The Development of a Ray Tracer

Emiliano Hinojosa Guzmán

June 6th, 2024

Introduction

Ray tracing is a method in computer graphics used to simulate lights, objects, and materials. During this project, I tackle the challenge of building my own ray tracer from scratch. In this report, I will explain the many development stages my ray tracer went through and I'll go over its inner workings.

Starting with basic functionality and simple shapes, I progressively enhanced the ray tracer by adding features such as different types of lights, more complex object shapes, and various shading techniques. Each version represents a step forward in complexity and capability, illustrating both the technical challenges encountered and the solutions implemented.

Some of the advancements were made as a group, others were individually. Throughout this report, you will find detailed descriptions of each version, from the initial, rudimentary implementation to the sophisticated, feature-rich final version.

Version 0.1

The first version of my ray tracer was a very rustic one. You would open a blank scene, add your camera, and the objects. The camera would have an origin, a resolution, and a field of view. The ray tracer class would take the resolution of the camera and, pixel by pixel, calculate the direction a ray needed to be cast from the camera onto that direction. Once we have the direction needed, we cast a ray and check if it happens to intersect with any of the objects currently in the scene. In this version, the only available objects are spheres. To calculate the intersection with a sphere, I used the geometric solution. If a ray happened to intersect with the shape, then it would return an Intersection object which stored information about the ray-sphere intersection; otherwise, nothing was returned. Information such as t0, t1, and the color of the object. If the ray misses all the shapes in the scene, the intersection returned is null, and we paint the pixel our background color, in this case, white. If the ray intersects with at least one shape, we update the intersection to contain the information of the closest intersection. Then, after having cycled over all the shapes, if the intersection returned was in fact not null, then we would paint the pixel the color of the shape with the closest intersection. These are the principles under which the first version of my ray tracer would operate and would serve as guidelines to build on top of.

Figure 1: V0.1

Version 0.2

The second version of the ray tracer was developed as a group activity. During this version, we built a better ray tracer with the same objective as version 0.1. The main differences between this version and my version 0.1 were that the intersection would now store coordinates, distance, the normal vector at the point of intersection, and a reference to the intersected object. Another significant change was the implementation of clipping planes. These planes are meant to ignore anything beyond them so that we don't have to perform additional costly operations.

Figure 2: V0.2

Version 0.3

Most 3D models are based on triangles since it is very easy to detect intersections with them and it is the shape with the fewest edges that can form any other shape. For the third version of the ray tracer, we were tasked with implementing triangles as a means to later be able to import OBJ models. For this, I developed the class Triangle. This class stores a position, 3 vertices to describe its shape, and a color. The way this version calculates triangle intersections is by first defining our ray and the three vertices of our triangle.

Using the Möller-Trumbore algorithm, we calculate if the ray intersects the triangle by solving a system of linear equations. This involves computing vectors, cross products, and dot products to check if the intersection point lies within the triangle's bounds and how far along the ray this intersection occurs. If all conditions are met, we get the exact intersection point and return it in the form of an Intersection.

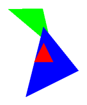

Figure 3: V0.3

Version 0.4

For the fourth version of my ray tracer, we implemented as a group an OBJ file reader. This works by reading the file line by line, parsing vertex (v) and normal (vn) data into lists. When it encounters face (f) data, it uses the vertex indices to create Triangle objects. These triangles are then used to construct a Model3D object with a specified origin and color. If any errors occur during reading, it catches and prints the exception, returning null if unsuccessful.

Figure 4: V0.4

Version 0.5

For the fifth version of my ray tracer, I implemented directional lights and flat shading for lambertian surfaces. For the lights, I implemented an abstract class Light from which all lights will inherit. All lights have an intensity and a light color. Then I created the class Directional Light whose lambertian reflection is the dot product of the intersection normal and the light's direction. Light has an abstract method to calculate diffuse by using the information from the lambertian surface.

