vec{S_{center}})^2 - {S_{radius}}^2 \tag{1.10}$$

$$(\vec{R_{direction}}^2)t^2 + 2(\vec{R_{direction}} \bullet (\vec{R_{origin}} - \vec{S_{center}}))t + (\vec{R_{origin}} - \vec{S_{center}})^2 - {S_{radius}}^2 = 0 \quad (1.11)$$

1.3.2.2 Ray-Plane Intersection

For any two-dimensional object, the plane it lies in is the first shape tested for intersection. Luckily, ray-plane intersection testing is fairly cheap and straightforward. The surface definition function of a plane is Function 1.12, with P representing the plane. The plane's *offset* is a point on the plane, and the plane's *normal* is a vector perpendicular to the plane. The intersection point between a ray and a plane is given by solving Equation 1.13 for t and then using Formula 1.7. These equations were derived from basic vector math, along with guidance from "Physically Based Rendering" [37].

$$f(\vec{p}) = (\vec{p} - \vec{P_{offset}}) \bullet \vec{P_{normal}} \quad (1.12)$$

$$(\vec{P_{normal}} \bullet \vec{R_{direction}})t + \vec{P_{normal}} \bullet (\vec{R_{origin}} - \vec{P_{offset}}) = 0 \quad (1.13)$$

1.3.2.3 Ray-Triangle Intersection

The method for testing intersection with triangles is a bit more complicated than the other tests we've covered so far. The mathematics for this section are again derived from vector mathematics. However, Jean-Colas Prunier's "Scratchapixel," an excellent and accessible online resource for 3D rendering [41], was also an indispensable guide in facilitating understanding along the way. First, intersection is tested with the plane the triangle lies in – this results in a ray-plane intersection point $\vec{Q}$, or no intersection. Then, if there was an intersection, another test must be performed to detect if $\vec{Q}$ is inside the triangle. We do this using the "inside-outside" method (as suggested by "Scratchapixel"): test if $\vec{Q}$ is on the left side of each edge.

To conduct the test, we label each triangle vertex $\vec{V_i}$, with i increasing in the counter-clockwise direction. We then have three triangle vertices: $\vec{V_0}$, $\vec{V_1}$, and $\vec{V_2}$. We can now use Function 1.14: if $f(i) > 0$ for each i, then $\vec{Q}$ is inside the triangle. Otherwise, $\vec{Q}$ is outside the triangle. Note that in Function 1.14, if $i = 3$, then $i = 0$ (i "wraps" to only valid values).

$$f(i) = P_{normal} \bullet ((\vec{V_{i+1}} - \vec{V_i}) \times (\vec{Q} - \vec{V_i})) \quad (1.14)$$

Function 1.14 can be complicated to visualize, so imagine this: we first form two vectors, both with $\vec{R_{origin}} = \vec{V_i}$. The first points along the triangle's edge, while the other points towards $\vec{Q}$. Both of these vectors will be in the plane of the triangle. Thus, if we cross

them, the vector produced will either be away from the plane in the same direction as $\vec{P_{normal}}$, or away from the plane in the opposite direction. According to the right-hand rule, if the crossed vector is in the same direction as $\vec{P_{normal}}$, then the vector pointing to $\vec{Q}$ is "to the left" of the vector pointing towards $\vec{V_{i+1}}$. We can then calculate the dot product between $\vec{P_{normal}}$ and the crossed vector – if it is positive then the crossed vector is in the same direction as the normal, and therefore $\vec{Q}$ is to the left of the edge. One last addendum to this intersection test is the fact that it can be difficult to perform the counter-clockwise numbering of vertices. We therefore simply expect vertices to be specified in counter-clockwise order. This is similar to what many other 3D rendering algorithms assume.

1.4 Development Ecosystem

RAYTERM's development relies on many other systems. The actual implementation code is hosted on GitHub, a public platform for projects using the Git version control system [17, 2]. A continuous integration system called Travis CI [11] is used for testing and environment management. Documentation is handled by `doxygen` [48] – all external facing functions, methods, and classes have a minimum of a few sentences of documentation. More details about the implementation tools are given in Section 1.4.1; testing hardware and low-level APIs are discussed in Section 1.4.2.

1.4.1 Programming Languages and Tools

With a large project such as RAYTERM, there are certain choices that must be made from a software development perspective. These choices inform how the project is modified, built, and eventually executed. In the case of RAYTERM, we used the C++ language [20] for main development, and Gradle [18] as a build system.

1.4.1.1 Implementation Language

The C++ language was used for three main reasons. First, the language has a huge ecosystem of low-level tools, with interfaces to libraries such as OptiX, CUDA, Eigen, and much more. This ensures that no matter the need, there was most likely a library out there that could fill that need. Secondly, C++ allows low-level C-like programming while keeping abstractions such as objects available. Lastly, C++ is a reasonably fast language: with no virtual machine, unlike Java or Kotlin, garbage collection is not a burden. Decades of work have gone into compiler toolchains such as `g++` and `clang`, enabling optimizations that would likely not be possible for newer languages.

RAYTERM attempts to keep most of its implementation simple and straightforward, using classes and greater abstractions only as necessary. This ensures that overhead is minimal, as well as guaranteeing readability for open-source contributors. Advanced features of

C++ such as templating are avoided so as to keep the knowledge entry barrier low. Finally, all code in RAYTERM's implementation conforms to the Google C++ style guide [13], as checked by `clang-tidy`.

1.4.1.2 Build System

Complex projects can be a nightmare to manage, especially when there are multiple contributors. Build systems are an important tool that simplify this headache. An opinionated build system such as Gradle also specifies sensible defaults for most situations, requiring minimal starting configuration. This is why Gradle was chosen as the build system for RAYTERM. Gradle was used to compile C++ code into an executable for distribution or testing using its Software Model feature. This feature allows the specification of executables and interdependent components, with sensible default source code and binary locations. Gradle efficiently manages long linking or dependency lists, without resorting to macro-infested and variable-filled makefiles; it also handles header inclusions seamlessly, without dealing with relative path issues. Gradle can even automatically retrieve dependencies if they are published as a Maven artifact. All of these features make ongoing development and maintenance an easier proposition.

1.4.2 Hardware

There are many implementation issues related to the performance aspects of RAYTERMsince high performance is a strict requirement for obtaining render times low enough for real-time display. For the purposes of testing, we use a machine with an Intel Core i7 4770k CPU and NVIDIA Geforce GTX 980 Ti GPU. Only recently has ray-tracing even been able to achieve real-time levels of performance; much of this progress is thanks to advances in GPU technology. The largest innovator in this space, GPU chip design company NVIDIA, released its RTX series of GPUs that have hardware support for ray-tracing. RTX hardware spread is slow, however, due to its high early adoption cost. Therefore we do not focus on RTX hardware; instead, we look to slightly older technologies to provide the implementation backbone; this works well with the existing test hardware, which is a few generations old. However, if we can easily optimize for RTX hardware we do choose to, as we have in using RTX mode for OptiX. Some additional discussion on this topic is available in Chapter 3, especially when comparing the sequential and parallel render engines.

1.5 Thesis Outline

In the remaining chapters of this thesis document, we will describe the features of RAYTERM, detail more advanced background information, and discuss implementation. Chapter 2 reviews a number of past approaches to ray-tracing, and provides background for some more

advanced ray-tracing algorithms and data structures. It also describes some tools that we used for RAYTERM's initial proposal and testing. Chapter 3 outlines the high-level conceptual algorithms, component design, and methodology used in this project. It is complemented by Chapter 4, which is an in-depth description of RAYTERM's implementation, including example output from many different stages of the project. Finally, Chapter 5 completes this document, describing the future work left to complete, a hope for open-source contributions, and a look back at the progress made throughout this development process.

Chapter 2

Related Work

In this chapter, we will discuss a few related research papers, detail how they informed RAYTERM's development, and show the improvements of RAYTERM over other systems. First, we will conduct a high-level discussion of the first ray-tracing paper through to basic details on modern-day optimizations. Following that, we will discuss two Github projects, one of which was a major inspiration for Unicode image composition and the algorithms involved (see Section 1.2.3).

2.1 Discovery of Ray-Tracing

Here we describe the first ray-tracing paper, as well as two seminal works that brought ray-tracing to the forefront of photorealistic computer rendering. It is important to note here that many of these papers are relatively old, since the algorithms being used in RAYTERM are correspondingly dated. This is because most recent advances in ray-tracing concern more advanced methods of photorealistic rendering, such as path-tracing [28], low-level optimizations [50], dynamic scene rendering [49], and volumetric ray-tracing hardware acceleration [26], to name just a few. All of these topics, while interesting and tangentially related, are too advanced; comparison with the algorithms used in RAYTERM would be unrealistic and meaningless, beyond showing how much more development RAYTERM needs to be truly state of the art when compared to modern ray-tracers.

2.1.1 The First Ray-Tracer

The very first ray-tracing algorithm was developed by Arthur Appel in his 1968 paper "Some techniques for shading machine renderings of solids" [3]. His algorithm is now known as a ray casting algorithm – it does not follow the approach we described in Section 1.2.1. Instead, rays are traced from a point light source to the object being shaded, and a plus symbol of varying size is rendered at that location, depending on the intensity of light at that point. When a photographic negative is taken, light spots that were not hit by the rays

(thereby darkening them) are now "in shadow", as the color levels were inverted. Today, Appel's work is not normally considered a real ray-tracing algorithm. However, his work informed much of the following research, especially his ideas and mathematics on light intensity. After Appel's work, there were some commercial applications of ray-tracing in the field of radiosity; ray-tracing was used to track radiation inside tanks at Mathematical Applications Group, Inc (MAGI) [1].

2.1.2 The Breakthrough

The next big entry to the ray-tracing field was Turner Whitted's 1980 paper "An Improved Illumination Model for Shaded Display" [51]. In this groundbreaking and approachable work, Whitted introduced the recursive ray-tracing algorithm we covered in Section 1.2.1. Whitted was not the first to use ray-tracing – Arthur Appel had first pioneered the field over a decade ago, and there were also emerging commercial applications. Whitted's real contribution was the idea for how to improve ray-tracing so that it could solve the problem of *global illumination*. Global illumination was not yet formalized, but the idea was to somehow gather the effect of all light in the scene on every single point. This can easily be categorized as taking into account both direct lighting, light coming directly from a source, and indirect lighting, light coming from reflections, refractions, or other non-direct travel methods.

Recursive ray-tracing approximates direct lighting easily, as each encountered point sends a ray to each light in the scene. Indirect lighting is slightly more difficult – pure reflection and refraction are easy, but as soon as reflections become more diffuse, and there are many, many directions a point gets lit from, recursive ray-tracing breaks down and path-tracing must take over. Despite this, ray-tracing was able to achieve remarkably good images, and thus Whitted-style ray-tracing was born. Even today, any simple recursive ray-tracing algorithms are known as Whitted-style ray-tracers. Indeed, RAYTERM uses a Whitted-style ray-tracer, with only some small enhancements, such as *heuristic emissive sampling*. Essentially, this sampling adds a new generation method for rays so that when a ray encounters a diffuse reflective surface, we attempt to sample the possible locations for other emissive (light-producing or light-reflecting) objects. This adds some rudimentary support for area lights and diffuse reflections, although it is still not photorealistic.

One element that is not mentioned in "An Improved Illumination Model for Shaded Display", but is in Whitted's retrospective on the paper [1], is adaptive super-sampling. This method generates more eye-rays, and therefore more definition, only in fragments that do not have sufficient detail. This is a relatively simple algorithm: the corners of each pixel provide four sample points creating a "sample square". Then, if each of the four points are sampled relatively close in value, and there isn't a small object visible through the pixel, the average sample is taken from each of the four points. If the points are not close in value, or there is a small object, then the sample square is subdivided and the process starts again, reusing some old values and calculating new ones. This method is inherently sequential and adds additional complexity to the ray-tracing algorithm. Therefore, it is not imple-

mented in the current version of RAYTERM; however, it could be a future optimization. This optimization could, in fact, treat the entire starting canvas as a single sample square, and then refine sample points from there. Depending on the complexity of the small object test, this could be an improvement.

2.1.3 Formalization

The path to fully photorealistic rendering was blazed soon after Whitted's paper. The mathematical basis for all of ray-tracing and photorealistic rendering was published by James Kajiya in 1986 [22]. In his paper "The Rendering Equation", he articulated a generalization for many different rendering algorithms; this generalization is shown as Equation 2.1 below. Although the idea behind the rendering equation was not completely new, Kajiya presented it in a form using vector mathematics, especially suited for computer graphics. The equation also gives direction for more advanced and photorealistic rendering techniques, leading up to path-tracing.

$$I(x,x') = g(x,x')\left[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right] \tag{2.1}$$

This equation is fairly complex, but a basic and simplified explanation will be given here. First, we will describe the terms: $I(x,x')$ is related to the intensity of light passing from point x' to point x; $g(x,x')$ is a "geometry" term; $\epsilon(x,x')$ is related to the intensity of emitted light from x' to x; $\rho(x,x',x'')$ is related to the intensity of light scatter from x'' to x by a patch of surface at x'. Note that inside the integral is a recursive reference – this is where we build an entire scene's lighting influence on a single point. We start with some definitions: x' is the point we just hit with our sample ray, x is the origin of our ray, and x'' is a point that we get through further ray-tracing – it is a point found by reflecting or scattering the impacting ray. The integral is actually the simplest element in the equation; it samples all possible incoming directions for light (referencing itself recursively), calculates the possible reflection from that light along the outgoing direction back towards x, and sums all these reflections up. The ϵ term is the amount of light emitted from x'. Finally, the g term is included in Kajiya's original paper, but most other formulations remove it in favor of attenuation terms inside the integral. Essentially, it controls how the geometry around x' affects the outgoing light.

Since for practical purposes we can't infinitely recurse, we will eventually hit a recursion level when evaluating the rendering equation. In this case, the incoming light is simply set to the ambient light level and no additional recursive calculations take place. This takes care of all reflection, refraction, and light travel; notice, however, that we must solve the integral over an entire sphere of directions, and we must do this for every single point we sample, for every single direction we sample from. This is *horrendously* inefficient. The significant part of this equation for RAYTERM is the integral, since if we can approximate that, RAYTERM would be photorealistic.

In RAYTERM, we use the rendering equation to inform the recursive ray generation, attempting to find areas of the integral where most of the light is incoming; this method we name *heuristic emissive sampling* (also mentioned above in Section 2.1.2). In the common case, all light will be from the direction of nearby light sources. However, we can also bias newly generated rays towards areas where refracted light may exist, such as around glass materials. Keeping this mathematical basis in mind was critical during implementation since the further we stray from this equation, the less photorealistic the final rendered images become.

