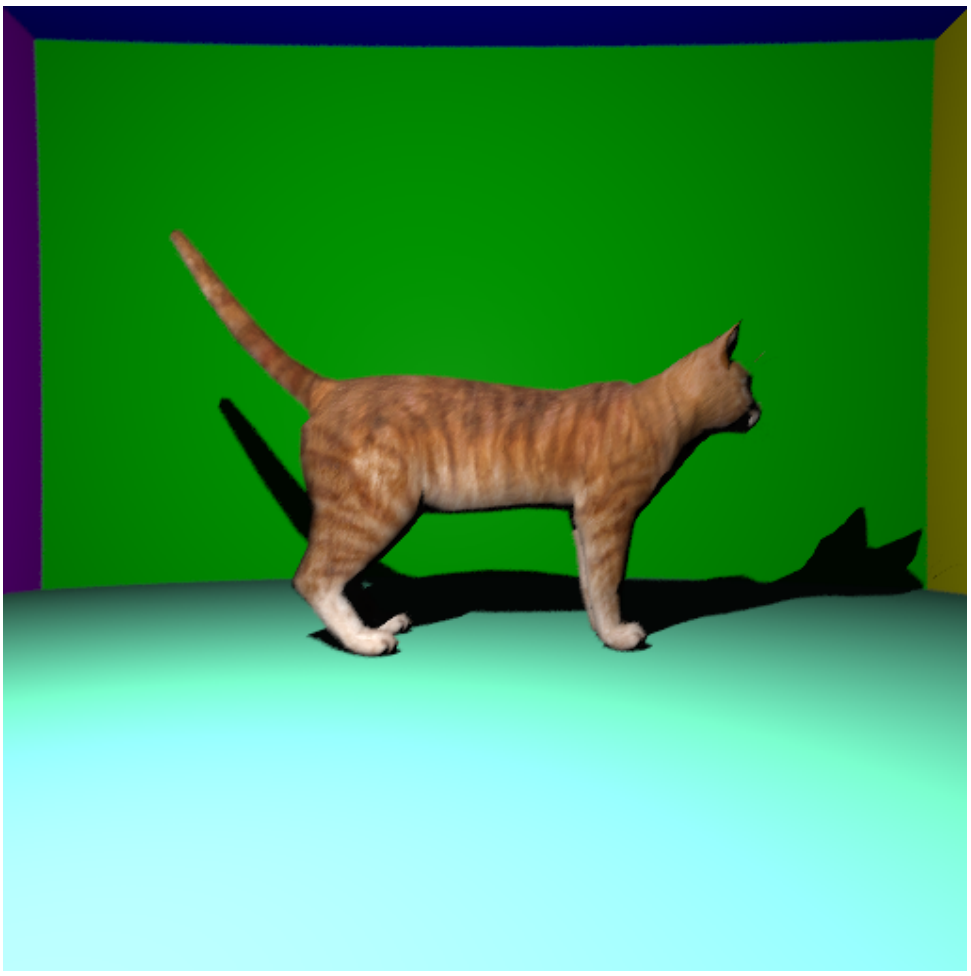
Ray Tracing Implementation Report

Anahi Tinel

May 2, 2025

Github repository: https://github.com/anahitinel/Computer-Graphics



Introduction

This project implements a ray tracing renderer in C++. Ray tracing simulates light paths by tracing rays from the camera through pixels into a scene. The implementation supports rendering 3D models with various materials and lighting effects. This report focuses on explaining the code structure and implementation details rather than the mathematical theory.

1 Code Structure Overview

The ray tracer is built with an object-oriented approach using several key classes:

- **Vector**: Handles 3D vector operations

- **Ray**: Represents a light ray with origin and direction
- **Object**: Abstract base class for scene objects
- **Sphere**: Implements spherical objects
- **MeshObject**: Handles importing and rendering 3D models
- **Scene**: Manages the collection of objects and lighting

2 Basic Ray Tracing with Spheres

2.1 Sphere Implementation

The `Sphere` class inherits from the `Object` base class and implements the `intersect` method to calculate ray-sphere intersections. Each sphere is defined by:

- Center position (Vector C)
- Radius (double R)
- Surface color/albedo (Vector)
- Properties like reflectivity, transparency, etc.

