RAYTERM

A Ray-Tracing Rendering Engine for XTerm-like Terminals

Saejin Mahlau-Heinert
Department of Computer Science
Allegheny College
mahlauheinerts@allegheny.edu
https://saejinmh.com

October 31, 2018

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 will render a fully three-dimensional (3D) scene, complete with lighting, shadows, and physically-based materials. RAYTERM can be used as an engine for terminal-based 3D tools, visualizations, games, and more; it will be fully open-source and ready for integration into other projects.

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 is abound in the gaming space. This proposal 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. With the current generation of powerful CPU and GPU chips, able to execute billions and trillions of calculations per second respectively, it is finally possible to do real-time, close to photorealistic rendering. While the goal of RAYTERM is not high-resolution photorealism, some approximation of photorealism will be obtained, albeit at a lower resolution.

This proposal describes RAYTERM, a system that will create what are, in essence, images on a terminal screen. This rendering engine will perform real-time updating of the displayed image, generating an animation. RAYTERM will use the recursive ray-tracing algorithm, simulating the path of light through the scene – this is described in more detail in Section 1.1. Rendered images will be displayed in two different modes: using single half-character pixels, or more complex

Unicode block characters. The mechanics of this image composition method are discussed in Section 1.2. Once an image is rendered, the `ncurses` C library [22] will be used to display it in a terminal. The final terminal output will be similar to Figure 1, which was generated by TerminalImageViewer [10] as a mockup. More details on this process are given in Section 1.3.

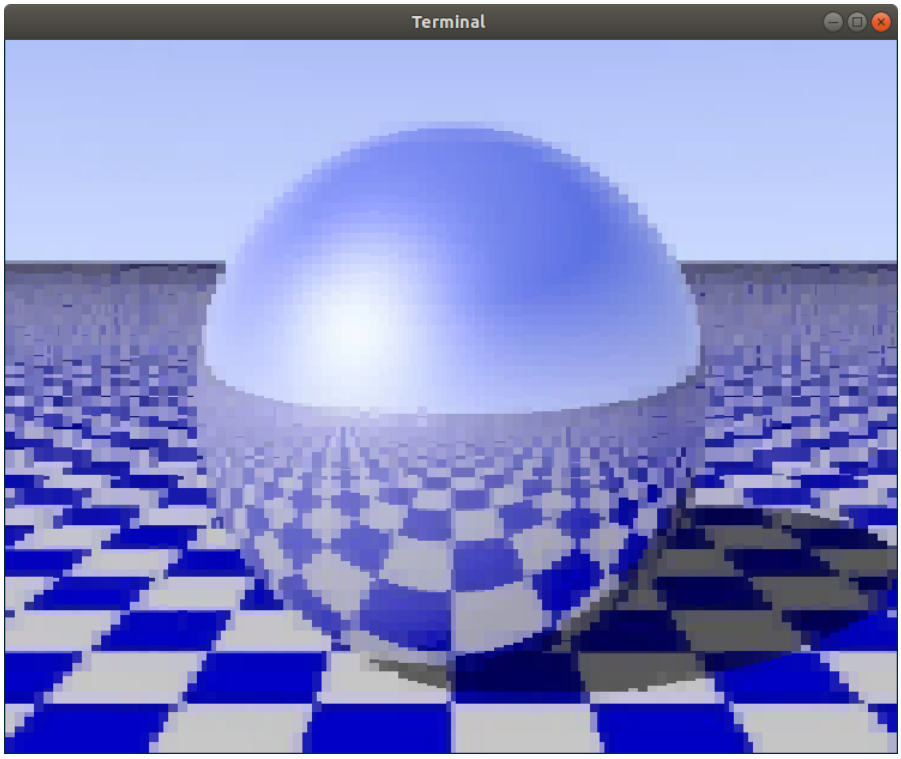


Figure 1: Example of proposed terminal output

1.1 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. In this proposal, the recursive ray-tracing algorithm first pioneered by Turner Whitted in his ground-breaking paper *An Improved Illumination Model for Shaded Display* [28], will be discussed and utilized. 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 renderer of choice for photorealistic images, 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 2 is a render created by the POV-Ray engine [3]. 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 3 provides more details on these advances, as well as an explanation of how ray-tracing will be utilized in RAYTERM.

Figure 2: POV-Ray render created by Gilles Tran [26]

Ray-tracing revolves around the idea of *rays*, a mathematical construct which can be 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 ray 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 can be used to get the coordinates of a point in three-dimensional space (assuming $\vec{R_{direction}}$ is a unit vector). This is called the *parametric form* of a ray. The mathematics behind ray-tracing are further explored in Section 3.

$$\vec{point} = \vec{R_{origin}} + t\vec{R_{direction}} \tag{1}$$

In ray-tracing, 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 will not be supported by the proposed RAYTERM, but would be a fascinating area of future work.

The basis for ray-tracing is the rendering equation, articulated by James Kajiya in 1986 [14]. The rendering equation models the outgoing light at some point given all incoming light, a bidirectional reflectance distribution function (BDRF), and a normal 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 engines that do this are known as *path-tracers* and are the most photorealistic rendering engines invented.

RAYTERM will not use a path-tracer – such algorithms are still many times too slow for most real-time rendering situations. Instead, the rendering equation will be solved by sampling the incoming light with rays. This is known as recursive ray-tracing, since it starts with a single ray that splits every time 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 four or more 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, splitting and generating new rays. These rays are biased towards directions that contain lots of incoming light – such as towards a point light source. The specific implementation of recursive ray-tracing that RAYTERM will use will only scatter into a set number of rays at each intersection point: one ray toward each light source, along with reflection and refraction rays towards the relevant directions.

