

# Assignment 3:

## Basic Ray Tracing

NAME: ZHOU YUKAI  
STUDENT NUMBER: 2023533109  
EMAIL: ZHOUYK2023@SHANGHAITECH.EDU.CN

### 1 INTRODUCTION

### 2 INTRODUCTION

This assignment implements a basic ray tracer following the structure provided in the framework. The required components include ray-triangle and ray-AABB intersections, a BVH accelerator, a direct lighting integrator with shadow testing, anti-aliasing, and a perfect refraction material. All modules are integrated into a working system that renders Cornell box-type scenes with both diffuse and transparent objects.

In addition, we implemented a GPU LBVH builder (Karras 2012) as an optional component to accelerate BVH construction for scenes containing a large number of primitives.

### 3 RAY-GEOMETRY INTERSECTIONS

#### 3.1 Ray-Triangle Intersection

The ray-triangle intersection follows the standard barycentric method. Given a ray  $r(t) = o + td$  and triangle vertices  $p_0, p_1, p_2$ , we solve:

$$o + td = p_0 + u(p_1 - p_0) + v(p_2 - p_0),$$

where  $u, v$  are barycentric coordinates.

The implementation computes the triangle edges, evaluates several cross products, and checks that  $u \geq 0$ ,  $v \geq 0$ , and  $u + v \leq 1$ . If valid, the intersection distance  $t$  is accepted and the interpolated position, normal, and UV coordinates are written to the `SurfaceInteraction`.

#### 3.2 Ray-AABB Intersection

Ray-AABB testing uses the standard “slab” method. For each axis we compute the entering and exiting  $t$  values for the bounding box, and accumulate a global range  $[t_{\min}, t_{\max}]$ . The ray hits the AABB if  $t_{\min} \leq t_{\max}$ .

This logic is used heavily during BVH traversal, so the implementation precomputes reciprocal ray directions and sign bits to reduce branches.

### 4 BVH CONSTRUCTION

#### 4.1 Overall Structure

The BVH is stored as a flat array of nodes. Each node contains: - a bounding box, - indices for its children or a primitive range, - a flag indicating whether it is a leaf.

#### 4.2 Build Algorithm

The CPU BVH uses a simple top-down recursive partition:

1. Compute the bounding box of primitives and the bounding box of their centroids.
2. If the number of primitives falls below a threshold, create a leaf.
3. Otherwise choose the split axis as the longest centroid extent.
4. Partition primitives into two groups along the split axis.
5. Recursively build both children.

This strategy is simpler than SAH but sufficiently effective for the assignment.

#### 4.3 Traversal

BVH traversal uses an explicit stack. For each node: - test the ray against the nodes AABB; - if overlapping, traverse children (internal) or intersect all primitives (leaf); - the traversal order uses the entry distance to reduce early misses.

This greatly reduces the number of triangle tests compared to brute force.

### 5 DIRECT LIGHTING AND ANTI-ALIASING

#### 5.1 Sampling and Anti-Aliasing

Each pixel is sampled multiple times by jittering subpixel sample points. These coordinates are transformed by the camera model to generate primary rays. The final pixel value is the average of all ray contributions.

#### 5.2 Direct Lighting

For each ray intersection: 1. Construct a `SurfaceInteraction`. 2. Query the BSDF for the surface response. 3. Sample the light direction and check visibility via a shadow ray. 4. If visible, compute:

$$L = f(\omega_o, \omega_l) L_i \max(0, n \cdot \omega_l).$$

Diffuse surfaces use a Lambertian model whose albedo may come from a texture.

### 6 OPTIONAL EXTENSIONS

This section describes the optional components implemented in addition to the required features, including support for multiple point lights and a softshadow rectangular area light.

#### 6.1 Multiple Point Lights

The original integrator supports only a single point light specified by its position and flux. We extend the system to handle an arbitrary number of point lights by maintaining a small array of light positions and intensities inside the integrator:

- Each light contributes independently using standard diffuse shading.

1:2 • Name: Zhou Yukai

student number: 2023533109

email: zhouyk2023@shanghai.tech.edu.cn

- A shadow ray is shot toward each light to determine visibility.
- Lighting contributions accumulate additively.

This extension allows scenes to contain fill lights or colored secondary lights. Figure ?? shows an example rendering using three point lights.

## 6.2 Rectangular Area Light and Soft Shadows

To approximate realworld lighting effects, we further implemented a rectangular area light that produces soft shadows. Unlike point lights which create sharp shadow edges, area lights illuminate the scene from a finite region, so the visibility varies smoothly across surfaces.

