Assignment 1:
Exploring OpenGL Programming

NAME: DING CHANGRUI
STUDENT NUMBER:2354457294
EMAIL: DINGCHR2023@SHANGHAITECH.EDU.CN

1 INTRODUCTION

This assignment implements the core components of a minimal educational ray tracer, including ray-geometry intersection, BVH acceleration structures, direct illumination, and interaction with refractive materials. The final renderer is able to produce a Cornell Box scene containing diffuse, shadow-casting objects and refractive materials such as glass.

The major implemented features include:

- Raytriangle intersection
- RayAABB intersection
- BVH construction and traversal
- Direct illumination integrator with shadow ray testing
- Perfect refraction (dielectric)
- Anti-aliasing using multi-sample jittering

Below, I will detail how each task is implemented.

2 IMPLEMENTATION DETAILS

2.1 Ray-Triangle Intersection

I implemented the function `TriangleIntersect` using the MöllerTrumbore algorithm. The algorithm solves for (t, u, v) and checks barycentric conditions:

```
bool TriangleIntersect(Ray &ray, const
    uint32_t &triangle_index,
  ...

  InternalVecType dir = Cast<
      InternalScalarType>(ray.direction);
  InternalVecType v0  = Cast<
      InternalScalarType>(vertices[v_idx[0]])
      ;
  InternalVecType v1  = Cast<
      InternalScalarType>(vertices[v_idx[1]])
      ;
  InternalVecType v2  = Cast<
      InternalScalarType>(vertices[v_idx[2]])
      ;

  const InternalVecType edge1 = v1 - v0;
  const InternalVecType edge2 = v2 - v0;
  const InternalVecType pvec  = Cross( dir,
      edge2);
  const InternalScalarType det = Dot(edge1,
      pvec);
  const InternalScalarType eps =
      InternalScalarType(1.0000e-12);

  if (abs(det) < eps) return false;

  const InternalScalarType inv_det =
      InternalScalarType(1) / det;
  const InternalVecType tvec = Cast<
      InternalScalarType>(ray.origin) - v0;
  const InternalScalarType u = Dot(tvec, pvec
      ) * inv_det;

  if (u < InternalScalarType(0) || u >
      InternalScalarType(1))
    return false;

  const InternalVecType qvec = Cross(tvec,
      edge1);
  const InternalScalarType v = Dot(dir, qvec)
       * inv_det;

  if (v < InternalScalarType(0) || (u + v) >
      InternalScalarType(1))
    return false;

  const InternalScalarType t = Dot(edge2,
      qvec) * inv_det;

  if (!ray.withinTimeRange(static_cast<Float
      >(t))) return false;
...
}
```

2.2 RayAABB Intersection

I implemented slab-based AABB intersection in `AABB::intersect`. The ray direction is inverted once and reused.

```
bool AABB::intersect(const Ray &ray, Float *
    t_in, Float *t_out) const {
  Vec3f inv_dir = ray.safe_inverse_direction;
```

```
  Vec3f t0 = (low_bnd   - ray.origin) *
      inv_dir;
  Vec3f t1 = (upper_bnd - ray.origin) *
      inv_dir;

  Vec3f t_enter = Min(t0, t1);
  Vec3f t_exit  = Max(t0, t1);
  Float t_enter_max = ReduceMax(t_enter);
  Float t_exit_min  = ReduceMin(t_exit);

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

  if (t_in != nullptr)  *t_in  = t_enter_max;
  if (t_out != nullptr) *t_out = t_exit_min;
  return true;
}
```

2.3 BVH Construction

The BVH is implemented in `BVHTree::build`. The algorithm uses recursive top-down construction:

- Compute bounding box for primitives in range $[L, R)$
- Choose split axis based on max dimension of centroid bounds
- Partition primitives along the chosen axis
- Recursively build left and right children

```
typename BVHTree<_>::IndexType BVHTree<_>::
    build(
    int depth, const IndexType &span_left,
        const IndexType &span_right) {
  ...
  if (depth >= CUTOFF_DEPTH || (span_right -
      span_left) <= 1)
  {
    // create leaf node
    const auto &node = nodes[span_left];
    InternalNode result(span_left, span_right
        );
    result.is_leaf = true;
    result.aabb    = prebuilt_aabb;
    internal_nodes.push_back(result);
    return internal_nodes.size() - 1;
  }

  InternalNode result(span_left, span_right);

  // const int &dim = depth % 3;
  const int &dim  = ArgMax(prebuilt_aabb.
      getExtent());
  IndexType count = span_right - span_left;
  IndexType split = INVALID_INDEX;

  if (hprofile == EHeuristicProfile::
      EMedianHeuristic) {
    use_median_heuristic:
    split = span_left + count / 2;
        split = span_left + count / 2;

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

    // clang-format on
  }
  ...
}
```