Version 0.6

For version 0.6, we implemented smoothing groups by reading an .obj file, parsing vertex and face data, and keeping track of which triangles belong to which smoothing group. After

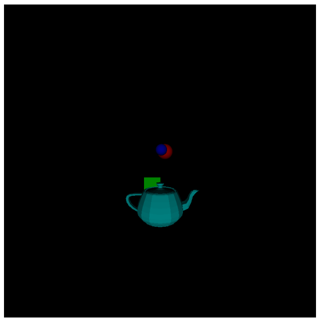

Figure 5: V0.5

reading all data, it averages the normals for vertices that are shared by multiple triangles within the same smoothing group, ensuring smooth transitions across these triangles. This enhances the visual quality of the rendered model by providing smoother shading and eliminating sharp edges between triangles that should appear smooth.

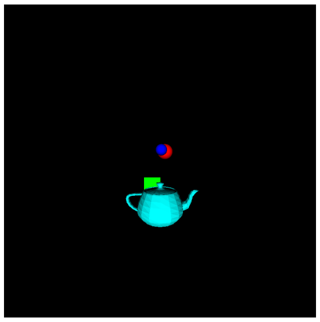

Figure 6: V0.6

Version 0.7

For version 0.7, we implemented spotlights. This was a very simple step since we could simply copy and modify the directional light. Instead of having a direction, the spotlight has a position. The logic to calculate diffuse color is now handled by the ray tracer class and the way to calculate NDotL is overridden.

Version 0.8

By version 0.8, we no longer had access to the code developed in class, so we were on our own. We were tasked with implementing light falloff and hard shadows. I implemented these features quite easily. For the light falloff, I just divided the intensity at the point of intersection by the distance squared. For the hard shadows, if I cast a ray from the intersection to every light and the intersection is not null, it means there is something in the path from the point to the light, which means the pixel is in the shadows and is painted the background color.

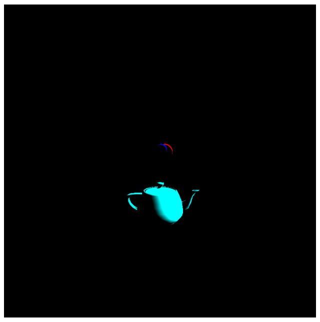

Figure 7: V0.7

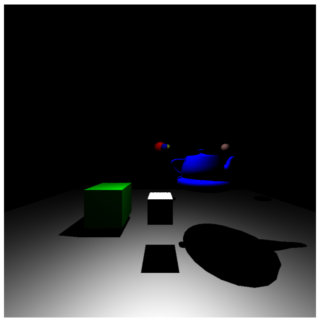

Figure 8: V0.8

Version 0.9

Version 0.9 is when things get interesting since we had no more backup from the group and had full creative control over our ray tracer. The main tasks to face with version 0.9 are parallelization, reflection, refraction, and the way different materials are handled.

I tackle parallelization by using Java's ExecutorService. Instead of processing each pixel sequentially, the image is divided into smaller tasks that can be executed concurrently by multiple threads. An ExecutorService is created with a fixed thread pool based on the number of available processors. Each pixel's color calculation is turned into a Runnable task and submitted to the executor for parallel execution. Once all tasks are submitted, the program waits for the executor to finish processing all tasks. This approach utilizes multiple CPU cores to render the image faster by dividing the workload across threads, improving performance significantly.

Reflection is handled by calculating a new ray that bounces off a surface at the intersection point, like a mirror. This is done by reflecting the incoming ray direction using the surface normal. A new reflected ray is created and traced to see what it hits next. The color from this new intersection is combined with the original object's color, simulating how light bounces off reflective surfaces and creating realistic mirror-like effects.

Refraction simulates how light bends when it passes through transparent materials like glass or water. When a ray hits a transparent surface, a new refracted ray is calculated based on the surface normal and the material's refraction index. This refracted ray bends according to

Snell's Law, and the program traces it to see what it hits next. The color from this refracted ray is combined with the object's color, simulating the bending and passing of light through transparent objects, creating realistic effects like the bending of light in water.

