# Assignment 3: Basic Ray Tracing

NAME: XU WEIHAN
STUDENT NUMBER: 2023533008
EMAIL: XUWH2023@SHANGHAITECH.EDU.CN

## 1 INTRODUCTION

In this assignment. I implemented the following tasks:

- Compile the source code and configure the language server environment. (MUST)
- Implement ray-triangle intersection functionality. (MUST)
- Implement ray-AABB intersection functionality.(MUST)
- Implement the BVH (Bounding Volume Hierarchy) construction. (MUST)
- Implement the IntersectionTestIntegrator and PerfectRefraction material for basic ray tracing validation, handing refractive and solid surface interactions (MUST)
- Implement a direct lighting function with diffuse BRDF and shadow testing. (MUST)
- Implement anti-aliasing via multi-ray sampling per pixel within a sub-pixel aperture.(MUST)
- Implement support for multiple light sources. (OPTIONAL)
- Implement environment lighting via environment maps. (OPTIONAL)
- Investigate texture mapping support in the code base, utilize them on an exmaple scene and understand the texture mapping (OPTIONAL)

## 2 IMPLEMENTATION DETAILS

### 2.1 Ray-Triangle Intersection

The ray-triangle intersection is implemented in the `TriangleIntersect` function (lines 73-152 in `src/accel.cpp`) using the **Möller-Trumbore algorithm**.

The implementation follows these key steps:

(1) Compute edge vectors: $\mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0$ and $\mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0$
(2) Calculate determinant: $\det = \mathbf{e}_1 \cdot (\mathbf{d} \times \mathbf{e}_2)$, reject if $|\det| < 10^{-8}$
(3) Compute barycentric coordinates $u$ and $v$, verify $u, v \geq 0$ and $u + v \leq 1$
(4) Calculate intersection distance $t$ and check if $t \in [t_{\min}, t_{\max}]$
(5) Update surface interaction with barycentric coordinates $(1 - u - v, u, v)$

The implementation uses double precision for intermediate calculations to ensure numerical stability, then converts results back to single precision.

### 2.2 Ray-AABB Intersection

The ray-AABB (Axis-Aligned Bounding Box) intersection is implemented in the `AABB::intersect` method (lines 32-65 in `src/accel.cpp`) using the **slab method**.

The implementation follows these steps:

(1) Compute intersection times with all three pairs of parallel planes:

$$\mathbf{t}_1 = (\mathbf{low} - \mathbf{o}) \cdot \mathbf{d}^{-1}, \quad \mathbf{t}_2 = (\mathbf{upper} - \mathbf{o}) \cdot \mathbf{d}^{-1} \tag{1}$$

where $\mathbf{d}^{-1}$ is the safe inverse direction of the ray
(2) Calculate near and far intersection points for each axis:

$$\mathbf{t}_{\text{near}} = \min(\mathbf{t}_1, \mathbf{t}_2), \quad \mathbf{t}_{\text{far}} = \max(\mathbf{t}_1, \mathbf{t}_2) \tag{2}$$

(3) Determine entry and exit times:

$$t_{\text{enter}} = \max(t_{\text{near}}), \quad t_{\text{exit}} = \min(t_{\text{far}}) \tag{3}$$

(4) Check validity: if $t_{\text{enter}} > t_{\text{exit}}$, no intersection exists
(5) Return $t_{\text{enter}}$ and $t_{\text{exit}}$ via output pointers

This method efficiently tests ray-box intersection by treating the AABB as the intersection of three slabs perpendicular to the coordinate axes.

### 2.3 BVH (Bounding Volume Hierarchy) Construction

The BVH construction is implemented in the `BVHTree::build` method (lines 122-220 in `include/rdr/bvh_tree.h`) using a recursive top-down approach.

