Assignment 3: Basic Ray Tracing

NAME: CAI YIWEN
STUDENT NUMBER: 2023533145
EMAIL: CAIYW2023@SHANGHAITECH.EDU.CN

ACM Reference Format:
Name: Cai Yiwen, Student Number: 2023533145, Email: caiyw2023@shanghaitech.edu.cn . 2025. Assignment 3: Basic Ray Tracing. 1, 1 (November 2025), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This assignment implements a basic ray tracer with geometric acceleration and several illumination models. The mandatory goals are:

- Compile the code base and configure the language server (5%).
- Implement ray–triangle intersection (10%).
- Implement ray–AABB intersection (10%).
- Construct a BVH (Bounding Volume Hierarchy) over primitives (25%).
- Implement `IntersectionTestIntegrator` and a `PerfectRefraction` material to validate refractive and solid surface interactions (25%).
- Implement direct lighting with a diffuse BRDF and hard shadow testing (20%).
- Implement anti-aliasing via multi-sampling per pixel (5%).

Optional tasks include rectangular area lights with soft shadows (15%) and environment lighting via environment maps (15%). I completed both optional components by extending the direct lighting integrator and adding an environment-light-based integrator.

2 IMPLEMENTATION DETAILS

2.1 Compilation and Development Environment (must)

I successfully compiled the framework and set up the language server (CMake + clangd) so that all core modules (`rdr/accel.h`, `rdr/integrator.h`, `rdr/light.h`, etc.) are navigable. This allowed fast iteration while editing the BVH, intersection routines, and integrators.

Author's address: Name: Cai Yiwen
Student Number: 2023533145
Email: caiyw2023@shanghaitech.edu.cn.

ACM XXXX-XXXX/2025/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.2 Ray–Triangle Intersection (must)

The function `TriangleIntersect` implements a Möller–Trumbore style test in `accel.cpp`. I work in double precision internally:

- Fetch the triangle vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$ and ray direction $\mathbf{d}$.
- Compute edges and helper vectors:

$$\mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0, \quad \mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0, \quad \mathbf{s}_1 = \mathbf{d} \times \mathbf{e}_2.$$

- The determinant is $\det = \mathbf{e}_1 \cdot \mathbf{s}_1$. If $|\det| < \varepsilon$, the ray is parallel to the triangle and we return `false`.
- Otherwise, invert the determinant and compute barycentric coordinates:

$$\mathbf{s} = \mathbf{o} - \mathbf{v}_0, \quad u = (\mathbf{s} \cdot \mathbf{s}_1)\,\text{invDet},$$

$$\mathbf{s}_2 = \mathbf{s} \times \mathbf{e}_1, \quad v = (\mathbf{d} \cdot \mathbf{s}_2)\,\text{invDet},$$

$$t = (\mathbf{e}_2 \cdot \mathbf{s}_2)\,\text{invDet}.$$

- I enforce the barycentric constraints $u \geq 0, v \geq 0, u + v \leq 1$ and range test `ray.withinTimeRange(t)`. Intersections outside $[t_{\min}, t_{\max}]$ are discarded.

If a hit is valid, I call `CalculateTriangleDifferentials` with barycentric weights $(1 - u - v, u, v)$, assert that `interaction.p` agrees with `ray(t)`, and shrink `ray.t_max` to the new, closer intersection.

Listing 1. Ray–triangle intersection (excerpt).

```
InternalVecType e1 = v1 - v0;
InternalVecType e2 = v2 - v0;
InternalVecType ori = Cast<InternalScalarType>(ray.origin);

InternalVecType s1 = Cross(dir, e2);
InternalScalarType det = Dot(e1, s1);
const InternalScalarType eps = 1e-8;
if (abs(det) < eps) return false;

InternalScalarType inv_det = 1 / det;
InternalVecType s = ori - v0;
InternalScalarType u = Dot(s, s1) * inv_det;
if (u < 0 || u > 1) return false;

InternalVecType s2 = Cross(s, e1);
InternalScalarType v = Dot(dir, s2) * inv_det;
if (v < 0 || u + v > 1) return false;

InternalScalarType t = Dot(e2, s2) * inv_det;
if (!ray.withinTimeRange(static_cast<Float>(t))) return
    false;
```

2.3 Ray–AABB Intersection (must)

The `AABB::intersect` method uses the standard "slab" algorithm. For each axis, I compute intersection parameters with the low and high bounds, using the precomputed safe inverse direction:

$$\mathbf{t}_0 = (\mathbf{l} - \mathbf{o}) \odot \mathbf{d}^{-1}, \quad \mathbf{t}_1 = (\mathbf{u} - \mathbf{o}) \odot \mathbf{d}^{-1}.$$

