Assignment 3: Basic Ray Tracing

NAME: YIFEI CHEN
STUDENT NUMBER: 2023533029
EMAIL: CHENYF2023@SHANGHAITECH.EDU.CN

1 INTRODUCTION

In this homework, I complete all the required tasks as well as three optional tasks: Support for multiple light sources, support for area lights and texture mapping.

2 IMPLEMENTATION DETAILS

2.1 Compile

I use clangd with Ninja generator to compile the code. The commands are as follows:

```
cmake -G "Ninja" -B build -
    DCMAKE_EXPORT_COMPILE_COMMANDS=ON -
    DCMAKE_BUILD_TYPE=RelWithDebInfo
cmake --build build
```

2.2 Ray-Triangle Intersection

The code below shows the implementation of the Moller-Trumbore algorithm for ray-triangle intersection.

```
InternalScalarType u = InternalScalarType(0);
InternalScalarType v = InternalScalarType(0);
InternalScalarType t = InternalScalarType(0);
auto e1 = v1 - v0;
auto e2 = v2 - v0;
auto s = Cast<InternalScalarType>(ray.origin)
     - v0;
auto s1 = Cross(dir, e2);
auto s2 = Cross(s, e1);
auto denom = Dot(s1, e1);
u = Dot(s1, s) / denom;
v = Dot(s2, dir) / denom;
t = Dot(s2, e2) / denom;
if (std::abs(denom) < InternalScalarType(1e
    -8) || u < InternalScalarType(0) ||
  v < InternalScalarType(0) || u + v >
      InternalScalarType(1) ||
  t < InternalScalarType(ray.t_min) || t >
      InternalScalarType(ray.t_max)) {
return false;
}
```

To be specific, given a ray defined as $R(t) = O + td$ and a triangle defined by its three vertices V_0, V_1 and V_2, we can solve the intersection by solving the equation

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{S_1 \cdot E_1} \begin{bmatrix} S_2 \cdot E_2 \\ S_1 \cdot S \\ S_2 \cdot d \end{bmatrix},$$

where $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$, $S = O - V_0$, $S_1 = d \times E_2$, $S_2 = S \times E_1$. If $t > 0$, $u > 0$, $v > 0$ and $u + v < 1$, then the ray intersects with the triangle at point $P = V_0 + uE_1 + vE_2$. In the code we return false if the requirements are not satisfied.

2.3 Ray-AABB Intersection

`AABB::intersect` is implemented as follows:

```
const Vec3f &invdir = ray.
    safe_inverse_direction;

Float tx1 = (low_bnd.x - ray.origin.x) *
    invdir.x;
Float tx2 = (upper_bnd.x - ray.origin.x) *
    invdir.x;
Float tmin = std::min(tx1, tx2);
Float tmax = std::max(tx1, tx2);

Float ty1 = (low_bnd.y - ray.origin.y) *
    invdir.y;
Float ty2 = (upper_bnd.y - ray.origin.y) *
    invdir.y;
tmin = std::max(tmin, std::min(ty1, ty2));
tmax = std::min(tmax, std::max(ty1, ty2));

Float tz1 = (low_bnd.z - ray.origin.z) *
    invdir.z;
Float tz2 = (upper_bnd.z - ray.origin.z) *
    invdir.z;
tmin = std::max(tmin, std::min(tz1, tz2));
tmax = std::min(tmax, std::max(tz1, tz2));

if (tmax < tmin || tmax < ray.t_min || tmin >
     ray.t_max)
  return false;

if (t_in)
  *t_in = tmin;
```

```
if (t_out)
  *t_out = tmax;
return true;
```

For each axis, it computes the parametric distances tx1, tx2, ty1, ty2, tz1, tz2 at which the ray intersects the lower and upper bounds of the box, using the precomputed safe_inverse_direction. For each axis, the entry and exit times are obtained using std::min and std::max, and the global intersection interval is refined by taking the maximum of all entry times (tmin) and the minimum of all exit times (tmax). If tmax < tmin or the interval does not overlap the ray's valid range [ray.t_min, ray.t_max], the ray misses the box; otherwise, the ray intersects the AABB and the values t_in = tmin and t_out = tmax are returned.