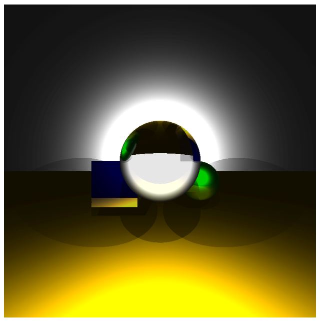

Figure 9: V0.9

Version 0.10

For the last and final version of my ray tracer, I dedicated myself to making it look more realistic, mostly when it comes to materials. That's why instead of adding biased weights for the reflection-refraction combination, I implemented reflectance through Fresnel's equations. I also added rotation and scaling for 3D Models so that I could place objects in a more natural way within my scenes.

0.1 My Biggest Challenges in Building the Ray Tracer

When I embarked on the journey of building a ray tracer from scratch, I knew it would be a complex and demanding project, but I didn't fully grasp the extent of the challenges that lay ahead. As someone who has never been particularly strong in geometrical analysis, the foundational aspects of ray tracing presented a significant hurdle. Understanding and implementing the intricate mathematics involved in calculating intersections, normals, and lighting was a steep learning curve.

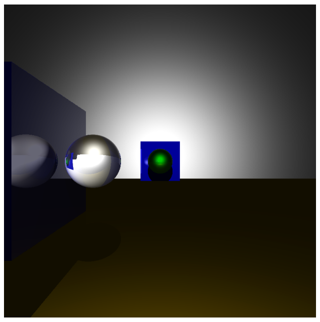

Figure 10: V0.10

0.1.1 Geometrical Analysis

Geometrical analysis forms the backbone of any ray tracing algorithm. From calculating the intersection points between rays and objects to determining surface normals and light reflections, every step requires precise mathematical computations. Initially, I struggled with concepts such as vector mathematics, dot products, and cross products. These are essential for understanding how light interacts with surfaces. I had to revisit and reinforce my understanding of these fundamental mathematical concepts, often feeling like I was starting from the basics.

0.1.2 Adding Fresnel Equations

A particularly challenging aspect was the addition of Fresnel equations to enhance the realism of reflections and refractions in the scenes. The Fresnel effect describes how light behaves at the boundary between two different media, changing the amount of reflection and refraction based on the angle of incidence. Incorporating Fresnel equations required a deep dive into complex mathematical concepts and physical optics principles. Implementing these equations correctly involved understanding how to blend reflection and refraction based on the angle of incidence and the material properties.

0.1.3 Overcoming the Challenges

Despite these challenges, I persisted. I broke down each problem into smaller, manageable tasks, focusing on understanding one concept at a time. I leveraged online resources, tutorials, and academic papers to build my knowledge base. Through continuous practice and debugging, I gradually became more comfortable with the geometrical and mathematical aspects of ray tracing.

Implementing Fresnel equations was particularly rewarding, as it significantly improved the visual realism of the rendered scenes. It required careful tuning and testing to ensure the equations accurately represented the physical phenomena, but the end result was worth the effort.

Conclusion

Final Renders

The Empty Man

I made this render mainly thinking about how no man to ever have walked this Earth is complete by himself. We all strive for community; it's such an important thing to our nature that without the views and opinions of others (represented by the spheres), we are not complete.

In this render, reflection can be visualized as many of the spheres play with reflection and different reflection indexes, simulating several materials.