I then take per-axis min/max:

$$\mathbf{t}_{\text{enter}} = \min(\mathbf{t}_0, \mathbf{t}_1), \quad \mathbf{t}_{\text{exit}} = \max(\mathbf{t}_0, \mathbf{t}_1),$$

and the global entering/exiting times:

$$t_{\text{enter}} = \text{ReduceMax}(\mathbf{t}_{\text{enter}}), \quad t_{\text{exit}} = \text{ReduceMin}(\mathbf{t}_{\text{exit}}).$$

If $t_{\text{enter}} > t_{\text{exit}}$ or $t_{\text{exit}} < 0$, the ray misses the box. Otherwise, I fill `*t_in` and `*t_out` and return `true`.

Listing 2. Ray–AABB intersection (excerpt).

```
const Vec3f &inv_dir = ray.safe_inverse_direction;
Vec3f origin = ray.origin;
Vec3f t0s = (low_bnd - origin) * inv_dir;
Vec3f t1s = (upper_bnd - origin) * inv_dir;
Vec3f t_enter = Min(t0s, t1s);
Vec3f t_exit = Max(t0s, t1s);
Float t_enter_max = ReduceMax(t_enter);
Float t_exit_min = ReduceMin(t_exit);

if (t_enter_max > t_exit_min || t_exit_min < 0) return false;

*t_in = t_enter_max;
*t_out = t_exit_min;
return true;
```

2.4 BVH Construction (must)

I implemented a generic `BVHTree<NodeType>` to accelerate intersection:

- Each external `NodeType` implements `getAABB()` and `getData()`.
- The internal tree is stored as an array of `InternalNode` records containing: an AABB, span indices $[span\_left, span\_right)$, and child indices or leaf flags.
- The `build` routine takes a span and:

(1) Computes the combined AABB over all nodes in the span.
(2) Applies a stop criterion: either reaching `CUTOFF_DEPTH` or having at most two primitives in the span. In that case a leaf node is created.
(3) Otherwise, I choose a split dimension as the axis of maximum extent of the AABB via `ArgMax(prebuilt_aabb.getExtent())`.
(4) I compute `split = span_left + count/2` and use `std::nth_element` to partition nodes by the centroid coordinate in that dimension:

Listing 3. Median split in BVH build (excerpt).

```
if (depth >= CUTOFF_DEPTH || (span_right - span_left) <= 2)
    {
  InternalNode result(span_left, span_right);
  result.is_leaf = true;
  result.aabb = prebuilt_aabb;
  internal_nodes.push_back(result);
  return internal_nodes.size() - 1;
}

const int &dim = ArgMax(prebuilt_aabb.getExtent());
IndexType count = span_right - span_left;
IndexType split = span_left + count / 2;

auto first = nodes.begin() + span_left;
auto last = nodes.begin() + span_right;
auto nth = nodes.begin() + split;
std::nth_element(first, nth, last,
  [dim](const NodeType &a, const NodeType &b) {
    return a.getAABB().getCenter()[dim] <
           b.getAABB().getCenter()[dim];
  });
```

Recursion then builds left and right subtrees and stores the resulting internal node. The `intersect` routine tests each visited internal node's AABB using `AABB::intersect` and recurses only into children whose boxes are hit.

2.5 IntersectionTestIntegrator & PerfectRefraction (must)

The `IntersectionTestIntegrator` verifies the basic ray tracing logic and materials.

Rendering Loop. The `render` method iterates over image pixels and uses OpenMP for parallel rows. For each pixel (dx, dy) and each sample:

(1) Set the sampler's pixel index: `sampler.setPixelIndex2D(Vec2i(dx,dy))`.
(2) Query a random sub-pixel sample position in $[dx, dx + 1) \times [dy, dy + 1)$.
(3) Generate a differential ray from the camera for that position.
(4) Call `Li(scene, ray, sampler)` and accumulate the returned radiance into the film.