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 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 [5]. Some critical characters that RAYTERM will use are known as the *block characters* – they are characters `U+2580` – `U+259F`. Relevant characters are shown in Figure 3. Contingent on the quality of the output using only block characters, additional sets could be used, such as triangles and lines. The core of image composition using Unicode is an algorithm coloring the characters and the background – RAYTERM will use 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.

There are two image modes that will be available for use in RAYTERM. The first is pure *pixel mode*, in which the Unicode "half-block" symbol (`U+2584`) is used. Since mono-spaced character output (such as in a terminal) is twice as tall as it is wide, the half-block can split a single

2581 LOWER ONE EIGHTH BLOCK
2582 LOWER ONE QUARTER BLOCK
2583 LOWER THREE EIGHTHS BLOCK
2584 LOWER HALF BLOCK
2585 LOWER FIVE EIGHTHS BLOCK
2586 LOWER THREE QUARTERS BLOCK
2587 LOWER SEVEN EIGHTHS BLOCK
2589 LEFT SEVEN EIGHTHS BLOCK
258A LEFT THREE QUARTERS BLOCK
258B LEFT FIVE EIGHTHS BLOCK
258C LEFT HALF BLOCK
258D LEFT THREE EIGHTHS BLOCK
258E LEFT ONE QUARTER BLOCK
258F LEFT ONE EIGHTH BLOCK

(a) Block Elements

2596 QUADRANT LOWER LEFT
2597 QUADRANT LOWER RIGHT
2598 QUADRANT UPPER LEFT
2599 QUADRANT UPPER LEFT AND LOWER LEFT AND LOWER RIGHT
259A QUADRANT UPPER LEFT AND LOWER RIGHT
259B QUADRANT UPPER LEFT AND UPPER RIGHT AND LOWER LEFT
259C QUADRANT UPPER LEFT AND UPPER RIGHT AND LOWER RIGHT
259D QUADRANT UPPER RIGHT
259E QUADRANT UPPER RIGHT AND LOWER LEFT
259F QUADRANT UPPER RIGHT AND LOWER LEFT AND LOWER RIGHT

(b) Quadrant Elements

Figure 3: Subset of `U+2580` – `U+259F` [5]

character into two 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 would mean that a typical 85 by 30 character terminal would result 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 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 eigth of a character in width or length. The differences between these two modes are highlighted in Figure 4a and 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 much more defined. The performance impact of both modes is another factor that must be assessed when making implementation decisions for RAYTERM. More details on the algorithm for ray-to-character translation is available in Section 3.1.2.

(a) Pixel Mode Image Output

(b) Character Mode Image Output

Figure 4: Examples of Image Mode Output

1.3 Terminal Output using `ncurses`

Once an image is generated, RAYTERM must somehow display that image in a terminal with no surrounding prompt or other formatting – just a simple *character field*. To do this, we will use `ncurses`, a C library that abstracts implementation details of the terminal to enable full-window 24-bit color character output. Using `ncurses`, the terminal window will be divided into two "panels", a main panel for the actual image output – updated 30 to 60 times per second – and an info panel to render information such as frames per second and logging information. With this dependency, RAYTERM can be used on all terminals that support `ncurses` – generally, any XTerm-like terminal will work, as long as it supports a terminal information database: either `termcap` or `terminfo` will work.

A possible limitation to `ncurses` is keyboard input handling – there is no mechanism to get events on an up or down movement of a key, as is possible in some other libraries. Therefore, research will need to be done on possible alternatives for the input aspect of the engine. This is not a priority, however, since it is not related to the rendering nature of RAYTERMand instead only helps with demonstrations of its capabilities.

2 Related Work

In this section, we will discuss related research papers, detail how they inform RAYTERM's development, and show the improvements of RAYTERM over similar 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 has already been utilized to create some of the figures in this document, such as Figure 4.

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* [1]. His algorithm is now known as a ray casting algorithm – it does not follow the approach we have seen so far. 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.

The Breakthrough

The next big entry to the ray-tracing field was Turner Whitted's 1980 paper *An Improved Illumination Model for Shaded Display* [28]. In this groundbreaking work, Whitted introduced the recursive ray-tracing algorithm we covered in Section 1.1. Whitted was not the first to use ray-tracing – Arthur Appel had first pioneered the field over a decade ago, and some commercial applications were also emerging in the field of radiosity. 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. Recursive ray-tracing approximates this very well, and thus Whitted-style ray-tracing was born. Even today, any simple recursive ray-tracing algorithms (like the one proposed for RAYTERM) are known as Whitted-style ray-tracers.

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 in general was published by James Kajiya in 1986 [14]. In his paper *The Rendering Equation*, he articulated a generalization for many different rendering algorithms; this generalization is shown as Equation 2 below. Although the idea behind the rendering equation was not completely new, Kajiya presented it in a form especially suited for computer graphics using vector mathematics. The equation also gives direction for more advanced and photorealistic rendering techniques, leading up to path-tracing.

$$I(x,x') = g(x,x')[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''] \tag{2}$$

We will not give a full explanation of the rendering equation here. The significant part for RAYTERM is the integral, since if we can approximate that, RAYTERM will be photorealistic. In RAYTERM, we will use the rendering equation to inform the recursive ray generation, attempting to find areas of the integral where most of the light is incoming. 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 as our implementation is built is critical, so that RAYTERM does not stray too far from photorealism.

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* [23].