2.1.4 Accelerated Intersections

Ray-tracing requires intersection tests with every object in the scene. If it were possible to drastically reduce the number of tests computed for each ray, perhaps by only testing the ray against objects in its general vicinity, massive speed increases would emerge. As it turns out, this was first explored right after Whitted's original paper. The Bounding Volume Hierarchy (BVH) acceleration structure was proposed in Steven Rubin and Turner Whitted's 1980 paper "A 3-dimensional representation for fast rendering of complex scenes" [42].

A BVH groups objects into hierarchical organizations, with each group covering a larger area than the groups inside it. Each group has a *bounding volume*, a geometric primitive that encloses all members of that group. In most common implementations of a BVH, this bounding box is a cube, since it is easy to determine the required box from a list of vertices inside it. Ray-tracing intersections are then done with the root group's bounding volume first. Once an intersection is found, instead of evaluating the rendering equation, we progress down the hierarchy from the bounding box we hit, and try more intersections. This can potentially skip the vast majority of geometry in the scene, thereby speeding up computation. There have been numerous improvements to this idea; however, BVHs continue to be an easy-to-implement and efficient acceleration structure [39]. The main acceleration structure used in RAYTERM is in fact a modified BVH from the OptiX library.

2.1.5 Parallelization

Ray-tracing is inherently parallelizable because each eye-ray and its descendants are independent – rays cannot collide, and one fragment's eye-ray does not affect another fragment's eye-ray. Thus, there have been many optimizations that target enhanced hardware acceleration to enable massively parallel ray-tracing. The first example of this was "Design and Analysis of a Parallel Ray Tracing Computer" [6], a 1983 paper by John Cleary and others. In it, Cleary describes a computing system that is functionally similar to the Graphics Processing Units (GPUs) available today. They were even able to build small prototypes, but calculated that a full-blown system would cost $50 million and be able to generate 1000 by 1000 pixel images in 0.15 seconds. Sadly, 1983 chip technology had not

yet progressed to the point where such a large multi-core processor was feasible.

However, all is not lost. Much of the research done in this area led to the design of systems such as OptiX and CUDA, enabling projects like RAYTERM to benefit from the massive speed increases estimated by early papers [36, 34, 1]. Instead of a $50 million monstrosity, relatively small GPU chips capable of the same performance are now available for only hundreds of dollars.

2.2 Related Applications

2.2.1 Terminal Images

A small program available on Github was the original inspiration for some details of RAYTERM. Called TerminalImageViewer (`tiv`) [16], it uses RGB ANSI escape codes and Unicode block characters to display images in a terminal window. RAYTERM uses many of the same ideas to produce its own animated output. The algorithm behind `tiv` is also a direct inspiration for the algorithm used for ray-to-character translation. RAYTERM, however, improves upon TerminalImageViewer in two incredibly impactful ways: the graphics produced are animated, and the input data is a three-dimensional scene and not an image. Fundamentally, `tiv` and RAYTERM are similar only in output style, while the purpose and internal workings are generally dissimilar.

2.2.2 Gaming with Termloop

A goal of RAYTERM is to support future games that might use the terminal as a graphical display. Games have utilized the terminal as a display mechanism for a long time – text-based adventure games were built with a terminal in mind, and RPGs also got their start with randomly generated levels displayed in 2D in a terminal window. Termloop is a game engine built to display games in a terminal [4]. It has support for collision detection, keyboard and mouse input, ASCII art, camera movement, and more. Its purpose is similar to RAYTERM's – namely, to facilitate game creation in the terminal. Termloop is also built in pure Go, a fairly portable programming language that has the advantage in simplicity when compared to C++. However, Termloop, like all other terminal game engines available today, has one glaring limitation: it is not 3D.

Unlike RAYTERM, Termloop cannot support complex environments, shaded models, or other modern graphical features common in game engines today. This one differentiating factor is huge; it means that no games can ever be made that don't fit into the strict field of two-dimensional graphics. RAYTERM changes this, enabling huge possibilities for future programs and applications built on top of RAYTERM. This also allows for existing games to be rewritten or "ported" to RAYTERM; perhaps in the future, "Skyrim" or "Doom" will be running in a terminal! While RAYTERM doesn't have the other niceties such as

collision detection, the ultimate goal of RAYTERM, beyond this initial implementation and development, is to grow to provide the same level of support for 3D games that Termloop has for 2D games.

2.2.3 Modern Ray-Tracing

Although we have described older methods for ray-tracing, and some modern optimizations, we have yet to mention a modern ray-tracer. OctaneRender is a professional real-time 3D renderer [19]. It is based on a GPU path-tracing algorithm, boasting physically correct shading while denoising its output using sophisticated machine learning algorithms. OctaneRender is far beyond RAYTERM in terms of raw optimization, power, and photorealism. It implements a full path-tracing algorithm, along with volumetric rendering support. Another big advantage OctaneRender has is the ability to render models and textures that do not fit into a GPU's memory. This is very impactful for large scenes, but luckily the benefits would not be very great for RAYTERM, since the low resolution prevents the details that take up that large memory space, like high-resolution textures, from making a difference.

In comparing OctaneRender to RAYTERM, it is important to keep in mind that although RAYTERM is neither as performant nor as photorealistic, it does provide a somewhat similar experience. This is because RAYTERM renders to a much lower resolution screen: the terminal. This introduces many advantages that OctaneRender does not have; first and foremost is the ability to run on slightly less powerful machines. The very latest GPU is not require to utilize RAYTERM, whereas OctaneRender will use all the compute power it can get and more. RAYTERM is also very forgiving when rendering 3D assets such as models – after all, the resolution is incredibly low. This makes development of visuals for RAYTERM-based game engines much, much easier and faster. Finally, RAYTERM has an aesthetic that can't be created through photorealism – it evokes memories of old, nostalgic video games and 90s movie CGI; RAYTERM fully embraces the retro theme, and benefits from doing so.

Chapter 3

Method of Approach

RAYTERM was developed using a modified agile approach; short development "sprints" were executed for key system development. We describe the product of some of those sprints in Section 3.1, primarily the ray-tracing engines. In Section 3.2 we also describe the Tickit interface and its algorithms. The overall structure of the project is twofold: there is an unfinished research prototype, called `rayterm-cpu`, as well as a finished prototype called `rayterm-gpu`, or simply RAYTERM. We will describe the development process of these implementations further in Chapter 4; this chapter focuses on the algorithms, design, and overall structure of the project.

3.1 Render Engines

While building RAYTERM, two separate render engine prototypes were implemented. The first, which we describe in Section 3.1.1, was a simple, single core sequential renderer [30]. This engine was developed in three main development sprints, over a period of about two months. The second, described in Section 3.1.2, uses CUDA [34] and OptiX [36] to leverage GPU compute power in parallel. This engine was developed over two sprints, in a little under a month.

3.1.1 Sequential

The sequential renderer, or `rayterm-cpu`, is a CPU-only ray-tracer. It supports three different types of materials: diffuse, metallic, and dielectric. These materials are complemented with two geometric primitives: disks, and spheres. Figure 3.1 shows an image of the `ppm` output of `rayterm-cpu` near the end of its development. The Tickit interface was never integrated into the CPU implementation, as this implementation is not performant enough for even simple scenes at low resolution. For example, Figure 3.1 was rendered in about twenty seconds on a modern CPU.

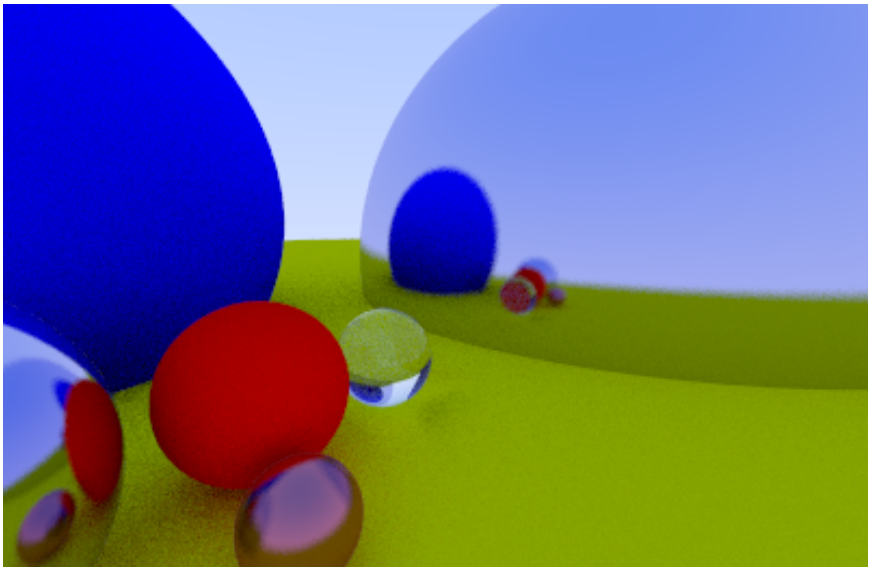


Figure 3.1: `rayterm-cpu` example `ppm` output

The main advantage of a sequential renderer is the lower development overhead since there is no need to interface with a complex device such as a GPU. This means that there is no need for handling parallel execution, and each ray is computed one after the other. Because of this simplicity, this engine's development was perfectly suited for the goal of gaining knowledge and experience in ray-tracing. The development also explored many fundamentals so that future contributions would be well-founded. With 219 commits and over a thousand lines of code in the final product, along with many thousands of additions and deletions, `rayterm-cpu` accomplished its goals.

3.1.1.1 Design

The sequential renderer's design relies on two main components, as well as the Eigen linear algebra library [15]. These components have a linear relationship, as can be seen in Figure 3.2; `raytrace` deals with tracing rays, `raymath` with intersection logic, and Eigen with linear algebra.

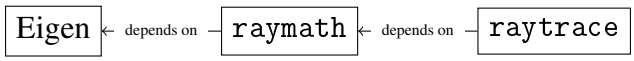


Figure 3.2: Component and dependency relationships in `rayterm-cpu`

Eigen [15] is a linear algebra for C++ that supports matrix and vector manipulations, various matrix decompositions and geometry features, and has many extensions for numerous other numerical operations. Eigen is also extremely well-optimized and uses SSE vectorized code, vastly improving performance on supported CPUs. In `raymath`, Eigen is used primarily for easy, tested representations and calculations involving vectors; all of the al-

gorithms implemented use equations derived from the ones in Section 1.3.2, and so benefit from easy implementation.

The base component in `rayterm-cpu` – `raymath` – provides support for various geometries and intersection routines. The geometries supported can be seen in Figure 3.3, and use Eigen structures to represent their geometric data. `raymath` also handles random number and vector generation, color handling, and intersection representation. The implementations behind these are covered in Section 4.1.1.

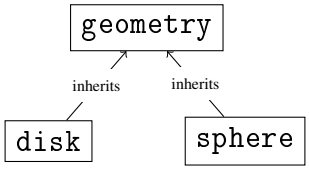


Figure 3.3: Geometric structures in `raymath`

The main component in `rayterm-cpu` is `raytrace`; it handles world representation, materials, and raytracing. `rayterm-cpu` only renders in `ppm` mode, since the Tickit interface was only integrated in the `rayterm-gpu` implementation, since that was the final prototype. `raytrace` represents objects with a `WorldObject` class, a list of which is stored in a `World`.

Figure 3.4 shows the function call path that is used when rendering a single ray. The main entry point is `raytrace_ppm`, which creates rays with a `Camera` class instance. This instance contains specific parameters to control the position and orientation of the virtual camera, and thus controls where the `ray` originates from, as well as its direction. Then, that `ray` is sent on to the `World` class, where an `intersection` is retrieved, and then used to generate a `color` from the given `ray`. This color is then finally returned to `raytrace_ppm`, which writes it to a `ppm` file. Unmentioned in this diagram are the recursive calls to `World::trace` from from `WorldObject::colorize`, which use the transformed "scatter" `ray` from `Material::scatter`. The implementation behind all of these structures and functions are covered in Section 4.1.2.

3.1.1.2 Algorithms

The sequential renderer implements single-branch Monte Carlo ray-tracing, a type of rendering that can produce physically accurate images with little development effort. The algorithm is largely based on randomness, only modifying the path of a single ray at a time as it bounces around the scene. When encountering objects, a particular Probability Distribution Function (PDF) is used to calculate where the ray will bounce. These PDFs vary based on the material and are hardcoded in the `Material` implementation. The result of this algorithm is extremely noisy because of this randomness (see Figure 3.5), so many samples must be taken and then averaged to gain a true value for a single pixel; this simulates the many millions of photons that would have traveled to that pixel in a real camera.

