Assignment 4: Global Illumination

NAME: ENTROPY-FIGHTER
STUDENT NUMBER: 2020533
EMAIL: XXXX@SHANGHAITECH.EDU.CN

1 INTRODUCTION

I do the must, bonus1, bonus2 and bonus4. In detail, I do the following things.

- (must)Path tracing with Monte Carlo integration, direct + indirect lighting (40 pts)
- (must)Ideal diffuse BRDF and area light source (20 pts)
- (must)Acceleration structure: BVH (30 pts)
- (optional)The ideal specular or glossy specular BRDF (10 pts)
- (optional)The translucent BRDF with refraction, e.g. glasses and diamonds. (10 pts)
- (optional)Advanced BVH with higher performance. (20 pts)

2 IMPLEMENTATION DETAILS

2.1 Path tracing with Monte Carlo integration

This part is implemented in "integrator.cpp". In this section, I complete the Integrator::render and Integrator::radiance methods. In general, to compute the radiance of each pixel in the image plane, I construct paths starting from the camera and compute the Monte-Carlo integration along these paths in order to solve the light transport equation. The iterative implementation of path tracing can be briefly summarized below.

```
Beta = 1, L = 0
Generate a ray from the camera to a pixel on the image plane
For i = 1 to max depth
(0) Suppose the marching ray hits at P(i)
(1) L += Beta * Direct lighting at P(i)
(2) Sample ONE next ray according to P(i)'s BRDF and find the corresponding Pdf
(3) Beta *= BRDF * cosΘ / Pdf
(4) Spawn and march the new ray
where L is the output radiance.
```

The most important step of the above pseudo code is to calculate the direct lighting and indirect lighting.

To calculate direct lighting, we need to sample the light source to generate a new ray. If the new ray is shadowed, we return. Otherwise, it it is not shadowed or blocked, we use the emission as L_i.

To calculate indirect lighting, we also need to sample to construct a new ray. After that, we need to test whether it hits the light. If it hits the light, it means it is a direct lighting ray. if it does not hit the light, it means it is not a direct lighting ray. Then we have to recursively calucate the radiance given the new ray.

2.2 Ideal diffuse BRDF and area light source

This part is implemented in the "bsdf.cpp" and "light.cpp".

As for ideal diffuse BRDF, we use cos-weighted sampling.

For function "evaluate", the return value should be $\frac{color}{\pi}$. For function "pdf", the return value should be $\frac{cos\theta}{\pi}$. For function "sample", we do the following things.

We firstly sample uniformly by the sampler.

```
Vec2f eta = sampler.get2D();
```

Then we do the transformation from (x,y) to (r, θ_0). The relation between them is

$$r = \sqrt{x}, \theta_0 = 2\pi y$$

After that, we do the projection onto the unit hemisphere, which means transforming from (r, θ_0) to (θ, ϕ)

$$\sin\theta = \sqrt{x_1}$$

$$\phi = \theta_0 = 2\pi y$$

```
float theta = asin(sqrt(eta[0]));
float phi = 2 * PI * eta[1];
```

Since we have sphere coordinates, we can compute x, y, z.

$$x = \sin\theta\cos\phi$$

$$y = \sin\theta\sin\phi$$

$$z = \cos\theta$$

```
Vec3f direction(sin(theta) * cos(phi), sin(theta)
    * sin(phi), cos(theta));
```

Finally, we need to transform from local space to world space.

```
Mat3f trans = Eigen::Quaternionf::FromTwoVectors(
    Vec3f(0.0f, 0.0f, 1.0f), interaction.normal).
    toRotationMatrix();
Vec3f res = trans * direction;
interaction.wo = res.normalized();
return pdf(interaction);
```

After sampling, we can get the new ray.