A BVH essentially 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. Ray-tracing intersections are done with the root group's bounding volume first, then progress down the hierarchy and 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 [20], and if there is development time, some form of BVH will be implemented in RAYTERM.

Parallelization

Ray-tracing is inherently parallelizable, because each ray-tree is independent – rays cannot collide, and one fragment's ray-tree does not affect another fragment's ray-tree. Thus, there have been many opimizations that target enhanced hardware acceleration to enable massively parrallel ray-tracing. The first example of this was *Design and Analysis of a Parallel Ray Tracing Computer*, a 1983 paper by John Cleary and others [4]. In it, Cleary describes a computing system that is functionally similar to the Grapics 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 was not lost. Much of the research done in this area led to the design of systems such as OptiX [18] (which we describe in Section 3.3), enabling projects like RAYTERM to benefit from the massive speed increases estimated by early papers. Instead of a $50 million monstrosity, relatively small GPU chips capable of the same performance are now available for only hundreds of dollars.

Terminal Images

A small program available on Github was the original inspiration for some details of RAYTERM. Called TerminalImageViewer (`tiv`) [10], it uses RGB ANSI escape codes and Unicode block characters to display images in a terminal window. RAYTERM will use the same ideas to produce its own animated output. Many of the example figures in this proposal were also created using `tiv`. The algorithm behind `tiv` is a direct inspiration for the ray-to-character algorithms described in Section 3.1.2. RAYTERM, however, improves upon TerminalImageViewer in a single, incredibly impactful way: the graphics produced are animated. Fundamentally, `tiv` and RAYTERM are similar only in output format, while the purpose and internal workings are generally dissimilar.

Gaming with Termloop

A stated 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 for the terminal [2]. It has a similar purpose to RAYTERM – namely, to facilitate game creation in the terminal. However, Termloop, like all other terminal game engines available today, has one glaring limitation: it is not 3D. 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 that, by allowing full 3D rendering in the terminal. It doesn't have the other niceties such as collision detection, but the ultimate goal of RAYTERM – beyond the stated goals in this proposal – is to grow to provide the same level of support for 3D games that Termloop has for 2D games.

3 Method of Approach

The methods, algorithms, and techniques used to implement RAYTERM are discussed here. In Section 3.1 we detail most of the mathematics involved; core algorithms for the `ncurses` interface are also covered there. Software tools and languages utilized to build RAYTERM are discussed in Section 3.2. Finally, hardware interfaces, along with the current landscape of hardware availability, are covered in Section 3.3. Additionally, discussion on parallelization is presented.

3.1 Algorithms and Mathematics

The mathematical basis for ray-tracing, which we describe in Section 3.1.1, relies on an understanding of vector math. These algorithms will be running millions of times per second in a highly parallelized environment. In this context, parallelization is a method of running two or more algorithm simultaneously; therefore even minor optimizations reduce the overall time to render, and are extremely important. We also cover details for the proposed algorithms for ray-to-character translations in Section 3.1.2. All of the ray-tracing mathematics described here are synthesized from Peter Shirley's excellent *Ray Tracing in One Weekend* [24] and the Morgan Kaufmann textbook *Physically Based Rendering: From Theory to Implementation* [19].

3.1.1 Ray-Surface Intersection Algorithms

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 a implicit surface definition function, in the form of Function 3, is intersectable with a ray [19]. In Function 3 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 4 for t and then using Formula 1 to calculate the coordinates of that point on the ray.

$$f(\vec{p}) = 0 \tag{3}$$

$$f(\vec{R_{origin}} + t\vec{R_{direction}}) = 0 \tag{4}$$

In RAYTERM, the only surfaces that will be supported are spheres, triangles, and infinite planes, since their surface definition functions are mathematically simple. If there is additional development time and resources available, other shapes such as quads, arbitrary polygons, and cubes may be considered. This may not be a huge development effort, as the algorithm discussed for triangle intersection testing is applicable to any convex polygon. A sphere is the simplest three-dimensional object to calculate ray intersection with, and therefore will be the first implemented for RAYTERM. In fact, during feasibility testing much of the math described in this section was implemented in the Go programming language.

Ray-Sphere Intersection

The surface definition function of a sphere is Function 5, with S representing the sphere. The intersection point between a ray and a sphere is given by solving Equation 6 for t and then using Formula 1. Both of these equations are directly adapted from *Ray Tracing in One Weekend* [24]. Notice that Equation 6 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.

$$f(\vec{p}) = (\vec{p} - \vec{S_{center}})^2 - {S_{radius}}^2 \tag{5}$$

$$({\vec{R_{direction}}}^2)t^2 + 2(\vec{R_{direction}} \cdot (\vec{R_{origin}} - \vec{S_{center}}))t + (\vec{R_{origin}} - \vec{S_{center}})^2 - {S_{radius}}^2 = 0 \tag{6}$$

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 straight forward. The surface definition function of a plane is Function 7, 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 8 for t and then using Formula 1. These equations were derived from basic vector math, along with guidance from *Physically Based Rendering* [19].

$$f(\vec{p}) = (\vec{p} - \vec{P_{offset}}) \cdot \vec{P_{normal}} \tag{7}$$

$$(\vec{P_{normal}} \cdot \vec{R_{direction}})t + \vec{P_{normal}} \cdot (\vec{R_{origin}} - \vec{P_{offset}}) = 0 \tag{8}$$

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 [21], 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 left-side 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 9: 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 9, if $i = 3$, then $i = 0$ (i "wraps" to only valid values).

$$f(i) = P_{normal} \cdot ((\vec{V_{i+1}} - \vec{V_i}) \times (\vec{Q} - \vec{V_i})) \tag{9}$$

Function 9 can be complicated to visualize, so imagine this: we first form two vectors, both with $\vec{R_{origin}} = \vec{V_i}$. One points along the triangle's edge, while the other points to $\vec{Q}$. 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 will therefore simply expect vertices to be specified in counter-clockwise order. This is similar to what many other 3D rendering programs assume.

