Ray Tracing Using Python

Prince Verma
University of Southern California

ABSTRACT

Ray tracing rendering technique emphasizes its ability to simulate realistic interactions between light and various materials. We implemented a ray tracing system from scratch using Python and extensively utilized the Numpy library to optimize performance on CPU architectures. Our system meticulously models the behaviors of different material types—Lambertian, Metallic, and Dielectric—demonstrating their unique reflective and refractive properties through a series of controlled simulations. Key elements such as vector operations for 3D graphics, advanced ray and camera class functionalities, and sophisticated scene management techniques were developed to enhance the realism and efficiency of our rendering engine. Results from the study vividly illustrate how different materials affect light behavior, including diffuse scattering, sharp reflections, and complex refractions, providing crucial insights into the practical applications of ray tracing in computer graphics, virtual reality, and visual effects. Future work will focus on expanding the material complexity and incorporating dynamic lighting models to push the boundaries of photorealistic rendering further.

1 INTRODUCTION

Rasterization is a rendering technique that determines the position of each object within a scene and transforms these 3D models into pixels on a 2D display. This method constructs an image progressively, layer by layer, similar to how a painter would create a painting.

Ray tracing is a different rendering approach that projects rays from the camera into the scene. In basic ray tracing setups, these rays bounce from the objects towards the light sources to determine the color at specific pixel locations. Ray tracing can simulate the true behaviors of light within a scene, though it requires significantly more computational power than rasterization.

Advanced Ray tracers use path tracing to extend the principles of ray tracing by attempting to solve the rendering equation. Unlike simple ray tracing, path tracing calculates the albedo using light that arrives from every conceivable direction, reflecting off surfaces throughout the scene. This method achieves more realistic lighting effects, including soft shadows, caustics, ambient occlusion, and indirect lighting. Owing to its precise and unbiased nature, path tracing is often utilized to create reference images to evaluate the performance of other rendering techniques.

2 TECHNOLOGY STACK

The code used in this project has been built from scratch using Python. The use of Numpy libraries accelerates computations on the CPU.

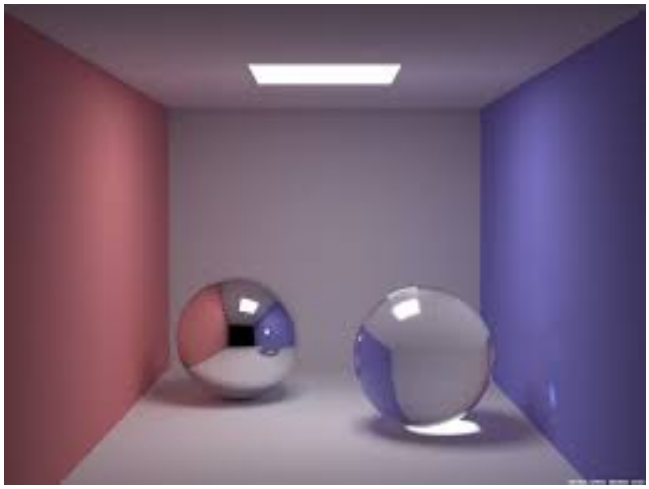

Figure 1: Ray Tracing using Monte Carlo Advanced path tracers

3 IMPLEMENTATION

I developed a class to manage 3D vectors using object oriented programming in python, which includes various operations crucial for 3D graphics. The formula represents a ray in our system:

$$p(t) = A + t \times B$$

where p indicates a position on a 3D line that originates from point A with direction vector B.

Initially, the camera is positioned at the origin, $(0, 0, 0)$, and we outline the coordinates of our screen space. For a sphere with center $C = (Cx, Cy, Cz)$ and radius R, an intersection with a ray occurs if there exists a scalar t such that the point $p(t)$ satisfies the quadratic equation:

$$B \cdot Bt^2 + 2B \cdot (A - C)t + (A - C) \cdot (A - C) - R^2 = 0.$$

This equation allows us to determine t and color the pixel influenced by the sphere.

Furthermore, we calculate surface normals to facilitate the shading of the sphere. Normals are treated as unit vectors, with their components x, y, and z translated into RGB values to visually represent normals, aiding in visualization in the absence of a defined light source.

