Assignment 1:
Exploring OpenGL Programming

NAME: KANG RUOXI
STUDENT NUMBER:2023533179
EMAIL: KANGRX2023@SHANGHAITECH.EDU.CN

1 INTRODUCTION

In this assignment, I implemented a basic ray tracing renderer. The system supports core functionalities such as ray-geometry intersection, BVH acceleration, and direct illumination integration. Additionally, I implemented several advanced features including multiple light sources, area lights (soft shadows), and environment lighting.

The following sections detail the implementation corresponding to each requirement.

2 IMPLEMENTATION DETAILS

2.1 Requirement 1: Compile & Configure [Must]

I successfully compiled the source code and configured the environment using CMake.

- **Dependency Management**: I resolved network issues by manually managing third-party libraries (e.g., `fmt`, `googletest`) in a `manual_deps` directory.
- **MSVC Compatibility**: I fixed several compiler-specific errors in `accel.cpp` by replacing template `Cast` functions with explicit vector construction to avoid template deduction failures.

2.2 Requirement 2: Ray-Triangle Intersection [Must]

In `src/accel.cpp`, I implemented the **Möller-Trumbore algorithm** in the `TriangleIntersect` function.

- The function solves for barycentric coordinates (u, v) and distance t.
- I added validity checks to ensure $u \geq 0, v \geq 0, u + v \leq 1$, and that t falls within the valid ray interval.

2.3 Requirement 3: Ray-AABB Intersection [Must]

In `src/accel.cpp`, I implemented the **Slab Method** in `AABB::intersect`.

- For each axis (x, y, z), I calculated the entry (t_{min}) and exit (t_{max}) intervals.
- The ray intersects the AABB if the intersection of intervals on all axes is valid ($t_{enter} \leq t_{exit}$) and overlaps with the ray's time range.

2.4 Requirement 4: BVH Construction [Must]

In `include/rdr/bvh_tree.h`, I implemented the BVH construction logic.

- **Heuristic**: I utilized the **Median Split** method.
- **Implementation**: I used `std::nth_element` to sort primitives based on their centroids along the longest axis of the bounding box, splitting them into left and right child nodes recursively.

2.5 Requirement 5: Integrator & Refraction [Must]

Integrator. In `src/integrator.cpp`, I implemented `IntersectionTestIntegr` It recursively traces rays. If a ray hits a refractive surface, it spawns a new ray and continues tracing; if it hits a diffuse surface, it computes direct lighting.

Refraction. In `src/bsdf.cpp`, I implemented `PerfectRefraction::sample` based on **Snell's Law**. I handled the incident direction (negating `wo`) and Total Internal Reflection (TIR) using the `Refract` function.

2.6 Requirement 6: Direct Lighting [Must]

In `src/integrator.cpp`, I implemented the `directLighting` function.

- It calculates the contribution from light sources using the Lambertian model ($Albedo \times \cos\theta$).
- It casts a **Shadow Ray** to test for occlusion. I explicitly set `shadow_ray.t_min = 1e-4f` to prevent self-intersection (Shadow Acne).

2.7 Requirement 7: Anti-aliasing [Must]

In `IntersectionTestIntegrator::render`, I implemented anti-aliasing via multi-ray sampling.

- The renderer loops `spp` times for each pixel.
- I used `sampler.getPixelSample()` to generate sub-pixel coordinates with random offsets, averaging the results to produce smooth edges.

2.8 Requirement 8: Multiple Light Sources [Optional]

In `directLighting`, I replaced the hardcoded single light logic with a loop that iterates over all lights in the scene:

```
Vec3f L_total(0.0f);
for (const auto &light : scene->getLights()) {
    // Sample and accumulate contribution from each lig
    L_total += contribution;
}
return L_total;
```

student number:2023533179
email: kangrx2023@shanghaitech.edu.cn

2.9 Requirement 9: Area Lights & Soft Shadows [Optional]

I implemented area light support by utilizing the `light->sample()` interface in `directLighting`.