3.1.2 Ray-Character Translation Algorithm

To translate the color result of a ray-trace to an actual character pixel, we propose two algorithms. These algorithms are inspired by the pixel-to-character translation algorithm used in Terminal-ImageViewer [10] – they are optimization attempts. RAYTERM will start with the Quadrant Algorithm, and then research and testing will be conducted to see if it is possible to improve towards the Search Algorithm, without incurring too much performance cost. If neither algorithm is able to produce an acceptable image quality and running TerminalImageViewer's algorithm is possible in real-time (it will require 32 rays per character pixel), then RAYTERM will use that algorithm instead of an optimized version.

Quadrant Algorithm

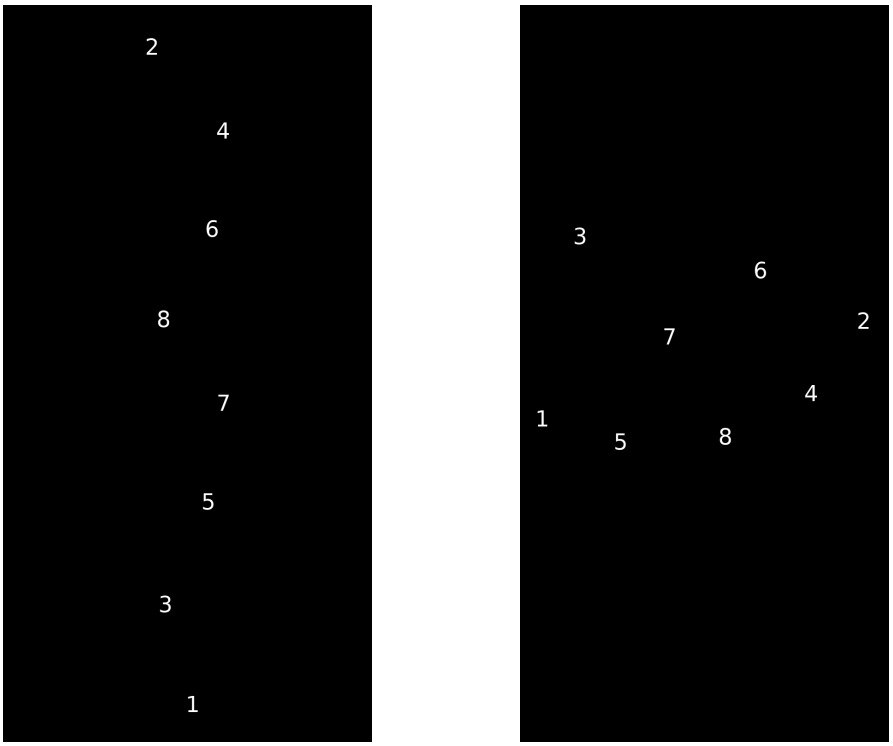
The Quadrant Algorithm works with only four rays per character pixel. These four rays will sample each of the four quadrants of the character and give color values. Once we have the four quadrant colors, we take the perceived brightness of each quadrant, using Formula 10, where C is the color in question [6]. Using the translated values we calculate an average brightness for the entire character. Any quadrant which has a higher brightness is then considered "on", while any darker quadrants are "off". Thus we determine which character from the set `U+2596` – `U+259F`, shown in Figure 3b, best represents the brightness gradient of the character pixel. After this, we must determine the color of the foreground ("on") and background ("off") quadrant sets. This is simply the average color value of the relevant rays – with this computed we have the colors and character to use for the character pixel.

$$brightness = \sqrt{0.299{C_{red}}^2 + 0.587{C_{green}}^2 + 0.114{C_{blue}}^2} \tag{10}$$

The Quadrant Algorithm's main problem is that it does not take into account color differences. Colors may be wildly different but have the same perceived brightness. If this is the case, the average color would not be an accurate representation of the character pixel. The main way to combat this would be to determine the "on"-ness of quadrants by a per-channel tonal range check similar to how TerminalImageViewer operates. Before making such optimizations, however, the algorithm must be tested with actual images.

Search Algorithm

The Search Algorithm starts with a completely different premise than the Quadrant Algorithm. It keeps track of what characters might be a good fit, and with each additional ray computed, narrows the list. This algorithm must again use the criterion of "on" and "off"-ness, to calculate what parts of the image should be foreground and which background. This re-introduces the previous problem of possible color inaccuracy. The Search Algorithm searches over characters from the set `U+2581` – `U+258F` – these characters are shown in Figure 3a. The character pixel is sampled with rays in a semi-randomized fashion: first a vertical search of the pixel is conducted, attempting to find an edge between "on" and "off". Then, a search across the horizontal axis of the pixel is conducted. In both of these searches the non-relevant axis is randomized to ensure that a significant edge is not missed. Figure 5 shows an example of one such set of searches.



(a) Example randomized search over the character's vertical axis

(b) Example randomized search over the character's horizontal axis

Figure 5: Example randomized searches – numbers indicate processing order

Once the searches are complete and an edge is found in both directions, the magnitude of the edges are tested. The larger edge is then given priority, and the relevant character is selected. For example, if the vertical edge is given priority, and rays 1, 3, and 5 (see Figure 5) are "on", then the three-eighths tall block character (`U+2583`) will be selected for that character pixel. This algorithm must also be tested with actual images to judge the quality of the translation. As with the Quadrant Algorithm, there are some possible optimizations for this algorithm – most only apply to a sequential implementation, however. This is because it is actually faster to ray-trace all 16 samples in parallel than to stop after every ray and check whether an edge has been found, as most optimizations would do.

