Assignment 3:
Ray Tracing with Direct Lighting

NAME:
STUDENT NUMBER: 2019533240
EMAIL: ZHANGSY3@SHANGHAITECH.EDU.CN

1 INTRODUCTION

- a pin-hole camera model, which is able to shoot a set of optical rays. And there should be at least one ray per pixel.
- the algorithm for the ray-triangle intersection (without the acceleration structure).
- the algorithm for the ray-cube intersection based on the ray-triangle intersection (without the acceleration structure).
- anti-aliasing for ray-tracing by using super-sampling with the rotated grid.

2 IMPLEMENTATION DETAILS

2.1 Pin-hole Camera Model and Ray Generate

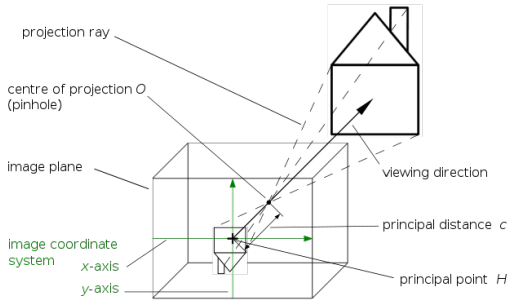

Fig. 1. Pinhole Camera Model

$$\begin{aligned} Forward &= Position - LookAt \\ Right &= RefUp \times Forward \\ Up &= Forward \times Right \end{aligned} \tag{1}$$

And for the ray generation, we just need to calculate the ray direction and the ray origin position, which is from a pixel on the camera panel to the position and go through the pinhole, and we can use a transformation

$$\begin{aligned} CameraPosition &= (px, py, -f) \\ RealWorldPosition &= trans \cdot CameraPosition + RealWorldCamPos \\ RealWorldRayDirection &= trans \cdot CameraPosition \end{aligned} \tag{2}$$

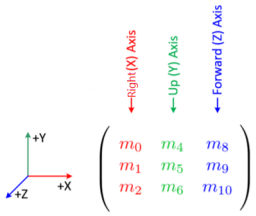

Fig. 2. transformation

2.2 Ray-Triangle Intersection

Then we follow the Möller-Trumbore algorithm to test whether the ray hits the triangle as follows:

```
bool RayIntersectsTriangle(Vector3D rayOrigin,
Vector3D rayVector,
Triangle* inTriangle,
Vector3D& outIntersectionPoint)
  {
  const float EPSILON = 0.0000001;
  Vector3D vertex0 = inTriangle->vertex0;
  Vector3D vertex1 = inTriangle->vertex1;
  Vector3D vertex2 = inTriangle->vertex2;
  Vector3D edge1, edge2, h, s, q;
  float a,f,u,v;
  edge1 = vertex1 - vertex0;
  edge2 = vertex2 - vertex0;
  h = rayVector.crossProduct(edge2);
  a = edge1.dotProduct(h);
  if (a > -EPSILON && a < EPSILON)
  return false;
    // This ray is parallel to this triangle.
  f = 1.0/a;
  s = rayOrigin - vertex0;
  u = f * s.dotProduct(h);
  if (u < 0.0 || u > 1.0)
  return false;
  q = s.crossProduct(edge1);
  v = f * rayVector.dotProduct(q);
  if (v < 0.0 || u + v > 1.0)
  return false;
  // At this stage we can compute t to find out
    // where the intersection point is on the line.
  float t = f * edge2.dotProduct(q);
```

student number: 2019533240
email: zhangsy3@shanghaitech.edu.cn

```
  if (t > EPSILON) // ray intersection
    {
    outIntersectionPoint = rayOrigin +
                                rayVector * t;
    return true;
  }
  else
    // This means that there is a line intersecti
    // but not a ray intersection.
  return false;
}
```

2.3 Area light

Seeing that we are rendering the scene with a panel of light, not a spot light, so we should sample the light source with a uniform distribution and distribute the lumen. So each sample is only one n^{th} of the original light source.

2.4 integrator

2.4.1 radiance. Seeing that we don't care the reflection for now, we just have two conditions: the ray hit the light, or the ray hit the object.
For case one, we just set the color to the light's color as we can see the light color directly from the camera.
For case two, we need to calculate the color of the object by using the Phong model. As same as the muti-light model in the project one, we can calculate the BRDF of each light source and sum all the BRDFs. Moreover we should test whether the ray can touch the light without be shadowed by other objects.

$$L_o(p, \omega_o) = \sum_{L_i} brdf(p) \cdot L_i \cos\theta_i \Delta A \tag{3}$$

and just use the phong model to calculate the diffusion and specular

2.4.2 rendering. Trace the ray from the camera to the world pixel by pixel.

- 1) generate ray.
- 2) insert into the scene.
- 3) get the radiance.
- 4) set pixel.