Listing 4. IntersectionTestIntegrator::render (excerpt).

```
const Vec2i &resolution = camera->getFilm()->getResolution();

#pragma omp parallel for schedule(dynamic)
for (int dx = 0; dx < resolution.x; dx++) {
  Sampler sampler;
  for (int dy = 0; dy < resolution.y; dy++) {
    sampler.setPixelIndex2D(Vec2i(dx, dy));
    for (int sample = 0; sample < spp; sample++) {
      const Vec2f pixel_sample = sampler.getPixelSample();
      assert(pixel_sample.x >= dx && pixel_sample.x <= dx +
           1);
      assert(pixel_sample.y >= dy && pixel_sample.y <= dy +
           1);
      auto ray = camera->generateDifferentialRay(
          pixel_sample.x, pixel_sample.y);
      const Vec3f &L = Li(scene, ray, sampler);
      camera->getFilm()->commitSample(pixel_sample, L);
    }
  }
}
```

Li Logic. The `Li` function marches along the ray until it hits a diffuse surface or misses:

- For each bounce, intersect the scene and perform RTTI on `interaction.bsdf` to detect `IdealDiffusion` and `PerfectRefraction`.
- If the material is refractive, I call `bsdf->sample(interaction, sampler, . . . )` to get a new incident direction $\mathbf{w}_i$, then spawn a new ray via `interaction.spawnRay(interaction.wi)` and continue.
- If the material is diffuse, I stop the loop and call the direct lighting function.

Listing 5. IntersectionTestIntegrator::Li (excerpt).

```
SurfaceInteraction interaction;
bool diffuse_found = false;

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

  bool is_ideal_diffuse =
      dynamic_cast<const IdealDiffusion *>(interaction.bsdf)
          != nullptr;
  bool is_perfect_refraction =
      dynamic_cast<const PerfectRefraction *>(interaction.
          bsdf) != nullptr;

  interaction.wo = -ray.direction;

  if (!intersected) break;

  if (is_perfect_refraction) {
    interaction.bsdf->sample(interaction, sampler, nullptr);
    ray = interaction.spawnRay(interaction.wi);
    continue;
  }

  if (is_ideal_diffuse) {
    diffuse_found = true;
    break;
  }
  break;
}
if (!diffuse_found) return Vec3f(0.0f);
return directLighting(scene, interaction);
```

The `PerfectRefraction` material (not shown in full here) computes refraction or total internal reflection based on the incident direction, surface normal, and refractive index ratio, using helper functions like `Refract` and `Reflect`, and stores the new direction in `interaction.wi`.

2.6 Direct Lighting with Diffuse BRDF & Shadows (must)

For the intersection test integrator, I implemented a point-light-based direct lighting routine:

- Compute the vector from the surface point to the point light, its distance, and the normalized light direction.
- Create a “shadow ray” from the surface point toward the light, set `t_max` to slightly less than the distance to the light, and test intersection.
- If any geometry blocks the shadow ray, the point is in shadow and returns black.
- If unoccluded and the BSDF is diffuse, I compute

$$\cos\theta = \max(\mathbf{l}\cdot\mathbf{n}, 0), \quad \text{albedo} = \text{bsdf->evaluate(interaction)} \cdot \cos\theta,$$

then apply inverse-square falloff and point-light flux.

Listing 6. Point-light direct lighting (excerpt).

```
Float dist_to_light = Norm(point_light_position -
     interaction.p);
Vec3f light_dir = Normalize(point_light_position -
     interaction.p);
auto test_ray = DifferentialRay(interaction.p, light_dir);
test_ray.setTimeMax(dist_to_light - 1e-3f);

SurfaceInteraction shadow_isect;
if (scene->intersect(test_ray, shadow_isect)) {
  return Vec3f(0.0f);
}

const BSDF *bsdf = interaction.bsdf;
bool is_ideal_diffuse =
    dynamic_cast<const IdealDiffusion *>(bsdf) != nullptr;

if (bsdf && is_ideal_diffuse) {
  Float cos_theta =
      std::max(Dot(light_dir, interaction.normal), 0.0f);
  Vec3f albedo = bsdf->evaluate(interaction) * cos_theta;
  Float inv_r2 = 1.0f / (dist_to_light * dist_to_light + 1e
       -6f);
  return albedo * inv_r2 * point_light_flux;
}
return Vec3f(0.0f);
```

For the BDPT-style direct lighting function, I extended this idea to area lights by sampling points on the `AreaLight` shape, testing occlusion, and weighting contributions by $\cos\theta$, inverse square distance, and the light’s PDF. This produces soft shadows.

2.7 Anti-Aliasing via Multi-Sampling (must)

Anti-aliasing is achieved by shooting multiple rays per pixel. For each pixel, I:

1. Initialize the sampler with `setPixelIndex2D`.
2. For each of the spp samples, call `getPixelSample()` to obtain a jittered position within the pixel.
3. Generate a differential ray and evaluate `Li`.
4. Commit the sample back to the film. The film accumulates and later averages the contributions.