To enhance image quality and reduce jagged edges, we implement antialiasing through supersampling. This technique involves averaging the colors from multiple rays that are sampled randomly within each pixel, leading to smoother and more visually appealing results.

Lambertian, or diffuse surfaces, do not emit light but reflect light in various directions. They acquire their color by absorbing some wavelengths of light from their surroundings and reflecting others, influenced by their inherent color. To model this:

- A sphere with unit radius is considered at the point of impact.
- A random direction within this sphere is chosen for the outgoing ray, simulating the random reflection.

Metallic surfaces are known for their smooth finish, reflecting light in a specific, predictable direction rather than scattering it randomly. The reflection direction for a metallic surface is given by:

$$V - 2 \cdot \text{dot}(V, N) \cdot N$$

where V is the incoming vector of light and N is the normal at the point of impact.

Dielectric materials such as glass or water alter the path of the incoming light ray by both reflecting and refracting it. Depending on random selection, the light ray may be:

- **Reflected** using the same formula as for metallic surfaces.
- **Refracted** according to Snell's law, which is expressed as:

$$n \cdot \sin(\theta) = n' \cdot \sin(\theta')$$

where n and n' are the refractive indices of the two media involved, with θ and θ' being the angles of incidence and refraction, respectively.

4 DESIGN AND ARCHITECTURE

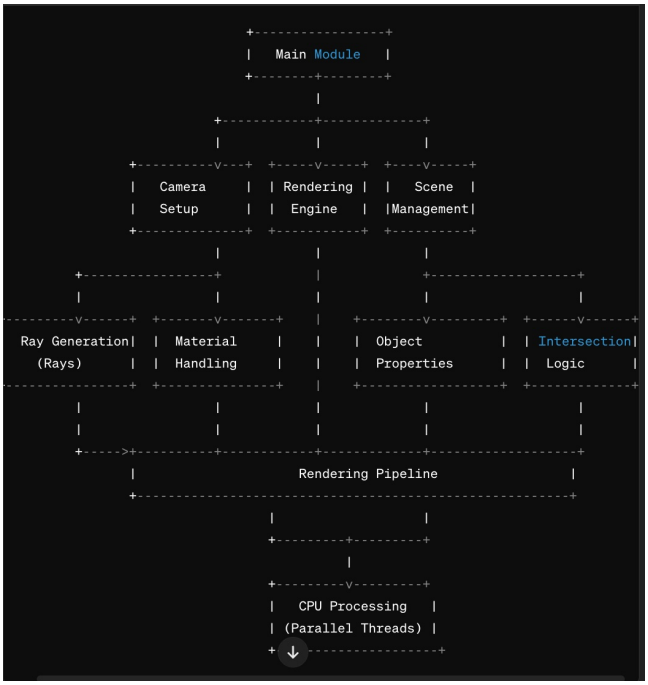

Figure 2: Block Diagram of the Ray Tracing Project Architecture Using Python

Various classes and components interact in my ray tracing system to simulate realistic lighting and shading effects. Here's a detailed breakdown of the critical components.

4.1 Vector Operations (Vec Class)

The *Vec class* is fundamental to 3D graphics computations, enabling essential vector arithmetic for geometry calculations. This class handles operations such as vector addition, subtraction, scalar multiplication, and complex operations like dot and cross products. These are vital for determining spatial relationships and behaviors in 3D rendering.

4.2 Ray Class

The *Ray class* defines and manipulates rays, which represent potential paths that light can travel through the scene. Each ray, characterized by an origin and direction, is traced to detect intersections with objects. This class is pivotal for understanding how light interacts with different surfaces.

4.3 Camera

The *Camera* component captures the scene from a specific viewpoint, defining the camera's position, orientation, and projection details necessary for ray generation. It sets the perspective and field of view, influencing the depth and realism perceived in the final render.

