High-Performance OBJ File Ray Tracer

Charlie Ratliff
Davidson College

Abstract

The purpose of this project was to extend a working ray tracer with Phong lighting into a comprehensive OBJ parser with the goal of rendering complex 3D models. This would allow the ray tracer to render triangle meshes with correct lighting, shading, and material properties. The implementation includes OBJ file parsing for vertice data, MTL parsing for material properties and texture mapping, normal interpolation for smooth shading across triangle surfaces, Bounding Volume Hierarchy using Axis-Aligned Bounding Boxes for efficient ray-triangle intersection testing, and multi-threading for parallel rendering. The final system successfully rendered a Mercedes model with over 330,000 faces at 2048x2048 resolution in under 55 seconds, demonstrating both correctness and efficiency in handling complex 3D model data.

1 Introduction

Ray tracing is a fundamental technique in computer graphics that simulates the physical behavior of light to generate photorealistic images. Prior to this project, I had built a working ray tracer that implemented the Phong lighting model and could render simple spheres and planes with solid colors. While this provided a foundation in understanding light and shading calculations, it lacked the capability to render the complex 3D models that are standard in modern computer graphics applications.

The goal of this project was to enhance my ray tracer in order to render the complex scenes that you actually see in production by parsing OBJ files. The OBJ format stores vertex positions, texture coordinates, surface normals, and face definitions, making it perfect for representing triangle meshes. Additionally, the associated MTL file format stores material properties including diffuse, ambient, and specular colors. After successfully rendering 3D objects, then arose the issue of efficiency and run time. To combat this, I chose to implement Bounding Volume Hierarchy in order to speed up the ray-triangle intersection tests that occurred at each pixel. I also implemented multi-threading to render each column of the image in parallel using the computer's eight CPUs.

The motivation for this project came from a desire to render real-world 3D models rather than simple geometric objects. Growing up, I always loved watching Pixar movies and was very intrigued with how they were made. After creating a working ray tracer and having the freedom to expand the product, I was very excited about the possibility of being able to render scenes just like the movies I grew up watching with my own code. This project not only reflected a childhood interest, but a hopeful pursuit in a career after college. I have loved working in the fields of computer graphics and image processing. The idea of working for a company and creating these images, movies, games, etc full time is exciting and I hope that I can one day get the chance to.

2 Background

2.1 Ray Tracing Fundamentals

Ray tracing works by casting rays from the camera through each pixel of the image plane and determining which objects those rays intersect. For each intersection, lighting calculations are performed using the Phong lighting model, which consists of three components: ambient lighting, diffuse reflection, and specular reflection. The Phong model computes the final color as:

$$I = I_a + I_d(\mathbf{L} \cdot \mathbf{N}) + I_s(\mathbf{B} \cdot \mathbf{N})^n \quad (1)$$

where I_a, I_d, and I_s are the ambient, diffuse, and specular colors of the object; $\mathbf{L}$ is the light direction; $\mathbf{N}$ is the surface normal; $\mathbf{B}$ is the bisector between the vertex direction and light direction; and n is the shininess exponent.

2.2 OBJ File Triangle Mesh Representation

Triangle meshes are the most common way to construct 3D surfaces in computer graphics. A mesh consists of vertex locations (x, y, z), vertex textures, vertex normals, and faces (triangles defined by three vertex indices). The OBJ file format stores this data using different line starting representations: vertices (v), texture coordinates (vt), normals (vn), and faces (f).

2.3 Spatial Acceleration Structures

Testing every ray against every triangle in a scene for intersection has $O(n)$ complexity per ray, making it difficult for models with hundreds of thousands of triangles. Several acceleration structures have been developed to address this:

Bounding Volume Hierarchies (BVH) organize objects into a tree structure where each node contains a bounding volume that encloses all objects in its subtree. Rays are tested against bounding volumes first, allowing entire subtrees to be skipped if the ray doesn't intersect the bounding volume. BVH essentially brings the overall runtime for ray-triangle intersection test down to $O(logn)$.

K-d Trees partition space using axis-aligned planes, creating a binary space partitioning structure. While K-d trees can achieve excellent performance for static scenes, they are more complex to implement and less robust to poorly distributed geometry.

Octrees recursively subdivide space into eight octants. They are simple to implement but can have poor performance with non-uniform distributions.

2.4 Texture Mapping

Texture mapping allows 2D images to be applied to 3D surfaces. Each vertex is assigned UV coordinates (values between 0 and 1) that map to a position in the texture image. For points inside triangles, UV coordinates are interpolated using barycentric coordinates, allowing smooth texture sampling across the surface.

