Alessandro MASSAAD

CSE306 - Computer Graphics: Report 1

1. Introduction

The objective of this project was to develop an advanced ray tracer capable of rendering scenes with realistic lighting effects and material properties. Throughout four labs, I implemented crucial features of ray tracing, including diffuse and mirror surfaces, direct and indirect lighting, antialiasing, and ray-mesh intersection. This report details the methods and techniques employed in my ray tracer. To get started with our ray tracer, follow these simple steps:

Clean the build directory by running: make clean

Compile the project by executing: make

Render the scene with surfaces by running: ./render_surfaces

Render the scene with lighting effects by running: ./render_light

Render the scene with ray mesh intersection by running: ./render_rmi

2. Diffuse and Mirror Surfaces

In Lab 1, we focused on implementing diffuse and mirror surfaces. The Sphere class was created to represent spherical objects in the scene. Each sphere has properties such as position, radius, color (albedo), and material type (diffuse, mirror, or transparent). The ray-sphere intersection method calculates the intersection of a ray with a sphere, determining the point of intersection and the surface normal at that point.

For diffuse surfaces, the color is calculated based on the Lambertian reflection model, where the intensity of the reflected light is proportional to the cosine of the angle between the light direction and the surface normal. For mirror surfaces, the reflection direction is computed using the law of reflection, and a new ray is traced in this direction to determine the reflected color.

Also in Lab 1, we implemented direct lighting and shadows. A point light source was added to the scene, and the direct illumination was computed by casting shadow rays from the intersection point to the light source. If a shadow ray intersects any object before reaching the light source, the point is in shadow, and only ambient light contributes to the color. Otherwise, the direct light intensity is calculated based on the inverse square law and the Lambertian reflectance.

3. Indirect Lighting

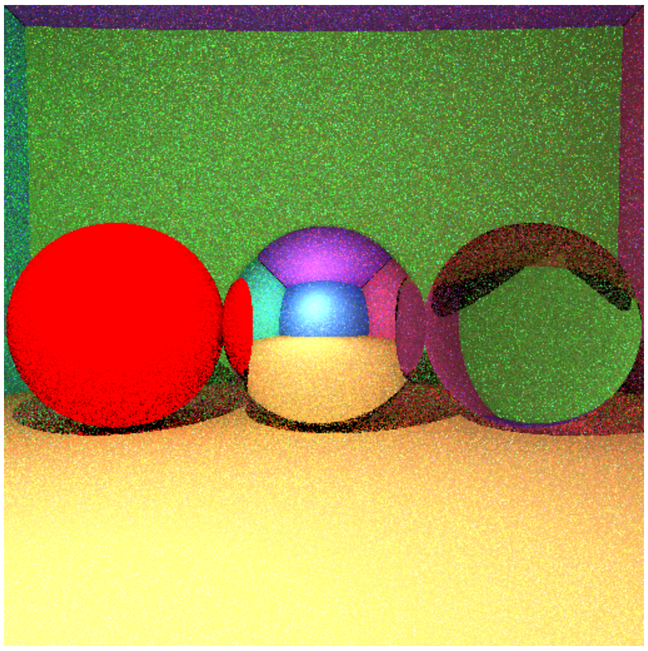
Figure 1: Rendered image with diffuse and mirror spheres.

In Lab 2, we extended the ray tracer to include indirect lighting. This was achieved by randomly sampling directions on the hemisphere around the intersection point and tracing rays in these directions. The indirect light contribution is then averaged over multiple samples. This Monte Carlo integration method captures the global illumination effects, where light bounces off multiple surfaces before reaching the camera.

To generate random directions, we used the random_cos function, which generates vectors uniformly distributed over the hemisphere. The getColor method in the Scene class recursively traces rays and accumulates both direct and indirect lighting contributions up to a specified bounce limit.