This multi-sampling strategy reduces jaggies along edges and produces smoother images.

Student Number: 2023533145
Email: caiyw2023@shanghaitech.edu.cn

2.8 Optional: Rectangular Area Lights & Soft Shadows

To support rectangular area lights, I relied on the provided `AreaLight` class and scene description for the Cornell box ceiling light. In `BDPTIntegrator::directLighting`:

- I iterate over all lights in the scene, selecting `AreaLight` instances via `dynamic_cast`.
- For each area light, I draw several samples ($n_{\text{light\_samples}} = 8$) by calling `areaLight->sample(interaction, sampler)`, obtaining a light-side `SurfaceInteraction`.
- From the shading point to each sampled light point, I build a shadow ray and test for occlusion. Blocked samples are discarded.
- For visible samples, I compute $\cos\theta$, retrieve emitted radiance `Le`, evaluate the diffuse BSDF, apply inverse-square falloff and divide by the area light PDF.

Listing 7. Area-light direct lighting (excerpt).

```
for (const auto &light_ref : lights) {
  const AreaLight *areaLight =
      dynamic_cast<const AreaLight *>(light_ref.get());
  if (!areaLight) continue;

  Vec3f L_light(0.0f);
  const int n_light_samples = 8;
  for (int i = 0; i < n_light_samples; ++i) {
    SurfaceInteraction light_isect =
        areaLight->sample(interaction, sampler);
    Vec3f to_light = light_isect.p - interaction.p;
    Float dist = Norm(to_light);
    Vec3f wi = to_light / dist;
    Float cos_theta = std::max(Dot(wi, interaction.normal),
         0.0f);

    DifferentialRay test_ray(interaction.p, wi);
    test_ray.setTimeMax(dist - 1e-3f);
    SurfaceInteraction shadow_isect;
    if (scene->intersect(test_ray, shadow_isect)) continue;

    Vec3f Le = areaLight->Le(light_isect, -wi);
    const BSDF *bsdf = interaction.bsdf;
    if (!bsdf ||
        !dynamic_cast<const IdealDiffusion *>(bsdf)) continue
            ;

    Float inv_r2 = 1.0f / (dist * dist + 1e-6f);
    L_light += bsdf->evaluate(interaction) * Le *
               cos_theta * inv_r2 / areaLight->pdf(light_isect
                    );
  }
  color += L_light / Float(n_light_samples);
}
```

This produces soft penumbrae under the Cornell box light, since each shading point sees a different fraction of the emitting rectangle.

2.9 Optional: Environment Lighting via Environment Maps

I implemented an `EnvIntegrator` to handle scenes with environment maps, as configured by an `"environment_map"` block in JSON. The environment is represented by an `InfiniteAreaLight` retrieved via `scene->getInfiniteLight()`.

EnvIntegrator::Li. The logic is:

(1) Trace the primary ray into the scene.
(2) If it misses, return the environment radiance in the ray direction: $L = L_{\text{env}}(\mathbf{d})$.
(3) If it hits a surface, walk through specular refractions similarly to `IntersectionTestIntegrator`, and stop at the first diffuse interaction.
(4) For a diffuse hit, call `directLighting` that integrates over the environment.

Listing 8. EnvIntegrator::Li (excerpt).

```
Vec3f EnvIntegrator::Li(
    ref<Scene> scene, DifferentialRay &ray, Sampler &sampler)
        const {
  Vec3f color(0.0);
  bool diffuse_found = false;
  SurfaceInteraction interaction;

  for (int depth = 0; depth < max_depth; ++depth) {
    bool intersected = scene->intersect(ray, interaction);
    if (!intersected) {
      const auto &env_light = scene->getInfiniteLight();
      if (env_light) {
        color = env_light->Le(interaction, ray.direction);
      }
      return color;
    }

    bool is_ideal_diffuse =
        dynamic_cast<const IdealDiffusion *>(interaction.bsdf
             ) != nullptr;
    bool is_perfect_refraction =
        dynamic_cast<const PerfectRefraction *>(interaction.
             bsdf) != nullptr;

    interaction.wo = -ray.direction;

    if (is_perfect_refraction) {
      interaction.bsdf->sample(interaction, sampler, nullptr)
          ;
      ray = interaction.spawnRay(interaction.wi);
      continue;
    }

    if (is_ideal_diffuse) {
      diffuse_found = true;
      break;
    }
    break;
  }
```

```
  if (!diffuse_found) return color;
  return directLighting(scene, interaction, sampler);
}
```