3.2 Software

RAYTERM's implementation will be hosted on Github, a public platform for projects using the Git version control system. Travis CI [8], a continuous integration system, will be used for testing and environment management. Documentation will be handled by `doxygen` [27] – all external facing functions, methods, and classes will have a minimum of a few sentences of documentation. More details about the implementation tools are given in Section 3.2.2. Additionally, Section 3.2.1 discusses some implementation specific practices for handling data.

3.2.1 Data Handling

Data for use in scenes and materials will be stored in plain text. This enables ease of use and human readability, as well as making scene editing extremely easy. Data storage will likely use the YAML format, leveraging a C/C++ library or similar resource to ease the implementation burden. The ideas described here are only informed guesses for how these entities should be represented; after the initial CPU ray-tracer implementation – see Section 5 – this model may be improved. Notably, one possible improvement is known as *instancing*, and involves allowing multiple copies of the same geometry to reference the same underlying data.

Materials

Materials are not fully user-specifiable. Instead, they simply reference a material type, such as lambertian or dielectric, and a texture. Texture mapping is a complex topic that will require much more research, especially as implementation becomes a possibility. For now, solid textures will be the only supported texture. Some additional values such as roughness control specific aspects of the generated scattering function. Materials also contain a user-set identification number to allow other objects to apply the material to themselves.

Objects

Objects are composed of geometric primitives or other objects. Since RAYTERM supports only spheres, infinite planes, and triangles, more complex objects require composition. A "box" object for example might contain four "side" objects which themselves contain two triangles. Each sub-object is associated with an offset from its parent. Thus, the data required for an object definition

is just an array of sub-objects and a position. Every object may also have a material ID – if not specified, the object's parent's material ID is used instead. This material ID dictates what material is used to scatter rays when they intersect with the object. Objects may also have an animation component – a *movement path* – ID. Similar to materials, movement paths are simply a collection of locations and a length of time to take to traverse those locations. In future versions of RAYTERM, movement paths are likely to be removed in favor of total object control in the form of scripting integration.

Scene Description

The scene description is a single file where all materials, objects, and movement paths in the scene are stored. At the top level it contains three lists: one of materials, another of objects, and a third of movement paths. One last special piece of data specific to the scene description is the camera description. This specifies the technical details of the camera such as focal length and field of view, as well as mechanical details such as its position and speed. The camera can also have a movement path defined, or optionally user control enabled. User control enables the user to move the camera around in real-time, fully experiencing the ray-traced scene around them.

3.2.2 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 chose to develop in the C++ language [13], and use Gradle [11] as the build system. Hardware APIs are covered in Section 3.3.

Implementation Language

The C++ language was chosen for three main reasons. First, the language has a huge ecosystem of low-level tools, with interfaces to libraries such as OptiX, CUDA, FORTRAN-implemented mathematics like Blitz++, and much more. This ensures that no matter the need, there is most likely a library out there that will 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 not be possible for newer languages.

RAYTERM will attempt to keep most of its implementation to a C-compatible level, using classes and greater abstractions only as necessary. This ensures that overhead will be minimal, as well as guaranteeing readability for the eventual open source release. Advanced features of C++ such as templating will be avoided so as to keep the knowledge entry barrier low. Finally, all code in RAYTERM's implementation will conform to the Google C++ style guide [9].

Build System

Complex projects can be a nightmare to manage, especially when there are multiple contributors. Build systems are an important tool that can simplify this headache. An opinionated build system such as Gradle also specifies sensible defaults for most situations, allowing minimal require configuration. This is why Gradle was chosen as the build system for RAYTERM. Gradle can be 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.

3.3 Hardware

There are a lot of unsolved technical issues that we may encounter as work continues for RAYTERM. Many issues are related to the performance aspects of RAYTERM, since it is a strict requirement for obtaining render times low enough for real-time display. 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 this proposal will not focus on RTX hardware and instead will look to slightly older technologies.

The main technical research for hardware acceleration will be done on the OptiX API [18], discussed in Section 3.3.2. The main goals of this research are to understand and use the technologies already available for non-real-time ray-tracing to do real-time work. Hardware testing will be done on an Intel Core i7 4770k CPU and NVIDIA Geforce GTX 980Ti.

3.3.1 GPU vs. CPU

Although a CPU-only ray-tracer is completely possible, and will be the first step in embarking on the creation of RAYTERM, it is not likely to be performant enough for real-time graphics. To reach this performance, the ray-tracer must take less than $\frac{1}{fps}$ seconds to render a single image. For 30 frames per second, the minimum required for real-time fluid animation, RAYTERM must render an image every 33 milliseconds. Therefore leveraging either CUDA (a compute API for NVIDIA GPUs [17]) directly, or utilizing OptiX will likely be required.

The main advantage to using a GPU instead of CPU is the massive parallelization possible – GPUs often have hundreds to thousands of processor cores, whereas a standard current-generation CPU has around eight. Ray-tracing is an inherently parallelizable algorithm because of the independence of each ray computation. Thus, taking advantage of thousands of cores provides massive boosts to performance, since many more rays are able to be simulated at once. However, interfacing with the highly complex GPU architecture is often done through a shader program instead of some general purpose language. Shader programs are bits of programmable code that

are inserted between fixed operations in rendering, such as vertex interpolation. Since ray-tracing requires a completely different set of fixed operations, namely ray creation and intersection testing, it is not possible to use traditional methods built for other rendering techniques. Instead, a new programming interface must be used, such as the afore mentioned OptiX or CUDA.