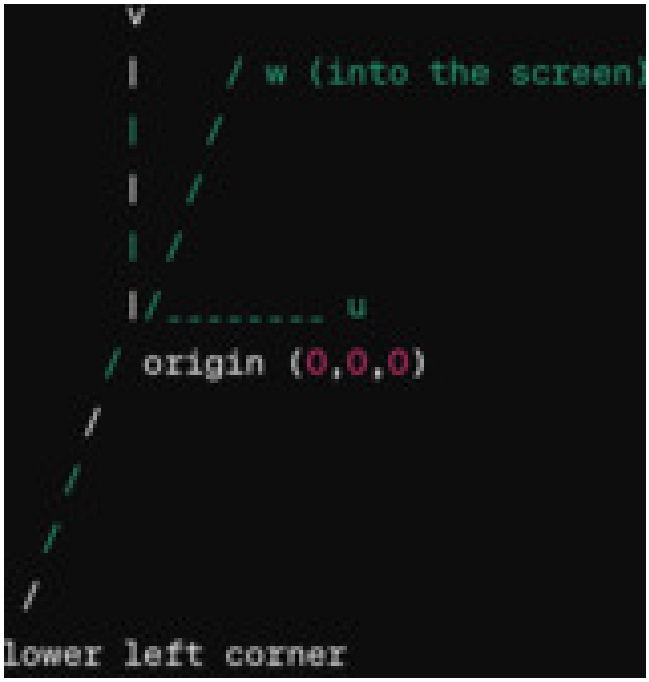

Figure 3: Camera Coorodinates

- Origin: The camera's position in 3D space (0.0, 0.0, 0.0).
- Lower Left Corner: The bottom left point of the camera's view (taking into account the aspect ratio).
- Horizontal: The vector defining the camera's horizontal axis, based on the aspect ratio and image width.
- Vertical: The vector defining the camera's vertical axis, based on the image height.

4.4 Materials

Materials determine how surfaces interact with light, crucial for rendering realistic textures and effects. They include:

- **Lambertian** materials for diffuse reflection.
- **Metal** materials for mirrored reflections.
- **Dielectric** materials for modeling transparent and refractive properties.

4.5 Intersection Logic

Intersection Logic calculates the outcomes of ray-object encounters. It determines the point of contact, the normal at the intersection, and other critical details that dictate how light is scattered or absorbed, affecting the visual outcome of the scene.

4.6 Scene Management

Scene Management, through structures like *AssetLists*, organizes and manages objects within the scene. Efficient scene management often employs techniques such as bounding volume hierarchies to expedite the ray tracing process by effectively reducing the number of intersection checks.

4.7 Rendering Engine

The *Rendering Engine* coordinates the entire rendering process. It oversees ray generation, intersection detection, and color accumulation based on the interactions of rays with objects, calculated by the material properties and lighting effects.

5 RESULTS

When compiling and rendering scenes using the comprehensive ray tracing system outlined above, the results demonstrate the profound impact of each component on the quality and realism of the final images.

