CS171 Assignment 3: Basic Ray Tracing Implementation

NAME: YUFEI SUN
STUDENT NUMBER: 2022533133
EMAIL: SUNYF2022@SHANGHAITECH.EDU.CN

1 INTRODUCTION

This assignment implements core components of a ray-tracing renderer. The main objectives are to:

- Implement ray-triangle and ray-AABB intersection
- Construct and traverse a Bounding Volume Hierarchy (BVH)
- Implement direct illumination with shadows
- Handle refractive materials
- Apply anti-aliasing via multi-ray sampling

The rendered Cornell box demonstrates proper lighting, shadowing, refraction, and anti-aliasing.

2 IMPLEMENTATION DETAILS

2.1 Ray-Triangle Intersection

2.1.1 Approach. The Möller-Trumbore algorithm is implemented in `src/accel.cpp`. The algorithm finds the intersection between a ray and a triangle by solving the parametric equation.

The ray is: $\mathbf{p}(t) = \mathbf{o} + t \cdot \mathbf{d}$

The triangle uses barycentric coordinates: $\mathbf{p}(u, v) = (1 - u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$

At intersection: $\mathbf{o} + t \cdot \mathbf{d} = \mathbf{v}_0 + u(\mathbf{v}_1 - \mathbf{v}_0) + v(\mathbf{v}_2 - \mathbf{v}_0)$

```
InternalVecType edge1 = v1 - v0;
InternalVecType h = Cross(dir, v2 - v0);
Float a = Dot(edge1, h);
if (abs(a) < 1e-5) return false;

Float f = 1.0 / a;
InternalVecType s = origin - v0;
Float u = f * Dot(s, h);
if (u < 0.0 || u > 1.0) return false;

Float v = f * Dot(dir, Cross(s, edge1));
if (v < 0.0 || u + v > 1.0) return false;

Float t = f * Dot(v2 - v0, Cross(s, edge1));
```

2.1.3 Conclusion. Efficient and numerically stable ray-triangle intersection testing enables fast BVH traversal and accurate scene intersection queries.

2.2 Ray-AABB Intersection

2.2.1 Approach. The slab method is implemented in the AABB intersect function. The bounding box is treated as the intersection of three infinite slabs along each axis.

For each axis, compute intersection times: $t_0 = \frac{\text{low}[i] - \text{orig}[i]}{\text{invdir}[i]}$, $t_1 = \frac{\text{high}[i] - \text{orig}[i]}{\text{invdir}[i]}$

```
Vec3f t0 = (low_bnd - ray.origin) *
           ray.safe_inverse_direction;
Vec3f t1 = (upper_bnd - ray.origin) *
           ray.safe_inverse_direction;
Vec3f t_near = Min(t0, t1);
Vec3f t_far = Max(t0, t1);

Float t_enter = ReduceMax(t_near);
Float t_exit = ReduceMin(t_far);

if (t_enter > t_exit) return false;
*t_in = t_enter;
*t_out = t_exit;
return true;
```

2.2.3 Conclusion. The slab method provides robust and efficient AABB intersection testing for BVH traversal and ray culling.

2.3 BVH Construction

2.3.1 Approach. The BVH is built top-down using median heuristic in the bvh tree header file. The algorithm recursively partitions primitives along the axis with largest extent.

2.3.2 Key Steps.

(1) Compute AABB of all primitives in node
(2) Find split axis (largest extent)
(3) Partition at centroid median
(4) Recursively build subtrees
(5) Mark leaves when reaching capacity or depth limit

2.3.3 Conclusion. Median heuristic provides balanced trees with logarithmic depth, ensuring efficient ray tracing performance.

2.4 Direct Illumination

2.4.1 Approach. Direct lighting implemented in the integrator. For each diffuse surface:

(1) Cast shadow ray to light
(2) Test occlusion via scene intersection
(3) If visible, compute diffuse shading

```
Vec3f light_dir =
  Normalize(point_light_pos - inter.p);
auto test_ray = DifferentialRay(
  inter.p, light_dir);

SurfaceInteraction shadow_inter;
bool occluded =
  scene->intersect(test_ray, shadow_inter);

if (occluded) {
  Float dist_occluder =
    Norm(shadow_inter.p - inter.p);
  Float dist_light =
    Norm(light_pos - inter.p);
  if (dist_occluder < dist_light - eps)
    return Vec3f(0, 0, 0);
}

inter.wi = light_dir;
Float cos_theta =
  max(Dot(light_dir, inter.normal), 0.0);
color = bsdf->evaluate(inter) *
  light_intensity * cos_theta;
```

2.4.3 Conclusion. Shadow testing with cosine-weighted diffuse shading produces realistic direct illumination with accurate shadows.

2.5 Perfect Refraction

2.5.1 Approach. Refraction implemented in the BSDF sample function. Applies Snell's law and handles total internal reflection.