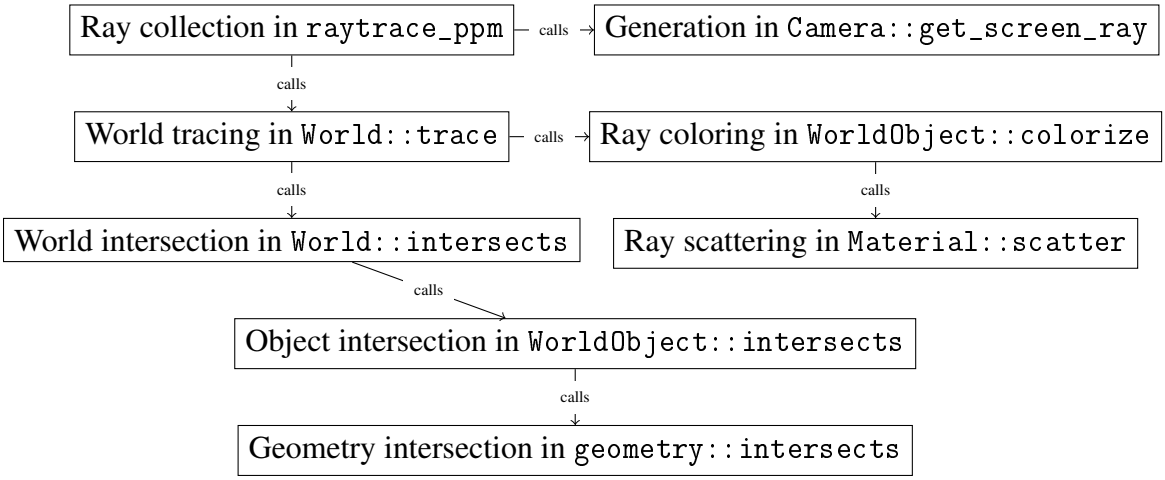


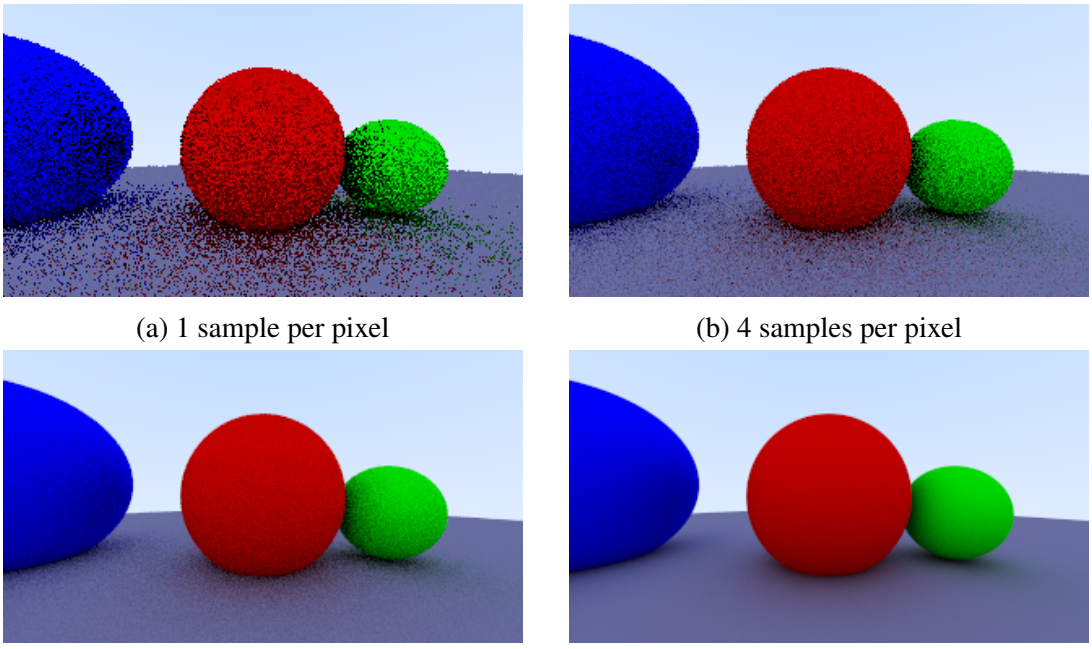
Figure 3.4: The journey of a `ray` and its `intersection`

The number of samples per pixel, sometimes termed "spp," is the number of rays generated from a random origin within the pixel; this is done for every pixel in the image. An example of various spp values is shown in Figure 3.5.

For an spp value of four, there are four rays generated for each pixel; thus, the rendering equation [22] is approximated at the first intersection by four random direction samplings (keep in mind the integration is over the sphere or hemisphere around the point of intersection for each ray). Thus, the more rays generated, the more accurate that sampling becomes. This is still not really solving the rendering equation and is a large simplification. Rather, the different eye-rays generated because of the spp value sample slightly different intersection points. These rays then also sample different directions out of those intersection points, and so are a kind of "fuzzy" approximation. An area for future work is implementing a kind of "stratification" of these samples, so that they are not randomly spread, but instead fully sample the sphere or hemisphere at a certain resolution; This would likely be combined with branched Monte Carlo rendering, which involves generating multiple rays for each intersection hit, thereby actually sampling the same integral. This would reduce the noise at low spp values significantly better than simply increasing the spp value, although at the slight cost of performance (more rays to calculate).

$$N_{ops} = R \cdot S \cdot D \cdot N \tag{3.1}$$

In discussing the complexity of this implementation, there are four main variables to keep in mind: first, the number of pixels to render, R; second, the number of samples per pixel, S; third, the maximum number of recursions per sample, D; and fourth, the number of intersectable objects in the scene, N. The number of pixels to render varies with the resolution of the desired image. The number of samples per pixel and the maximum number of recursions are both user-specified configurable values. The number of objects in the scene can vary wildly, but most test scenes in `rayterm-cpu` have less than 10 objects. This gives us



(a) 1 sample per pixel

(b) 4 samples per pixel

(c) 32 samples per pixel

(d) 1024 samples per pixel

Figure 3.5: Sample per pixel differentiated output from `rayterm-cpu`

Formula 3.1 to calculate the number of base "ray tracing operations," which we call N_{ops}. These operations consist of a single ray-tracing intersection operation, which involves testing a ray against a single parametric equation for intersection. As long as that operation can be considered constant, $O(R \cdot S \cdot D \cdot N)$ is the time complexity of the Monte Carlo algorithm used in `rayterm-cpu`. For spheres and disks, the only geometry supported, the intersection routine depends on both $sqrt()$ and $dot()$. However, the time complexities of these functions probably do not vary with respect to the value given to them. For taking the square root, there is likely system support for fast calculation independent of the input value; for vector dot products, the size is always constant at three entries. Because of this, we can consider the time complexity of a single intersection test to be constant. Thus, the time to complete a render varies proportionally to the number of objects in the scene, the resolution of the image, the number of samples per pixel, and the ray-tracing depth.

As an example, take a scene with 3 spheres and a disk. Let us render an 80 by 52 pixel image; we will choose to use 32 samples per pixel to get a reasonable image quality. For maximum recursion depth, a value of 5 should be plenty for the simple scene we have – a more complex scene could probably benefit from a few more, but since attenuation reduces the impact each successive ray has on a fragment, it is very unlikely to matter past a few tens. Thus, we have all the information we need: $R = 4160$, $S = 32$, $D = 5$, $N = 4$. We can calculate the final number of basic ray-tracing intersections by simply multiplying these variables together: we get 2662400. Since this kind of image can usually be rendered in around 200 milliseconds by `rayterm-cpu`, we can get a sense of how fast the existing

intersection is already: one test takes about 75 nanoseconds. There's not much hope of drastically improving this time. Even to get render time down to 33 milliseconds, which is needed for 30 frames per second playback, we would need an intersection test to take no more than 12.4 nanoseconds. This is nigh impossible with modern CPUs – for comparison, basic multiplication takes an average 19 nanoseconds on an i7-4770k, a relatively modern processor – the code to calculate this value on any computer can be seen in Appendix A.1. Therefore any further major performance improvement needs to break out of the sequential world this algorithm was written in.

The algorithms used for intersection are already covered in Section 1.3.2; however the scattering algorithms are discussed below. There are three materials supported in `rayterm-cpu`: Lambertian, metallic, and dielectric. The Lambertian material, sometimes known as diffuse, is very simple; the scattered ray originates at the intersection location, and the direction is random in the unit hemisphere oriented about the surface normal of the intersected object. Essentially, when a ray hits a Lambertian object the ray reflects around the tiny crevices and cracks on the surface of the object so that the direction is entirely random once it leaves the object.

$$\vec{R_{reflect}} = \vec{R_{in}} - 2(\vec{S_{normal}} \bullet \vec{R_{in}})\vec{S_{normal}} \tag{3.2}$$

The metallic material scatters rays in a similar way as the Lambertian but includes a bias towards the reflection direction. The reflection direction is calculated using Formula 3.2 [40], where $\vec{R_{reflect}}$ is the reflected ray, $\vec{R_{in}}$ is the incoming ray, and $\vec{S_{normal}}$ is the surface normal. The bias is generated by using a ray direction that is linearly interpolated between the pure reflection direction from Formula 3.2 and a random direction generated as in Lambertian materials. The amount of linear interpolation is known as the "roughness" – a roughness of zero gives perfect reflection (a mirror), whereas a roughness of one gives the same results as a Lambertian material. A slightly more accurate roughness ramp could be attained by using spherical linear interpolation, however, because of the significant performance penalty that was not used.

Finally, the dielectric material is the most complicated. A dielectric material supports both reflection (as in the other materials), and refraction. Since we scatter only one ray at a time, the scatter function uses probability to determine which kind of trajectory will be used. This probability is based on Schlick's approximation of the Fresnel factor, as seen in Formula 3.4 [43, 8, 40]. In this approximation, θ is the angle between the incoming ray direction and the surface normal; n_1 and n_2 are the indices of refraction of the object and air – these are reversed if the ray is leaving the object. Finally, R_0 is the value of the Fresnel term when there is minimal reflection *i.e.* when $\theta = 0$.

$$R_0 = (\frac{n_1 - n_2}{n_1 + n_2})^2 \tag{3.3}$$

$$R(\theta) = R_0 + (1 - R_0)(1 - cos\theta)^5 \tag{3.4}$$

This approximation gives values close to 0 when the view angle is high, for example when looking directly through an object; it gives values close to 1 when the view angle is low, for example on the oblique sides of a sphere. This can be seen in Figure 3.6: notice that the gradient is not linear and the white is mostly concentrated on the oblique edges of the sphere. The approximation is then used to determine if the ray will be refracted (low values) or reflected (high values) by testing against a random scalar in the range $[0,1)$. This will result in high viewing angles (black in Figure 3.6) generating refraction rays, and low viewing angles (white in Figure 3.6) generating reflection rays. This is a close approximation of the effect that glass and other dielectrics have when viewed in the real world.

Figure 3.6: Schlick approximation for a sphere (white is 1, black is 0) [8]

Now that the kind of ray has been chosen, the refraction or reflection direction is calculated. There is an additional possibility that the specific ray we have cannot be refracted because of total internal reflection – in this case, we switch over to a reflection ray. The ray direction in the reflection case is calculated with Formula 3.2 as in previous materials. The ray direction in the refraction case is calculated with Formula 3.7. In this formula, $\vec{R_{refract}}$ is the refracted ray; n_{from} is the index of refraction of the medium we are transfering from, and n_{to} is the corresponding index; $\vec{R_{in}}$ is the incoming ray; and $\vec{S_{normal}}$ is the surface normal. There is some additional handling in the implementation of this formula to ensure both ray directions (exiting and entering an object) are supported. This involves flipping the n values and negating I or $\vec{S_{normal}}$. This formula is derived from the excellent "Scratchapixel" [40] and "Raytracing in One Weekend" [44], with some significant rewriting.

$$\eta = \frac{n_{from}}{n_{to}} \tag{3.5}$$

$$I = (\vec{R_{in}} \bullet \vec{S_{normal}}) \tag{3.6}$$

$$\vec{R_{refract}} = \eta \vec{R_{in}} + (\eta I - \sqrt{1 - \eta^2 \cdot (1 - I^2)})\vec{S_{normal}} \tag{3.7}$$

3.1.1.3 Outcome

The sequential renderer is only a research prototype and was never finished. This is primarily because more performance was needed for the final RAYTERM implementation; the parallel renderer fulfills that need. Because of the prototype nature of `rayterm-cpu`, the final implementation only renders images through a Google Test [46] case, which can be run with the `test.sh` script in the implementation repository [30], or with `gradle test`. This generates a `test_image.ppm` file, examples of which we show from various stages of development in Section 4.1.2. Because of this, `rayterm-cpu` is very much a proof-of-concept and learning example, rather than a fully-fledged library.

That is not to say that `rayterm-cpu` was not a useful system. Many of the algorithms implemented carry over to the parallel renderer, and so the development of that version was aided by existing work. An excellent example of this is the random generation in `raymath` and the material scatter functions in `raytrace`; these algorithms are core to raytracing and having a prior implementation simplified further development. Although they may not carry over unmodified, they are the first iteration in a long improvement cycle.

3.1.2 Parallel

The parallel render engine, `rayterm-gpu`, uses a Graphics Processing Unit (GPU) to massively parallelize the raytracing algorithm. This improves performance dramatically because each eye-ray's children (the result of a collision between an object and a ray) are dependant on only their parent eye-ray. Thus, the computation of a single pixel's color is completely independent of every other pixel.

A short introduction to NVIDIA GPUs may be useful here; they consist primarily of small units of computation circuitry called "multiprocessors" which themselves are controllers for several CUDA cores [33], distributing workloads. Multiprocessors work on data and code called "thread blocks." Each individual CUDA core in a multiprocessor executes an individual thread in a thread block. These CUDA cores execute individual threads on the GPU; they can each handle an individual intersection test, and with thousands of them per GPU, we can easily cut the render time down. There are many restrictions to code that can run in a CUDA thread; CUDA cores act somewhat like advanced floating point arithmetic computation units. However, OptiX takes care of much of the device handling and thus can perform some advanced optimizations for the individual cores. A discussion on the specific hardware used for development and testing is available in Section 1.4.2.

With 227 commits and over 1.5 thousand lines of code, along with many thousands of additions and deletions, `rayterm-gpu` has a good start on its life. There are many, many more improvements possible, however. We detail some of them in the context of the larger RAYTERM implementation in Section 5.2. The implementation repository for `rayterm-gpu` [29] is also the same used for RAYTERM as a whole; thus, while we only describe `rayterm-gpu` in this section, our discussion here also applies to the final RAYTERM library. This section

specifically covers the `raytrace` component of the final RAYTERM library; Section 3.2 covers the `rayterm` component. Both of these are combined into a single `rayterm.so` library for use by other applications. An example application is also bundled in the implementation repository [29], as the `rtexplore` component. All of these are discussed in more detail in Section 4.2.

3.1.2.1 Libraries

Utilizing the massively parallel computation power of a GPU can be difficult – in fact, before CUDA's release in 2007, there was no well-defined way to do so programmatically. CUDA is an API and computing platform that enables massively parallel computations [34]. A special extension of C++, known as CUDA C/C++, can be compiled with `nvcc`, a custom LLVM-based C/C++ compiler. This extension enables library support for GPU accelerated multi-threading, physics, and linear algebra, along with many implementations of common data transformation algorithms. A significant limitation, especially for ray-tracing, is that performance can be significantly affected for inherently divergent tasks – tasks which do not consistently follow the same control flow – such as traversing an acceleration structure.

OptiX is an API interface developed for current generation NVIDIA GPUs that enables GPU accelerated ray-object intersection calculations in a programmable graphics pipeline [36, 35]. It does this by building on top of the CUDA platform, adding abstract and algorithm implementations for common ray-tracing tasks. OptiX allows user-defined single-ray programs to be compiled into a single program, called a "kernel," that runs on the GPU. These user programs are written in C++ and then transpiled to PTX, an instruction set for general purpose parallel programming. Other useful algorithms are also included in the OptiX library, such as BVH and similar acceleration structure implementations. The main method of communication between host (CPU) and device (GPU) are `Buffers`, which are simply arrays managed by OptiX. Depending on the type of `Buffer`, data could be moved in one direction or both, depending on the context; an example of this would be the pixel buffer, which is written to on the device and then loaded into CPU-accessible memory for display.