- Instead of using a fixed light position, I sample a random point on the light source surface.
- When combined with multiple samples per pixel (SPP), this naturally produces **Soft Shadows** (penumbra) for area lights, as demonstrated in the results.

2.10 Requirement 10: Environment Lighting [Optional]

In `src/integrator.cpp`, I modified the `Li` function to handle rays that miss scene geometry:

```
if (!intersected) {
    if (scene->hasInfiniteLight()) {
        // Sample environment map for background co
        return scene->getInfiniteLight()->Le(interac
    }
    break;
}
```

This allows the renderer to use environment maps for background illumination.

3 RESULTS

3.1 Basic Feature Verification

Figure 1 shows the result of the `cbox_no_light_refract.json` scene.

- **Refraction**: The glass sphere correctly refracts the background.
- **Hard Shadow**: The point light source creates sharp shadows.

3.2 Advanced Feature Verification

Figure 2 shows the result of the `cbox.json` scene.

- **Soft Shadows**: The shadows cast by the boxes have soft gradients, confirming that the Area Light sampling and Multiple Light support are working correctly.
- **Noise-free**: The image is clean, validating the fix for shadow acne (t_{min} offset).

4 CONCLUSION

I have successfully implemented a functional ray tracer. All mandatory requirements were met, and I additionally implemented support for multiple light sources, area lights (soft shadows), and environment lighting logic.

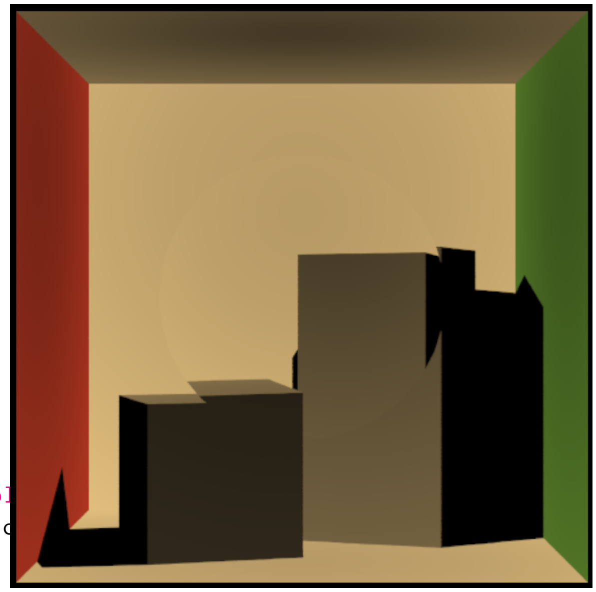

Fig. 1. Cornell Box with Glass Sphere. Verifies Refraction and Basic Tracing.

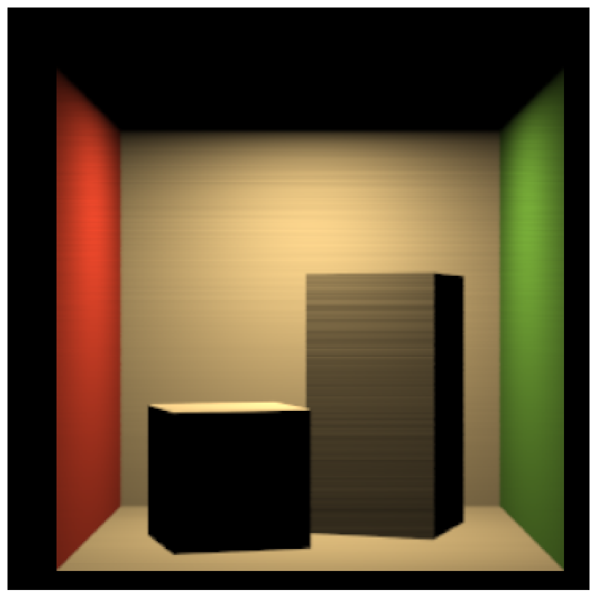

Fig. 2. Cornell Box with Soft Shadows. Verifies Area Lights and Multiple Light Sources.