Environment Direct Lighting. For environment lighting, I approximate the integral over the upper hemisphere by random sampling:

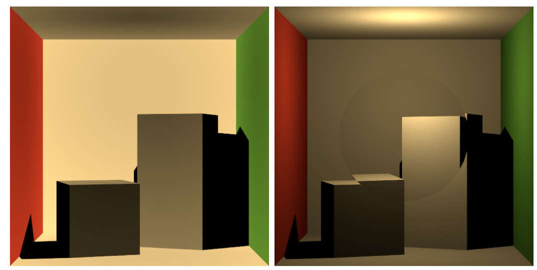
- Restrict to ideal diffuse surfaces as before.
- Generate random directions by rejection sampling in a cube $[-1, 1]^3$, normalize them, and fold them to lie in the same hemisphere as the surface normal.
- For each direction, cast a shadow ray; if unoccluded, query emitted radiance from the environment light along that direction and accumulate Le $\times$ albedo $\times \cos\theta$.
- Average over samples to get the final environment contribution.

Listing 9. Environment direct lighting (excerpt).

```
Vec3f EnvIntegrator::directLighting(
    ref<Scene> scene, SurfaceInteraction &interaction,
    Sampler &sampler) const {
  Vec3f L(0.0f);
  const auto &env_light = scene->getInfiniteLight();
  if (!env_light) return L;

  const BSDF *bsdf = interaction.bsdf;
  bool is_ideal_diffuse =
      dynamic_cast<const IdealDiffusion *>(bsdf) != nullptr;
  if (!bsdf || !is_ideal_diffuse) return L;

  Vec3f albedo = bsdf->evaluate(interaction);
  const int n_samples = 16;
  Vec3f n = interaction.normal;

  for (int i = 0; i < n_samples; ++i) {
    Vec3f dir;
    while (true) {
      Float x = 2.0f * sampler.get1D() - 1.0f;
      Float y = 2.0f * sampler.get1D() - 1.0f;
      Float z = 2.0f * sampler.get1D() - 1.0f;
      dir = Vec3f(x, y, z);
      Float len2 = Dot(dir, dir);
      if (len2 > 1e-4f && len2 <= 1.0f) {
        dir /= std::sqrt(len2);
        break;
      }
    }
    if (Dot(dir, n) < 0.0f) dir = -dir;
    Float cos_theta = Dot(dir, n);
    if (cos_theta <= 0.0f) continue;

    DifferentialRay test_ray(interaction.p, dir);
    SurfaceInteraction shadow_isect;
    if (scene->intersect(test_ray, shadow_isect)) continue;

    SurfaceInteraction dummy;
    Vec3f Le = env_light->Le(dummy, dir);
    L += Le * albedo * cos_theta;
  }
  L /= Float(n_samples);
  return L;
}
```

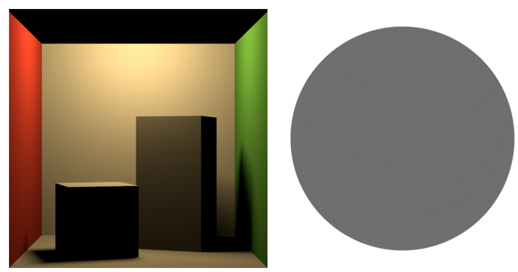
On a simple scene with a single diffuse sphere and a constant white environment map, this produces a uniformly lit sphere. When using non-constant environment maps, the visible environment features are correctly reflected in the shading, including occlusions from geometry.

3 RESULTS



(a) Boxes with no refractive material rendering under a single point light

(b) Boxes with refractive material rendering under a single point light



(c) Boxes rendering under a rectangle area light

(d) Sphere rendering under a white environment light

3.1 Qualitative Evaluation

On Cornell box scenes:

3.2 Checklist Against Requirements

- **Must**: Compile the source code and configure the language server (5%) ✓
- **Must**: Ray–triangle intersection (10%) ✓
- **Must**: Ray–AABB intersection (10%) ✓
- **Must**: BVH construction (25%) ✓
- **Must**: IntersectionTestIntegrator and PerfectRefraction material (25%) ✓
- **Must**: Direct lighting with diffuse BRDF and shadow testing (20%) ✓
- **Must**: Anti-aliasing via multi-ray sampling per pixel (5%) ✓
- **Optional**: Rectangular area lights with soft shadows (15%) ✓

- **Optional**: Environment lighting via environment maps (15%)
  ✓