Ray Tracing

Project Report

Eric Farmer
`edf63@msstate.edu`

December 12, 2019

Figure 1: This computer-generated image shows the power of ray tracing [1].

1 Introduction

Ray tracing is a rendering technique for generating images that simulates light propagating through space and shades objects based on complex inter-object interactions. While computationally costly, ray tracing is capable of producing images with nearly-realistic quality. Because of its ability to produce images with a very high degree of realism, it has become heavily integrated with the image and film industries. Ray tracing is a powerful tool for integrating computer-generated graphics with real-life video captured on filming sets, which has allowed the film industry to continue producing higher quality films. An example of the power of ray tracing is shown in Figure 1, which is a computer-generated scene rendered using ray tracing with many advanced lighting components.

2 Background

One of the foundational tasks in computer graphics is rendering three-dimensional objects, which takes geometric objects arranged in three-dimensional space and produces a two-dimensional image that shows the objects as viewed from a certain viewpoint [2]. This process takes as input a sete of 3-D objects and produces as output a 2-D array of pixels. Rendering can occur in two orders – object-order rendering and image-order rendering. In object-order rendering, each object is processed and the pixels that it influences are calculated and updated. In image-order rendering, each pixel is processed, and the objects that influence it are determined and the pixel value is found. Ray tracing is an image-order rendering algorithm.

As an image-order rendering algorithm, the ray tracing algorithm begins by iterating over each pixel in the raster image. For each pixel, the objective is to determine the geometric object observed by the ray emanating from the viewpoint in the direction of the pixel. The object of interest is the object nearest the viewpoint because it occludes any objects farther into space along the ray's path. After the object intersection is determined, the shading computation is performed. The complexity of the shading model can vary, but an increase in complexity generally produces a higher quality rendering.

Thus, a ray tracer has three core components:

1. ray generation, which computes the origin and direction of each pixel's viewing ray;
2. ray-object intersection, which finds the closest object intersecting the viewing ray;
3. shading, which computes the appropriate pixel color based on the results of the ray-object intersection [2].

These components are examined in more detail in Sections 2.1-2.3.

One large drawback to ray tracing in its simplest form is the computation cost per raster image. For a simple ray tracer, the performance depends on the number of rays being cast, which is influenced by the following:

1. raster image resolution,
2. multisample anti-aliasing level,
3. number of objects and lights in the scene, and
4. number of reflection bounces allowed.

This performance quickly becomes undesirable for anything but the simplest scenes. Techniques exist to improve this performance, but are not discussed in this document.

As demonstrated in Figure 1, ray tracing is capable of generating incredibly realistic scenes. This is done through complicated shading models that attempt to model the physical interactions of objects and light.

2.1 Ray Generation

The concepts of ray tracing are rooted in the physical propagation of light rays throughout the environment. In reality, the light rays propagate from the light source throughout the scene until all the photonic energy has been absorbed. This method of propagating rays is called *forward ray tracing*, and it has been implemented in computer graphics through Monte Carlo simulations. The advantage and disadvantage to this method is that the scene lighting is independent of the viewport. The light propagates throughout the scene until an equilibrium is reached. Thus, the advantage is that the camera can be positioned at any location and readily obtain the correctly lighted scene. The disadvantage is that it can take numerous iterations to reach a lighting equilibrium, and many light rays are unused because they never intersect the viewing plane.

An alternative method, called *backward ray tracing*, was developed to improve the rendering speed while maintaining the realistic lighting. In backward ray tracing, the theory is that only the light gathered at the camera is necessary, and it is unnecessary to light the entire scene to generate one raster image. Therefore, instead of shooting rays from each light source, the rays are shot from the camera through each pixel in the viewing plane. Because of its computational advantages, backward ray tracing has become the more common approach, and is usually the method referred to in the generic 'ray tracing' term. A comparison of the two methods of ray tracing is shown in Figure 2.