The implementation follows these key steps:

(1) **Pre-compute Bounding Box**: Calculate the AABB that encompasses all primitives in the current span [span_left, span_right)
(2) **Stopping Criteria**: Create a leaf node if either:
  - Recursion depth ≥ 22 (CUTOFF_DEPTH), or
  - Number of primitives ≤ 1
(3) **Splitting Strategy**: Use median split heuristic:
  - Select splitting dimension as the longest axis of the bounding box
  - Partition primitives at the median using `std::nth_element`
  - Sort by centroid position along the chosen axis
(4) **Recursive Construction**: Build left and right subtrees:

$$\text{split} = \text{span\_left} + \frac{\text{count}}{2} \tag{4}$$

Left subtree: [span_left, split), Right subtree: [split, span_right)
(5) **Node Organization**: Store internal nodes in a separate array with pointers to children and associated AABBs

The BVH accelerates ray tracing by hierarchical pruning: rays only traverse subtrees whose AABBs intersect with the ray, significantly reducing intersection tests.

## 2.4 IntersectionTestIntegrator and PerfectRefraction Material

This implementation provides basic ray tracing validation with support for refractive and solid surface interactions.

*2.4.1 IntersectionTestIntegrator.* Implemented in `src/integrator.cpp` (lines 28-213), this integrator traces rays through the scene and handles different material types:

(1) **Ray Generation**: For each pixel, generate rays using Monte Carlo sampling:
   - Sample pixel positions using `Sampler::getPixelSample`
   - Generate differential rays via `Camera::generateDifferentialRay`
(2) **Ray Tracing Loop**: Recursively trace rays up to `max_depth` bounces:
   - Test ray-scene intersection using `scene->intersect`
   - Identify surface type via RTTI (IdealDiffusion or PerfectRefraction)
   - Handle infinite lights for missed rays
(3) **Surface Handling**:
   - *Diffuse surfaces*: Terminate and compute direct lighting
   - *Refractive surfaces*: Sample new direction and spawn continuation ray
   - *Other surfaces*: Terminate ray path
(4) **Direct Lighting**: Compute illumination from point lights and environment maps with shadow testing and Monte Carlo integration

*2.4.2 PerfectRefraction Material.* Implemented in `src/bsdf.cpp` (lines 77-123), this material simulates perfect specular refraction and total internal reflection:

(1) **Interface Detection**: Determine ray direction relative to surface normal:

$$\text{entering} = \cos\theta_i = \mathbf{n} \cdot \mathbf{w}_o > 0 \tag{5}$$

(2) **IOR Adjustment**: Correct refractive index based on ray direction:

$$\eta_{\text{corrected}} = \begin{cases} \eta & \text{if entering} \\ 1/\eta & \text{if exiting} \end{cases} \tag{6}$$

(3) **Refraction Calculation**: Attempt to compute refracted direction using Snell's law:

$$\mathbf{w}_i = \text{Refract}(\mathbf{w}_o, \mathbf{n}_{\text{corrected}}, \eta_{\text{corrected}}) \tag{7}$$

(4) **Total Internal Reflection**: If refraction fails (critical angle exceeded), perform perfect reflection:

$$\mathbf{w}_i = \text{Reflect}(\mathbf{w}_o, \mathbf{n}_{\text{corrected}}) \tag{8}$$

The material returns a delta distribution (pdf = 1.0) since it represents perfect specular interactions.

## 2.5 Direct Lighting with Diffuse BRDF and Shadow Testing

The direct lighting function is implemented in `IntersectionTestIntegrator::directLighting` (lines 146-213 in `src/integrator.cpp`). It computes illumination from both point lights and environment maps with visibility testing.

*2.5.1 Point Light Illumination.* For each point light in the scene:

(1) **Light Direction and Distance**: Compute normalized direction and distance to light:

$$\mathbf{l} = \text{Normalize}(\mathbf{p}_{\text{light}} - \mathbf{p}), \quad d = \|\mathbf{p}_{\text{light}} - \mathbf{p}\| \tag{9}$$

(2) **Shadow Testing**: Cast shadow ray from surface point to light source with $t_{\max} = d - \epsilon$. If intersected, add ambient term and skip to next light
(3) **Direct Lighting Computation**: For visible lights, compute radiance using diffuse BRDF:

$$L_d = \frac{\rho}{\pi} \cdot \Phi_{\text{light}} \cdot \max(0, \mathbf{n} \cdot \mathbf{l}) \cdot \frac{1}{d^2} \tag{10}$$

where $\rho$ is the albedo from BSDF evaluation, $\Phi_{\text{light}}$ is the light flux