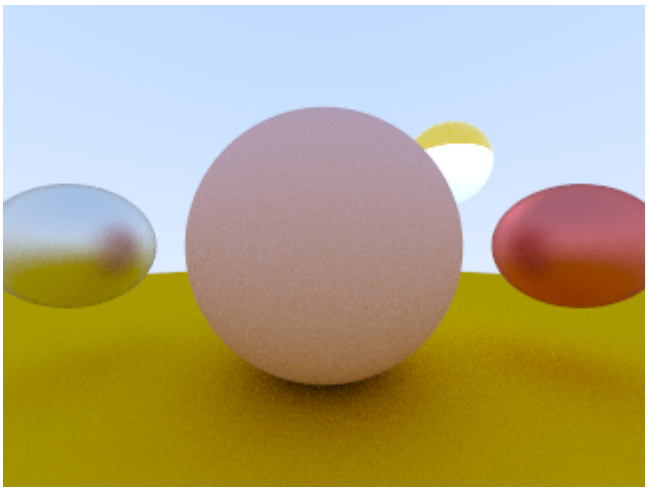

Figure 4: Objects with different materials

In the figure depicted (Figure 4), Sphere s1: Positioned at (0, 0, -1) with a radius of 0.5, made of Lambertian material in pinkish hue, demonstrating diffuse light scattering. Sphere s2: A large sphere with a radius of 100, centered at (0, -100.5, -1), crafted from Lambertian material, acting as a transparent ground plane that affects reflections and lighting. Sphere s3: Located at (1, 0, -1), this small sphere with a radius of 0.25 features a red Metal material, showcasing sharp, mirror-like reflections. Sphere s4: At (-1, 0, -1) with a radius of 0.25, made from a silvery Metal material, reflecting its environment with high fidelity. Sphere s5: Positioned at (0.8, 0.9, -1.5) with a radius of 0.25, using the same Dielectric material with Refractive index 2, it illustrates light refraction and transparency.

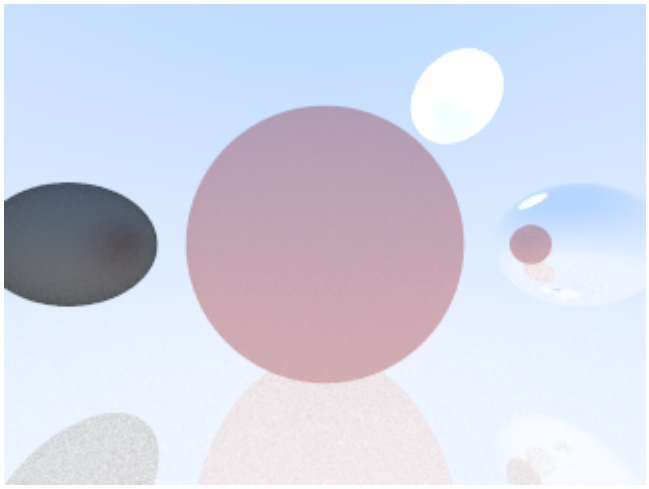

Figure 5: Reflections and refraction

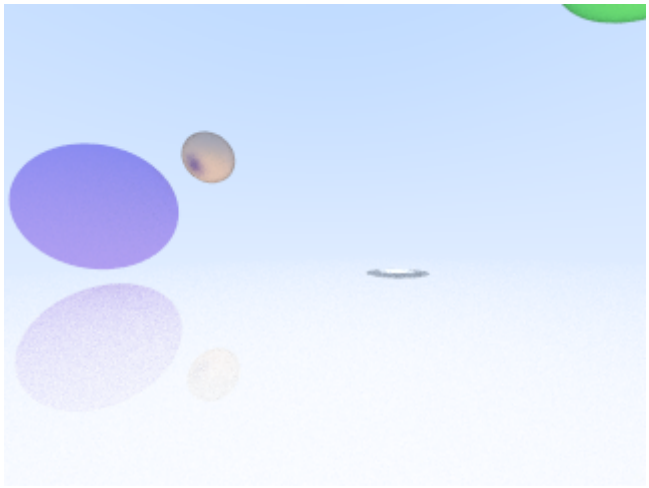

Figure 6: More reflection

In the figure depicted (Figure 5) and (Figure 6), the configuration is set up to test the extreme behaviors of materials with high levels of reflectivity or refraction within a ray tracing scene. Sphere s1: Located at (0, 0, -1) with a radius of 0.5, made of Lambertian material in a pink hue (RGB: 0.8, 0.5, 0.5). This sphere exemplifies standard diffuse reflection, scattering light softly in all directions. Sphere s2:

Positioned at (0, -100.5, -1) with a large radius of 100, crafted from a highly refractive Dielectric material (RGB: 1.0, 1.0, 1.0) with an index of refraction of 3. This ground sphere tests the limits of light bending and splitting, simulating an extremely clear and refractive surface like crystal or high-index glass. Sphere s3: At (1, 0, -1) with a radius of 0.25, this sphere features a perfectly reflective Metal material (RGB: 1.0, 1.0, 1.0), with no roughness (reflectivity = 0). It serves as a test for mirror-like reflection, reflecting its environment with maximum fidelity. Sphere s4: Found at (-1, 0, -1) with a radius of 0.25, composed of a dark Metal material (RGB: 0.2, 0.2, 0.2) with high roughness (reflectivity = 1). This sphere explores the effects of a metallic surface that diffuses light more than a typical mirror finish. Sphere s5: Located at (0.8, 0.9, -1.5) with a radius of 0.25, made from another highly refractive Dielectric material (RGB: 1.3, 1.3, 1.3) with an index of refraction of 3. Similar to s2, it tests high refraction in a smaller, localized setting, examining detailed bending and splitting of light. Below are some more images showing reflection and refraction for different configurations.