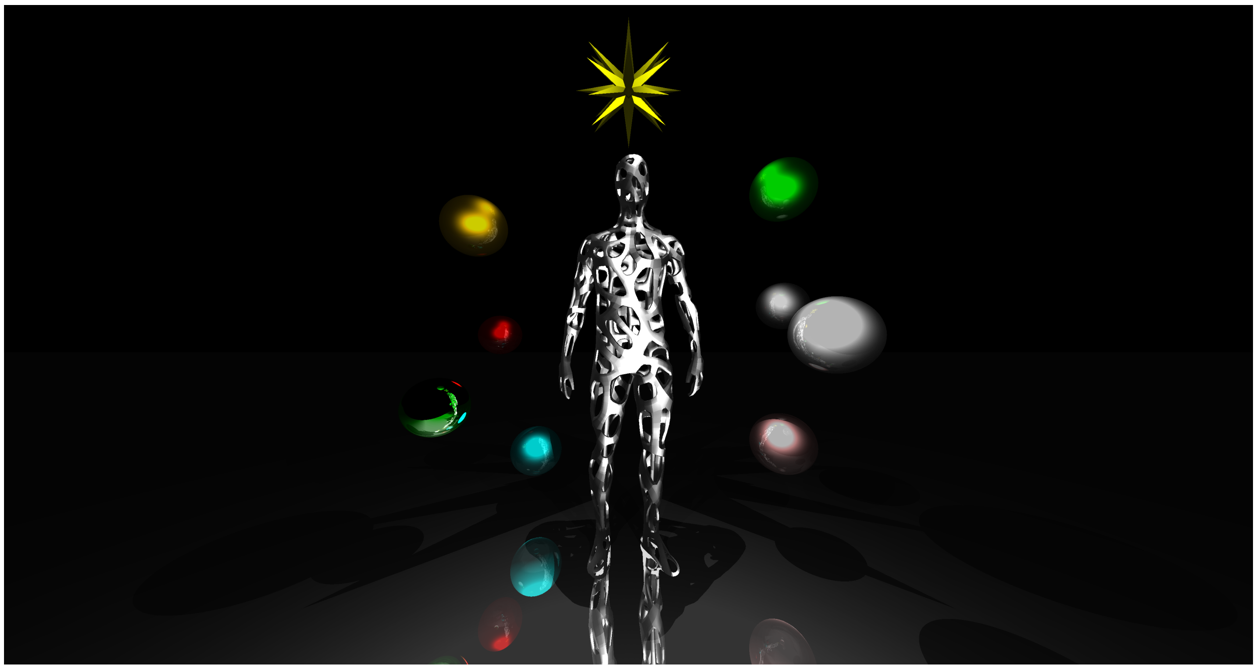

Figure 11: The Empty Man

He Who Pours

I made this render about the finite resources we have on this earth. A gargoyle is supposed to be a guardian against evil, but what happens when the gargoyle itself is the one causing mischief? The gargoyle represents our government, turning a blind eye to the evil caused, or even facing it and deciding to ignore it, failing its most sacred vows.

In this render, reflection can be visualized on the teapot, the tables, and even the ground.

The Algorithm

I made this render as a social commentary. The different colored panels are meant to represent a TV or any modern media by default. We were once free to decide the kind of media we wanted to consume, the type of culture we would surround ourselves with. With modern social media, that's no longer an option. There is only a feed of videos we scroll through decided by an algorithm. Yes, we can modify the feed to fit our desires, but it's no secret that this algorithm is also prone to falling into bubbles with narratives that may or may not benefit certain individuals. It really makes you think about what's behind these algorithms. The sphere in the scene represents a lens, and through that lens, what happens in the media is distorted and reveals what usually would be a sign of hope (star) into what seems to be an evil man. In this render, refraction can be visualized on the sphere.