3 OBJ File Parsing Implementation

The OBJ parser is responsible for converting text-based OBJ files into data structures suitable for rendering. The implementation handles all standard OBJ features required for rendering complex models.

3.1 File Structure and Parsing Strategy

The parser reads the OBJ file line by line, identifying different data types by their prefix: `v` for vertices, `vt` for texture coordinates, `vn` for normals, `f` for faces, `mtllib` for material library references, and `usemtl` for material assignment. All vertex data is stored in arrays, and face definitions reference these arrays using 1-based indices.

A key challenge that is faced when creating an OBJ parser is handling the various face definition formats. OBJ supports multiple

formats:

- `f v1//vn1 v2//vn2 v3//vn3` - Vertex and normal indices
- `f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3` - All three
- `f v1//vn1 v2//vn2 v3//vn3 v4//vn4` Information for four vertices

The parser splits face definitions by the forward slash character and checks for empty strings to determine which components are present. The present components are then stored in the corresponding arrays for that data type in the order that they are presented in the OBJ file.

3.2 Triangulation of Polygonal Faces

While many OBJ files contain only triangles, the format supports arbitrary polygons. The parser implements a triangulation strategy, which represents polygons using multiple triangles. For a face with vertices $[v_1, v_2, v_3, v_4]$, triangles are created as $[v_1, v_2, v_3]$ and $[v_2, v_3, v_4]$. This ensures all faces are converted to triangles for consistent rendering.

3.3 Material Library Integration

The parser detects `mtllib` directives and uses the MTL parser to load material definitions. When a `usemtl` directive is encountered, the current material is set, and all subsequent faces are assigned that material. This allows each part of the model to have their own correct visual properties.

4 Materials and Texture Mapping

4.1 MTL File Parsing

The Material Template Library (MTL) format stores material properties in a companion file to the OBJ file. Each material is defined with a name (used for reference in the OBJ file) and properties including ambient color (`Ka`), diffuse color (`Kd`), specular color (`Ks`), shininess (`Ns`), and optional texture maps (`map_Kd` for diffuse textures).

The MTL parser reads these files and creates Material objects that store all relevant properties, similar to the format of the OBJ parser. Color values are stored as normalized RGB components (0.0 to 1.0).

4.2 Texture Loading and Sampling

Textures are loaded using the ImageSharp library. The Texture class implements bilinear interpolation for smooth texture sampling. Given UV coordinates, the sampler:

1. Converts UV coordinates (0-1 range) to pixel coordinates
2. Identifies the four nearest pixels surrounding the sample point
3. Computes fractional distances for interpolation weights
4. Performs bilinear interpolation

4.3 Barycentric Coordinate Interpolation

To determine UV coordinates at any point on a triangle's surface, barycentric coordinates are computed. For a triangle with vertices v_1, v_2, v_3 and a point p on its surface, barycentric coordinates (u, v, w) satisfy:

$$p = u \cdot v_1 + v \cdot v_2 + w \cdot v_3 \tag{2}$$

where $u+v+w = 1$. These coordinates are computed by solving a system of dot products and then used to interpolate UV coordinates and normals.

5 Smooth Shading with Normal Interpolation

5.1 Flat vs. Smooth Shading

Flat shading uses a single normal vector per triangle, computed from the triangle's surface. This creates visible faceting, especially on curved surfaces approximated by triangle meshes. Smooth shading interpolates normals across the triangle surface, creating the illusion of smooth curves even with relatively coarse geometry.

5.2 Implementation

When vertex normals are provided in the OBJ file, they are stored in the Triangle class as (`N1`, `N2`, `N3`). During rendering, when a ray intersects a triangle, the interpolated normal is computed using barycentric coordinates:

$$\mathbf{N}_{smooth} = u \cdot \mathbf{N}_1 + v \cdot \mathbf{N}_2 + w \cdot \mathbf{N}_3 \tag{3}$$

The smoothed normal is then normalized and used in the lighting calculations, replacing the geometric normal. This technique dramatically improves visual quality for curved surfaces.

Figure 1: *Image of Car Using Geometric Normal*

Figure 2: *Image of Car Using Normal Smoothing*

From the above figures, the visual difference the normal interpolation creates is obvious. By interpolating the normals across each shape, the correct coloring was matched with each pixel rather than using one solid color across each shape.

