Assignment 1:
Exploring OpenGL Programming

NAME: XINYU JIA
STUDENT NUMBER:2023533102
EMAIL: JIAXY2023@SHANGHAITECH.EDU.CN

1 INTRODUCTION

This project involves building a ray tracing renderer from a provided C++ framework. The primary goal is to implement fundamental ray tracing components to render a Cornell Box scene with various materials and lighting effects. The implementation covers ray-geometry intersections, acceleration structures for performance, and lighting calculations for realism. This report outlines the implementation details for each required and optional task undertaken.

2 IMPLEMENTATION DETAILS

2.1 Required Tasks

2.1.1 Ray-Triangle Intersection. **Function:** `TriangleIntersect`

The ray-triangle intersection test is implemented using the efficient Möller–Trumbore algorithm. This method directly solves for the barycentric coordinates (u, v) and the ray distance t without needing to pre-calculate the plane normal. The intersection point $\vec{P}$ is defined by the ray equation and the triangle equation:

$$\vec{P} = \vec{O} + t\vec{D} = (1 - u - v)\vec{V_0} + u\vec{V_1} + v\vec{V_2}$$

where $\vec{O}$ and $\vec{D}$ are the ray's origin and direction, and $\vec{V_0}, \vec{V_1}, \vec{V_2}$ are the triangle's vertices. This can be rearranged into a linear system:

$$\begin{bmatrix} -\vec{D} & \vec{V_1} - \vec{V_0} & \vec{V_2} - \vec{V_0} \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \vec{O} - \vec{V_0}$$

The implementation uses Cramer's rule to solve for (t, u, v). The determinant of the matrix is first calculated. If it is near zero, the ray is parallel to the triangle plane. Otherwise, a valid intersection is registered only if $u \geq 0, v \geq 0, u + v \leq 1$, and t is within the ray's valid range $[\text{t\_min}, \text{t\_max}]$.

2.1.2 Ray-AABB Intersection. **Function:** `AABB::intersect`

The ray-AABB intersection is implemented using the slab test method. An AABB can be seen as the intersection of three pairs of parallel planes (slabs) aligned with the coordinate axes. A ray intersects the AABB if and only if the ray's intersection intervals with these three slabs overlap. The algorithm maintains a single valid interval $[t_{min}, t_{max}]$ for the ray segment inside the volume. For each axis, it computes the intersection distances with the two corresponding planes and updates the interval. For an axis-aligned ray, the inverse direction component becomes infinite, but the logic correctly handles this case due to floating-point arithmetic rules, effectively testing if the ray's origin is within the slab for that dimension. An intersection occurs if the final interval is valid (i.e., $t_{min} \leq t_{max}$).

2.1.3 BVH Construction. **Function:** `BVHTree::build`

The Bounding Volume Hierarchy (BVH) is constructed recursively to accelerate ray intersection tests. While the Surface Area Heuristic (SAH) often yields higher quality trees, the simpler and faster median split heuristic is employed here.

(1) **Termination:** The recursion stops if a node contains only one primitive or the maximum depth is reached, creating a leaf node.
(2) **Partitioning:** The dimension with the largest AABB extent is chosen as the splitting axis. **`std::nth_element`** is then used to partition the primitives in $O(N)$ time based on the median of their centroids along this axis.
(3) **Recursion:** The function recursively calls itself for the two resulting subsets, creating left and right children. The AABBs of these children are then merged to form the parent node's AABB.

Listing 1. BVH median split logic.

```
// Find the axis with the largest extent
const int &dim = ArgMax(prebuilt_aabb.getExtent());
// Find the split point
split = span_left + (span_right - span_left) / 2;
// Partition primitives using nth_element
std::nth_element(nodes.begin() + span_left, nodes.be
    nodes.begin() + span_right,
    [dim](const NodeType &a, const NodeType &b) {
      return a.getAABB().getCenter()[dim] <
             b.getAABB().getCenter()[dim];
    });
```

2.1.4 Direct Illumination and Refraction. The `IntersectionTestIntegra` handles direct lighting and perfect refraction.

- **Direct Lighting:** For diffuse surfaces, the outgoing radiance L_o is an integral over the hemisphere Ω. For direct lighting from a point light, this simplifies to a sum. A shadow ray is cast towards each light source. If unoccluded, the light's contribution is computed using the BRDF: $L_o = f_r(\omega_i, \omega_o) \cdot L_i \cdot \max(0, \vec{n} \cdot \omega_i)$, where f_r is the diffuse BRDF (ρ/π), L_i is the incoming radiance, and the dot product is the cosine term.

student number:2023533102
email: `jiaxy2023@shanghaitech.edu.cn`

- **Refraction:** For refractive materials, `PerfectRefraction::sample` calculates the new ray direction using Snell's law: $n_1 \sin\theta_1 = n_2 \sin\theta_2$. The implementation correctly determines the relative index of refraction $\eta = n_1/n_2$ based on whether the ray is entering or exiting the medium. Total internal reflection is handled by checking if the term under the square root for the transmitted angle is negative. If so, a reflection ray is generated instead.