3.1.2.2 Programmability

Both OptiX and RAYTERM itself support programmability in some form. We will define programmability as the possibility of using user-specified code in some operational capacity at certain points in the program's execution. For example, it is possible in OptiX to define a CUDA function that will be executed to test if a ray intersects with an object; this enables part of the capability to render any intersectable object with OptiX. RAYTERM supports material programs that are passed on to OptiX to deal with material scatters; because of this it should one day be possible to use NVIDIA MDL materials [24] in RAYTERM. We utilize programmability in OptiX in a fairly generic way, because much of the imple-

mentation burden for a specific application lies on the application author, and RAYTERM simply provides an interface. Before we describe RAYTERM's approach, let us consider Figure 3.7.

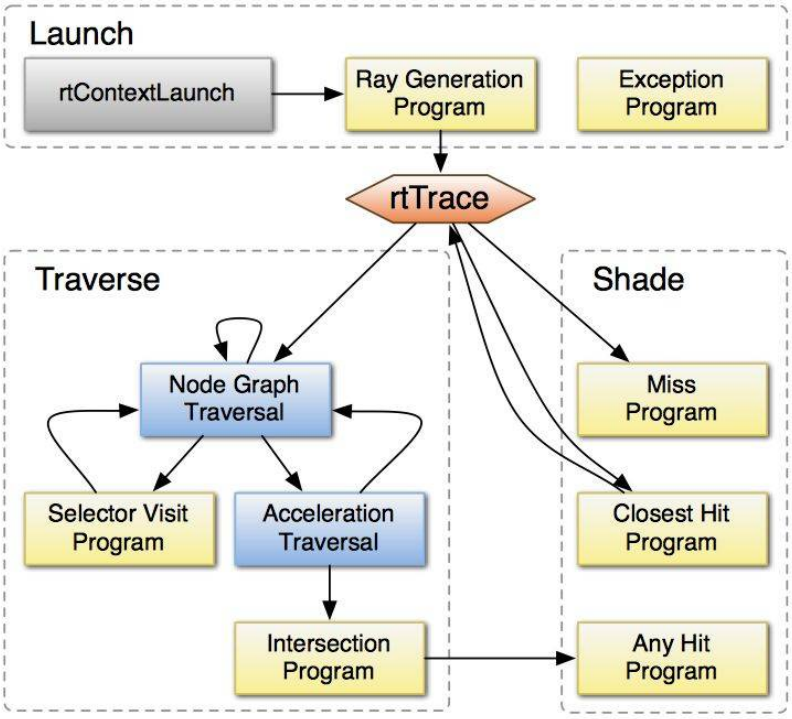


Figure 3.7: OptiX Program call graph

In RAYTERM, we only use part of Figure 3.7 – the "Traverse" section is unused in *RTX* mode, because it enables custom intersections and acceleration structures. In `rayterm-gpu`, only triangle intersections are supported, and acceleration structure calculations are handed off to OptiX's "Trbvh" algorithm [23]. Triangle intersection and acceleration structure builds are highly optimized in OptiX, more so than any user library could achieve. In newer NVIDIA RTX GPUs these constraints also enable real hardware acceleration, as opposed to GPU accelerated computation. In Figure 3.7, yellow boxes are user-defined programs. RAYTERM defines the Ray Generation Program as a pinhole camera model. The Exception Program is also defined, with switches to only enable debug prints and exception handling if RAYTERM is compiled in debug mode. Lastly, RAYTERM defines the Miss Program to shade the background of the scene a smooth sky gradient. The Closest Hit and Any Hit Programs, however, are completely user-defined. These programs control the scattering of rays when they encounter an object. How RAYTERM handles the passthrough configuration of these programs is detailed in the next section.

RAYTERM supports loading triangle models (in the `OBJ` format), material definitions, and CUDA functions (transpiled to PTX) from arbitrary locations on disk. These locations are currently specified as compile-time defines `PROGRAM_DIRECTORY` and `ASSET_DIRECTORY`.

When RAYTERM or a user application requests a CUDA function, material definition, or object model, these directories are used to search for and load the requested data, which is then inventoried and given an id that the user can reference it with. This enables easy asset reuse and configurable definition.

3.1.2.3 Configurability

At the time of writing, RAYTERM's test scene is hard-coded in the `raytrace` component. However, it is trivial to provide a loading mechanism so that user applications can specify the scene and any changes to it. Although the scene itself cannot currently be configured, a wide array of possibilities are available for material and object definitions. Objects are loaded from an `OBJ` model file, which should specify at minimum the vertices and vertex normals of the model. RAYTERM uses TinyObjLoader [12] to load and triangulate `OBJ` files. Then, the basic attributes – vertex positions, normals, shapes, etc. – are stored in a `Mesh` object that is inventoried in the resource system. For materials, a similar process is implemented. First, a custom parser decodes the given material file, which should conform to the format specified below. Then, an OptiX `Material` is created and inventoried. Finally, the resource system is capable of generating an OptiX `GeometryInstance`, which links a material and an OptiX `GeometryTriangles` generated from `Mesh` data. This `GeometryInstance` can then be added to the scene graph and rendered.

```
closest:diffuse,closest_hit,0
any:diffuse,any_hit,0
attrib:diffuse,triangle_attributes
var:float3,varAttenuation,0.15,0.86,0.21
```

Figure 3.8: Sample material definition

In Figure 3.8 a sample material definition is shown. The format of a material definition follows a simple pattern: each line is a single "command," the type of which is determined by the element before the first ':'. There are four types of commands: `closest`, `any`, `attrib`, and `var`. The `closest` command specifies a CUDA function to use for the Closest Hit Program. It has the format `closest:<file_name>,<function_name>,<ray_type>`. The `any` command is similar, except that it specifies the Any Hit Program. The `attrib` command sets the attribute program for the `GeometryTriangles` instance this material is eventually paired with; it operates on all ray types, and thus does not need a specifier. Finally, the `var` command specifies a variable to pass through to the GPU CUDA function; the format is `var:<type>,<name>,<value>`. The possible types are `int` to `int4`, and `float` to `float4`; the values are specified in a comma-separated list. In Figure 3.8, a variable named `varAttenuation` is defined as $(0.15, \vec{0.86}, 0.21)$. This variable is used in the `diffuse` PTX file as the color to attenuate the recursively traced ray by – it is the color of the object this material is applied to. Any number of commands are allowed, and

any later commands will overwrite previous commands if they contain similar information, such as an attrib command. There is not currently a way to programmatically generate a `Mesh` or `Material`, however, these are possible future improvements to RAYTERM.

3.1.2.4 Design

OptiX is heavily utilized in `rayterm-gpu`. Much of the `raytrace` component deals with initializing and configuring the OptiX context correctly. The main class is `Renderer` – it represents a render engine, and has state about the world and everything in it. `Renderer` is designed to be a singleton, although no code in RAYTERM assumes this; rather, each instance requires its own OptiX context. This could be a problem: according to the OptiX Programming Guide, "multiple contexts can be active at one time in limited cases, but this is usually unnecessary." In that quote, "limited cases" really mean that it is untested and not well supported, although technically possible [35]. So depending on future OptiX changes, multi-renderer programs may or may not be a possibility.

A `Renderer` has three main components that operate on the OptiX context associated with the instance. The `Camera` member instance controls the current view of the world and can be used to translate or rotate that view. The `Programs` member instance holds an inventory of loaded OptiX CUDA functions and can be used to load more or get existing ones. Finally, the `Resources` member instance holds an inventory of `Materials` and `Meshs`; it can also load more or associate existing assets together to produce `GeometryInstances`, which could be added to the OptiX scene graph. There are other smaller supporting classes, such as `PixelBuffer`, which helps wrap existing OptiX functionality to ensure safe usage.

3.1.2.5 Algorithms

The parallel renderer uses many of the same algorithms as `rayterm-cpu`, so the previous discussion of those still apply. There are a few new algorithms that are unique to `rayterm-gpu`, however. First, a modification of the pinhole camera model was made. This was inspired by the camera models available in the advanced OptiX samples [27], and simply rewrites the equations used before in terms of ϕ and θ, as polar coordinates. In this new `Camera`, ϕ represents the rotation about the vertical world axis, and θ the rotation about the horizontal axis. This enables easier viewing modification so that the look direction can be controlled with just two control axis. Second, the color output space is transformed into BT.709 [21]; this ensures color accuracy and gamma correctness, as well as being the assumed color space for `ppm` files.

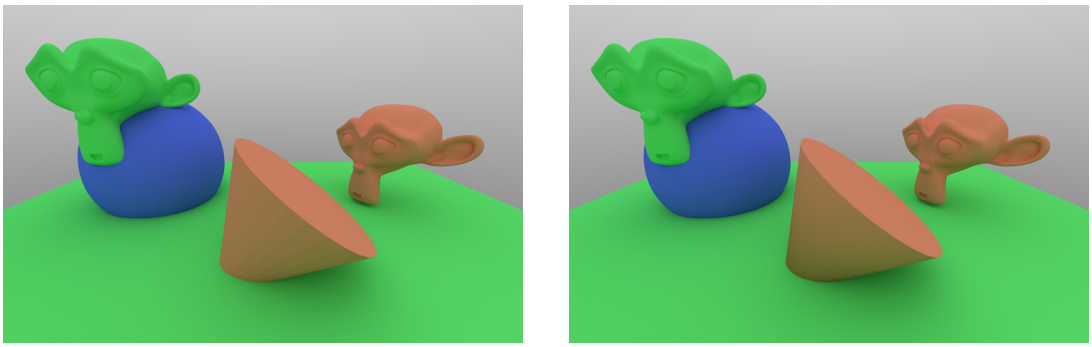
$$L = \begin{cases} \frac{V}{4.5} & V < 0.081 \\ (\frac{V+0.099}{1.099})^{\frac{1}{0.45}} & V \geq 0.081 \end{cases} \tag{3.8}$$

Formula 3.8 shows the piecewise function definition for conversion between a raw color

value V and a luminance component L. This is done for each color component to get the final color for each pixel. Additional algorithms also had to be developed as replacements for CPU-based algorithms in CUDA functions. For example, a `drand48` implementation is provided for CUDA functions to use, since the existing implementation is CPU-only. The implementation directly mirrors the UNIX specification for `drand48` [14]. Additionally, the sampling algorithms based off of `drand48` were replaced with analytical solutions, to prevent sample looping and provide better uniform distribution. The math behind the analytical solution is based on resources from "Scratchapixel" as well as an older paper by the same author as "Raytracing in One Weekend" [38, 45]. We also use an OptiX provided sampler, `cosine_sample_hemisphere`, to generate the initial sample; the further analytical solution simply transforms that sample to the correct normal space.

$$N_{interpolated} = B_x\vec{N_1} + B_y\vec{N_2} + (1 - B_x - B_y)\vec{N_0} \tag{3.9}$$

Finally, we use an algorithm to calculate the interpolated shading normal. This is very loosely derived from the OptiX samples [27], and uses the OptiX default `GeometryTriangles` attribute program to calculate the barycentric coordinates of the intersection point. Formula 3.9 shows the calculation for obtaining the interpolated normal; this result is normalized and can then be used for reflection or refraction calculations. An interpolated value is extremely useful because it takes into account the normals of each individual vertex and attempts to "smooth" the shape approximated by the triangle, thus making shading more accurate despite the physical differences. In Formula 3.9, N_i is the vertex normal of the i-th vertex in the triangle, and B is the two dimensional barycentric coordinate of the intersection point. The calculated result, $N_{interpolated}$, is the smoothed normal. This smoothing can be disabled by using an `OBJ` model that has had its normals exported as "flag" – this causes each triangle's vertices to have their normals all match the face normal, instead of having individual normals. This is useful for flat faces since the interpolated normal is then simply the same as the face normal. Figure 3.9 shows a comparison between smoothed normals and face normals (sometimes called "geometric normals"). Larger versions of these images are available in Appendix B as Figure B.2



(a) Geometric normals (unsmoothed) (b) Interpolated normals (smoothed)

Figure 3.9: Comparison of geometric and interpolated normals

3.1.2.6 Outcome

As part of a larger system and due to reduced available development time, `rayterm-gpu` has a relatively sparse unit test suite. Unlike `rayterm-cpu`, it is tested primarily through integration and comparison tests. However, `rayterm-gpu` does meet its goal of increasing performance, so much so that with a GTX 980 Ti GPU, the average render time for a dense scene with tens of thousands of triangles is under 30 milliseconds. Of course, this is still at a low resolution with only 64 spp; however, this is a vast improvement over the CPU research prototype, which would probably take upwards of an hour on the same scene. Much of this is due to the massively parallel architecture of OptiX and CUDA.

Just as with `rayterm-cpu`, a `ppm` test image can be generated through a Google Test [46] case, which can be run with the `test.sh` script in the implementation repository [30], or with `gradle test`. This generates a `test_image.ppm` file, examples of which we show from various stages of development in Section 4.2.1. However, because `rayterm-gpu` integrates with the larger RAYTERM system, rendered images can also be seen with the `rtexplore` sample application, which we discuss in more detail in Section 3.2.2.

3.2 Terminal Interface

The RAYTERM library depends on two main components: the render engine, and the terminal interface. This terminal interface uses the render engine to generate images for display in the terminal; it also converts those images from raw pixel colors to unicode characters. This interface is defined in the actual library, which we discuss in Section 3.2.1, as well as the user application. We provide and discuss a sample user application in Section 3.2.2.

3.2.1 RayTerm

RAYTERM is distinct from `rayterm` – RAYTERM is the entire system, and the name of the project, whereas `rayterm` is the name of the terminal interface component in the larger implementation. The interface depends on the `raytrace` component discussed in Section 3.1.2 to provide a render engine. Another dependency is `libtickit` [9], a library that supports ANSI escape code detection and subsequent usage. Because of the relative obscurity of Tickit, we adapted the main codebase to our needs [32]; some of these fixes and changes have made their way upstream.

Because of the `libtickit` usage, RAYTERM works best in a real XTerm terminal; color output should additionally be enabled by setting the `COLORTERM` environment variable to `truecolor`. These settings are only supported on relatively recent versions of XTerm. More information of direct color compatibility and the relevant ANSI escape codes can be found in Anton Kochkov's incredibly useful GitHub gist "TrueColour: Colours in terminal" [25].