2.4 BVH Construction

The stop criteria is filled in as follows:

```
if (span_right - span_left <= 1 || depth >=
    CUTOFF_DEPTH) {
  // create leaf node
}
```

span_right - span_left <= 1 checks if there is one or zero triangle in the current span, in which case we create a leaf node. depth >= CUTOFF_DEPTH checks if the current depth exceeds a predefined cutoff depth, preventing excessive recursion and ensuring balanced tree structure.

The median split is implemented as follows:

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

cmp is a lambda function that compares two nodes based on the center of their AABBs along the specified dimension dim.

std::nth_element is then used to rearrange the nodes such that the node at the split index is the median along the chosen dimension, effectively partitioning the nodes into left and right subtrees for BVH construction.

2.5 Integrator and Refractive Materials

For each ray generated by the camera, it traces the ray's path through the scene.

- **Refractive Surfaces:** If the ray hits a refractive material, we do not terminate the path. Instead, we call the material's BSDF::sample method to generate a new ray direction. The path tracing continues with this new ray, allowing light to pass through transparent objects.
- **Diffuse Surfaces:** If the ray intersects a diffuse surface, the recursive tracing stops. This intersection point is then passed to the directLighting function to calculate the illumination from light sources.
- **No Intersection:** If a ray misses all objects, it contributes no light, and black is returned.

This process is repeated up to a predefined max_depth to prevent infinite recursion. The implementation in Li is shown below:

```
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 = 0.0f;
    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;
  }
```

We have called the PerfectRefraction::sample method in Li to handle refraction materials. Now we detail its implementation:

```
Vec3f PerfectRefraction::sample(
    SurfaceInteraction &interaction, Sampler
    &sampler, Float *pdf) const {
// The interface normal
Vec3f normal = interaction.shading.n;
// Cosine of the incident angle
Float cos_theta_i = Dot(normal, interaction.
    wo);
// Whether the ray is entering the medium
bool entering = cos_theta_i > 0;
// Corrected eta by direction
Float eta_corrected = entering ? eta : 1.0F /
     eta;
Vec3f wt;
bool success = Refract(interaction.wo, normal
    , eta_corrected, wt);

if (success) {
  interaction.wi = wt;
} else {
  interaction.wi = Reflect(interaction.wo,
      normal);
}

// Set the pdf and return value, we dont need
     to understand the value now
if (pdf != nullptr)
*pdf = 1.0F;
return Vec3f(1.0);
}
```

The Refract function is called to compute the refracted direction. If refraction is not possible, it computes the reflected direction instead.

2.6 Direct Lighting Function

After the loop in function Li, if a diffuse surface was found, we call directLighting to compute the lighting at that point.

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

The directLighting function calculates the contribution of a point light source to a given surface point. First, it tests occlusion:

```
SurfaceInteraction shadow_isect;
bool occluded = false;
if (scene->intersect(test_ray, shadow_isect))
     {
  Float hit_dist = Norm(shadow_isect.p -
      interaction.p);
  if (hit_dist + 1e-6f < dist_to_light)
  occluded = true;
}
if (occluded) {
  return Vec3f(0.0f);
}
```

The above code first checks if the ray intersects any object, then uses hit_dist and dist_to_light to determine if the light source is occluded by this object. If it is occluded, the function returns black. If not occluded, compute the contribution using perfect diffuse diffuse model.

```
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);
  Float attenuation = 1.0f / (4.0f * PI *
      dist_to_light * dist_to_light);
    Vec3f flux = point_light_flux;
    color = bsdf->evaluate(interaction) *
        cos_theta * flux * attenuation;
    color *= 2;
}

return color;
```

The above code calculates the cosine of the angle between the light direction and the surface normal, computes the attenuation based on the inverse square law, and evaluates the BSDF at the interaction point.