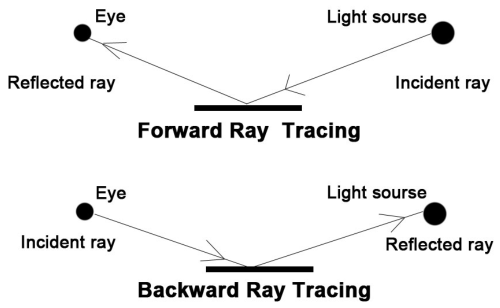


Figure 2: A comparison of forward and backward ray tracing methods. [3].

During the Italian Renaissance, artists began designing using the principle of linear perspective to increase the realism of their work. The same principles are used today in computer graphics for the same reasons. Ray tracing can naturally incorporate one-point perspective projection by casting each ray from some camera point e through each pixel in the 2-D viewing plane. This process is seen in Figure 3.

Generating a ray requires two pieces of information – the ray origin and the ray direction. To simplify the ray generation process, the scene is processed in camera space. This means that the camera is located at the origin e and is looking down the $-w$ axis with the image plane aligned with the uv plane. The left and right image dimensions are defined by the edges of the image, l and r, along the u axis. The top and bottom image dimensions are defined by the edges of the image, t and b, along the v axis. Since the image plane is generally centered about e, $l < 0 < r$ and $b < 0 < t$. The image must fit $n_x \times n_y$ pixels into the rectangle of size $(r-l) \times (t-b)$ with each pixel represented by its center. Therefore, the (u, v) image plane coordinates of a pixel (i, j) in the raster image are calculated by

$$\begin{aligned} u &= l + \frac{(r-l)(i+0.5)}{n_x} \\ v &= b + \frac{(t-b)(j+0.5)}{n_y}. \end{aligned} \tag{1}$$

Additionally, in perspective-based ray tracing, the image plane is offset from the camera location by some focal length d. With this information, the primary rays for the raster image can be generated using the following procedure:

1. Compute u and v from Equation (1)
2. The ray direction is $-dw + uu + vv$ or, alternatively, $(-d, u, v)$ in camera space.
3. The ray origin is e.

2.2 Ray-Object Intersection

Once the primary viewing rays have been generated, the ray must be checked against the scene geometry to find the geometric object closest to the viewing plane. Many geometric objects allow for ray intersection calculations, but this project focused on triangle intersections. Recalling that the equation for a ray is

$$p = e + td, \tag{2}$$

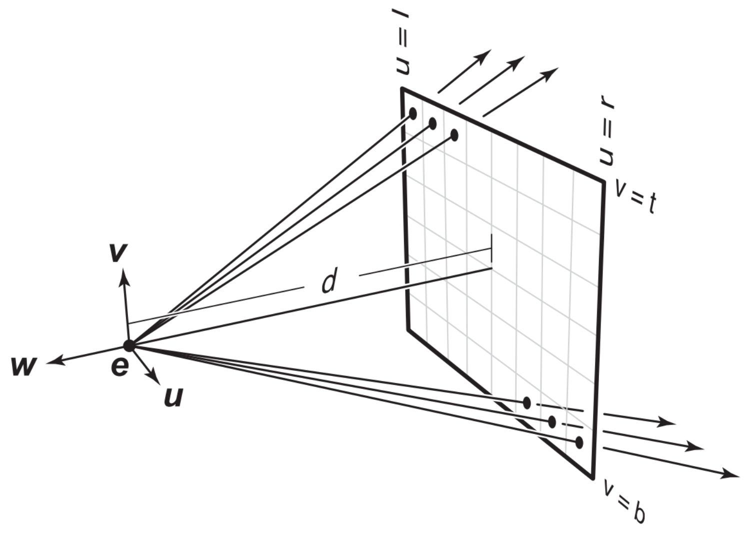


Figure 3: Primary ray generation for perspective projection. [2]