3.2.1.1 Design

There are only two important structures in `rayterm`: the `UnicodeBuffer` and the `Terminal`. The `UnicodeBuffer` class is a direct analog to `PixelBuffer`, with the `translate_halfpixel` function handling translations, as we discuss in Section 3.2.1.2. The `Terminal` class represents an actual terminal application, with the ability to render a frame, set the information string, resize the terminal, etc. The `rayterm` component is actually extremely simple, with just under three hundred lines of code. Much of the work is handed off to `libtickit`, which handles the actual character rendering. In user applications we suggest using `libtickit` for event loops and other input/output handling, as integrating a different utility may prove difficult given the dependencies of RAYTERM.

3.2.1.2 Algorithms

There is only one real algorithm implemented in `rayterm`. This is the character translation algorithm that converts a `PixelBuffer` from the `raytrace` component to a `UnicodeBuffer`. This algorithm, called `translate_halfpixel`, has a straightforward implementation: it simply iterates over the pixels in the `PixelBuffer` two rows at a time, converting the values to `unicode_cell` structs. This two-row iteration is necessary because each unicode character is actually two pixels; the foreground is the "upper" pixel, while the background represents the "lower" pixel. The `unicode_cell` struct thus holds both a foreground and background color.

This translation algorithm is used to translate each frame's `PixelBuffer` before rendering the screen; at the time of writing, since only `translate_halfpixel` is supported, all characters drawn to the screen are the unicode upper half block character `U+2580` (see Figure 1.3a for similar example characters). This causes the foreground color of a `unicode_cell` to apply to the character portion of the character cell, and the background color to apply to the empty portion of that same cell. Through trial and error, we've found that varying the font size through odd values gives the best pixel-like results; even pt sizes can sometimes cause gaps which confuse the viewer.

3.2.1.3 Outcome

While developing the terminal interface was a long and iterative process, the result is a simple and straightforward interface available for use by consumers of the RAYTERM library. As with `rayterm-gpu`, due to time constraints `rayterm` has a sparse unit test suite; however, its simplicity and reliance on `libtickit` (which is extremely well tested) lowers the importance of such testing. For usage, user applications should at minimum simply allocate a new `Terminal` with the `new` keyword. Then, the Tickit `run` or `tick` routines should be called to handle re-exposure; otherwise, the terminal will not receive render updates. A new frame can be drawn (perhaps after modification of the scene graph or camera through

the `Renderer` available in `Terminal`) with the `renderFrame` member function. With this extremely simple interface, complex applications can be designed.

3.2.2 RTExplore Sample

Within the RAYTERM implementation repository [29] we provide `rtexplore`, a simple terminal-based real-time ray-traced viewport into the test world of `raytrace`. This sample acts as both an integration test and proof of concept for other users of RAYTERM. When running this sample, or indeed any RAYTERM consumer application, five shared libraries must be linkable; these are listed below. Depending on how `libtickit` was compiled into RAYTERM, `libtickit.so` may also need to be available. A video demonstration of `rtexplore` is available as well [31].

- `libtermkey.so`
- `libunibilium.so`
- `libcudart.so`
- `liboptix.so`
- `librayterm.so`

The `rtexplore` sample uses a custom Tickit event loop to control input/output, as well as key and mouse event listeners. The key listener detects key presses in the terminal from the user, and modifies the `Camera` used for rendering by small amounts; The A, W, S, and D keys control camera position, while the arrow keys control orientation. The mouse listener is not used for input, but the caught events are still displayed in the info string, as are past key events. Thus, most major features of RAYTERM are utilized by `rtexplore`. Of course, this application has huge room for improvement, as does the rest of RAYTERM. Some options for future work are provided in Section 5.2.

Chapter 4

Implementation

In this chapter we discuss the implementation journey, and all the challenges and solutions we encountered. Because much of the development effort for RAYTERM was focused on ray-tracing, many of the topics here are also related to ray-tracing.

4.1 CPU Prototype

The CPU research prototype was the most challenging component to implement. This was likely because much of the research was difficult, and for every implementation effort afterward, we were able to reference the prototype. Some of the bigger challenges involved accurately replicating physical phenomenon, since the mathematics and physics behind them are often extremely complicated, and physics textbooks take a long time to read. The below sections chronicle the implementation journey, with various figures at milestones along the way, illustrating what the `ppm` output looked like at the time.

4.1.1 `raymath`

The `raymath` component was really straightforward to implement, and there are no significant challenges. This was also an introduction to the Eigen library [15] and C++ in general. One big decision made in `raymath` was to `typedef` certain Eigen template types to ensure that the rest of `rayterm-cpu` code used only types such as `vector` or `ray`, instead of the more complicated Eigen equivelents `Eigen::Matrix<scalar, 3, 1>` and `Eigen::ParametrizedLine<scalar, 3>`. A `scalar` type was also defined, in theory for the purpose of changing from `double` to `float` calculations if desired. However there are some dependencies in `raytrace` on `double` calculations, so changing that now is likely to introduce problems.

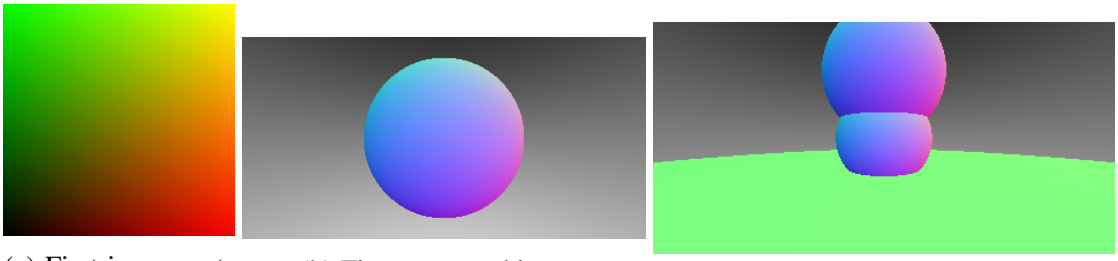
One of the only challenges in `raymath` implementation was dealing with memory alignment. Any structure that uses Eigen structs needs to be correctly aligned, and the error

messages for discovering this are not easily interpretable. Thus, it was only after some research that the `EIGEN_MAKE_ALIGNED_OPERATOR_NEW` macro was discovered and put to use. The `raymath` component is also one of the better-tested modules, with tests to ensure the mathematics of intersection work for all situations.

4.1.2 `raytrace`

The `raytrace` component had the most development time put into it throughout the entire RAYTERM development process. It started off as an extremely simple Monte Carlo path-tracer and graduated to increasingly more complex materials, algorithms, and implementations. Here we will describe some of the steps through this process.

The first image output ever produced is shown in Figure 4.1a. This tested the `ppm` writing code, and not much else was implemented beyond that. Next, sphere intersection was implemented as seen in Figure 4.1b. This first ray-traced image was colored by showing the normal vector at the intersection point, shifted into the RGB color range; the x component to the red component, the y to the green, etc. Finally the first iteration of `World` was implemented, as shown in Figure 4.1c. Here, multiple spheres are able to be intersected, and each ray is tested against each object in the scene. This uses the same normal visualization as before.



(a) First image output

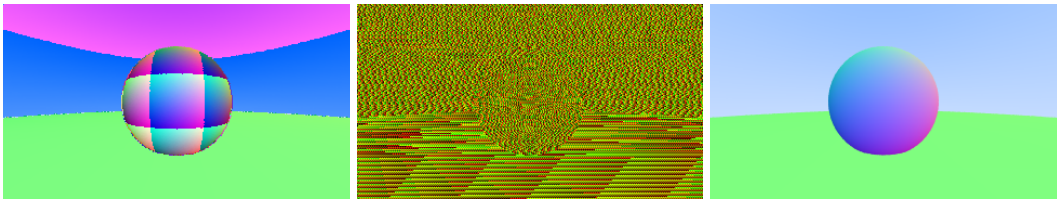
(b) First raytraced image

(c) First multi-object image

Figure 4.1: Ray-tracing firsts

The next steps were to enable multiple samples per pixel. Figure 4.2 shows some of the difficulties we encountered. The biggest challenge here was ensuring that the ray direction was correct; during this time `Camera` was rewritten into the final version, with much better robustness for situations like this. In `rayterm-cpu` we use random subpixel jitter for every sample – this was what caused the majority of the issues, and took some time to work out.

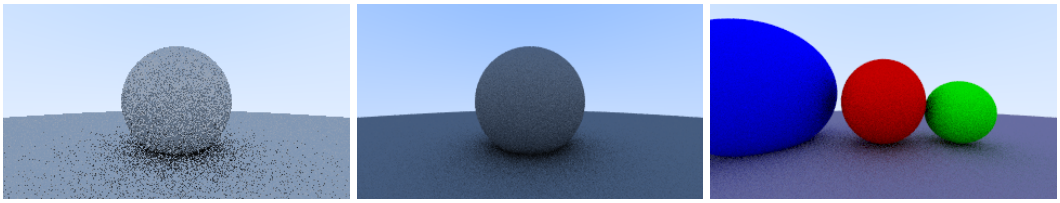
The first material development used a different scattering function than currently; the development sequence is shown in Figure 4.3. In the beginning, Lambertian scatter was achieved through picking a random vector in a sphere and offsetting that vector by the normal. This provides a good approximation, but later versions replaced this with a non-uniform hemispherical sampling algorithm, which generates a vector in the entire visible hemisphere



(a) Difficulties (b) More difficulties (c) Finally working

Figure 4.2: Multiple samples per pixel

centered on the normal from the intersection point. There is an existing bug with this version, however: the sampling is not uniform. There is a concentration around the normal vector due to the random sampling method we used. This is fixed in `rayterm-gpu` – you can see the difference in Figure 4.13.



(a) First Lambertian (b) Refined Lambertian (c) Attenuated Lambertian

Figure 4.3: Lambertian Development

Figure 4.4 was the first high-quality render attempted, and it actually revealed a subtle bug not seen before. This render took 141 seconds and had an spp of 1600. The bug revealed is easily visible at the intersection between the small and large spheres – there is a "ledge" or bright spot where it should be completely occluded. This was a result of too high of a limit being placed on the scattered rays; rays that had an intersection less than 0.001 were discarded. However, these rays would have darkened the crevice between the two spheres. They essentially bounced back and forth between the spheres, attenuating at each bounce, but were stopped too soon while the distance they traveled was still visible. This was easily fixed by decreasing the limit to 0.00001.

Now that the ray-tracing basics were taken care of, we moved on to more diverse materials. Figure 4.5 is a test scene with various metallic or glossy materials, along with some Lambertian. The only real difference between the two is that glossy materials have a bias towards the reflection direction when scattering rays. That bias is controlled by the roughness – the large sphere to the right has a very low roughness, while the closer small red sphere's roughness is slightly larger. At lower values, the roughness controls the blurriness of reflections; at larger values, it controls how "diffuse" the object looks. Again in the first implementation, the math behind the scattering was similar to the Lambertian mathematics, except instead of a sphere offset by the normal, the random vectors were generated in a sphere with radius equal to the roughness value, and offset by the normalized reflection

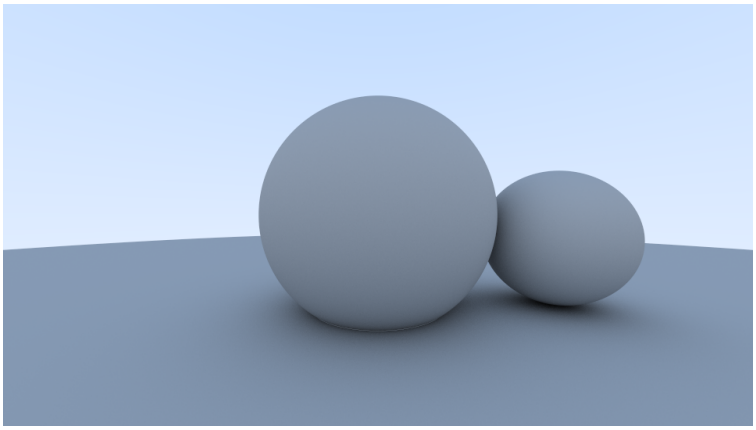


Figure 4.4: Bugged intersection cutoff

direction. This also provides a workable approximation but was soon replaced, just as the Lambertian math was. The replacement is considerably better; we sample a hemisphere centered on the normal, and then linearly interpolate the reflection direction towards that random vector based on the roughness. Thus, a roughness of one causes metallic materials to be mathematically identical to Lambertian. This method is discussed in more detail in Section 3.1.1.2.

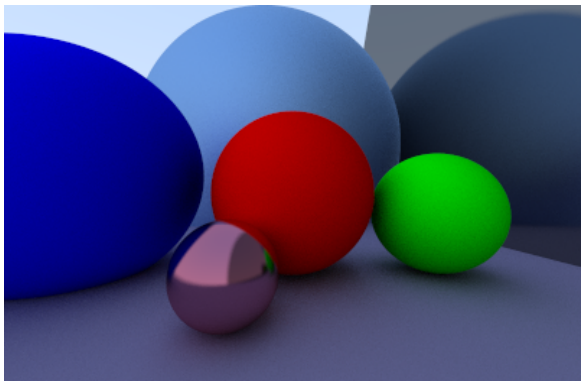
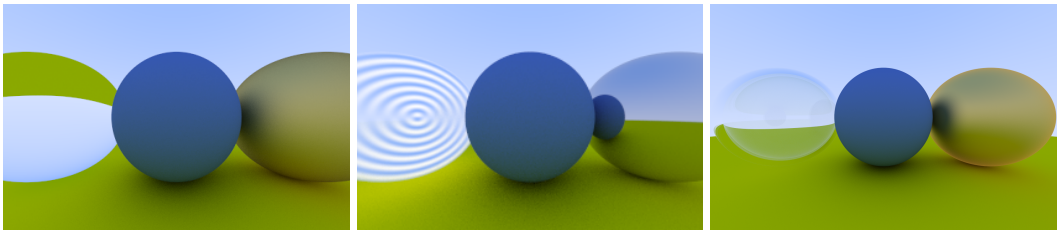


Figure 4.5: Metallic materials

Next on the list of material types to implement were dielectrics. These cover a wide range of object looks, all involving both reflection and refraction; common examples are water and glass. The development cycle along with some problems encountered are depicted in Figure 4.6. One huge issue was that once reflection and refraction were working correctly, we discovered a bug in metallic materials that would appear whenever we corrected glass bugs. This turned out to have been caused by the poor approximation in the metallic scatter function, but it took a significant amount of effort to find that. This bug is discussed later on as well.