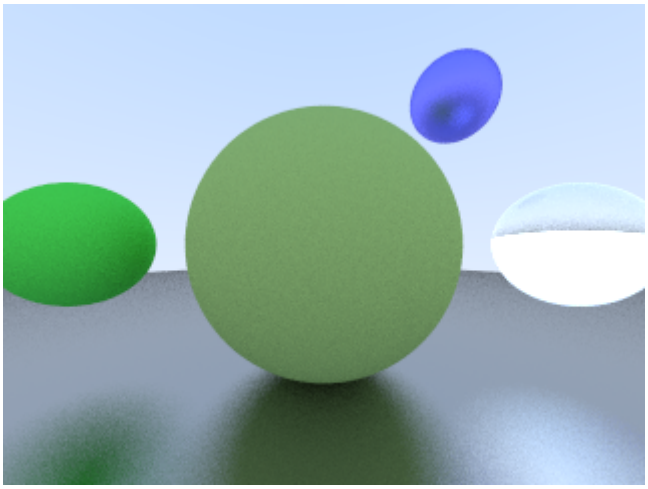

Figure 7: Metal surface shadowing

In the figure depicted (Figure 7),a variety of materials and colors are utilized to explore different visual effects and interactions with light, particularly focusing on shadowing.

In the figure depicted (Figure 8), a variety of materials and colors are used for two spheres to explore different visual effects and interactions with light, mostly noise.

6 CONCLUSION

This project successfully demonstrates the complex interplay between light and various material types within a simulated ray tracing environment. Our setup, involving spheres with distinct reflectivity, diffusion, and refraction properties, has allowed for an in-depth analysis of how light behaves when interacting with different surfaces. The Lambertian materials illustrated the effects of diffuse scattering, providing a soft and uniform light spread. In contrast, metallic and dielectric materials showcased sharp reflections and complex light-bending phenomena, respectively, highlighting their potential for creating visually compelling and realistic scenes.

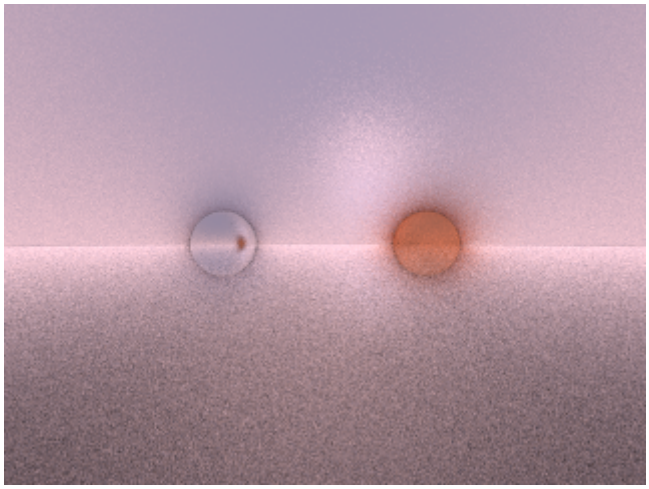

Figure 8: Noisy Sampling

Furthermore, the introduction of spheres with different colors and roughness levels added depth to our understanding of shadow casting and the influence of surface properties on the coloration and intensity of shadows. The use of a simulated environment has proven invaluable in isolating and studying specific aspects of light-material interactions, which are often challenging to replicate in real-world settings.

The findings from this project enhance my theoretical understanding and have practical implications for fields such as computer graphics, virtual reality, and visual effects, where accurate rendering of materials under diverse lighting conditions is crucial. Future work could explore more complex geometries and heterogeneous materials, as well as the integration of dynamic lighting models to push further the boundaries of what can be achieved with ray tracing technology.