GPU-Based Ray Tracing of Triangle Meshes Using OpenGL

Oliver Oxford

December 20, 2025

1 Introduction

This project explores the implementation of a basic ray tracer on the GPU using modern OpenGL and GLSL. The goal was to load arbitrary triangle meshes, construct an acceleration structure on the CPU, and perform ray–triangle intersection tests in a fragment shader to render the mesh. Rather than using the traditional OpenGL rasterization pipeline, the project renders a full-screen quad and computes visibility entirely through ray tracing in the fragment shader.

The work combines CPU-side preprocessing (mesh loading, normalization, and BVH construction) with GPU-side ray traversal and shading, providing a practical introduction to real-time ray tracing concepts without relying on dedicated ray tracing APIs.

2 Mesh Loading and Normalization

Meshes are loaded using the Assimp library, which provides support for common formats such as OBJ. Upon loading, each mesh is triangulated and vertex normals are generated if missing. The axis-aligned bounding box (AABB) of the mesh is computed from the vertex data.

To ensure consistent rendering regardless of the original mesh scale, the mesh is normalized into a unit bounding box. This is achieved by computing the mesh center and maximum extent, then translating and uniformly scaling all vertices so that the mesh fits within $[-0.5, 0.5]$ in all axes. The mesh bounding box is updated accordingly after normalization.

From the normalized mesh, a flat array of triangle data is constructed, where each triangle stores its three vertex positions and a face normal. This triangle array is later uploaded to the GPU using a Shader Storage Buffer Object (SSBO).

3 BVH Construction

To accelerate ray–triangle intersection tests, a Bounding Volume Hierarchy (BVH) is constructed on the CPU. Each BVH node contains:

- An axis-aligned bounding box (AABB)
- Indices of left and right child nodes
- An index range into the triangle array for leaf nodes

The BVH is built recursively by splitting triangles along alternating axes based on their centroids. Triangles are reordered in-place so that all triangles belonging to a node are contiguous in memory. This allows leaf nodes to reference triangles using a `firstTri` index and a `triCount`, rather than storing explicit triangle lists. To ensure that triangles were always enclosed in boxes, the child bounding boxes are expanded slightly to fully encapsulate any triangles with centroids in their partition but corners which cross the boundary. To avoid infinite recursion, a node is declared as a leaf if a split in every axis direction always results in one child containing all of it's parents triangles.

Once constructed, the BVH node array is uploaded to the GPU in a second SSBO, enabling traversal entirely within the fragment shader.

4 Rendering Approach

Instead of rendering geometry traditionally, the application draws a single full-screen triangle or quad. Each fragment corresponds to one pixel on the screen and computes a ray originating from the camera into the scene.

The ray origin and direction are derived from normalized screen-space coordinates (`uv`) and a simple camera transform. Because the mesh is normalized into a known coordinate range, a minimal camera setup is sufficient to view the entire object.

5 Ray Tracing in the Fragment Shader

In the fragment shader, ray tracing is implemented using two main intersection routines:

- Ray–AABB intersection using the slab method
- Ray–triangle intersection using the Möller–Trumbore algorithm

For naive rendering, the ray is tested against every triangle in the scene. For accelerated rendering, the ray traverses the BVH iteratively using an explicit stack to avoid recursion, which is not supported in GLSL.

When a leaf node is reached, the ray is tested against all triangles contained within that node, and the closest intersection is selected. If a hit is found, the fragment is shaded using the triangle's surface normal, mapped into RGB space for visualization.

6 Results

The bounding volume hierarchy is visualized by increasing pixel intensity for each leaf node bounding box intersected by a ray fired in the view direction. This shows that boxes more tightly clustered close to the mesh surface. In figure 1. looking in areas of empty space, such as the bottom right corner or between the camels legs reveals how space is subdivided by the bounding volume hierarchy.