2.4 Direct Illumination Integrator

The provided integrator computes per-pixel color by:

(1) Casting *spp* primary rays with subpixel jitter (anti-aliasing).
(2) Finding closest intersection using BVH.
(3) Casting a shadow ray toward the point light.
(4) Evaluating diffuse BRDF for non-occluded paths.

Phong-like diffuse shading:

$$L = \rho \max(0, n \cdot \omega_{light})$$

```
void IntersectionTestIntegrator::render(ref<
    Camera> camera, ref<Scene> scene) {
  // Statistics
  std::atomic<int> cnt = 0;

  const Vec2i &resolution = camera->getFilm()
      ->getResolution();
#pragma omp parallel for schedule(dynamic)
  for (int dx = 0; dx < resolution.x; dx++) {
    ++cnt;
    if (cnt % (resolution.x / 10) == 0)
      Info_("Rendering:␣{:.02f}%", cnt *
          100.0 / resolution.x);
    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();

        auto ray = camera->
            generateDifferentialRay(
            pixel_sample.x,pixel_sample.y);

        assert(pixel_sample.x >= dx &&
            pixel_sample.x <= dx + 1);
        assert(pixel_sample.y >= dy &&
            pixel_sample.y <= dy + 1);
        const Vec3f &l = Li(scene, ray,
            sampler);
        camera->getFilm()->commitSample(
            pixel_sample, l);
      }
    }
  }
}
Vec3f IntersectionTestIntegrator::
    directLighting(
    ref<Scene> scene, SurfaceInteraction &
        interaction) const {
  Vec3f color(0, 0, 0);
  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 - 1.000e
      -4F);
  //Perform shadow detection
  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 != nullptr && is_ideal_diffuse) {
    Float cos_theta =
        std::max(Dot(light_dir, interaction.
            normal), 0.0F);  // one-sided

    Vec3f albedo = bsdf->evaluate(interaction
        );
    color = albedo * cos_theta / (
        dist_to_light * dist_to_light);
  }

  return color;
}
```

2.5 Implement a Direct Illumination Integrator

Support for refractive materials is added by extending the BSDF to implement perfect refraction. When a ray hits a transparent surface, the renderer determines whether it is entering or exiting the medium and applies Snells law to compute the transmitted direction. If refraction is not possible, the material falls back to perfect reflection. By integrating this behavior into the sampling and shading pipeline, the renderer correctly handles light bending and total internal reflection, enabling realistic rendering of glass-like objects.

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

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

    // Perform RTTI to determine the type of
        the surface
    bool is_ideal_diffuse =
        dynamic_cast<const IdealDiffusion *>(
            interaction.bsdf) != nullptr;
    bool is_perfect_refraction =
        dynamic_cast<const PerfectRefraction
            *>(interaction.bsdf) != nullptr;

    // Set the outgoing direction
    interaction.wo = -ray.direction;

    if (!intersected) {
      break;
    }

    if (is_perfect_refraction) {
      Float pdf = 1.0;
      Vec3f bsdf_val = interaction.bsdf->
          sample(interaction, sampler, &pdf);
      ray = interaction.spawnRay(interaction.
          wi);
      continue;
```

```
      }

      if (is_ideal_diffuse) {
        // We only consider diffuse surfaces
            for direct lighting
        diffuse_found = true;
        break;
      }

      // We simply omit any other types of
          surfaces
      break;
    }

    if (!diffuse_found) {
      return color;
    }

    color = directLighting(scene, interaction);
    return color;
}
```

3 RESULTS

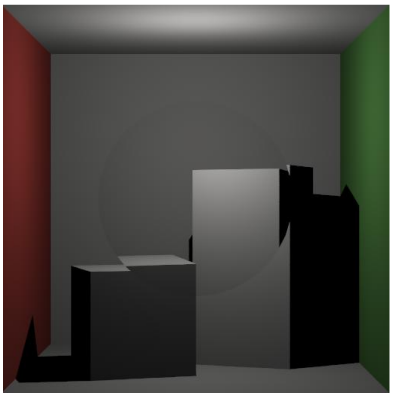

Fig. 1. final picture