*2.5.2 Environment Map Illumination.* For scenes with infinite area lights:

(1) **Monte Carlo Sampling**: Generate 30 samples from environment map using `InfiniteLight::sample`
(2) **Visibility Testing**: For each sample direction $\mathbf{w}_i$, cast shadow ray to test occlusion
(3) **Radiance Integration**: Accumulate unoccluded contributions:

$$L_{\text{env}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\rho}{\pi} \cdot L_e(\mathbf{w}_i) \cdot \max(0, \mathbf{n} \cdot \mathbf{w}_i) \tag{11}$$

where $N = 30$ is the number of environment samples

## 2.6 Anti-Aliasing via Multi-Ray Sampling

Anti-aliasing is implemented in the `IntersectionTestIntegrator::render` method (lines 41-77 in `src/integrator.cpp`) using stochastic sub-pixel sampling.

(1) **Samples Per Pixel (SPP)**: For each pixel $(dx, dy)$, generate spp rays with randomized sub-pixel positions
(2) **Sub-Pixel Jittering**: Use `Sampler::getPixelSample` to generate random positions within the pixel aperture $[dx, dx + 1] \times [dy, dy + 1]$
(3) **Ray Generation**: Create differential rays for each sample position via `Camera::generateDifferentialRay`
(4) **Monte Carlo Integration**: Accumulate radiance contributions:

$$L_{\text{pixel}} = \frac{1}{\text{spp}} \sum_{i=1}^{\text{spp}} L_i(\mathbf{r}_i) \tag{12}$$

where $\mathbf{r}_i$ is the $i$-th ray sampled at a random sub-pixel position
(5) **Film Accumulation**: Each sample is committed to the film using `Film::commitSample`, which internally averages all contributions

This approach effectively reduces aliasing artifacts by averaging multiple samples with different ray directions per pixel, smoothing out high-frequency discontinuities at geometric edges.