2.7 Anti Aliasing

In IntersectionTestIntegrator::render, we generate #spp rays for each pixel and use Monte Carlo integration to compute radiance.

```
const Vec2f &pixel_sample = sampler.
    getPixelSample();
auto ray = camera->generateDifferentialRay(
    pixel_sample.x, pixel_sample.y);
```

When we call sampler.getPixelSample(), it returns a 2D sample with a small offset within the pixel, thus achieving anti-aliasing. See Figure 1 for the result.

2.8 Area Light

We modify the .json file to support area lights. An example is shown below:

```
"light_type": "area",
"light_position": [
  0.0,
  1.99,
  0.5
],
"light_u": [
  1.0,
  0.0,
  0.0
],
"light_v": [
  0.0,
  0.0,
  1.0
],
"light_size": [
  0.5,
  0.5
],
"light_radiance": [
  25.5,
  18.0,
  7.5
],
"max_depth": 16,
"spp": 512
```

In integrator.h, we modify the IntersectionTestIntegrator class to support area lights:

```
IntersectionTestIntegrator(const Properties &
    props) : Integrator(props) {
  std::string ligth_type =
    props.getProperty<std::string>("
       light_type", "point");
  if (ligth_type == "area") {
    use_area_light = true;
    area_light_pos =
      props.getProperty<Vec3f>("
         light_position", Vec3f(0.0f, 1.9f,
         0.0f));
    area_light_u = Normalize(
      props.getProperty<Vec3f>("light_u",
         Vec3f(1.0f, 0.0f, 0.0f)));
    area_light_v = Normalize(
      props.getProperty<Vec3f>("light_v",
         Vec3f(0.0f, 0.0f, 1.0f)));
    area_light_size =
      props.getProperty<Vec2f>("light_size",
         Vec2f(0.5f, 0.5f));
    area_light_radiance =
      props.getProperty<Vec3f>("
         light_radiance", Vec3f(10.0f));
    area_light_normal = Normalize(Cross(
       area_light_u, area_light_v));
    area_light_area = area_light_size.x *
       area_light_size.y;
  } else {
    use_area_light = false;
    point_light_position = props.getProperty<
       Vec3f>("point_light_position",
                                Vec3f(0.0F, 5.0F,
                                    0.0F));
    point_light_flux =
      props.getProperty<Vec3f>("
         point_light_flux", Vec3f(1.0F, 1.0F
         , 1.0F));
  }

  max_depth = props.getProperty<int>("
     max_depth", 16);
  spp = props.getProperty<int>("spp", 8);
}
```

The following variables are added to the protected section of the class:

```
// Area light params
bool use_area_light;
Vec3f area_light_pos;
Vec3f area_light_u;
Vec3f area_light_v;
Vec2f area_light_size;
Vec3f area_light_radiance;
Vec3f area_light_normal;
Float area_light_area;
```

Then we modify the `directLighting` function to support area lights:

```
Vec3f color(0, 0, 0);
Float dist_to_light;
Vec3f light_dir;
Vec3f L_incoming;
if (use_area_light) {
  Vec2f uv = sampler.get2D();
  Vec3f sample_pos = area_light_pos +
            area_light_u * (uv.x - 0.5f) *
               area_light_size.x +
            area_light_v * (uv.y - 0.5f) *
               area_light_size.y;

  Vec3f diff = sample_pos - interaction.p;
  Float dist_sq = Dot(diff, diff);
  dist_to_light = std::sqrt(dist_sq);
  light_dir = diff / dist_to_light;
  Float cos_light = Dot(area_light_normal, -
     light_dir);
  if (cos_light <= 0.0f)
  return Vec3f(0.0f);
  L_incoming = area_light_radiance *
     area_light_area * (cos_light / dist_sq)
     ;
} else {
  dist_to_light = Norm(point_light_position -
       interaction.p);
  light_dir = Normalize(point_light_position
      - interaction.p);
  Float attenuation = 1.0f / (4.0f * PI *
      dist_to_light * dist_to_light);
  L_incoming = point_light_flux * attenuation
      ;
}
```

The code samples a point on the area light source and computes the incoming radiance at the interaction point, taking into account the area of the light source and the angle of incidence. See Figure 2 for the result.