2.5 Anti-Aliasing

By using up sampling and with rotated grid pattern, we can reduce the aliasing effect of the image.

```
for (int i = 0; i < num_sample; i++) {
  for (int j = 0; j < num_sample; j++) {
    vec2f beam_origin(
    pixel_pos.x() + pixel_size.x() * (i + 0.5f) / num_sample,
    pixel_pos.y() + pixel_size.y() * (j + 0.5f) / num_sample);

    vec2f offset = beam_origin - center_pos;
    beam_origin = rotate_matrix * offset + center_pos;
    ...
```

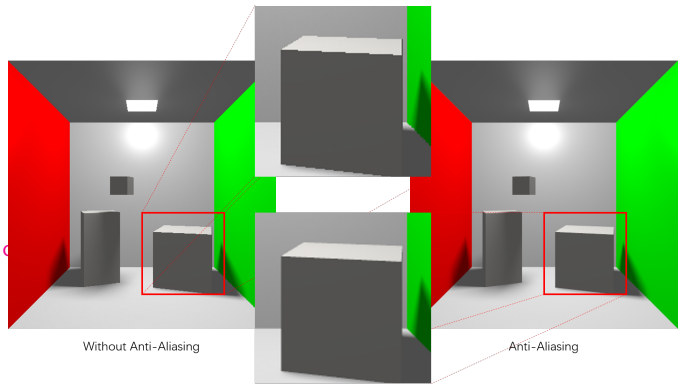

Fig. 3. comparison of the output of anti-alising and without anti-alising

```
    generateray();
    insert();
    getradiance();
    setpixel();
  }
}
```

3 RESULTS

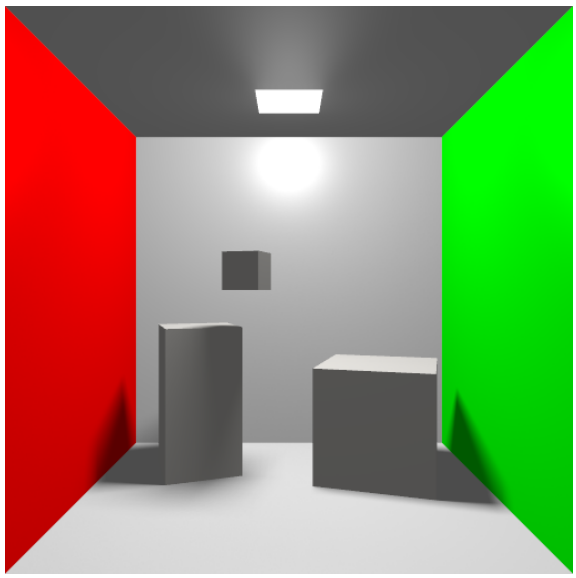

Fig. 4. case 1

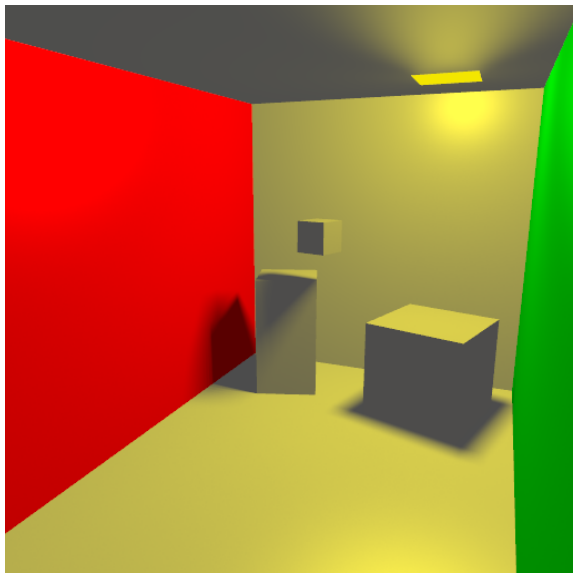

Fig. 5. case 2

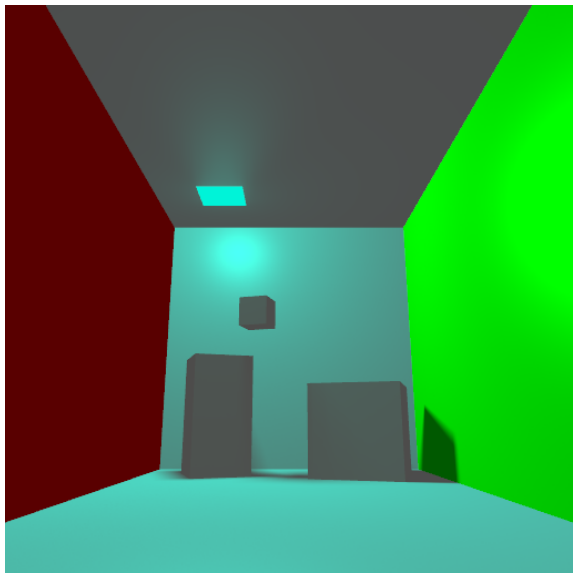

Fig. 6. case 3