3.3.2 OptiX

OptiX is an API interface developed for current generation NVIDIA GPUs that enables GPU-accelerated ray-object intersection calculations in a programmable graphics pipeline [18]. OptiX allows user-defined single-ray *programs* to be compiled into a single program 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 acceleration structure implementations like BVH. Additionally, lightweight scene representation formats are defined and usable, although are not required for use of the main API.

Further practical research on the uses and implementation of OptiX is required, especially on the topic of C++ to PTX transpilation. Additional research on the source code of OptiX itself may be useful as well, even if RAYTERM does not eventually use OptiX as an API.

3.3.3 CUDA

Utilizing the massive 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. 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 in the field of ray-tracing, is that performance is significantly effected for inherently divergent tasks – tasks which do not consistently follow the same control flow – such as traversing an acceleration structure.

RAYTERM may use CUDA C++ to perform ray-tracing intersection tests and dispatch ray simulations. If OptiX, a more ray-tracing-specific API, does not work out due to implementation issues or other reasons, CUDA is the fall-back resource.

4 Evaluation

Evaluation and testing of a graphics engine is a difficult proposition – it often involves extremely subjective reasoning, especially as to the realism of an algorithm. Given it's subjectiveness, some researchers use paired-comparison and rating-scale experiments [16]. These experiments generally ask observers to rate or compare the accuracy of some generated image. RAYTERM will not be using these methods, because its main purpose is not photorealistic graphics – although adequate realism is expected – but instead a unique pixelated style, real-time display, and easy extension.

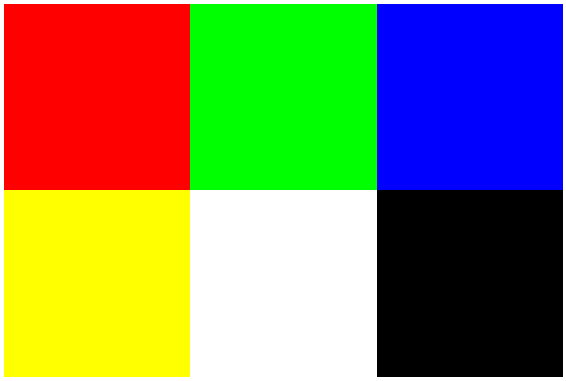
The original retro pixelated style has many critics and advocates, as can be seen in popular video game culture. Therefore, an observer-based test is likely to be heavily biased. Instead of a comparison test with many participants, simple "adequate realism" tests will be conducted throughout development to detect bugs and focus implementation in the correct direction. Descriptions of these tests, along with details on code testing, are available in Section 4.3.

4.1 Portable Pixmap Output Mode

The first step in evaluating and testing a ray-tracer is to get the output in an versatile format. RAYTERM will have a debug mode called *`ppm` mode* that can generate image files in the portable pixmap format. This mode will be a different interface alongside the `ncurses` interface, allowing much higher resolution output for testing. It is unlikely that this mode will be real-time capable, but render times for `ppm` images with millions of fragments will serve as good benchmarks for algorithm optimizations. The image format `ppm` was chosen for its human readability along with ease of use for development. There are many versions of the `ppm` format – RAYTERM will be using the ASCII varient. The file content format is extremely simple: a `ppm` file is shown in Figure 6a, with Figure 6b being its output.

```
P3 # header
3 2 # width x height
255 # max color value
# image data -- RGB triplets
# (extra whitespace is ignored)
255   0   0     0 255   0
  0   0 255   255 255   0
255 255 255     0   0   0
```

(a) `ppm` file contents



(b) `ppm` image

Figure 6: Portable pixmap format example

Using this `ppm` image output mode will enable graphical quality comparisons, testing, and bug-fixing. It will also provide output and results data for output-based regression testing to ensure that any changes are purposeful. Lastly, `ppm` mode will be used to test the fidelity of RAYTERM's ray-to-character translation algorithm against other available algorithms in programs such as TerminalImageViewer by providing a single, identical input image for translation. Both algorithms will be run, and the resulting character field evaluated for differences.

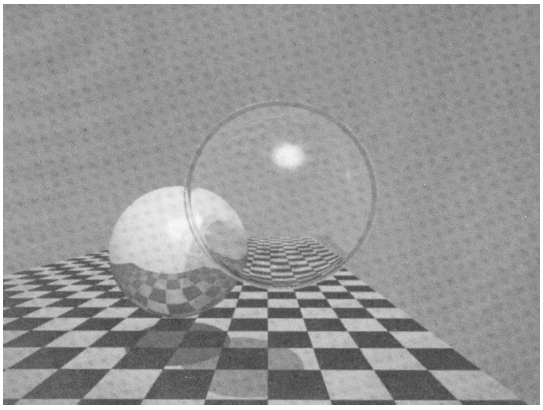
4.2 Demonstration Scenes

In addition to `ppm` mode, demonstration scenes will also be built to enable real-time testing of the ray-tracing algorithm and RAYTERM's associated systems. These scenes will provide data

for "adequate realism" tests as well as provide demonstrations of RAYTERM's capabilities. The final project will include at least the following scenes, but may be more extensive.

- Whitted's Sphere
- Cornell's Box
- Shadow Demo
- Moving Object
- Environment Exploration

The first two scenes are famous ray-tracing reference scenes and will be used to benchmark the performance of RAYTERM against other ray-tracing engines, such as OctaneRender [12] or Blender [7]. Figure 7 shows versions of both scenes.