where any point on the ray is calculated by adjusting t. Thus, the objective of ray-object intersection is to determine the value t, if it exists, for each geometric object. One method of calculating ray-triangle intersection is to utilize Barycentric coordinates. Thus, the intersection of a ray with a triangle is

$$e + td = a + \beta(b-a) + \gamma(c-a), \tag{3}$$

where t, β, and γ are unknown values to calculate. In the Barycentric approach, the vertices of the triangle map out the plane that contains the triangle. Therefore, solving Equation (3) can be broken down into two operations – determining the offset t where the ray intersects this plane and if that intersection point is inside the triangle. The intersection point is inside the triangle iff $\beta > 0$, $\gamma > 0$, and $\beta + \gamma < 1$. If no solutions exist, then the ray is parallel to the triangle plane or the triangle is degenerate.

To solve Equation (3) for the unknown variables, it can be rewritten as the system of linear equations

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}. \tag{4}$$

The variables can be solved for using Cramer's rule. Substituting for the standard column-order dummy variables and solving gives

$$\beta = \frac{j(ei-hf) + k(gf-di) + l(dh-eg)}{M} \tag{5}$$

$$\gamma = \frac{i(ak-jb) + h(jc-al) + g(bl-kc)}{M} \tag{6}$$

$$t = -\frac{f(ak-jb) + e(jc-al) + d(bl-kc)}{M}, \tag{7}$$

where

$$M = a(ei-hf) + b(gf-di) + c(dh-eg). \tag{8}$$

The speed of this algorithm can be improved by using early termination. Since the ray offset t must fall within some bounds, t may be computed first to determine if the intersection occurs at a point closer to the ray origin than already seen. If not, then β and γ are not calculated. Additionally, early termination may be performed on γ to ensure $\gamma \in (0,1)$.

Using the Barycentric coordinates where $\alpha = 1 - \beta - \gamma$, interpolation can be performed at the intersection point. This interpolation allows the intersection point to return its interpolated position, normal vector, and colors. These values are useful in the next stage of ray tracing – shading.

2.3 Shading Model

The Blinn-Phong shading model is used to determine the color at each pixel of a raster image. For ray tracing, the total light intensity I received at the surface is

$$I = I_a + I_d + I_s + I_e + I_r + I_t, \tag{9}$$

where

$$\begin{aligned}
I_a &= k_a \left(I_{a_{global}} + \sum_i I_{a_i} \right) && (10)\\
I_d &= \sum_i \frac{k_d}{a + bD_i + cD_i^2} \left(l_i \cdot n \right) I_{d_i} && (11)\\
I_s &= \sum_i \frac{k_s}{a + bD_i + cD_i^2} \left(n \cdot h_i \right)^s I_{s_i} && (12)\\
I_e &= k_e && (13)\\
I_r &= \frac{k_r}{a + bD_r + cD_r^2} I\left(u_r \right) && (14)\\
I_t &= \frac{k_t}{a + bD_t + cD_t^2} I\left(u_t \right). && (15)
\end{aligned}$$

The vectors are shown in Figure 4, and

$$\begin{aligned}
h &= \frac{l_i + v}{\|\|l_i + v\|\|} && (16)\\
u_r &= u - \left(2u \cdot n \right) n && (17)\\
u_t &= \frac{\eta_i}{\eta_t} u - \left(\cos\theta_t - \frac{\eta_i}{\eta_t} \cos\theta_i \right) n. && (18)
\end{aligned}$$

An advanced version of this shading model is shown in Figure 5, but not used in this project.

3 Results

The original goal for this project was to render the MINI Cooper model used during class in the Cornell box. However, due to the amount of rendering time required, the MINI Cooper model has not completed rendering in time for this report. In it's place, a reflective box was placed in the Cornell box.

