# CSE306 Project 1 Report - Ray Tracer

Arjun Bommadevara

May 2024

## 1 Introduction

This project focuses on the development and implementation of a path tracer using C++, creating a scene with various geometric objects, implementing light interaction models, and using efficient algorithms for intersection testing and light transport simulation. The path tracer generates images by simulating numerous light rays per pixel, achieving high realism. Path tracing simulates the behavior of light as it interacts with surfaces in a scene, capturing lighting effects like soft shadows and global illumination. In our project, we consider both direct and indirect light sources by recursively tracing rays.

All the code was run on a MacBook Air, with the apple silicon M1 chip.

## 2 Spheres

### 2.1 Ray-Sphere Intersection

In the initial phase of our path tracing project, we focused on implementing a basic ray-sphere intersection algorithm to validate fundamental computations and establish a solid foundation for more complex scenes. We placed a single sphere at the center of the scene. We then checked whether a ray intersects the sphere. If a ray intersected the sphere, the corresponding pixel was colored white; otherwise, it remained black. The resulting image, a simple white sphere on a black background, confirmed that our ray-sphere intersection calculations were functioning correctly. The time taken to get this image was 150ms.

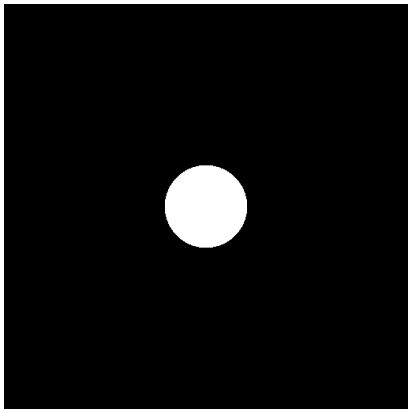
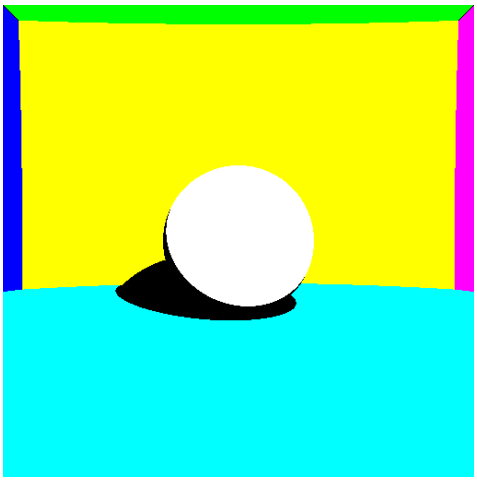


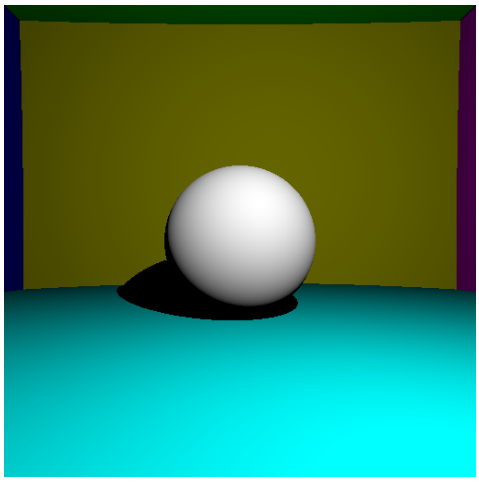
Figure 1: Ray Sphere Intersection

The next step in the was to create a more complex scene, where we have a ball in a room. A scene featuring a central white sphere surrounded by walls of various colors, effectively demonstrating the interaction of light with different surfaces. The shading and shadows computation was done using the Lambertian model as described in the lecture notes.

Initially,the scene was rendered without applying gamma correction, as shown in Figure 2a below. In this rendering, the colors appear very saturated and unrealistic. Figure 2b below showcases the scene with gamma correction applied. The colors are richer and more natural, and the overall lighting appears more realistic. The shadow beneath the sphere and the color gradients on the walls are improved through gamma correction. Figure 2a took 350ms to compute and Figure 2b took 390 ms. They were both computed with $I = 2 * 10^{10}$.