(a) Original Whitted's Sphere figure [28]

(b) Cornell Box by Kari Kivisalo [15]

Figure 7: Classic ray-tracing reference scenes

With the addition of the `ppm` image mode, output images will be visually compared to detect errors in visual effects such as specular reflections. The other three scenes are demonstrations designed to showcase certain aspects of RAYTERM. The Shadow Demo scene is a simple scene similar to NVIDIA's RTX shadow demo, seen in Figure 8. It will demonstrate soft shadows and show the limitations of RAYTERM's ray-tracer. Since only point lights will be supported at this point, this scene will not look as realistic as Figure 8. When this scene is implemented, and if there is time in the development schedule, area lights may be implemented through multiple random sampling. This simple addition to recursive ray-tracing simply adds more shadow rays to each scatter, randomly spread between any area lights in the scene.

The Moving Object scene will be modeled after a typical Cornell Box scene, however, the main point of the scene will be object and light source movement in real time. This will stress the ray-tracer and demonstrate the real-time shadows, lighting, and rendering to the fullest extent. Finally, the Environment Exploration scene is a scene where the user will be able to "explore" and move the camera around to experience a wider area. Currently envisioned as a set of complex

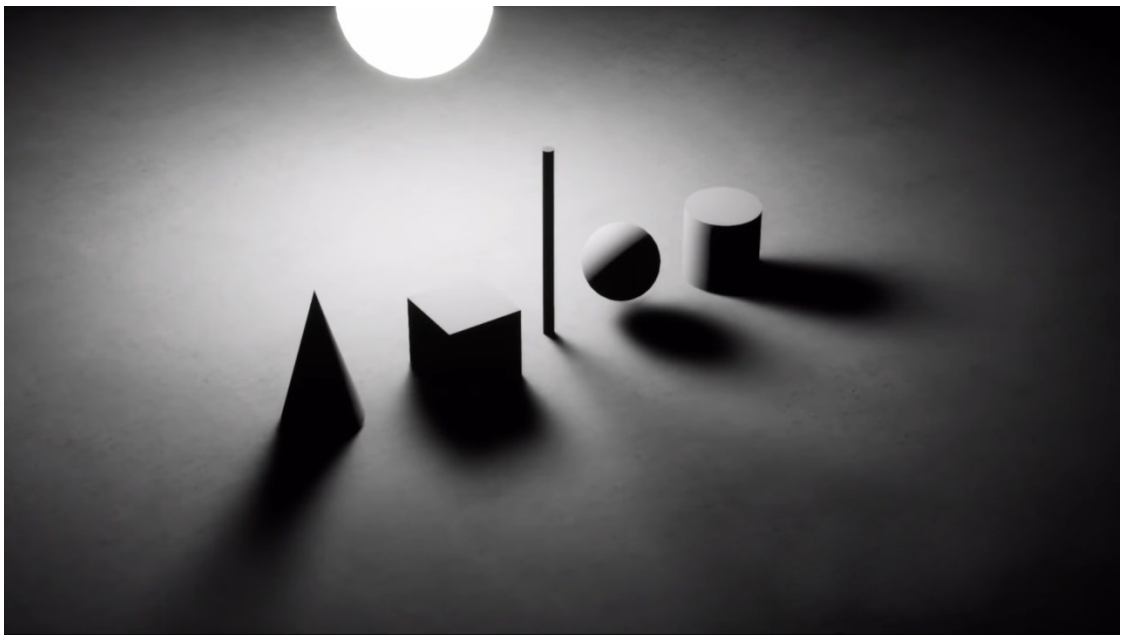


Figure 8: NVIDIA Shadow Scene – shown at CES 2018, NVIDIA Keynote

models of a street corner or otherwise realistic location, it will be the stress test for RAYTERM. As many thousands of objects and possibly millions of rays are computed per second, this scene will ensure that the ray-tracing algorithm is well-designed and optimized, and if an acceleration structure is implemented, that structure performs adequately.

Along with the scene demonstrations there will also be material demonstrations – although simple in nature, they will show off the different implementations of lambertian, dielectric, and metal materials, along with other possible effects such as glass, tessellation, and ray-traced bump mapping. Many of these effects are contingent upon available development time; however, at least the lambertian and metal demonstrations will be available in the final project.

4.3 Testing

Testing will focus on reference image comparison, specifically with reference images generated by *Cycles*, the rendering engine used by the Blender modeling tool [7]. These reference images will be compared with the same scene rendered with RAYTERM. Visual effects such as reflections, specular highlights, transparency, shadows, and ambient occlusion will be examined in both images to determine if there is any loss of quality with RAYTERM's ray-tracing algorithm. Some loss of quality is acceptable; however, anything that could be noticed by a viewer as appearing unrealistic or otherwise not making physical sense violates the "adequate realism" criterion. An example of such a violation might be "fireflys" – pixel abnormalities often caused by numerical errors that result in pure black or white pixels. Additional violations may occur in object seams or other areas where geometry is not fully defined – these situations should be fixed if and when they occur.

The reference images and RAYTERM output will also be mathematically compared by highlighting portions of the images that are not pixel-perfect copies (often referred to as "diffing"). This is likely to have many false positive bug identifications as small variations can easily happen. However, diffing may point out possible inaccuracies that do not appear physically incorrect but may still be a bug or unintended side effect. Additional testing will focus on render times – the time it takes a rendering engine to complete an image – specifically comparing RAYTERM to OctaneRender [12] and its real-time ray-tracing algorithm. This will involve creating identical scenes and then gathering and comparing performance data for both engines. In most cases, since RAYTERM will be implementing only a subset of the effects OctaneRender uses, it should be faster. As a counter-point, however, OctaneRender is highly optimized and thus may have the edge in raw performance for complex scenes.