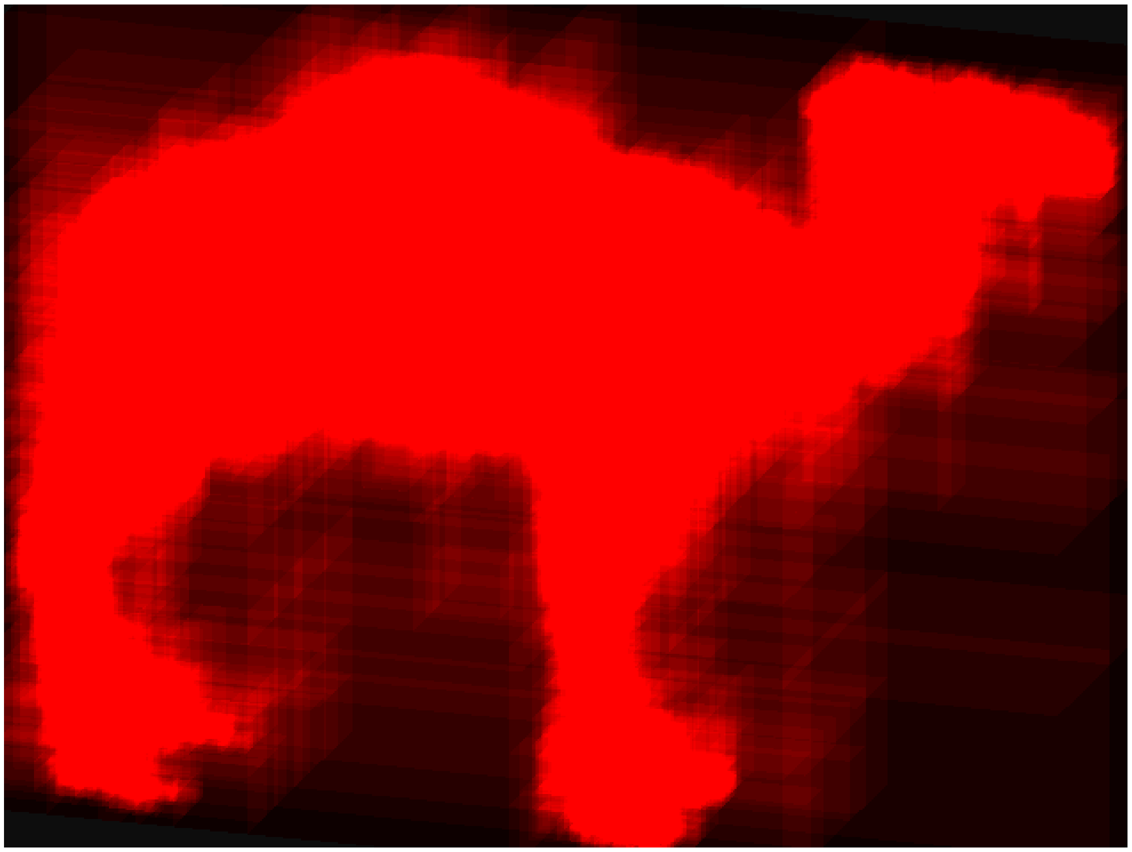

Figure 1: Bounding volume hierarchy visualization of a camel

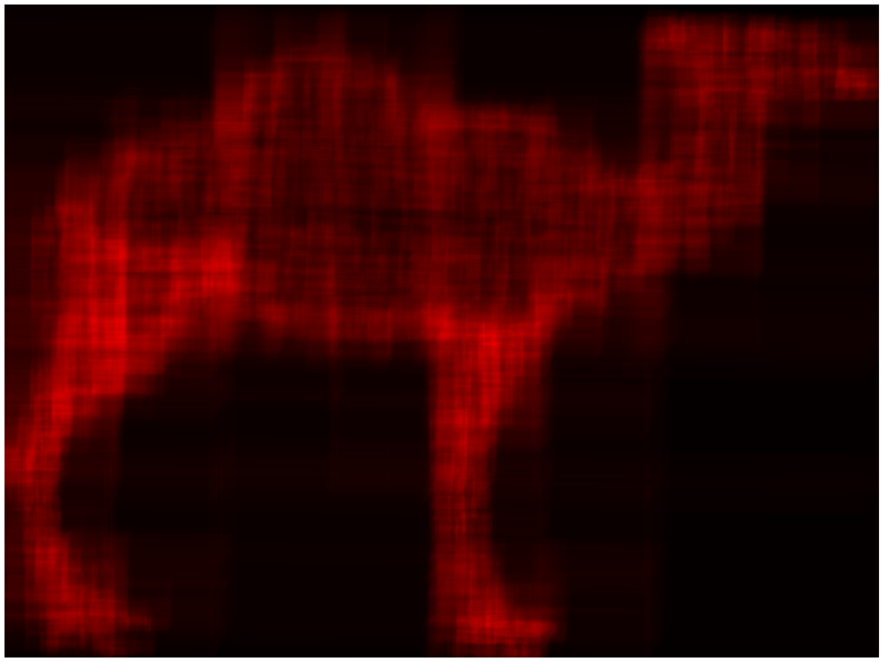

Figure 2: With significantly lower bounding box alpha (more transparent), we get a heat map showing where bounding boxes are most tightly clustered

6.1 Results

To analyze the correctness of the project, we compare our BVH-accelerated rendering with a naive implementation where ray-triangle intersection is tested for every triangle in the mesh for each ray. As show below, the results are identical. See appendix for more examples.