# 3 RESULTS

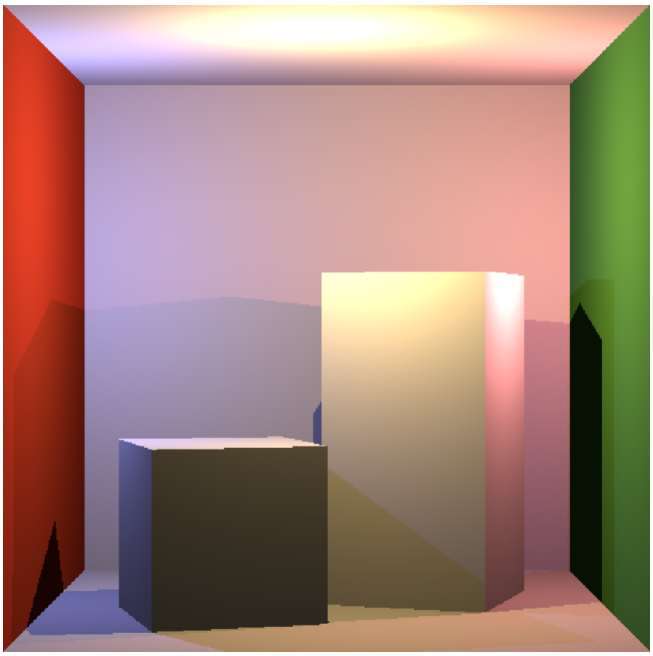

Fig. 1. cbox with no refract

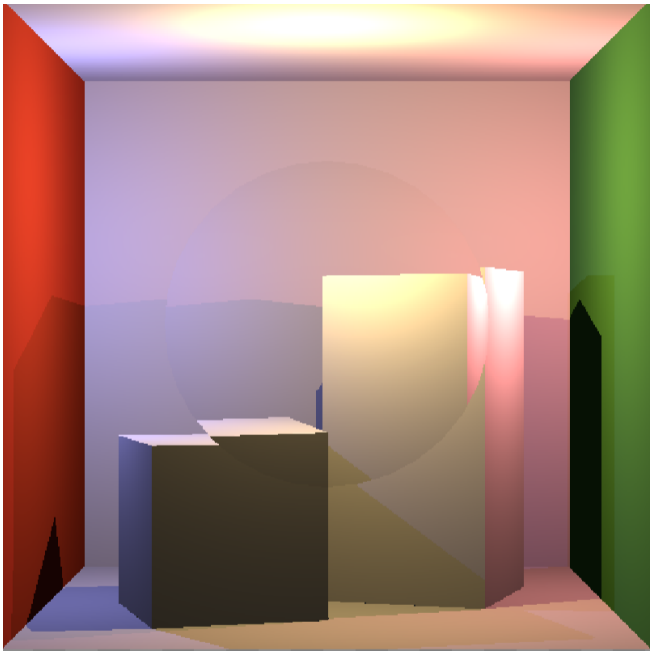

Fig. 2. cbox with refract

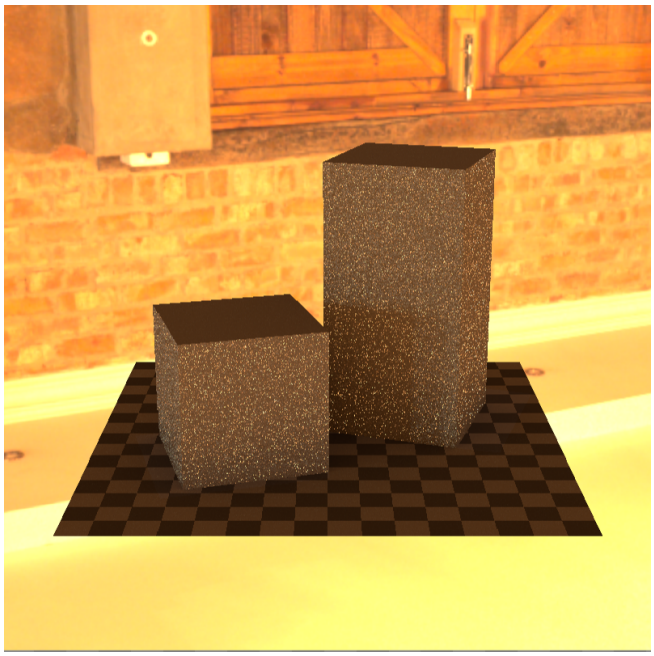

Fig. 3. environment light

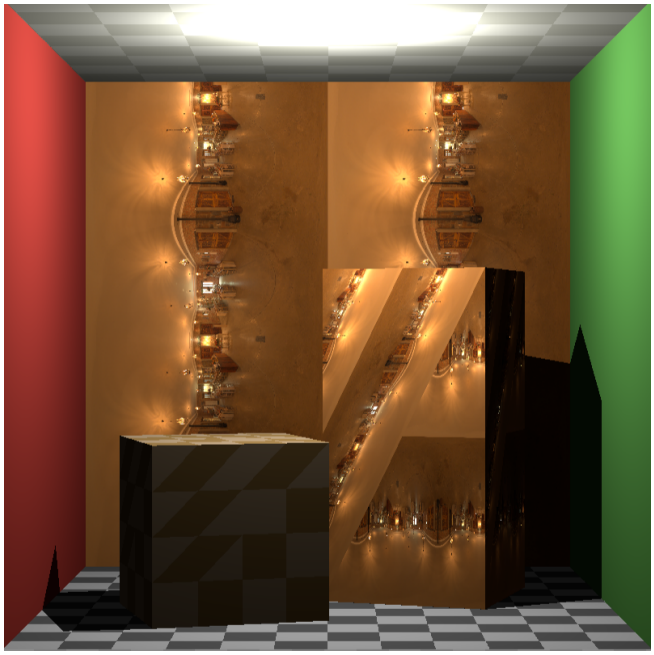

Fig. 4. texture mapping