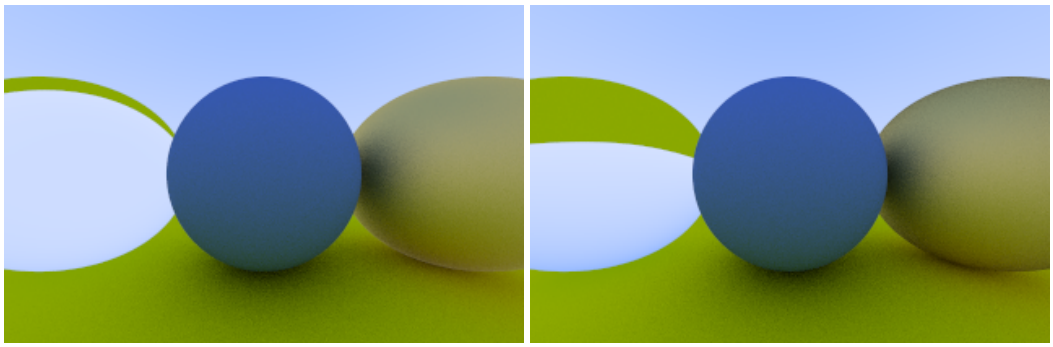
Figure 4.7 shows a bug that appeared after first implementing dielectrics, where metals with a high roughness value were seemingly not shading correctly. Another manifestation



(a) Dielectric Refraction only (b) Fresnel difficulties (c) Dielectric "bubble"

Figure 4.6: Dielectric Development

was found where glass materials did not correctly internally refract/reflect. This bug was solved by adding tests to check for internal intersection capability since that was a likely suspect. That turned out to be the issue; we then added implementation to pass those tests by searching for the smallest non-negative t when testing intersections. Before this change, the t found for intersections was always the result of $\frac{-b-\sqrt{discriminant}}{2a}$; now we consider $\frac{-b\pm\sqrt{discriminant}}{2a}$. This bug fix also exposed the metallic material halo bug, which we talk about next.

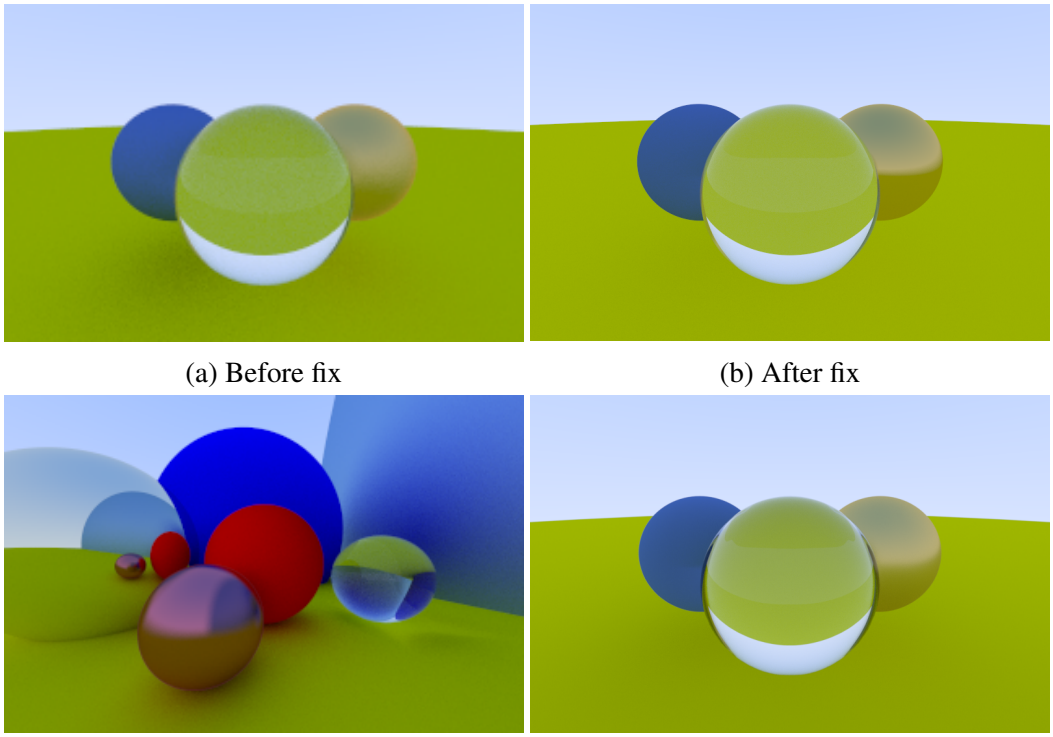


(a) Before fix (b) After fix

Figure 4.7: Smallest t intersection bug

The most challenging and complicated bug we encountered during `rayterm-cpu` development was the metal halo bug; Figure 4.8 shows various informative outputs from the debugging effort. It manifested as a halo of lighter color around the edges of metallic spheres with high roughness values. This turned out to have been caused by the old sphere mathematics in the metallic scatter function. It manifested the way it did because when roughness values were high, the generated rays sometimes intersected the surface of the sphere and were not discarded; instead, the lighter color that was not visible to the camera leaked over the edges. This was a fascinating bug to tackle, and resulted in a lot of learning experiences. We additionally did the first comparison test between a Blender [10] scene and our own. This test showed that once the solution was implemented, our metallic material is very similar to Blender's. We also learned from this comparison that the dielectric material is not quite physically accurate – notice that there is a darker edge around the glass ball in

the Blender reference image, which is much smaller in the `rayterm-cpu` generated image. This could be caused by offsets in Blender's ray-tracing or inaccuracies in the refraction mathematics, however, it is probably the Schlick approximation [43] that is to blame.



(a) Before fix

(b) After fix

(c) Glass material and metal halo bug

(d) Blender reference image

Figure 4.8: Metallic halo bug

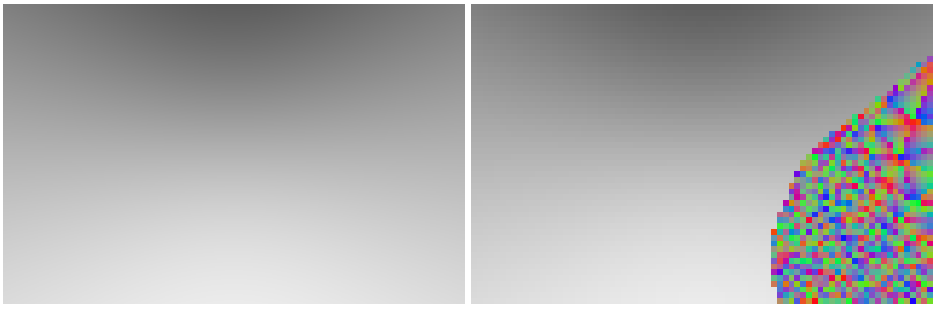
4.2 Final Implementation

The challenges faced during `rayterm-gpu` and later `rayterm` implementation differ substantially from the prior challenges. First, since we are now using GPU computation, continuous integration was difficult to set up, and ultimately does not run unit tests or compile CUDA code; instead, only the C++ is compiled and linted. This is still useful, but not as much as it could be if remote GPU testing was a possibility. The next biggest challenge `rayterm-gpu` faced was the amount and size of the dependencies needed. CUDA alone is almost two gigabytes, and OptiX adds another large chunk. This challenge was fairly enjoyable to tackle, however, and with submodules and git repositories set up for remote cloning, dependency management was fairly easy. Related to that challenge was integration with Gradle [18], which is always complicated when dealing with the Software Model. Again, however, trial and error proved enough, and the build script available in the RAYTERM implementation repository [29] is extremely full-featured and supports a

number of special tasks. Combined with custom shell scripts to run specific tests or applications, and the development environment was a joy to work with.

4.2.1 `raytrace`

The `raytrace` component was the main focus of the initial development efforts in `rayterm-gpu`. It started off as a less featureful clone of `rayterm-cpu`, and as time went on, was optimized more and more for the specific GPU application it was in. In this section, we will describe some of the big milestones encountered throughout the development process. As always, the first few days of development were filled with a lot of learning, especially on the OptiX library and its various quirks. Figure 4.9 shows the first ray-traced image and the first loaded mesh. This was the first hint of the oddities in OptiX; the normal x and z axis were switched: x was now forward and backward, whereas z was left and right. It is likely that this is due to some particularity with the pinhole camera implementation but actually works just fine for most operations.



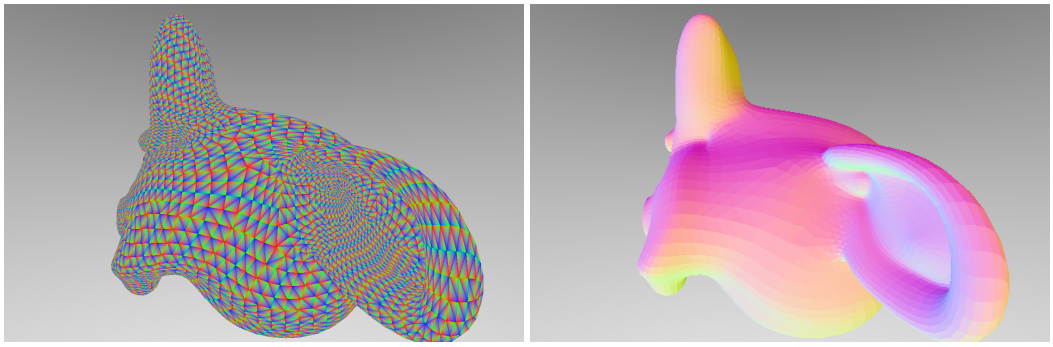
(a) First sky image (b) First mesh render

Figure 4.9: GPU rendering firsts

In Figure 4.10 we can see the first mesh imported. 4.10a displays the barycentric coordinates in much the same way as normals were displayed for `rayterm-cpu`, and 4.10b displays the actual geometric normals. Already there is a massive improvement in performance – these images render in less than a second and contain over fifteen thousand triangles. That would probably cripple `rayterm-cpu`.

After the initial development to enable loading `OBJ` files, additional configuration was easy. In Figure 4.11 the first iteration of the test scene can be seen. Each instance of the same model is created with an OptiX `Matrix4x4` dictating orientation and position; this makes setting up a simple scene extremely easy.

Next, we tackled Lambertian materials. This was incredibly simple because we already had working algorithms. The only development required was to translate the random generation functions. This was an adventure in and of itself since a direct translation was not actually possible; see Section 3.1.2.5 for more. Figure 4.12a was the first test render. At first the



(a) Triangle barycentric coordinates (b) After fix

Figure 4.10: Triangle geometric normals

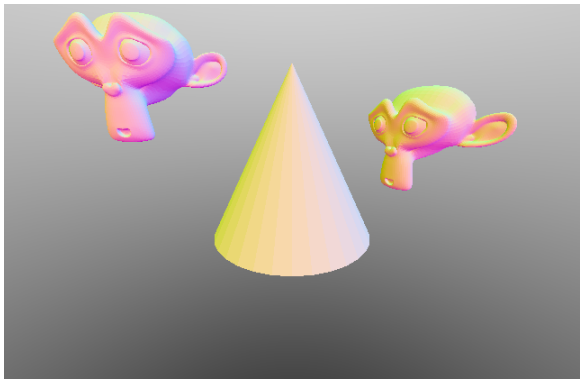
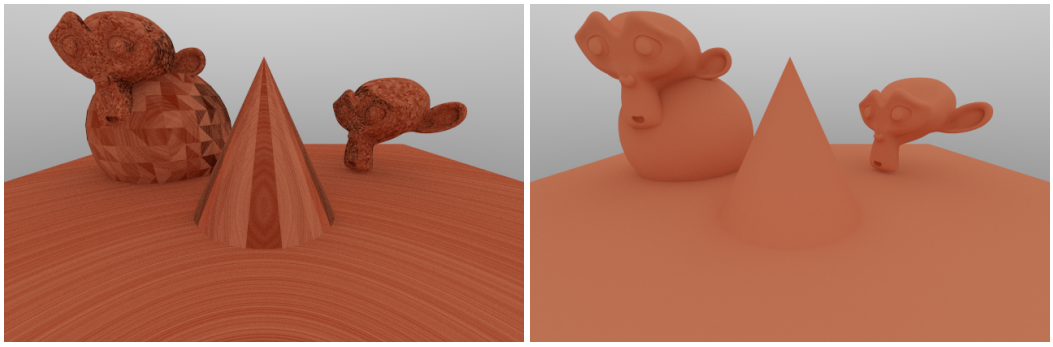


Figure 4.11: Test scene creation

artifacts were quite confusing; however, after a little debugging it was evident that rays were somehow colliding with their origin triangle. To fix this, a lower limit of 0.00001 was set on the ray intersection calculation – any intersection distance smaller than that limit was ignored. This cleaned up the image nicely, generating Figure 4.12b for an almost complete test scene.

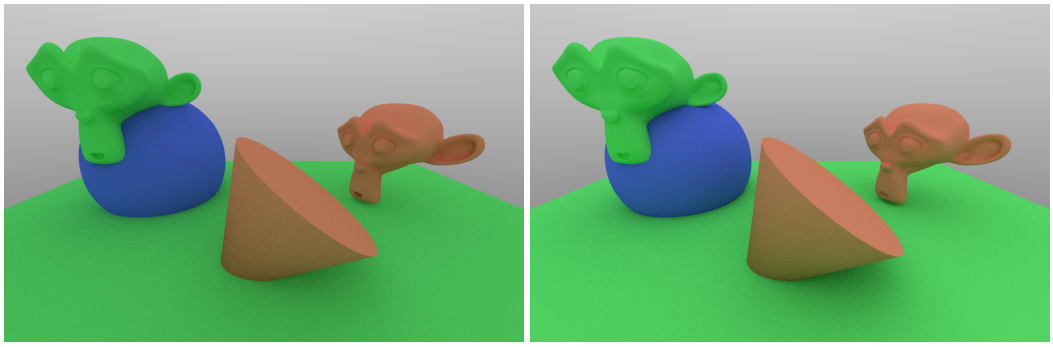
In between Figure 4.12 and Figure 4.13, attenuation was implemented. Now, a few improvements and optimizations were in order. First, we threw out the older semi-direct translation of `random_in_uhemisphere` and wrote a new one that used cosine hemispherical sampling to uniformly sample the possible ray directions. Surprisingly, this resulted in a noticeable improvement to realism; a before and after comparison is available in Figure 4.13, with larger versions in Appendix B as Figure B.3. The main difference is an overall lightening and increased contrast of shadows and ambient occlusion.