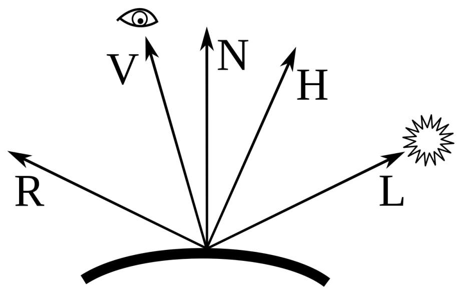


Figure 4: The vectors associated with the Blinn-Phong shading model [4].

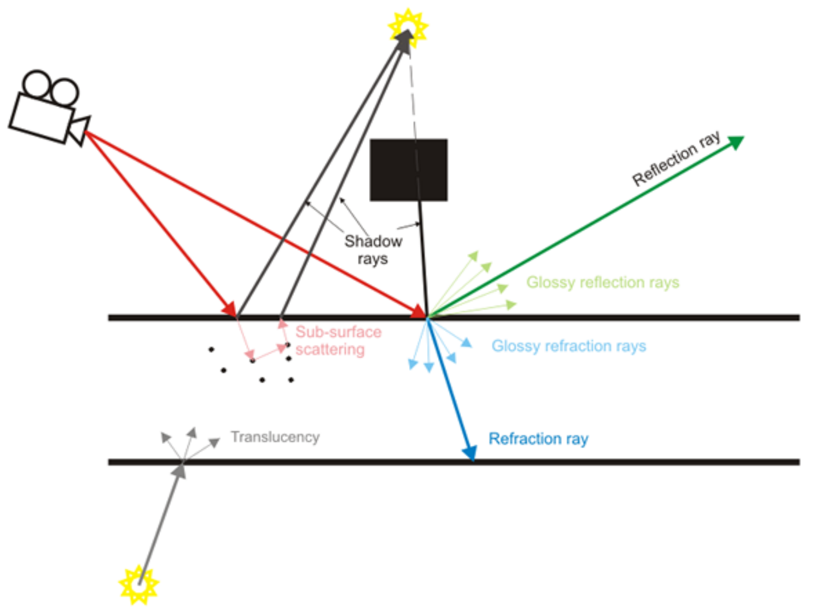
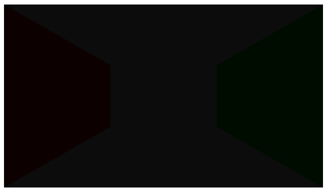


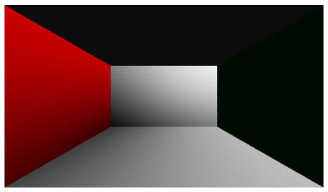
Figure 5: Shading vectors associated with advanced shading models [5].

The first step in the process was generating the Cornell box with simple ambient lighting. Numerous difficulties arose here due to generating the box geometry by hand. First, the bottom and top planes of the box were accidently placed at different heights than the walls, which led to an awkward gap. After fixing this, a ray-traced Cornell box with ambient lighting was generated, shown in Figure 6a. To improve the render quality, 16x multisample antialiasing was added and used for all the images shown in this report.

The next step was to add diffuse lighting to the raytracer. Diffuse lighting requires a light source. For simplicity, the first light source implemented was a directional light source. The light source for the following renderings was oriented toward $[-1, -1, -1]$. While attempting to add diffuse lighting, it was discovered that the hand-generated point normals for the Cornell box model were incorrect. This led to incorrect diffuse lighting, and difficulty in debugging. However, after this problem was remedied, the Cornell box was rendered again with ambient and diffuse lighting from a directional light, shown in Figure 6b.

