Assignment 3: Basic Ray Tracing

NAME: WANGYUJIE
STUDENT NUMBER: 2023533006
EMAIL: WANGYJ2023@SHANGHAITECH.EDU.CN

ACM Reference Format:
Name: wangyujie , student number: 2023533006 , email: wangyj2023@shanghaitech.edu.cn . 2025. Assignment 3: Basic Ray Tracing. 1, 1 (November 2025), 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In this assignment, I implemented a CPU-based ray tracing renderer. The system supports core rendering functionalities including ray-geometry intersections, acceleration structures using Bounding Volume Hierarchies (BVH), and a physically based integrator capable of handling direct illumination, shadows, and refractive materials. Furthermore, I implemented texture mapping with bilinear interpolation and optimized the BVH construction using the Surface Area Heuristic (SAH).

2 Implementation Details

2.1 Ray-Triangle Intersection

For ray-triangle intersection, I utilized the standard barycentric coordinate method. Given a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ and a triangle with vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$, the intersection point can be expressed as:

$$P = (1 - u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2 \tag{1}$$

The algorithm solves the linear system to find t, u, v. The intersection is valid if $u \geq 0, v \geq 0, u + v \leq 1$ and t is within the valid ray interval. Upon intersection, I compute the geometric properties (normals, UVs) and use the helper function `CalculateTriangleDifferentials` to compute partial derivatives $\partial\mathbf{p}/\partial u$ and $\partial\mathbf{p}/\partial v$ for texture mapping.

2.2 BVH Construction

I implemented the `BVHTree::build` function to construct a binary acceleration structure. My implementation supports two splitting heuristics:

2.2.1 *Median Split.*

The basic approach sorts primitives along the longest axis of the bounding box and splits them at the median index. This ensures a balanced tree topology.

Author's Contact Information: Name: wangyujie
student number: 2023533006
email: wangyj2023@shanghaitech.edu.cn.

2.2.2 *Surface Area Heuristic (SAH).*

To optimize ray traversal performance, I implemented the SAH method using a binning approach (as seen in `bvh_tree.h`). The algorithm divides the space into $B = 12$ buckets. For each possible split plane between buckets, the cost is evaluated as:

$$C = C_{trav} + \frac{S_L}{S_{total}} N_L C_{isect} + \frac{S_R}{S_{total}} N_R C_{isect} \tag{2}$$

where S_L, S_R are the surface areas of the left and right child bounding boxes, and N_L, N_R are the primitive counts. I use `std::partition` to efficiently reorder the primitives based on the optimal split bucket. This significantly reduces the number of intersection tests during rendering.

2.3 Integrator and Shading

The `IntersectionTestIntegrator` handles the light transport simulation.

2.3.1 *Ray Generation and Anti-Aliasing.*

I utilized OpenMP for parallelizing the loop over image pixels. To perform anti-aliasing, I implemented a jittered sampling strategy. For each pixel (x, y), multiple samples are generated:

$$P_{sample} = (x + \xi_1, y + \xi_2), \quad \xi_i \in [0, 1) \tag{3}$$

These samples are averaged in the `Film` class to produce the final pixel color.

2.3.2 *Recursive Tracing (Li Function).*

The `Li` function traces the ray path.

- **Refraction:** If the ray hits a `PerfectRefraction` surface, I invoke `bsdf->sample` to compute the transmitted direction (handling Snell's law and total internal reflection) and spawn a new ray using `interaction.spawnRay`.
- **Diffuse Surfaces:** If an `IdealDiffusion` surface is hit, the recursive tracing stops, and direct lighting is computed.

2.3.3 *Direct Lighting and Shadows.*

In `directLighting`, I calculate the contribution from a point light source.

(1) A shadow ray is constructed from the hit point towards the light position.
(2) I use `scene->intersect` to check for occlusions. If the shadow ray is blocked, the contribution is zero.
(3) If visible, the radiance is computed using the Lambertian cosine law:

$$L_{out} = \rho \cdot \max(0, \mathbf{n} \cdot \mathbf{l}) \tag{4}$$

2.4 Texture Mapping

I implemented texture support in `texture.h`.

- **UV Mapping:** The `UVMapping2D` class applies scaling and translation to the UV coordinates provided by the mesh.
- **Image Texture:** The `ImageTexture` class evaluates color using bilinear interpolation. Given UV coordinates, I map them to pixel coordinates (u', v') and fetch the four nearest texels $(c_{00}, c_{10}, c_{01}, c_{11})$. The final color is interpolated based on the fractional offsets:

$$C = \mathrm{lerp}(\mathrm{lerp}(c_{00}, c_{10}, s), \mathrm{lerp}(c_{01}, c_{11}, s), t) \tag{5}$$

3 Results

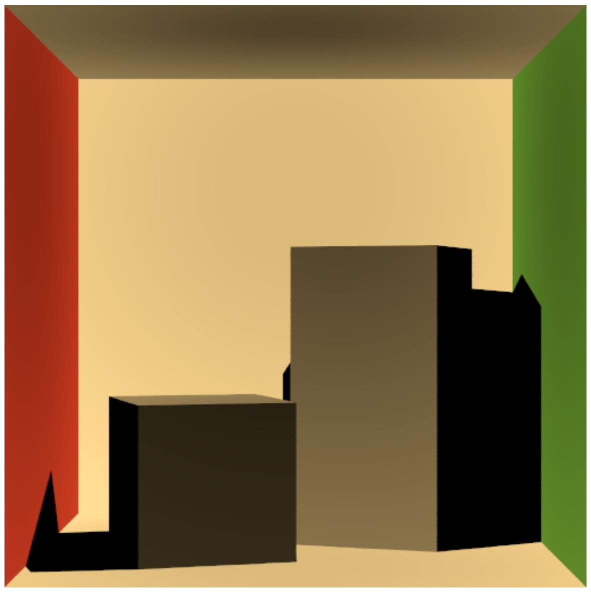

Fig. 1. Cornell Box rendered with direct lighting and hard shadows.

As shown in Figure 1, the renderer successfully produces correct geometry, visibility, shading, and refractive effects. The hard shadows confirm the correctness of the shadow ray logic, and the glass sphere demonstrates the recursive ray tracing capability.