6 Bounding Volume Hierarchy Acceleration

6.1 Motivation and Algorithm

For complex models with hundreds of thousands of triangles, testing every ray against every triangle is computationally expensive. The Mercedes model used in testing contains over 360,000 faces, which would require over 1.5 trillion intersection tests per frame for a 2048x2048 image without acceleration.

The BVH algorithm organizes triangles into a binary tree structure. Each node contains an Axis-Aligned Bounding Box that encloses

all objects in that subtree. During ray traversal, if a ray doesn't intersect a node's bounding box, all triangles in that subtree can be skipped. This drastically reduces the number of intersection tests needed at each pixel.

6.2 Construction Algorithm

The BVH is built top-down using a recursive splitting strategy:

1. Compute the bounding box for all triangles at the current node
2. If the number of triangles is below a threshold (4 in this implementation), create a leaf node
3. Otherwise:
    - Determine the longest axis of the bounding box
    - Sort triangles by their centroid along that axis
    - Split the list at the median
    - Recursively build left and right subtrees

The splitting strategy ensures balanced trees, though more sophisticated surface area heuristics could provide even better performance.

6.3 Ray-AABB Intersection

Ray-box intersection uses the slab method, which treats the box as the intersection of three pairs of parallel planes (one pair for each axis). For each axis, the algorithm computes the distances along the ray to the near and far planes:

$$t_{min} = \frac{box_{min} - ray_{origin}}{ray_{direction}} \quad (4)$$

$$t_{max} = \frac{box_{max} - ray_{origin}}{ray_{direction}} \quad (5)$$

The ray intersects the box if and only if the intervals $[t_{min}, t_{max}]$ overlap for all three axes. This test is extremely fast and allows quick rejection of entire subtrees.

6.4 Traversal Algorithm

During rendering, rays are traced through the BVH recursively:

1. Test ray against current node's bounding box
2. If no intersection, return immediately
3. If leaf node, test ray against all triangles in the leaf
4. If internal node, recursively traverse left and right children
5. Track the closest intersection found so far

This algorithm reduces the average complexity from $O(n)$ to $O(\log n)$ per ray, providing dramatic speedups for complex scenes.

7 Multi-Threading for Parallel Rendering

Modern CPUs have multiple cores that can execute instructions in parallel. Since each pixel can be computed independently of one another, multi-threading is a great option to improve performance.

The implementation uses C#'s `Parallel.For` construct to distribute pixel rendering across available CPU cores. The outer loop processing image width is parallelized, with each thread processing a full column of pixels.

The parallelization provides nearly linear speedup with the number of cores.

8 Ray-Triangle Intersection

The Möller-Trumbore algorithm is used for efficient ray-triangle intersection. This algorithm directly computes the barycentric coordinates and intersection distance without first computing the ray-plane intersection. Given a ray with origin $\mathbf{O}$ and direction $\mathbf{D}$, and a triangle with vertices $\mathbf{V}_1$, $\mathbf{V}_2$, $\mathbf{V}_3$, the algorithm computes:

$$\mathbf{E}_1(edge1) = \mathbf{V}_2 - \mathbf{V}_1 \quad (6)$$

$$\mathbf{E}_2(edge2) = \mathbf{V}_3 - \mathbf{V}_1 \quad (7)$$

$$a = \mathbf{E}_1 \cdot (\mathbf{D} \times \mathbf{E}_2) \quad (8)$$

If a is near zero, the ray is parallel to the triangle. Otherwise:

$$f = 1/a \quad (9)$$

$$\mathbf{s} = \mathbf{O} - \mathbf{V}_1 \quad (10)$$

$$u = f(\mathbf{s} \cdot (\mathbf{D} \times \mathbf{E}_2)) \quad (11)$$

$$\mathbf{q} = \mathbf{s} \times \mathbf{E}_1 \quad (12)$$

$$v = f(\mathbf{D} \cdot \mathbf{q}) \quad (13)$$

$$t = f(\mathbf{E}_2 \cdot \mathbf{q}) \quad (14)$$

The ray intersects if $u \geq 0$, $v \geq 0$, and $u + v \leq 1$. This algorithm is highly efficient, requiring only one division and a few cross and dot products.

9 Results and Performance Analysis

9.1 Test Models and Rendering Quality

The system was tested with the Mercedes 3D model, which contains 360,000+ triangular faces and solid material definitions. The model was rendered at 2048x2048 resolution with an orthographic camera projection.

The rendered images demonstrate correct geometry, smooth shading through normal interpolation, and proper Phong lighting including ambient, diffuse, and specular components. The materials correctly reflect their appropriate shininess values producing realistic highlights.