(a) No Gamma correction



(b) Gamma corrected

Figure 2: First Rendering of a Scene

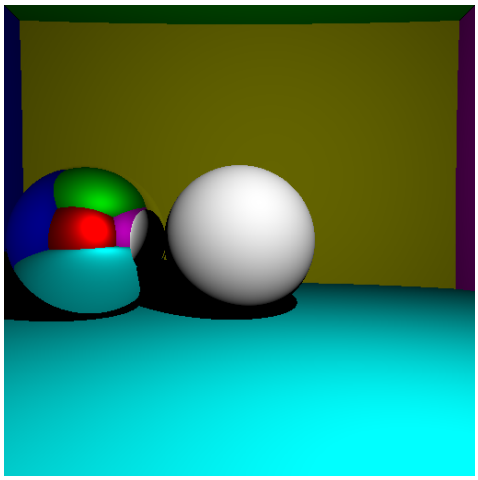
### 2.2 Sphere textures

The next step in our path tracing project was to add surfaces with reflective (mirror) and transparent properties, aiming to simulate more complex light interactions and enhance realism.

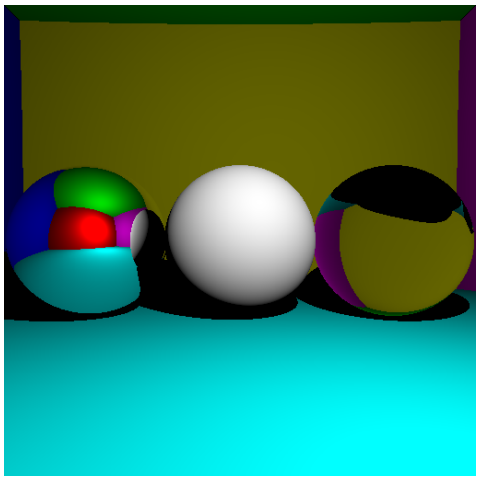
To implement mirror surfaces, we used the law of reflection, as described in our lecture notes. Figure 3a shows the scene with a reflective sphere added to the left of the central white sphere, and the mirror clearly shows the surrounding environment in the sphere's reflections.

For transparent surfaces, we applied Snell's Law to calculate light refraction as it passes through different media, taking into account the refractive indices. Figure 3b includes both a reflective and a transparent sphere, positioned to the left and right of the central white sphere, respectively. The transparent sphere correctly demonstrates light bending and partial internal reflection, adding another layer of realism to the scene.

Figure 3a took 436ms to render, and Figure 3b took 566ms to render.



(a) Mirror Sphere



(b) Mirror Sphere and Transparent Sphere

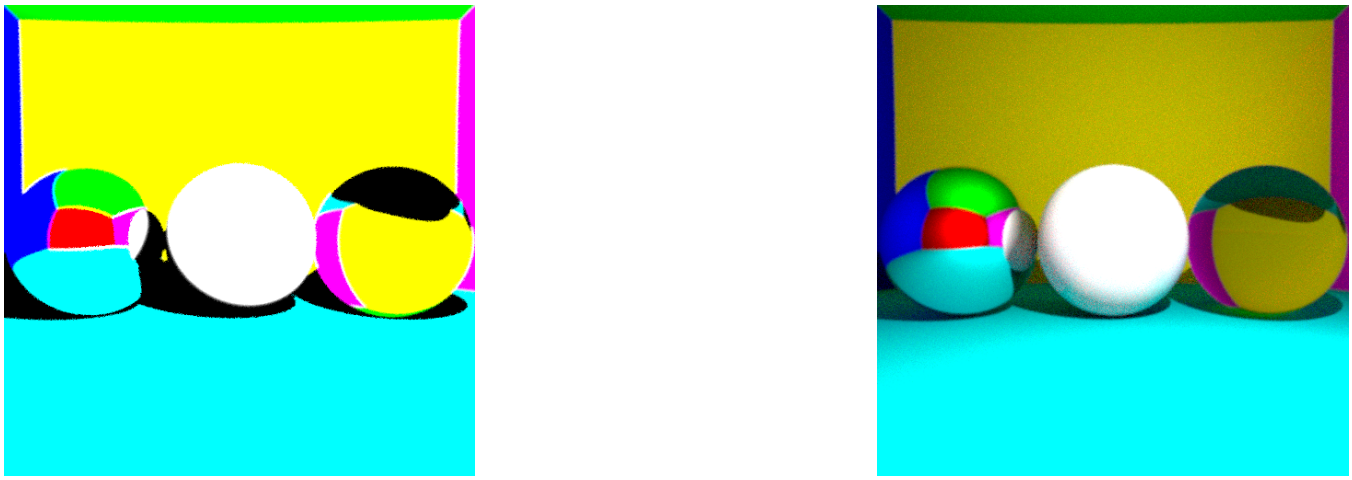
Figure 3: Reflective and Transparent surfaces

#### 2.2.1 Indirect Lighting and Antialiasing

In the next phase of our project, we focused on implementing indirect lighting and antialiasing to enhance the realism of our renders. Indirect lighting simulates the effect of light bouncing off surfaces, while antialiasing reduces jagged edges, resulting in smoother images. The random cos function I used was heavily inspired from one I found on the internet, because my own implementation had unexplained red lines in the middle of the image which i did not understand why they appeared.