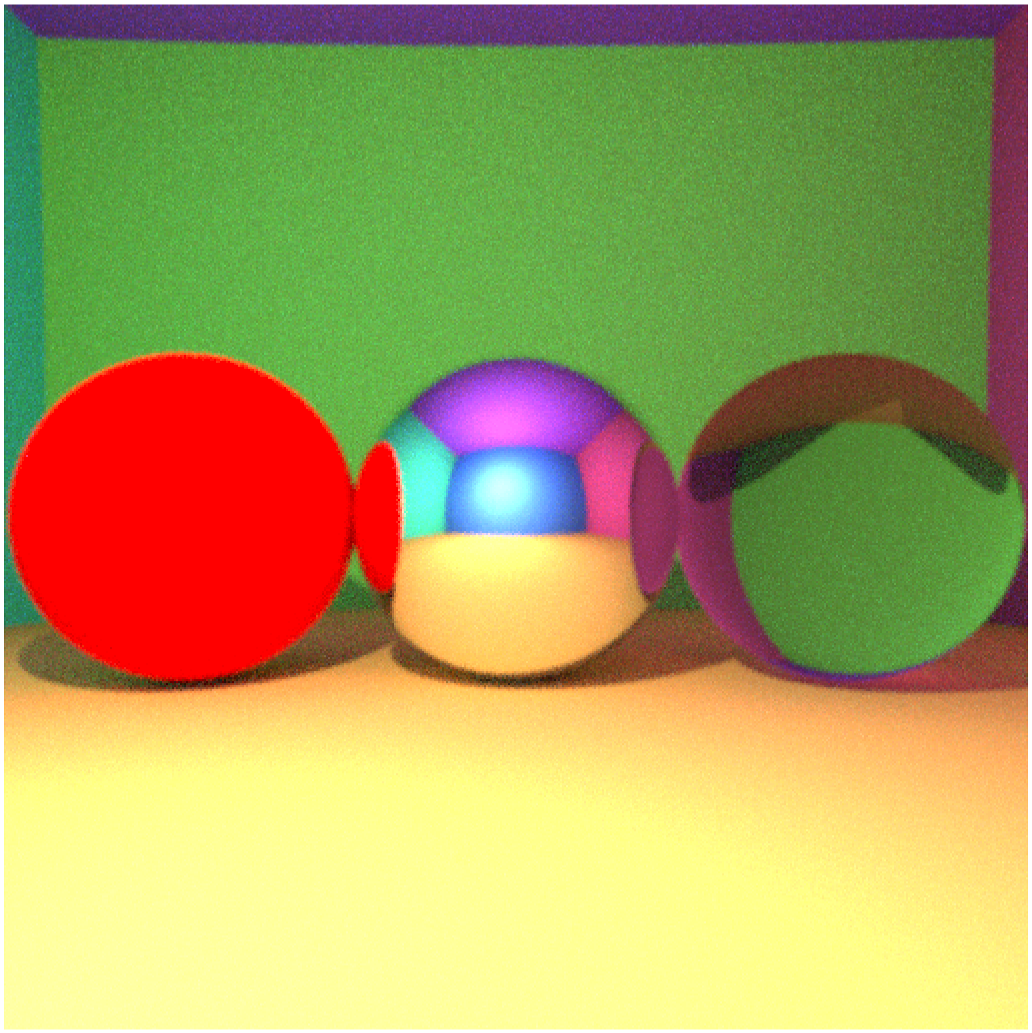

Figure 2: Rendered image with indirect lighting.

To reduce aliasing artifacts, we implemented antialiasing in Lab 2. This was done by jittering the ray origin within each pixel. Multiple rays are cast per pixel, and their colors are averaged to produce the final pixel color. The boxMuller function was used to generate normally distributed random offsets, which were added to the pixel coordinates before casting rays.The execution time for rendering the scene with diffuse and mirror surfaces was 21.3665 seconds. This indicates the computational complexity involved in handling reflections and diffuse lighting calculations.

CSE306 - Computer Graphics: Report 1

4. Ray-Mesh Intersection and BVH

In Labs 3 and 4, we implemented ray-mesh intersection and Bounding Volume Hierarchy (BVH) for efficient ray tracing of complex scenes. The TriangleMesh class was created to load and represent 3D models, such as the provided "cat.obj". The mesh is divided into triangles, and a BVH is constructed to accelerate intersection tests.