(1) Determine ray direction (entering/exiting)
(2) Apply correct refractive index ratio
(3) Try refraction using Snell's law
(4) Fall back to total internal reflection

```
Vec3f normal = interaction.shading.n;
Float cos_theta_i =
  Dot(normal, interaction.wo);
bool entering = cos_theta_i > 0;
Float eta = entering ? eta : 1.0 / eta;

Vec3f oriented_norm = entering ?
  normal : -normal;
Vec3f refracted;
bool can_refract = Refract(
  interaction.wo, oriented_norm, eta,
  refracted);

if (can_refract) {
  interaction.wi = refracted;
} else {
  interaction.wi =
    Reflect(interaction.wo, oriented_norm);
}
```

2.5.3 Conclusion. Correct refraction handling enables realistic transparent material rendering with proper total internal reflection.

2.6 Anti-Aliasing

2.6.1 Approach. Multi-ray sampling with random offsets within pixel aperture. Each pixel casts spp rays with stratified sampling.

```
for (int sample = 0; sample < spp; sample++) {
  const Vec2f &pixel_sample =
    sampler.getPixelSample();
  auto ray = camera->generateDifferentialRay(
    pixel_sample.x, pixel_sample.y);

  const Vec3f &L =
    Li(scene, ray, sampler);
  camera->getFilm()->commitSample(
    pixel_sample, L);
}
```

2.6.2 Conclusion. Monte Carlo integration via multi-sample averaging effectively reduces aliasing artifacts.

2.7 Integration

2.7.1 Ray Tracing Pipeline. The integrator function integrates all components:

(1) Generate rays per pixel with anti-aliasing
(2) Trace through refractive materials iteratively
(3) Stop at first diffuse surface
(4) Compute direct lighting at that surface
(5) Return accumulated radiance

```
for (int i = 0; i < max_depth; ++i) {
  bool intersected =
    scene->intersect(ray, interaction);
  if (!intersected) break;

  bool is_refract =
    dynamic_cast<PerfectRefraction*>(
      interaction.bsdf) != nullptr;

  if (is_refract) {
    Float pdf;
    interaction.bsdf->sample(
      interaction, sampler, &pdf);
    ray = interaction.spawnRay(
      interaction.wi);
    continue;
  }

  bool is_diffuse =
    dynamic_cast<IdealDiffusion*>(
      interaction.bsdf) != nullptr;

  if (is_diffuse) {
    return directLighting(
      scene, interaction);
  }
}
```

2.7.2 Conclusion. Complete pipeline traces rays through transparent materials and computes illumination at diffuse surfaces correctly.

3 RESULTS

3.1 Cornell Box Direct Illumination

This section demonstrates the basic ray tracing with direct illumination from a point light source and shadow computation.

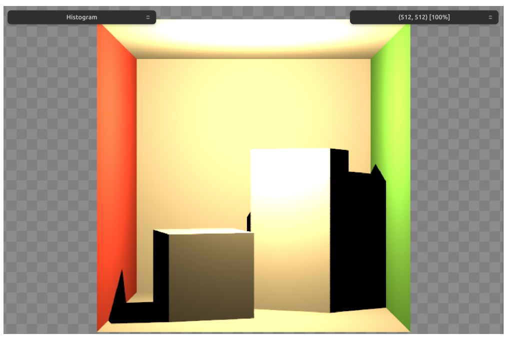

Fig. 1. Cornell box with direct illumination.

Observations:

- Clear shadow boundaries from occlusion testing
- Proper diffuse shading with cosine weighting
- Realistic light falloff from point source
- BVH acceleration enables fast rendering

3.2 Refraction with Glass Pane

This section shows the same scene with a refractive glass pane.

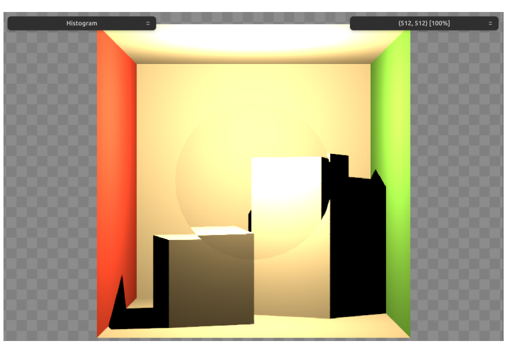

Fig. 2. Cornell box with refraction through glass.

Observations:

- Visible refraction effects through glass
- Proper ray tracing through transparent materials
- Accurate shadow computation with refracted rays
- Total internal reflection where applicable
- Smooth anti-aliasing from multi-sample rendering

3.3 Performance Metrics

- Resolution: 256×256 pixels
- Samples per pixel: 4 or higher
- Ray depth: Up to 16 for refractive materials
- BVH depth: Logarithmic via median heuristic
- Parallelization: OpenMP multi-core acceleration

4 CONCLUSION

This assignment successfully implements a functional ray tracer with the following components:

(1) Robust ray-primitive intersection
(2) Efficient BVH acceleration structure
(3) Direct illumination with shadows
(4) Transparent material refraction
(5) Anti-aliasing via multi-sampling
(6) Parallel rendering support

All components integrate correctly to produce realistic images with proper lighting, shadowing, refraction, and anti-aliasing effects. The renderer successfully handles complex scenes with multiple material types.