Our implementation models the area light directly inside the integrator without modifying the frameworks light interface. A rectangular emitter is parameterized by:

- a center position  $c$ ,
- two spanning directions  $e_x, e_y$  defining width and height,
- a constant radiance value.

For each surface interaction, the integrator performs Monte Carlo sampling:

- (1) A random point is sampled on the rectangle:

$$p_L = c + (u - \frac{1}{2})e_x + (v - \frac{1}{2})e_y, \quad u, v \sim U[0, 1].$$

- (2) A shadow ray is cast toward  $p_L$  to test occlusion.
- (3) If unoccluded, a diffuse contribution proportional to  $\max(0, n \cdot \omega_i)$  is added.
- (4) The result is averaged over all samples.

This produces soft penumbra regions whose width increases with distance from the occluder, matching the expected behavior of an area light.

Figure ?? demonstrates the resulting soft shadows inside the Cornell box. The edges of the box shadows become smooth gradients instead of hard discontinuities.

## 6.3 Implementation Notes

- The area light is sampled purely inside the direct lighting integrator and does not require modifications to the frameworks Light or Scene classes.
- Increasing the number of samples improves the smoothness of shadows but increases render time.
- Compared to point lights, area lighting produces more natural shading on curved refractive objects, such as the glass sphere used in our scene.

## 7 PERFECT REFRACTION MATERIAL

The perfect refraction BSDF uses Snells law. We determine whether the ray is entering or exiting the medium, compute the appropriate  $\eta$  ratio, and attempt refraction. If total internal reflection occurs, reflection is used instead.

The integrator recursively traces rays through transparent materials until a solid surface or the environment is reached. A maximum recursion depth is enforced for safety.

## 8 GPU LBVH CONSTRUCTION (OPTIONAL)

### 8.1 Motivation

When scenes contain many primitives, CPU BVH build becomes slow. LBVH constructs the BVH in parallel by first sorting primitives according to Morton codes and then building the topology using local prefix comparisons.

### 8.2 Pipeline Summary

The GPU LBVH follows these steps: 1. Compute centroids and Morton codes. 2. Sort primitives by Morton codes. 3. Determine internal node ranges using prefix lengths. 4. Assign child links. 5. Perform a bottom-up bounding box reduction.

### 8.3 Validation

We verified correctness using small test scenes and comparing node structures with the CPU BVH. Renderings using CPU and GPU BVHs match closely. Large scenes observe significant performance improvement.

## 9 RESULTS

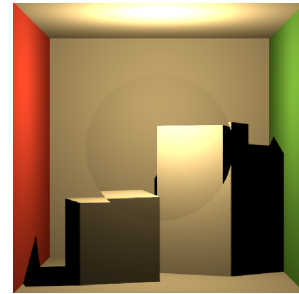


Fig. 1. Render result(single light).

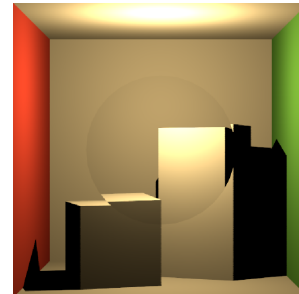


Fig. 2. Building LBVH on GPU.

## 10 CONCLUSION

We implemented a complete ray tracer capable of rendering diffuse and refractive objects with direct illumination. The BVH greatly accelerates intersection tests, and anti-aliasing improves image quality. The optional GPU LBVH builder further reduces BVH build time for complex scenes.

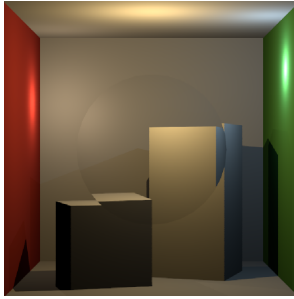


Fig. 3. Multiple light result.

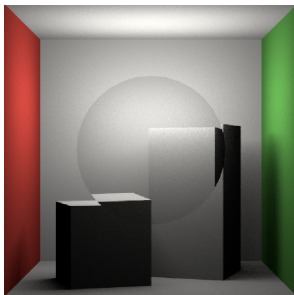


Fig. 4. Area light result.

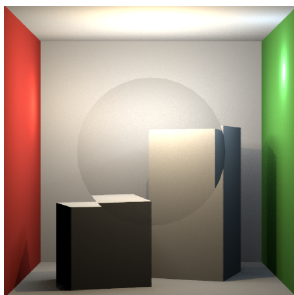


Fig. 5. Final result.

Future improvements may include microfacet BRDFs, environment lighting, multiple importance sampling, and full path tracing.