To complete the standard Phong lighting model, specular lighting was added. One difficulty encountered here was that the pixel colors were now exceeding one, which was causing a wrap around to occur when


(a) Ambient


(b) Ambient & Diffuse for Directional Light

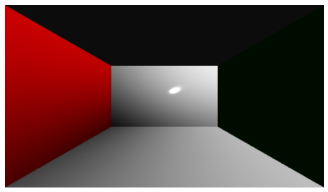
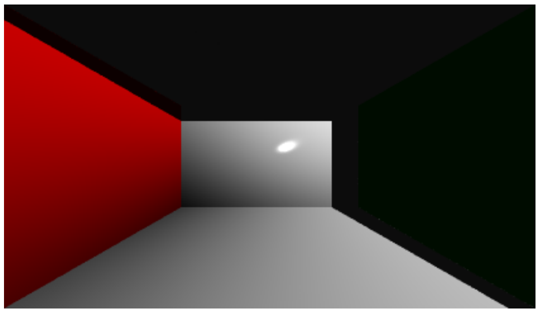

(c) Phong Lighting for Directional Light

Figure 6: The development of the ray tracing algorithm with the Cornell box.


(a) Ambient

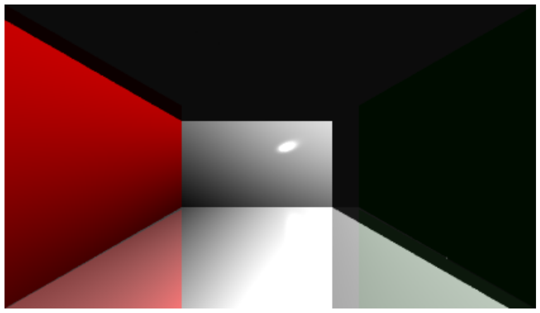

(b) Shadows & Reflections for Directional Light

Figure 7: The addition of more complex lighting.

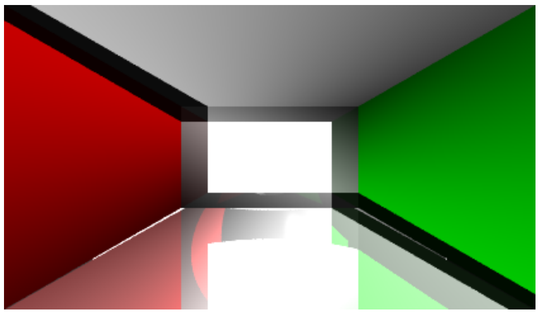
converting from float64 to uint8. This caused the specular areas to appear black instead of white. This was remedied by clamping all values over 1.0 to 1.0. The result is shown in Figure 6c.

With Phong light stably implemented, the next step was to use the power of ray tracing to generate shadows. The result of adding shadows with a directional light is shown in Figure 7a. Due to the sharp edges in the scene and simple method used for calculating shadows, the shadows look unrealistic. A better approach would have been to use soft shadows by multisampling the shadow feeler ray.

The next task was to generate reflections. This required an extension to the geometry framework to incorporate the reflective properties of the materials. For the Cornell box, the bottom and back faces were set as reflective. The integration of reflections went fairly smoothly, and the results of adding reflective lighting to the Cornell box is shown in Figure 7b

With the demonstration of a successful lighting model, the raytracer was expanded to process multiple lights. The addition of multiple lights was simple because it had been designed for from the start. The only change made was that more lights were added to the scene when it was loaded. The light sources for Figure 8a were $[-1,-1,-1]$ and $[1,1,-1]$. The light sources for Figure 8b extended the two ligth case by adding a light towards $[0,1,-1]$.

Finally, to increase the complexity of the scene, a reflective box was added inside the Cornell box. This process was merely tedious to generate another set of geometry for the box. The result was impressive though, and can be seen in Figure 9.


(a) Two Directional Lights

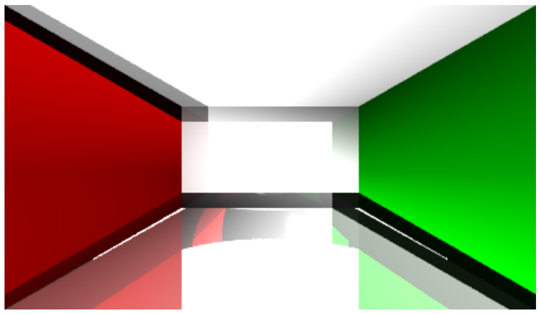

(b) Three Directional Lights

Figure 8: The addition of more complex lighting.