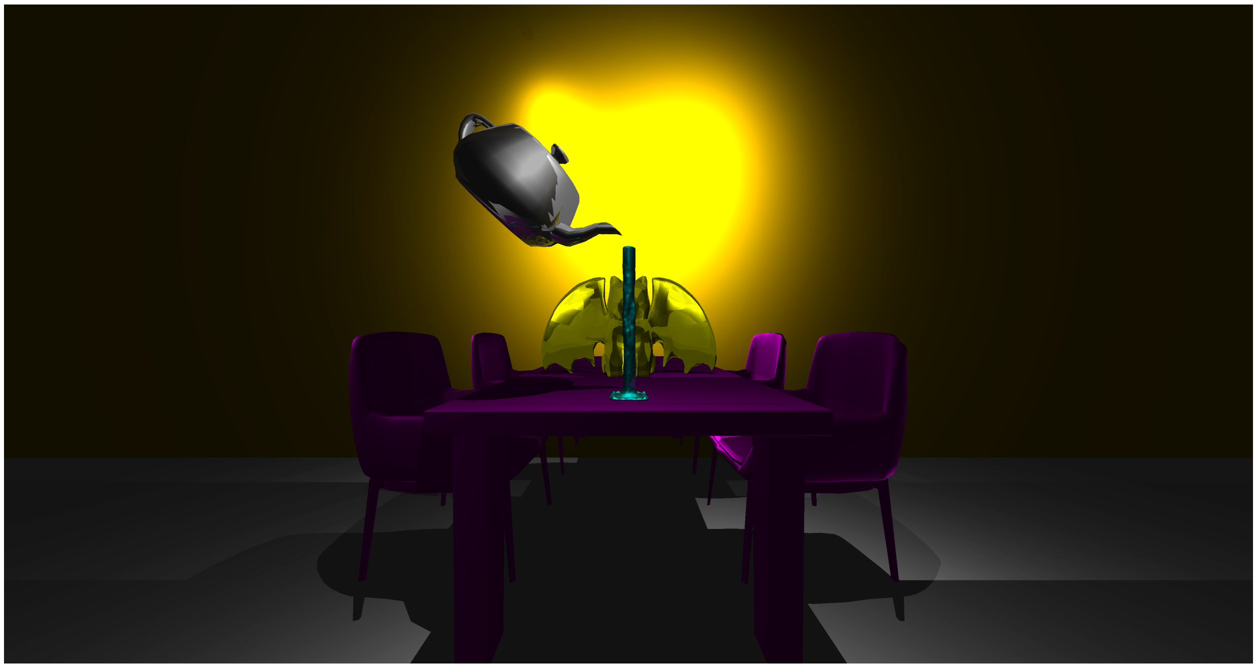
Figure 12: He Who Pours

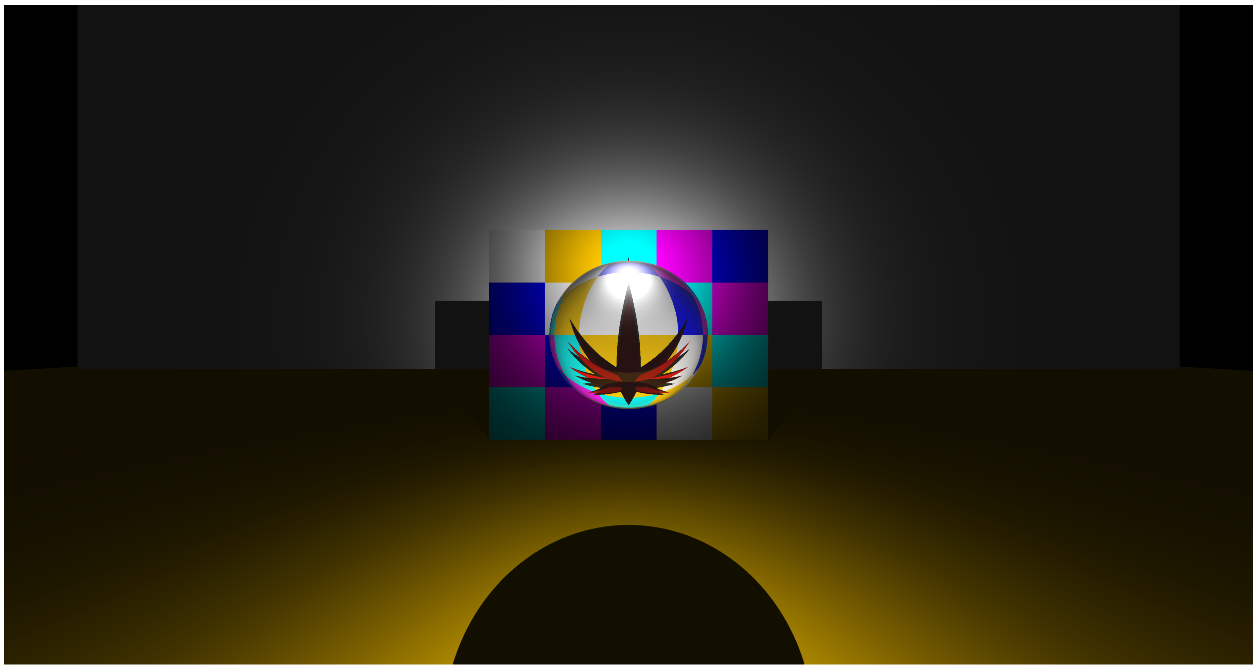
Figure 13: The Algorithm