9.2 Performance Measurements

Performance was measured on a modern MacBook Pro with 8 CPU cores with the following results:

- **Without BVH:** Over 16 Hours
- **With BVH (single-threaded):** 15 minutes 12 seconds
- **With BVH and multi-threading:** 54.7 seconds

The final system achieves real-time production-quality renders and produces high-resolution images in under a minute.

9.3 Visual Results

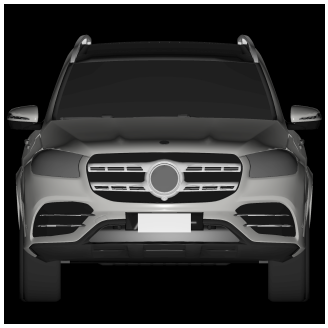

Figure 3: *Mercedes Render*

Figure 4: *Teapot Render*

10 Discussion

10.1 Technical Challenges

Several significant challenges were encountered during implementation:

OBJ Format Variations: Different 3D modeling tools export OBJ files with varying conventions. Some use different face formats. The parser was made robust through careful validation and support for all standard face formats.

Texture Mapping: The MTL parser and color equations underwent several tweaks in order to correctly display images using texture mapping. The biggest change was implementing a practice of setting an objects ambient color to be five percent of its diffuse color. Before this change, many files had the objects ambient color set to white which was overtaking my lighting equation causing each object to be completely white.

BVH Construction Quality: Early BVH implementations produced unbalanced trees that provided minimal speedup. Switching to median splitting along the longest axis dramatically improved tree quality and traversal performance.

10.2 Lessons Learned

This project reinforced several important computer graphics concepts:

Importance of Choosing the Correct Data Structure: The difference between $O(n)$ and $O(\log n)$ was tremendous as the file sizes grew. Adding in BVH made it so that my project could render images in reasonable time.

File Format Complexity: Real-world file formats have numerous edge cases and variations that simple test files and classroom exercises don't reveal. Robust parsing requires extensive testing.

Incremental Development: Building the system in stages (basic parsing, then materials, then textures, then acceleration) allowed each component to be thoroughly tested before adding complexity.

11 Reflection

This project has taught me many valuable skills in the field of computer graphics. I learned about rendering techniques, file formats and file parsing, efficiency techniques and more. I thoroughly enjoyed working on this project. I liked that I was able to create visible results that I could see happening in real time. This allowed me to feel like I had the freedom to create whatever I could imagine which I really enjoyed. Over the course of the semester in Computer Graphics, I have learned a lot about rendering images and how big companies create the things we see on screen with such realism. I also learned about many efficiency techniques that are used at companies such as NVIDIA.

In order to complete this project, knowledge from my other classes was crucial. Specifically, data structures, algorithms, and image processing. These classes all gave me the knowledge I needed to know how to correctly deal with images and how to handle the data in the most efficient way.

12 Conclusion

This project successfully implemented a high-performance ray tracer capable of rendering complex OBJ models with materials and textures. The system handles industry-standard file formats, implements smooth shading through normal interpolation, utilizes BVH acceleration, and leverages multi-core processors for parallel rendering.

The Mercedes model with 360,000+ faces renders at 2048x2048 resolution in under 55 seconds, demonstrating both correctness and efficiency. The model architecture allows for future enhancements while maintaining code clarity and maintainability.

12.1 Future Work

There are several enhancements that would further improve the project that I would like to add:

Advanced Materials: Support for physically-based rendering materials with metallic and roughness parameters to allow more realistic material representation.

Global Illumination: Implementing recursive ray tracing for reflections, refractions, and indirect lighting would produce photorealistic images with realistic light transport.

Improved Acceleration: Surface area heuristic for BVH construction would optimize tree quality. GPU acceleration could provide substantial speedup.

References

3D Models for Free - Free3d.Com. https://free3d.com/. Accessed 8 Dec. 2025.

Ray-Tracing a Polygon Mesh. Scratchapixel.

https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-polygon-mesh/polygon-to-triangle-mesh.html. Accessed 8 Dec. 2025.

Triangle Meshes. Physically Based Rendering, 4th Edition.

https://www.pbr-book.org/4ed/Shapes/Triangle_Meshes.Accessed 8 Dec. 2025.

Watkins, K. Triangles and Meshes. MrKWatkins, 18 Aug. 2024.

https://www.mrkwatkins.co.uk/triangles-and-meshes/.