The last optimization attempt was to implement stratified sampling. Although this was not successful, mostly because of complications with the current architecture, it did produce some wonderfully weird renders. These can be seen in Figure 4.14. The attempt here is to essentially break the `drand48` return values into "stratas", and pick a random value in each strata for every `drand48` call. However, these strata were advanced linearly, so the



(a) First Lambertian shading render (b) After adding lower limit

Figure 4.12: First Lambertian material render



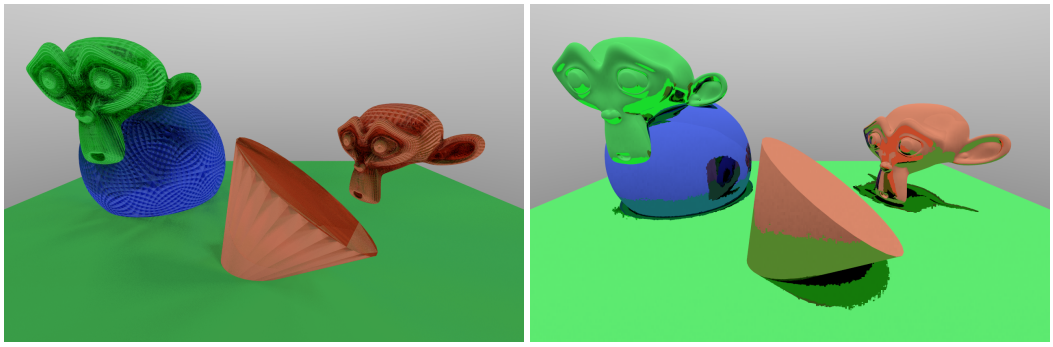
(a) Before uniform hemispherical sampling (b) After uniform hemispherical sampling

Figure 4.13: Uniform hemispherical sampling

first random value would always be in the first strata, and so on. At low stratification levels (meaning only 10 or so stratas), Figure 4.14 is produced. An even more fascinating render, in Figure 4.14b, is produced when the number of stratas grows larger than a few hundred. We theorize that this is due to the `drand48` function's randomness breaking down past a certain level. The artifacts are reminiscent of bad compression, which may give some hint as to their source. Regardless of the outcome, this was the last development effort in `raytrace`; after this, the focus was shifted to `rayterm`.

4.2.2 `rayterm`

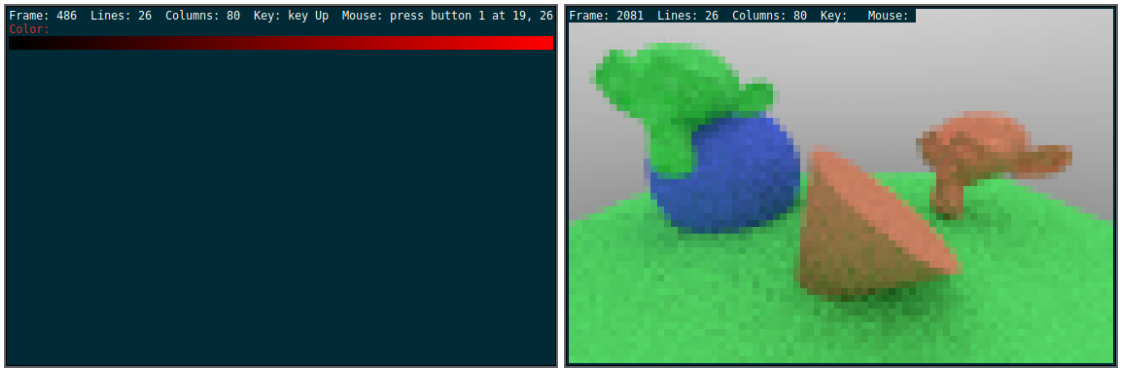
The `rayterm` component is fairly simple in design. It contains a `Terminal` and a few other utility methods and algorithms draw a `PixelBuffer` to the screen. The main development effort when working on `rayterm` was actually updating the `libtickit` library [9, 32] to work correctly with RGB direct color. This took a surprising amount of development for relatively few changes, although there is now a GitHub repository with over 70 commits, Travis CI, and testing enabled; ensuring confidence in the Tickit version in use by RAYTERM. Figure 4.15a is an example of one of the tests that were conducted to ensure



(a) First stratified sampling attempt (b) Bit compression at high stratification

Figure 4.14: Stratified rendering attempts

`libtickit` was working as expected. Beyond that development, `rayterm` came together with very few challenges.



(a) `rayterm` test output (b) `rtexplore` terminal output

Figure 4.15: Terminal output

4.2.3 `rtexplore`

The `rtexplore` user application is the face of RAYTERM. It is not, however, the actual product of this work. Developed in just a few hours with the aim of showing the capabilities of RAYTERM, it is heavily based on Tickit examples [9]. There are many future improvements possible to the program, however. With some additional configurability upgrades to `rayterm`, a much better user experience could be achieved. Additionally, more documentation and other sample applications would probably be useful to potential users of RAYTERM. The current state of `rtexplore` is shown in Figure 4.15b. This can easily be replicated by cloning the RAYTERM implementation repository [29] and running the `run.sh` script after setting up dependencies.

Chapter 5

Discussion and Future Work

5.1 Summary of Results

Over the many years of innovation in the field of computer graphics, advances in rendering have led to massive increases in the fidelity of engaging, satisfying, and realistic computer visualizations. With its initial development complete, RAYTERM is a new and unique entry into the ranks of rendering engines. It is reminiscent of retro aesthetics of the seventies and eighties but also uses new technology like ray-tracing to deliver compelling possibilities. RAYTERM can now be used as an engine for terminal-based 3D tools, visualizations, games, and more. The current implementation has been completely open-sourced under the AGPL v3 license, and any contributions are welcome.

Throughout the development of RAYTERM new possibilities were entertained, sometimes discarded, but ultimately kept and distilled to the immature product we have now. In its current form, RAYTERM could easily be used to implement a terminal viewer for model files, arguably an incredibly useful tool that does not yet exist. As RAYTERM is designed in a simple, open, and accessible format, we hope that future improvements will make RAYTERM even better, more configurable, and extensible.

5.2 Future Work

There is a lot of room for improvement in RAYTERM; it is, after all, an extremely early version of an ambitious idea. One major improvement possible is the creation of a scene description language. Currently, the scene must be set programmatically; if the same flexibility was given to scenes as is materials, user application development would be even easier. Another equally useful improvement would be support for "Material Definition Language" [24] materials. This could involve a large reworking of the asset loading system but should provide an easy development process that does not require any CUDA programming knowledge to create high-quality ray-traced scenes in a terminal.

Numerous other improvements are possible, such as environment mapping, explicit lighting, motion blur, triangle animation (which is actually somewhat supported), and much more. All of these improvements are possible with future collaborators on the open-sourced RAYTERM implementation.

5.3 Conclusion

RAYTERM is a complex system, the implementation and design of which is an ongoing process. The system currently supports real-time 3D scene viewing through a terminal emulator. This is made possible through the OptiX library allowing ray-surface intersection calculations to be done a highly parallelized manner on a GPU. The Tickit interface and its half-pixel translation algorithm, along with `libtickit` itself, enable RAYTERM to treat a terminal as an image canvas. RAYTERM is now a fully real-time capable ray-tracing engine, rendering 30 or more unicode character images per second to a terminal window on a moderately powerful computer. Since the initial startup development of RAYTERM is complete, all contributions to RAYTERM development are now welcome. In pursuit of maintaining good open-source readability and ease of development, documentation has been and will continue to be a priority throughout the transition process. We hope that future projects find RAYTERM inspiring and useful, whether as an engine or as a jumping-off point for real-time ray-tracing and terminal-based applications.

Appendix A

Code

This appendix lists some miscellaneous code snippets. However, the main body of work for RAYTERM is located in two GitHub repsitories, listed in the bibliography as [30] and [29].

A.1 Time Multiplication Operations

```
#include <chrono>
#include <cstdio>
#include <cstdlib>
#include <string>

using namespace std;
using namespace std::chrono;

int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: rtime <N> <WARMUP>\n");
        exit(1);
    }
    int n      = stoi(argv[1]);
    int warmup = stoi(argv[2]);

    steady_clock::time_point begin, end;
    int64_t times[n + warmup];

    for (int i = 0; i < n + warmup; i++) {
        if (i >= warmup) {
            printf("Start run %d\n", i);
        }
        begin    = steady_clock::now();
```

```
        float y = 1251.612 * 212.541;
        end     = steady_clock::now();
        if (i > warmup - 1) {
            printf("End run %d, ", i);
        }

        times[i] = duration_cast<std::chrono::nanoseconds>(end - begin).count();
        if (i >= warmup) {
            printf("time: %ldns\n", times[i]);
        }
    }

    int64_t average_time = 0;
    for (int i = warmup; i < n + warmup; i++) {
        average_time += times[i];
    }
    average_time /= n;

    printf("\nSUMMARY\n\nMultiplied %d floats in %ldns on average\n", n, average_time);
    exit(0);
}
```

Appendix B

Larger Figures

This appendix shows larger images for some figures.

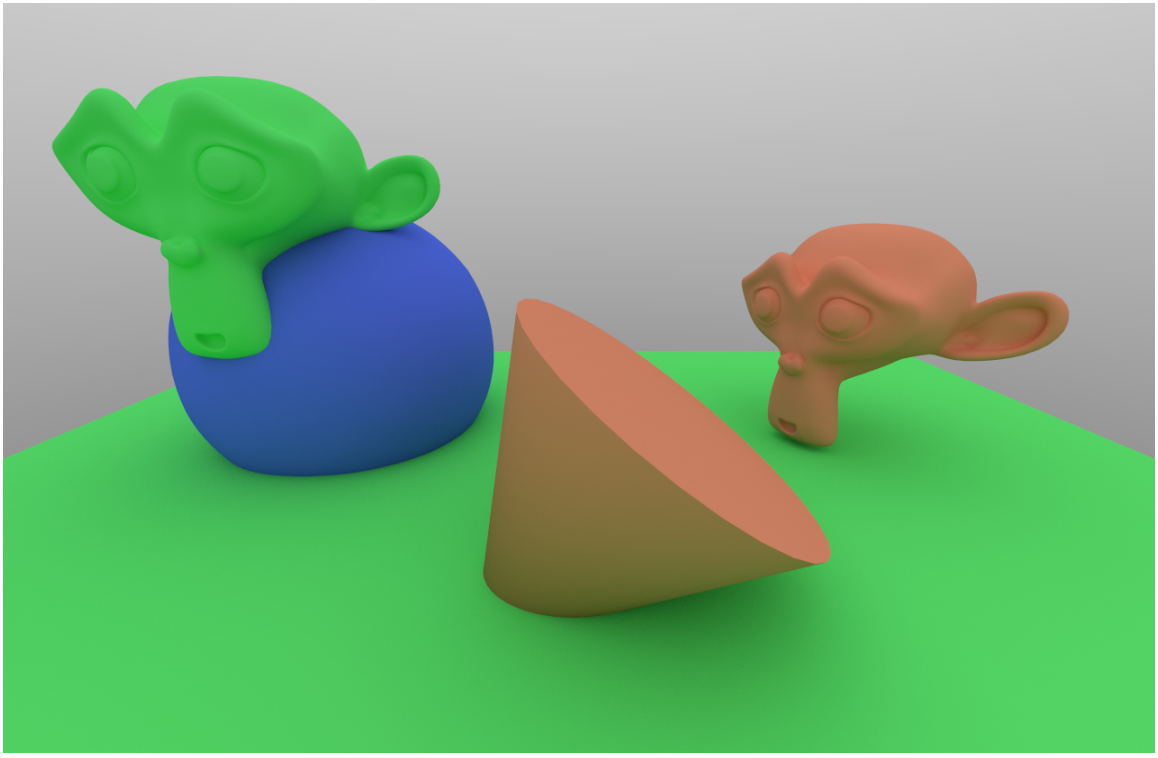
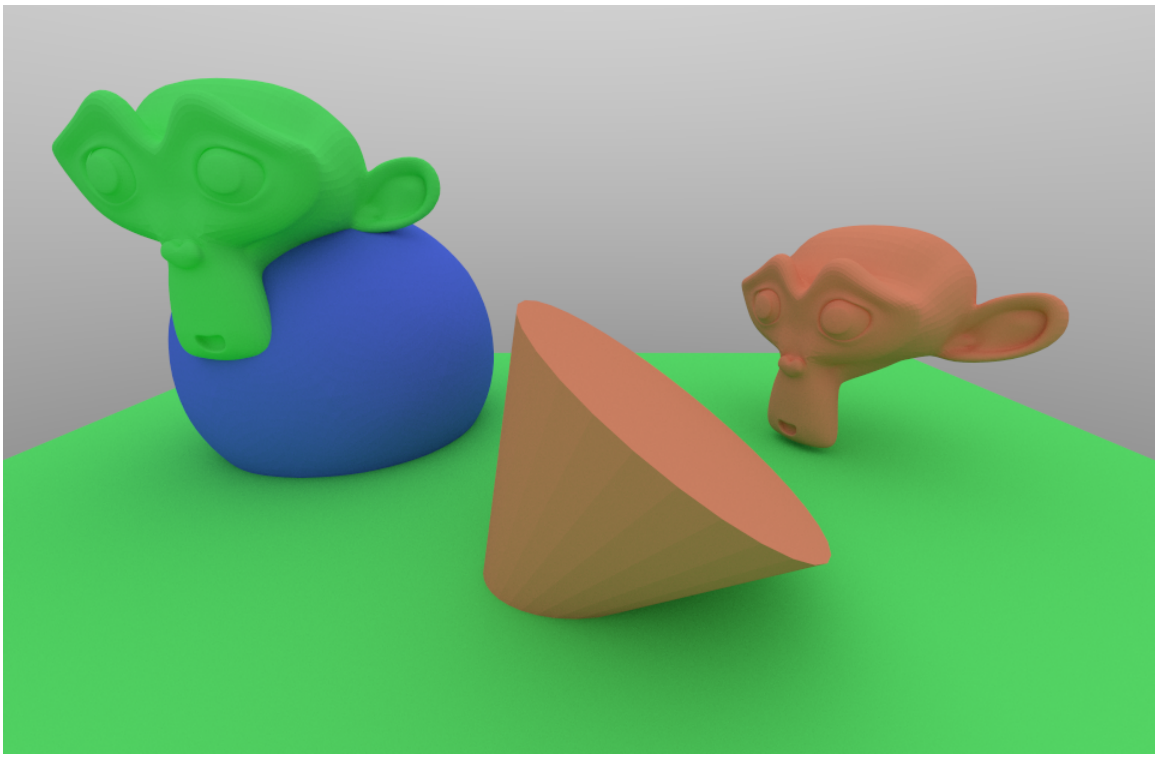
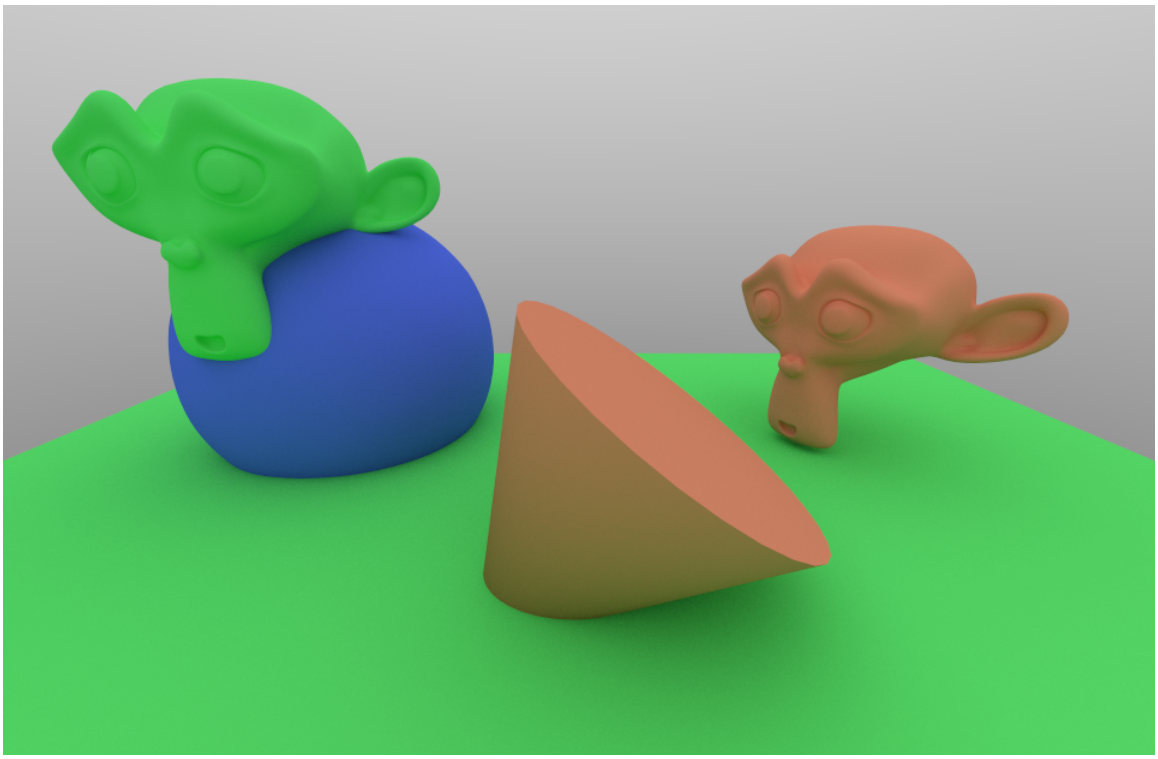