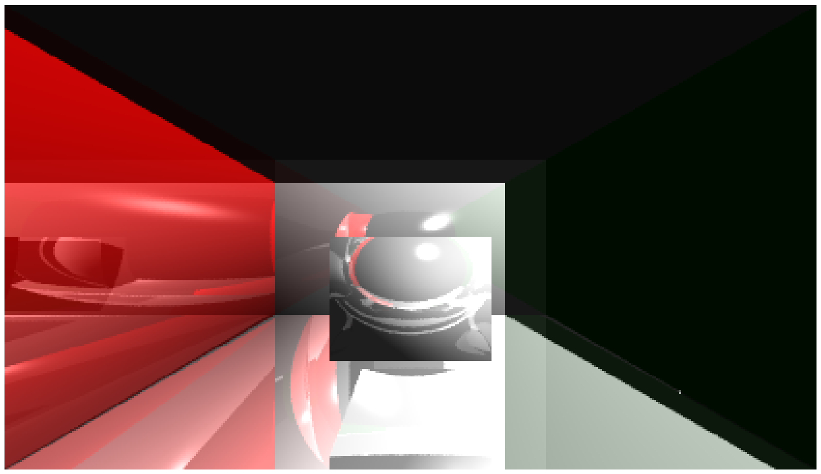


Figure 9: Reflective multibox rendering with 16xMSAA and a single directional light source.

4 Conclusion and Future Work

In conclusion, this project successfully demonstrated some of the fundamental features of ray tracing for producing 2-D raster images from 3-D scenes. While the final results used a simpler model than desired, it still demonstrated the lighting effects intended.

The future work for this project can be split into three categories – optimization, usability, and realism. Work in the optimization category attempts to improve the performance of the program in speed or memory. The usability category attempts to improve the compatibility of the program with various model formats. Improvements to realism attempt to render the scene using more advanced techniques to generate higher quality images. The following list shows a breakdown of the future work for this project:

4.1 Optimization Improvements

Bounding Box Intersections This improvement checks each mesh object by its bounding box before iterating over the contained geometry. Using this approach, more complex scenes can be rendered faster because each geometric object is not checked for each ray.

Multiprocessing This improvement is to parallelize the primary ray casting. Each primary ray is independent of all other primary rays, so multiple pixels may be calculated in parallel. The simplest implementation here is to utilize the central processing unit cores available. More advanced implementations could leverage graphics processing units to achieve much higher parallelization at the cost of redesigning to account for the memory management necessary.

Spatial Search Structures Using a data structure that supports spatial search can improve the search time for geometric objects during the intersection test. One commonly used structure is Kd-trees.

4.2 Usability Improvements

Additional Geometries Currently, the program lacks the ability to process anything except triangles. The addition of various geometric primitives would increase the usability of the program. Using the appropriate geometric representation of objects can also increase render speed. For example, a circle composed of triangle is a lot of objects to check for intersection, whereas a circle object would be only one intersection check.

Extended File Formats The program lacks the ability to load models from standard file formats. This improvement would allow the ray tracer to interact with current modeling software more easily, which allows for more complex scene generation.

4.3 Realism Improvements

The ray tracer can be expanded to do soft shadows and more advanced lighting models. Additionally, higher fidelity models are required to calculate some lighting effects.

References

[1] B. Caulfield, “What’s the difference between ray tracing and rasterization?,” Mar 2018.

[2] S. Marschner, *Fundamentals of Computer Graphics*. A K Peters/CRC Press, dec 2015.

[3] “Ray tracing algorithm: Different types and uses of ray tracing algorithm,” Dec 2019.

[4] M. Kraus, “Blinn vectors,” Jun 2011.

[5] “Basic ray tracing.”

Assignment Code

The source code for this assignment is hosted on GitHub.

`https://github.com/WhoFama24/cse6413-raytracer.git`