The recursive_call method builds the BVH by recursively partitioning the mesh triangles and creating bounding boxes for each node. During ray tracing, the BVH is traversed to quickly eliminate large portions of the scene that do not intersect the ray, significantly improving performance.

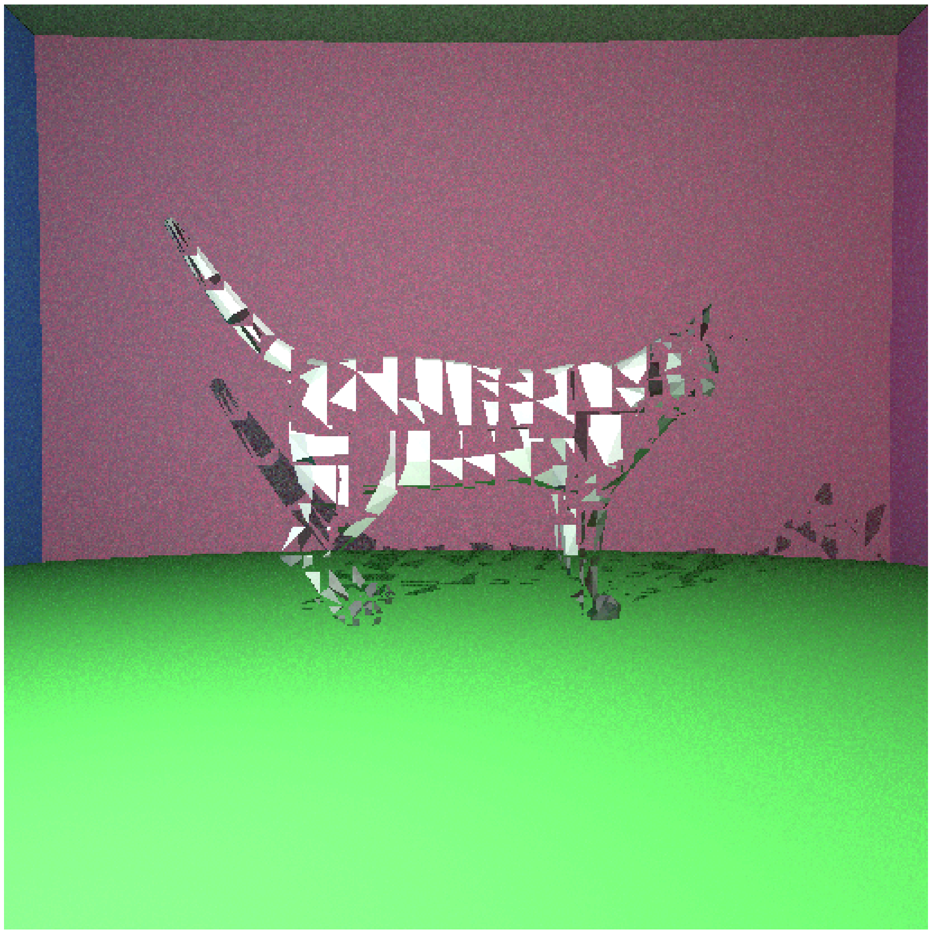

Figure 5: Rendered image with ray-mesh intersection and BVH.

When rendering the scene with the ray-mesh intersection and BVH, the number of triangles in the mesh was 1,318, and the number of vertices was 2,247. The execution time for this scene was 15.8128 seconds, demonstrating the efficiency of the BVH in handling complex geometries and reducing computational overhead.

5. Conclusion

This project successfully implemented a ray tracer capable of rendering scenes with various materials, lighting effects, and geometric complexities. Through a series of labs, we developed key features such as diffuse and mirror surfaces, direct and indirect lighting, antialiasing, and efficient ray-mesh intersection using BVH. The resulting images demonstrate the capability of our ray tracer to produce realistic and high-quality renders, showcasing the techniques learned and applied throughout the project.

6. References

- Lecture notes on Ray Tracing, pages 14 to 47.
- stb_image and stb_image_write libraries for image handling.
- Monte Carlo integration for indirect lighting.
- Bounding Volume Hierarchy (BVH) for efficient ray tracing.