Code testing will use the *Google C++ Testing Framework* for unit testing [25]. Code coverage should cover all mathematics routines at minimum, and greater than 90% algorithm coverage is the goal. CUDA or OptiX code testing will be done with a mix of CPU-side Google Test code and custom test handlers for GPU testing. Unit testing coverage for the `ncurses` interface should also be greater than 90% if possible; however, additional research into this area will be required, as creating a mock-up terminal for testing may prove to be more work than it is worth. Lastly, test integration with Travis CI is also a goal. Travis CI environment tests will also be used for different terminal types if additional research determines this is possible.

5 Research Schedule

Table 1 proposes a schedule for the implementation and research work needed to complete RAYTERM. First, a CPU implementation of the recursive ray-tracing algorithm will be developed, at first tested in `ppm` output mode. Then, as the ray-tracing code grows to a testable state, the `ncurses` interface will be developed. This involves implementing the ray-to-character translation algorithm, as well as standardizing an interface with which another render engine (specifically the future GPU ray-tracer) can interface with the displayed terminal image.

Task	Begin Date	End Date
Proposal defense preparation	Early Nov.	Mid Nov.
CPU implementation	Late Nov.	Early Jan.
`ncurses` interface implementation	Mid Dec.	Late Jan.
Thesis introduction/related works	Late Nov.	Mid Dec.
OptiX/CUDA research	Mid Dec.	Early Jan.
GPU implementation	Early Jan.	Early Feb.
Scene/material creation	Mid Dec.	Late Mar.
Thesis chapter writing	Early Feb.	Early Apr.
Graphical testing/comparison	Early Mar.	Mid Mar.
Thesis defense preparation	Early Apr.	Mid Apr.

Table 1: Proposed work schedule

Once both the CPU ray-tracer and `ncurses` implementations are nearing completion, research will begin on OptiX/CUDA integration. The design and architecture of the GPU ray-tracer will

take the lessons learned in the implementation of the CPU ray-tracer and use them to improve organization and understandability. Before the design phase for the GPU ray-tracer starts, thesis writing will commence in earnest. The first stages of writing will d323ocument the challenges and lessons learned from completing the CPU ray-tracer, including performance testing and evaluation; these will eventually end up in the Methods of Approach section of the thesis document. Additional research and writing for the Introduction and Related Works sections will also begin at this time.

Finally, implementation of RAYTERM's actual GPU ray-tracing engine will begin in early January. From this point onward, implementation and demonstration will be the primary goals. By late February, final demonstration scene creation should be wrapping up and material demonstrations should be well on their way to completion. The tempo of thesis chapter writing picks up early March, with discussions of the various new GPU-based techniques used for the GPU ray-tracer as the primary focus. After everything else is complete, end result testing and graphical comparisons will be conducted and documented, along with preparation for the thesis defense.

6 Conclusion

RAYTERM is a complex system, the implementation and design of which will not be fully known until initial testing and research is completed in the form of the CPU ray-tracer. The proposed system will support real-time 3D scene viewing through a terminal emulator. This will made possible by leveraging one of the hardware interfaces discussed in Section 3.3 to compute ray-surface intersections in a highly parallelized manner on a GPU. Ray-to-character translation algorithms are also needed to transform the rendered image into Unicode characters. These algorithms, along with the `ncurses` library, enable RAYTERM to treat a terminal as an image canvas.

The development of RAYTERM will proceed at first in an experimental stage while the abstractions and foundations for ray-tracing are created. The end result of this experimental stage is a working CPU ray-tracer, which may or may not be capable of real-time rendering. The second and final stage involves rewriting the CPU ray-tracer to utilize GPU compute power, enabling faster render times and real-time performance. The lessons learned from the experimental stage are crucial for a good design and implementation at this point. The end result of this stage is the ultimate goal of RAYTERM: a fully real-time capable ray-tracing engine, rendering 30 Unicode character images per second to a terminal window.

After the initial development period discussed in Section 5, all contributions to RAYTERM development will be welcome. In pursuit of maintaining good open-source readability and ease of development, documentation will be a priority throughout the initial implementation process. Additionally, tight integration with Github and Travis CI will ensure the integrity of public contributions. We hope that future projects will find RAYTERM inspiring and useful, whether as an engine or as a jumping-off point for real-time ray-tracing and terminal-based games.

References

[1] 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.

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

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

[4] 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.

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

[6] Darel Rex Finley. Hsp color model – alternative to hsv (hsb) and hsl. 2006.

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

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

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

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

[11] Gradle Inc. Gradle. `https://gradle.org/`, 2018.

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

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

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

[15] Kari Kivisalo. Cornell box. `https://commons.wikimedia.org/wiki/File:Cornell_box.png`, 2003. This image was rendered in POV-Ray.

[16] Jiangtao Kuang, Hiroshi Yamaguchi, Changmeng Liu, Garrett M Johnson, and Mark D Fairchild. Evaluating hdr rendering algorithms. *ACM Transactions on Applied Perception (TAP)*, 4(2):9, 2007.

[17] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 2011.

[18] 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.

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

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

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

[22] Eric S. Raymond, Thomas E. Dickey, et al. ncurses. `http://invisible-island.net/ncurses/`, 1993.

[23] 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.

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

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

[26] 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.

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

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