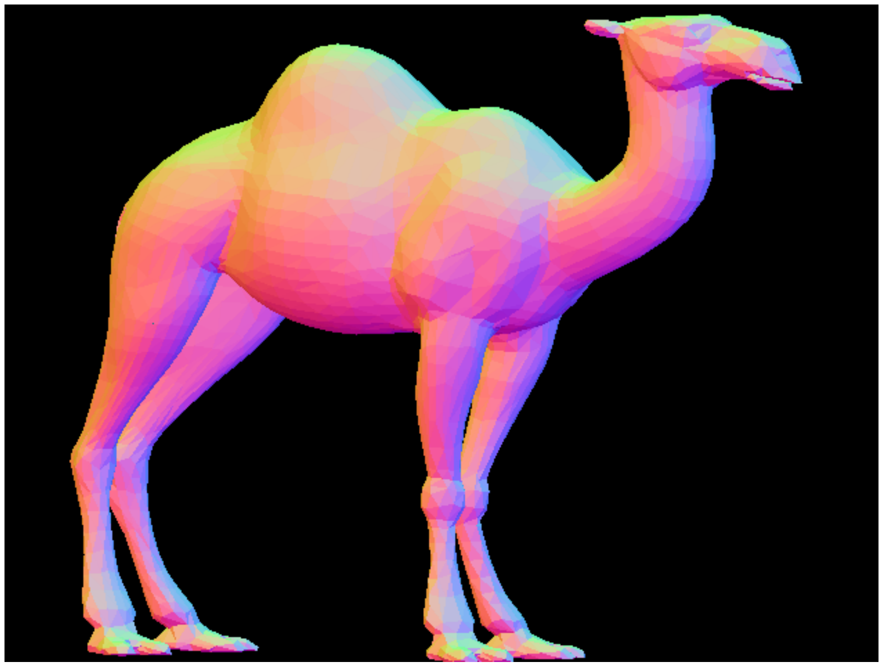

Figure 3: BVH accelerated camel rendering

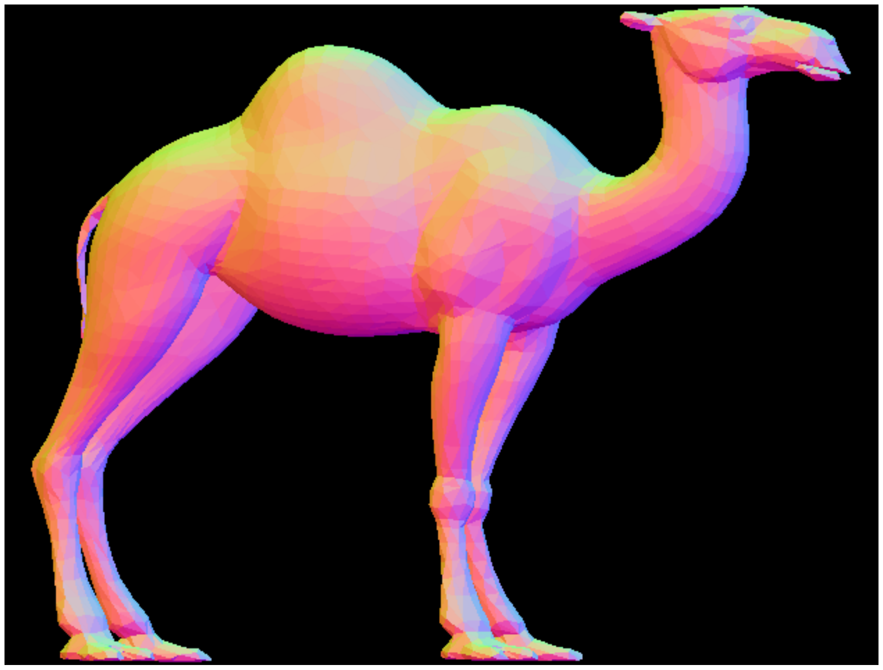

Figure 4: Naive camel rendering

7 Future Work

Ray tracing on meshes is useful for much more that simply rendering. As future work the algorithms and data structures of this project could be used in a scene with lighting where shading is determined based off of the paths of light rays and intersections with the mesh.

While the greatest benefits of BVH based ray tracing occur in scenes with multiple objects and complex geometry, it would be informative to measure the difference in performance and rendering speed between the BVH-accelerated and naive renderings.

8 Conclusion

This project demonstrates a complete pipeline for GPU-based ray tracing using OpenGL and GLSL, including mesh loading, normalization, BVH construction, and ray traversal. While performance is limited compared to hardware-accelerated ray tracing, the implementation provides clear insight into the underlying algorithms and data structures used in modern ray tracers.

Future work could include more efficient BVH splitting heuristics, support for reflections and shadows, and improved camera controls. Overall, the project serves as a strong foundation for understanding real-time ray tracing techniques.

9 LLM usage

For this project LLMs (ChatGPT) were used for writing the report and setting up the project environment (loading the meshes, creating the OpenGL program, linking buffers, etc.) and debugging. The Bounding Volume Hierarchy data structure and traversal was written by me by hand.

10 Citations

Peter Shirley and Steve Marschner. *Fundamentals of Computer Graphics.* CRC Press, 4th edition, 2020.

The following websites were also used as references in designing the algorithms and data structures `https://graphics.stanford.edu/~boulos/papers/efficient_notes.pdf`
`https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm`
`https://en.wikipedia.org/wiki/Slab_method`

11 Appendix: Further examples