Figure 4a shows the indirect lighting effect, before we implemented antialiasing, and Figure 4b

shows it after we implemented antialiasing. As we can see, there is a huge change in realism and light saturation after we implement the antialiasing. Without parallelisation, it took around 15 minutes to generate each image. After parallelisation, it took around 2 seconds. These images were computed with 50 rays per pixel. There is a bit of noise in these images.



(a) No antialiasing

(b) After antialiasing

Figure 4: Indirect Lighting and Antialiasing

The number of rays cast also played quite a difference in computation time. Without Parallelisation, it took around 45 minutes to compute 1000 rays per pixel, while with it took around 4 minutes with parallelisation. Figure 5 below shows 1000 rays per pixel, and we see much less noise.

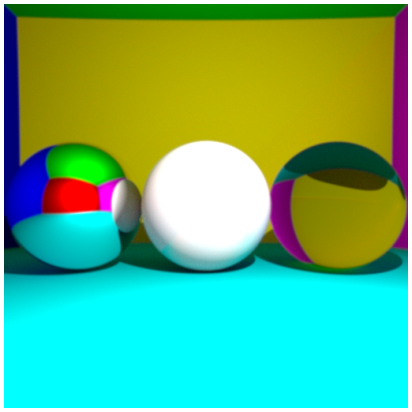


Figure 5: 1000 rays per pixel

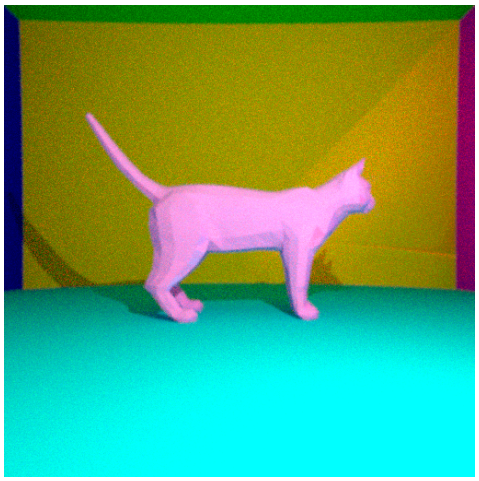
The final step in our path tracing project involved implementing mesh intersection and integrating a complex 3D model: a cat.

To achieve this, we utilized the Möller–Trumbore intersection algorithm, the method described in lecture notes for determining if a ray intersects a triangle. In our implementation, the intersect

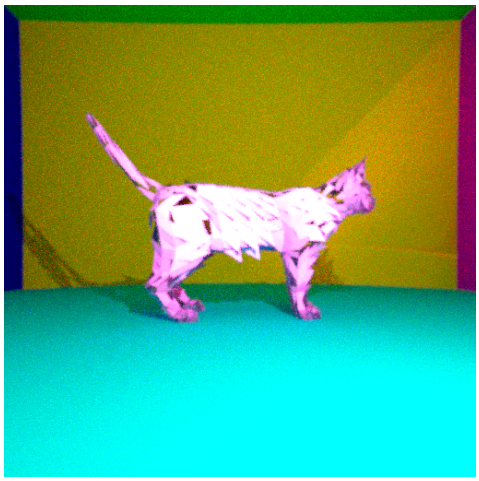
method of the TriangleMesh class uses this algorithm to check for intersections between rays and the triangles that make up the mesh. When a ray intersects a triangle, the algorithm provides the exact intersection point and the corresponding surface normal.

Figure 6a shows the pink cat model integrated into the scene with realistic lighting and shadows, confirming the successful implementation of mesh intersections. The detailed geometry of the cat is rendered accurately, demonstrating the robustness and versatility of our path tracer.

Without BVH, this took around 18 minutes to render. I attempted to implement BVH, which took around 2min to run, however, I faced issues that I have not been able to fix. As seen in Figure 6b, although it looks very cool, the cat has major geometry issues, so therefore I decided to omit BVH from my final code. I will try my best to implement BVH correctly in the next few days, and I will upload an updated file to the github when I do so.



(a) Mesh Intersection with Cat



(b) BVH attempt

Figure 6: Mesh Intersection and BVH attempt