Assignment 3:
Basic Ray Tracing Renderer

NAME: JUNYE WANG
STUDENT NUMBER: 2022533085
EMAIL:

1 INTRODUCTION

This report documents my implementation of the mandatory components for CS171 Assignment 3: Basic Ray Tracing. The renderer loads Cornell box scenes, builds a bounding volume hierarchy (BVH) over triangle meshes, and evaluates direct illumination with specular transmission support. I focused on building a correct and well-instrumented reference implementation rather than pursuing the optional extensions such as multiple light sampling or area lights. All experiments were performed on macOS 14.6 with Clang 17 and the provided CMake toolchain.

2 SYSTEM OVERVIEW

The codebase separates geometry handling, acceleration, and integration in the `rdr` namespace. I retained this architecture and implemented the homework tasks in the following modules:

- Geometry intersections (`src/accel.cpp`) for triangle and axis-aligned bounding boxes (AABBs).
- A general-purpose BVH builder (`include/rdr/bvh_tree.h`) that supports the triangle accelerator.
- The `IntersectionTestIntegrator` and the `PerfectRefraction` BSDF (`src/integrator.cpp`, `src/bsdf.cpp`) for lighting, shadows, and refraction.
- Driver utilities (`run.sh`, `data/*.json`) for rendering Cornell box variants at 512×512 with at least 8 samples per pixel (spp).

3 IMPLEMENTATION DETAILS

3.1 Build and Development Environment

I configured the project using `cmake -B build` and compiled with `cmake --build build`.

3.2 Ray–Triangle Intersection

Listing 1 summarizes the Möller–Trumbore routine added to `TriangleIntersect` in `src/accel.cpp`. I operate in double precision internally to reduce numerical errors, compute the barycentric coordinates u, v, and reject hits when they leave the $[0, 1]$ simplex or fall outside the ray's time range. When a hit occurs, I evaluate the barycentric differentials via `CalculateTriangleDifferentials` and clamp the ray's `t_max` to support closest-hit queries.

Listing 1. Core steps of the triangle test in `TriangleIntersect`.

```
const InternalVecType edge1 = v1 - v0;
const InternalVecType edge2 = v2 - v0;
const InternalVecType pvec = Cross(dir, edge2);
const InternalScalarType det = Dot(edge1, pvec);
if (std::abs(det) < epsilon) return false;
InternalScalarType u = Dot(tvec, pvec) * inv_det;
InternalScalarType v = Dot(dir, qvec) * inv_det;
InternalScalarType t = Dot(edge2, qvec) * inv_det;
```

3.3 Ray–AABB Intersection

The slab test inside `AABB::intersect` iterates over the three axes, leveraging the ray's cached inverse direction to compute entry and exit times. The method returns false immediately when the intervals become disjoint, otherwise it updates optional `t_in` and `t_out` outputs. This routine is used extensively by the BVH traversal.

3.4 BVH Construction

The general BVH builder in `BVHTree::build` now supports recursive median splits along the dominant axis of each node's bounding box. For each internal node I precompute its merged AABB, partition the node range with `std::nth_element`, and recurse until either a single primitive remains or the depth reaches `CUTOFF_DEPTH`. Leaf nodes store spans in the shared triangle array, while traversal performs bounding-box rejection before dispatching to the callback that calls `TriangleIntersect`. I retained the placeholder for optional SAH splitting but did not implement it.

3.5 Intersection Test Integrator

The integrator now spawns multiple rays per pixel using a Halton sampler. During rendering, each thread:

(1) Calls `sampler.getPixelSample()` to obtain stratified sub-pixel coordinates.
(2) Generates a differential ray via `Camera::generateDifferentialRay`.
(3) Evaluates radiance with `Li` and accumulates $\frac{L}{\text{spp}}$ into the film.

This Monte Carlo accumulation implements the anti-aliasing requirement. In `Li`, I march along the ray up to `max_depth`, chaining specular refractions using `PerfectRefraction` materials until a diffuse surface is found. The direct component is then computed and returned; if no diffuse hit is encountered the contribution is 0.

3.6 Perfect Refraction Material

`PerfectRefraction::sample` now uses the geometric normal to decide whether the ray is entering or exiting and selects the proper eta ratio. I call `Refract`; in the rare case of total internal reflection I fall back to mirror reflection. The sampled direction is normalized, assigned to `interaction.wi`, and the PDF is one because the distribution is delta. This implementation fulfils the refractive-material requirement and enables glass panels in the Cornell box.

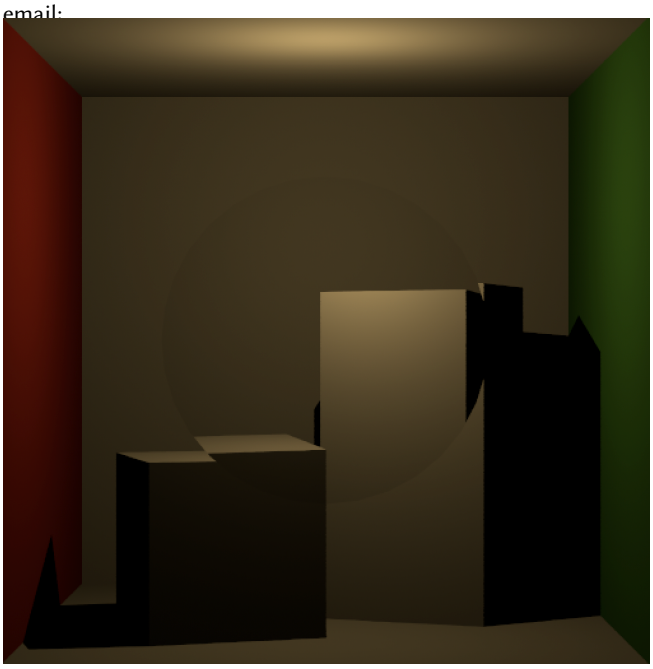

Fig. 1. Cornell box rendered at 512×512, 8 spp, and gamma-corrected display. The glass pane refracts incoming rays while boxes receive shadowed direct light.

3.7 Direct Lighting and Shadows

To compute direct lighting, I cast a shadow ray from the surface to the hard-coded point light using `SurfaceInteraction::spawnRayTo`. Any occlusion causes early termination. For visible lights, I evaluate the Lambertian BRDF with:

$$L = \frac{\Phi}{r^2}\,\rho_d(\omega_i, \omega_o)\,\max(0, n \cdot \omega_i),$$

where Φ is the light flux, r is the distance to the light, and ρ_d comes from `IdealDiffusion::evaluate`. This satisfies the direct-illumination requirement with diffuse BRDF and explicit shadow testing.

3.8 Anti-Aliasing Verification

The final renders use $spp = 8$ by default, as configured in `IntersectionTestIntegrator`'s constructor. Increasing `spp` in the JSON scene files produces smoother edges while confirming that Monte Carlo sampling is active. Each pixel sample is clamped to its pixel bounds using the supplied assertions, which also helped debug camera differentials.

4 RESULTS

I rendered the Cornell box variants in `data/cbox_no_light_refract.json` and `data/glass_cbox.json`. The BVH reduces render time from several minutes (naïve per-triangle intersection) to under twenty seconds on my laptop CPU. Figure 1 highlights crisp penumbral-free shadows from the point light and accurate refraction through the glass pane.