Figure B.1: Final test scene render

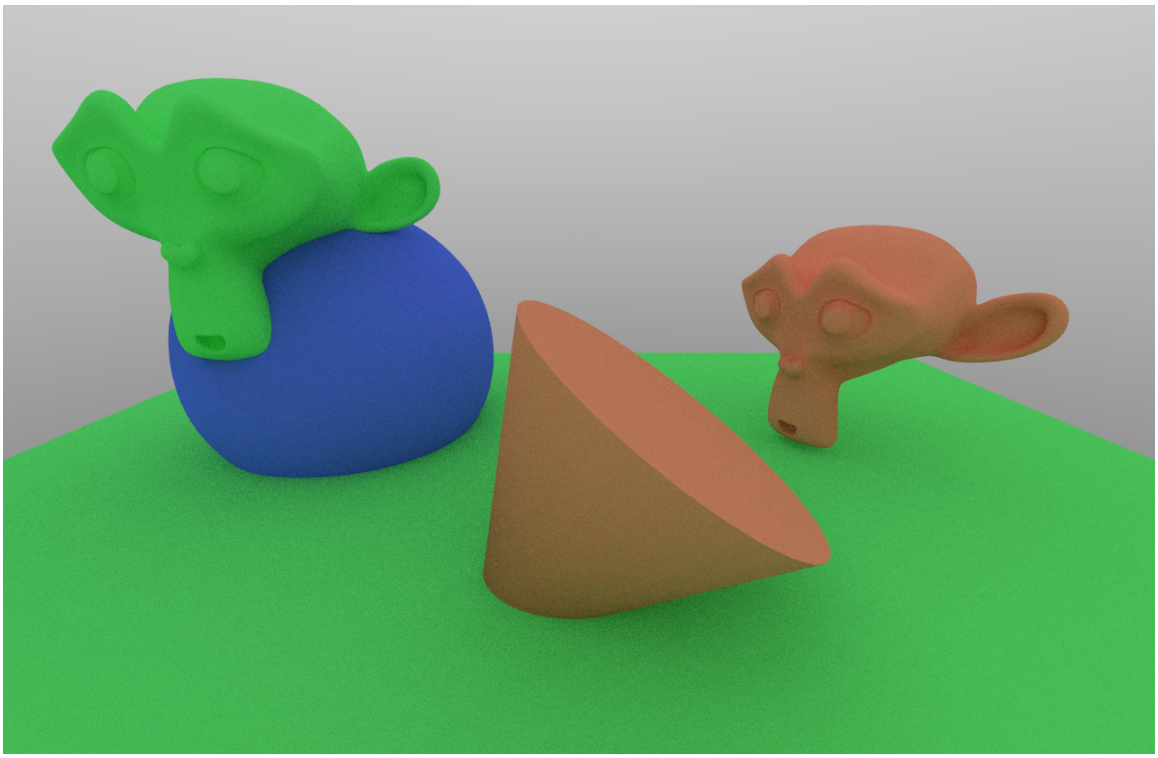


(a) Geometric normals (unsmoothed)

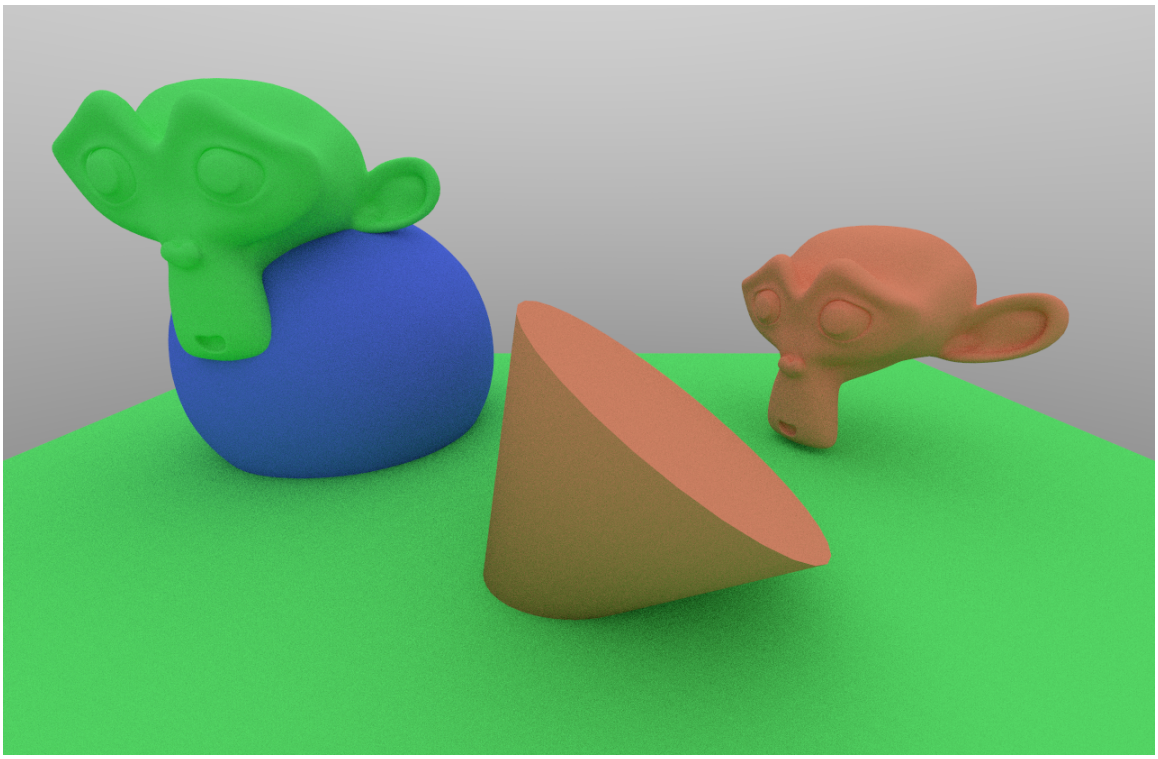


(b) Interpolated normals (smoothed)

Figure B.2: Comparison of geometric and interpolated normals



(a) Before uniform hemispherical sampling



(b) After uniform hemispherical sampling

Figure B.3: Uniform hemispherical sampling

Bibliography

[1] A ray-tracing pioneer explains how he stumbled into global illumination. https://blogs.nvidia.com/blog/2018/08/01/ray-tracing-global-illumination-turner-whitted/, 2018. Accessed: 2018-12-11.

[2] Git version control system. https://git-scm.com/, 2019.

[3] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, pages 37–45. ACM, 1968.

[4] Joel Auterson et al. Termloop. https://github.com/JoelOtter/termloop, 2015.

[5] CJ Cason, T Froehlich, N Kopp, R Parker, et al. Pov-ray. *Persistence of Vision Raytracer Pty. Ltd, Victoria, Australia*, 2003.

[6] John G Cleary, Brian M Wyvill, G Birtwistle, and R Vatti. Design and analysis of a parallel ray tracing computer. In *Graphics Interface*, volume 83, pages 33–38, 1983.

[7] The Unicode Consortium. *The Unicode Standard*. 2018.

[8] Joey de Vries. Pbr theory. https://learnopengl.com/PBR/Theory, 2016.

[9] Paul Evans. Tickit. http://www.leonerd.org.uk/code/libtickit/, 2017.

[10] Blender Foundation. Blender. https://www.blender.org/, 2002.

[11] Travis Foundation. Travis ci. https://github.com/travis-ci/travis-ci, 2011.

[12] Syoyo Fujita. tinyobjloader v1.0.6. https://github.com/syoyo/tinyobjloader, 2017.

[13] Google. Google c++ style guide. https://google.github.io/styleguide/cppguide.html, 2008.

[14] The Open Group. The single unix specification – drand48(void). https://pubs.opengroup.org/onlinepubs/7908799/xsh/drand48.html, 1997.

[15] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[16] Stefan Haustein, David Tschumperlé, et al. Terminalimageviewer. https://github.com/stefanhaustein/TerminalImageViewer, 2016.

[17] GitHub Inc. Github. https://github.com/, 2019.

[18] Gradle Inc. Gradle. https://gradle.org/, 2019.

[19] OTOY Inc. Octanerender. https://home.otoy.com/render/octane-render/, 2016.

[20] ISO. *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. International Organization for Standardization, fourth edition, 2014.

[21] ITU-R. Parameter values for the hdtv standards. Recommendation BT.709-6, International Telecommunication Union, Geneva, 2015.

[22] James T Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

[23] Tero Karras and Timo Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 89–99. ACM, 2013.

[24] L. Kettner, M. Raab, D. Seibert, J. Jordan, and A. Keller. The Material Definition Language. In Reinhard Klein and Holly Rushmeier, editors, *Workshop on Material Appearance Modeling*. The Eurographics Association, 2015.

[25] Anton Kochkov et al. Truecolour – colours in terminal. https://gist.github.com/XVilka/8346728, 2019.

[26] Jens Kruger and Rüdiger Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38. IEEE Computer Society, 2003.

[27] Dylan Lacewell, Detlef Roettger, et al. Advanced optix samples. https://github.com/nvpro-samples/optix_advanced_samples, 2017.

[28] Eric P Lafortune and Yves D Willems. Bi-directional path tracing. 1993.

[29] Saejin Mahlau-Heinert. Rayterm. https://github.com/Michionlion/rayterm, 2019.

[30] Saejin Mahlau-Heinert. Rayterm-cpu. https://github.com/Michionlion/rayterm-cpu, 2019.

[31] Saejin Mahlau-Heinert. Rayterm preliminary demonstration. https://www.youtube.com/embed/mr3k4GER3Rs?loop=1&playlist=mr3k4GER3Rs&controls=0&autoplay=1, 2019.

[32] Saejin Mahlau-Heinert and Paul Evans. libtickit. https://github.com/Michionlion/libtickit, 2019.

[33] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi. *NVIDIA, Santa Clara, Calif, USA*, 2009.

[34] NVIDIA. Cuda c programming guide. *NVIDIA Corporation*, 2011.

[35] NVIDIA. Nvidia optix 6.0 documentation. http://raytracing-docs.nvidia.com/optix_6_0/index.html, 2019.

[36] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics (TOG)*, 29(4):66, 2010.

[37] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2016.

[38] Jean-Colas Prunier. Global illumination and path tracing. https://www.scratchapixel.com/lessons/3d-basic-rendering/global-illumination-path-tracing, 2017.

[39] Jean-Colas Prunier. Introduction to acceleration structures. https://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure, 2017.

[40] Jean-Colas Prunier. Introduction to shading. https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading, 2017.

[41] Jean-Colas Prunier. Ray-tracing: Rendering a triangle. https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/, 2017.

[42] Steven M Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *ACM SIGGRAPH Computer Graphics*, volume 14, pages 110–116. ACM, 1980.

[43] Christophe Schlick. An inexpensive brdf model for physically-based rendering. In *Computer graphics forum*, volume 13, pages 233–246. Wiley Online Library, 1994.

[44] Peter Shirley. Ray tracing in one weekend. 2016.

[45] Peter Shirley and Kenneth Chiu. A low distortion map between disk and square. *Journal of Graphics Tools*, 2(3):45–52, 1997.

[46] Google's Testing Technology team. Google test. https://github.com/google/googletest, 2015.

[47] Gilles Tran. Pov-ray render using radiosity. https://commons.wikimedia.org/wiki/File:Glasses_800_edit.png, 2006. This image was created with POV-Ray 3.6 using Radiosity. The glasses, ashtray and pitcher were modeled with Rhino and the dice with Cinema 4D.

[48] Dimitri Van Heesch. Doxygen: Source code documentation generator tool. 2008.

[49] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics (TOG)*, 26(1):6, 2007.

[50] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. In *Computer graphics forum*, volume 20, pages 153–165. Wiley Online Library, 2001.

[51] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23, 1980.