2.1.5 Anti-Aliasing. Anti-aliasing is achieved by multi-sampling, a form of stochastic Monte Carlo integration over the pixel area. For each pixel, `spp` rays are generated. `sampler.getPixelSample()` provides a uniformly random sample in $[0,1)^2$, which is then mapped to a unique sub-pixel position. The final pixel color is the average of the radiance values computed for these `spp` samples, effectively smoothing jagged edges.

2.2 Optional Tasks

2.2.1 Texture Mapping. Following the assignment update, the focus was on analyzing the provided texture mapping framework.

- **Invocation:** Texture color is evaluated via the chain `Integrator::Li` $\rightarrow$ `BSDF::evaluate` $\rightarrow$ `Texture::evaluate`.
- **Mipmap Support:** The framework robustly supports Mipmapping to prevent aliasing. The `ImageTexture` constructor loads an image and immediately builds a `MIPMap` object, which generates and stores a pyramid of downsampled textures. The `mipmap->LookUp` function performs trilinear filtering. It uses texture coordinate differentials (`dstdx`, `dstdy`) to estimate the Level of Detail (LOD) required, then bilinearly interpolates within the two nearest Mipmap levels and linearly interpolates between them.
- **Sphere UVs and Differentials:** For spheres, UVs are derived from the spherical coordinates of the intersection point: $\phi = \text{atan2}(p_y, p_x)$, $\theta = \text{acos}(p_z)$, leading to $u = \phi/(2\pi)$ and $v = \theta/\pi$. The differentials are essential for high-quality texture filtering, as they inform the Mipmap system about the size of the pixel's footprint on the texture, allowing for the selection of the correct level of detail.

3 RESULTS

The implemented features were tested on the Cornell Box scene. Figure 1 shows the result with all required tasks completed, demonstrating correct BVH traversal, direct lighting with hard shadows, and refraction through the glass sphere. Figure 2 showcases the result of the texture mapping analysis, featuring a brick texture applied to the floor.

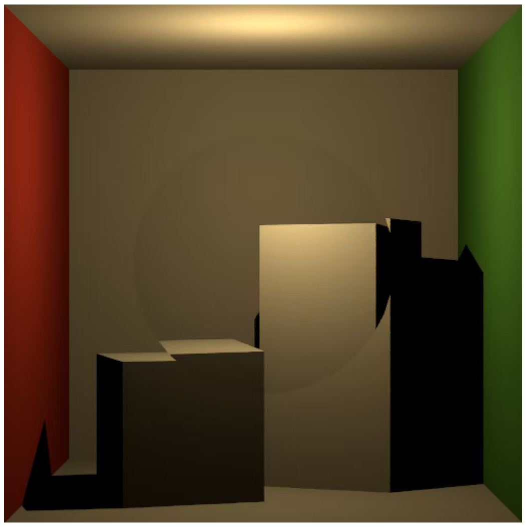
Fig. 1. Rendered image with all required tasks implemented.

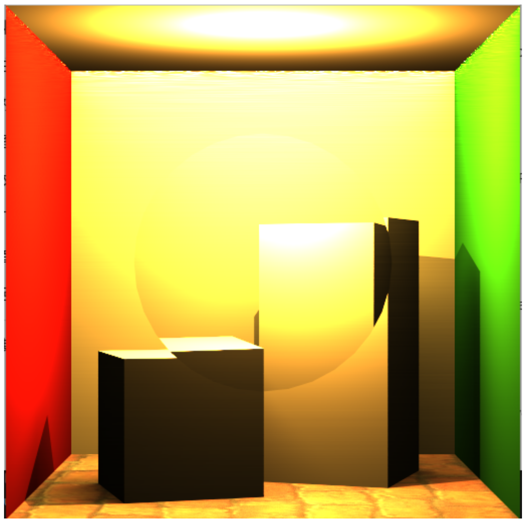
Fig. 2. Rendered image with optional task: texture mapping.