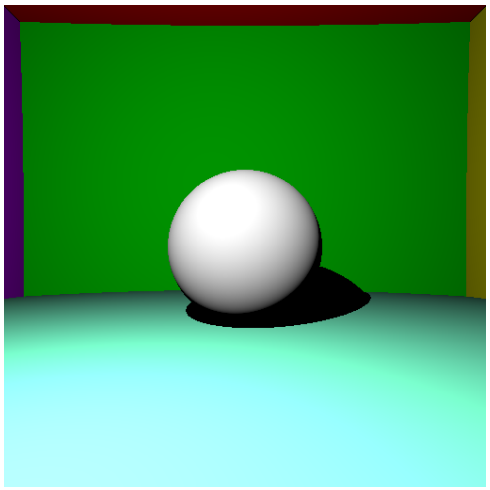
The intersection logic calculates the closest intersection point, and if found, returns the surface position, normal, and material properties.

2.2 Scene Rendering

The `Scene` class maintains a collection of objects and handles:

- Finding the closest intersection for each ray
- Computing lighting at intersection points
- Casting shadow rays to determine visibility

The main function sets up the scene with spheres and a light source, then iterates through each pixel, casting rays and computing colors.



3 Advanced Effects: Reflection and Refraction

3.1 Mirror Reflection

The code implements mirror reflection by:

- Adding a `mirror` boolean property to objects
- Using the `isMirror()` method to check if reflection should be calculated
- Computing reflected ray direction when a mirror surface is hit
- Recursively calling the color function for the reflected ray

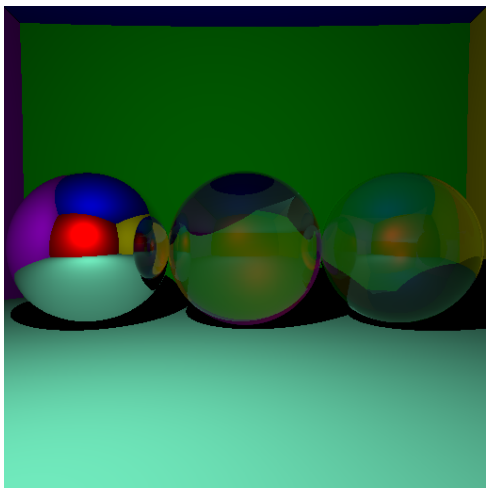
Reflection is controlled by the `ray_depth` parameter, which limits the number of recursive reflections to prevent infinite loops.

3.2 Refraction

For transparent objects like glass, the implementation includes:

- A `transparent` boolean property
- The `isTransparent()` method to identify refractive objects
- Code to calculate refracted ray direction based on Snell's law
- Handling of internal reflections when appropriate

The code balances between reflected and refracted components to simulate realistic glass-like materials.



4 Mesh Rendering and Advanced Effects

4.1 Mesh Loading and Rendering

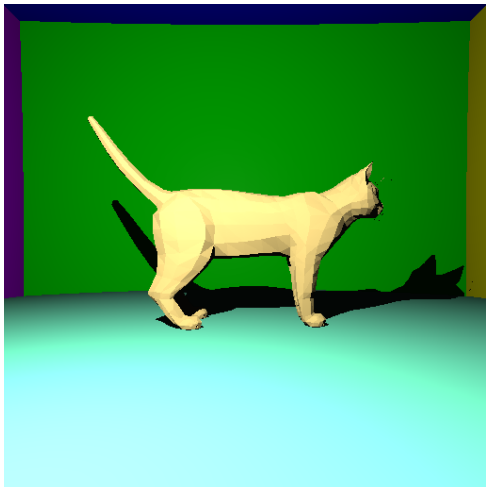
The `MeshObject` class extends `Object` to support rendering 3D models:

- Uses the `Mesh` class to load OBJ files and associated materials
- Parses vertices, texture coordinates, normals, and faces
- Implements ray-triangle intersection using the Möller–Trumbore algorithm

- Uses a bounding box for efficient intersection testing

The mesh loading process:

1. Reads the OBJ file format line by line
2. Processes vertex coordinates, texture coordinates, and normals
3. Creates triangular faces and associates them with materials
4. Loads and manages texture images using the STB image library



4.2 Texturing

Texture mapping is implemented by:

- Storing texture coordinates in the mesh data
- Calculating barycentric coordinates at intersection points
- Interpolating texture coordinates using these barycentric weights
- Sampling textures with proper gamma correction

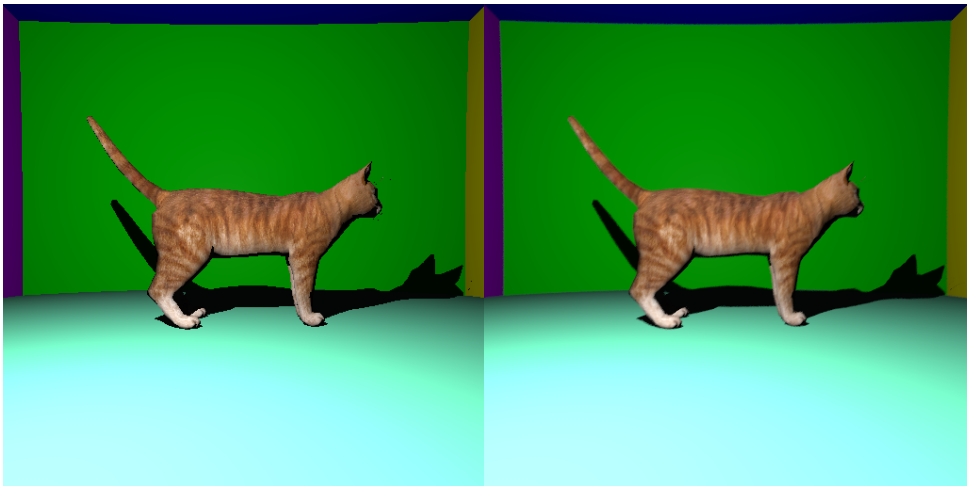
When a ray hits a textured surface, the code retrieves color from the texture based on the UV coordinates rather than using a flat material color.

4.3 Antialiasing

Antialiasing is implemented through supersampling:

- The `boxMuller` function generates Gaussian random offsets
- Multiple rays (defined by `n_rays`) are cast for each pixel
- Ray origins are slightly jittered from the exact pixel center
- Final pixel color is the average of all sample colors

This method effectively reduces jagged edges in the rendered image, creating smoother transitions between objects.



This is the cat model with antialiasing using 32 rays per pixel.

4.4 Indirect Lighting

The indirect lighting implementation adds realism through global illumination:

- The `random_cos` function generates random directions in a hemisphere
- Secondary rays are cast from intersection points in these random directions
- Recursive calls to `getColor` accumulate indirect illumination
- A scaling factor controls the intensity of indirect light

The code creates a coordinate system around the surface normal to properly sample the hemisphere, ensuring physically plausible lighting.

Below are images before and after adding indirect lighting (using 5 bounces):

Conclusion

This ray tracer implementation demonstrates a well-structured object-oriented approach to rendering realistic images. By separating concerns into distinct classes and leveraging inheritance, the code achieves a clean design while supporting advanced features like mesh rendering, texturing, antialiasing, and global illumination. The progressive development from basic spheres to complex scenes shows how the code architecture scales to handle increasingly sophisticated rendering techniques.