2.9 Multiple Light Sources

We introduce a new struct LightData and create a vector of it to support multiple light resources.

```
struct LightData {
  bool is_area_light;
  Vec3f position;
  Vec3f emission;

  Vec3f u, v;
  Vec2f size;
  Vec3f normal;
  Float area;
};

std::vector<LightData> lights;
```

We modify the directLighting function, adding a for loop to add every light's contribution:

```
Vec3f color(0, 0, 0);
for (const auto &light : lights) {
  ...
  // same code as before
  color += bsdf->evaluate(interaction) *
      L_incoming * cos_theta;
}
return color;
```

See Figure 3 for the result.

2.10 Texture

I modify the .json file to replace the floor with checkerboard texture.

```
// In "textures":
"floor_checker": {
  "type": "checkerboard",
  "color0": [0.1, 0.1, 0.1],
  "color1": [0.9, 0.9, 0.9],
  "tex_coordinate_generator": {
    "type": "uvmapping2d",
    "scale": [6.0, 6.0],
    "delta": [0.0, 0.0]
    }
}

// In "materials":
"floor_mat": {
  "type": "diffuse",
  "texture_name": "floor_checker"
}

// In "objects":
{
  "type": "mesh",
  "path": "assets/cbox/floor.obj",
  "material_name": "floor_mat"
}
```

See Figure 4 for the result.

3 RESULTS

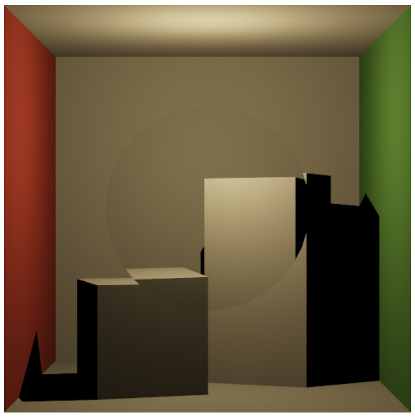
Fig. 1. Rendering result with refraction and anti-aliasing.

student number: 2023533029
email: `chenyf2023@shanghaitech.edu.cn`

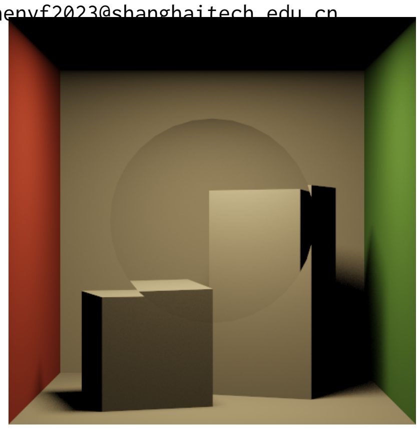

Fig. 2. Rendering result with area light source.

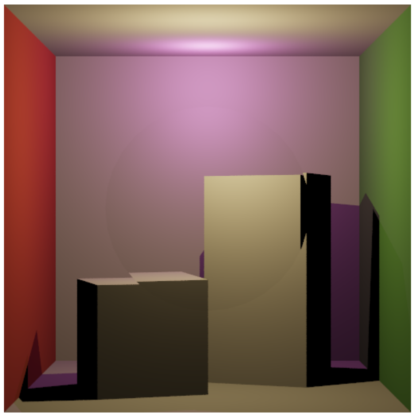

Fig. 3. Rendering result with multiple light sources.

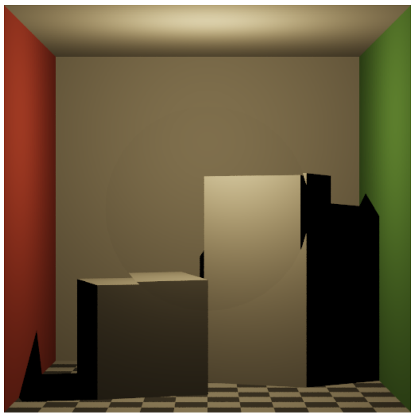

Fig. 4. Rendering result with texture.