As for area light source sampling, we just uniformly sample a point on the rectangle area. The pdf should be $\frac{1}{A}$. The code for "sample" function ia shown below.

```
Vec2f s = sampler.get2D();
Vec3f pos = position + Vec3f((s[0] - 0.5f) * size
    [0], 0.0f, size[1] * (s[1] - 0.5f));
*pdf = 1 / float(size[0] * size[1]);
interaction.wo = (pos - interaction.pos).
    normalized();
return pos;
```

student number: 2020533
email: xxxx@shanghaitech.edu.cn

In "pdf" function, we do the things below.

```
float cos= std::max(0.0f, -interaction.wo.dot(
    Vec3f(0, -1, 0)));
float distance = (pos - interaction.pos).norm();
return cos / distance / distance;
```

Solid angle's definition is $dw = \frac{dA \cos\theta'}{distance^2}$, where $\cos\theta' = w_i \cdot n_l$, *distance* is the distance between position of samples on light an the intersection position. Now, our basic rendering function is

$$L_o(x, w_o) = \int_\Omega L_i(p, w_i) f(p, w_i, w_o) \frac{\cos\theta \cos\theta'}{distance^2} dA$$

2.3 Acceleration structure: BVH

This part is implemented most in the "geometry.cpp". BVH is a new data structure whose name is bounding volume hierarchy. Pbrt tells us that BVHs are an approach for ray intersection acceleration based on primitive subdivision, where the primitives are partitioned into a hierarchy of disjoint sets. All objects are included in tree nodes of th bounding volume. The scene or each geometry/object can all be bounded by the recursive bounding volume.

- Organize objects into a tree
- Group objects in the tree based on spatial relationships
- Each node in the tree contains a bounding volume of all the objects below it

Here, we just do some simple introduction to how to construct, traverse a BVH tree. We would do some advanced implementations in the bonus part, and more details will be given in section 2.6.

```
class BVH {
 public:
  void Build(const std::vector<Vertex>& new_vertices,
             const std::vector<Vector3u>& new_triangles) {
    std::vector<Vector3u> triangles_copy = new_triangles;
    Build(new_vertices, triangles_copy, nodes);
  }

 private:
  void Build(const std::vector<Vertex>& vertices,
             std::vector<Vector3u>& triangles,
             BVHNode*& node) {
    // set this
    node = newNode(vertices, triangles);

    // split
    unsigned axis = SelectAnAxisToDivide();
    std::vector<Vertex3u> triangles_left, triangles_right;
    DivideToLeftAndRightPart(vertices, triangles,
                             triangles_left, triangles_right);

    // build left and right
    Build(vertices, triangles_left, node->left);
    Build(vertices, triangles_right, node->left);
  }
};
```

(a) construct

```
bool RayCast(const BVHNode* node, const Ray& ray,
             RayCastResult& result) {
  // empty node
  if (!node) return false;

  // not hit AABB
  if (!RayCast(node->bound, ray))
    return false;

  // recursive test
  bool hit = false;
  if (node->IsLeaf()) { // leaf node
    for (const Vector3u& triangle : node->triangles) {
      if (RayCast(triangle, ray, result)) {
        ... // record something into `result`
      }
    }
  }
  else { // interior node
    if (RayCast(node->left, ray, result)) hit = true;
    if (RayCast(node->right, ray, result)) hit = true;
  }

  return hit;
}
```

(b) raycast

2.4 (optional)The ideal specular BRDF

This part is implemented in the "bsdf.cpp". Implementation of ideal specular BRDF is easy. we just sample the reflect light, which has a pdf of 1. As for the "sample" function, the implementation is shown below.

```
float cos_theta = (-interaction.wi).dot(
    interaction.normal);
interaction.wo = (2 * cos_theta * interaction.
    normal + interaction.wi).normalized();
return pdf(interaction);
```

2.5 (optional)The translucent BRDF with refraction, e.g. glasses and diamonds

Reference: I learn this part from https://www.pbr-book.org/3ed-2018/Reflection_Models/Specular_Reflection_and_Transmission.

This part is implemented in the "bsdf.cpp". Implementation of the translucent BRDF with refraction is a bit harder than the ideal specular BRDF. What we need to use here is snell law.

$$\eta_1 \cdot \sin\theta_1 = \eta_2 \cdot \sin\theta_2$$

Snell law gives us the direction and also tells us that how the radiance changes when the ray goes between different media with different indices of refraction.

In this part, we need to write a refract() function, which computes the refracted direction w. The core code(pbrt vesion) is given below.

```
inline bool Refract(const Vector3f &wi, const
    Normal3f &n, Float eta, Vector3f *wt) {
   Float cosThetaI = Dot(n, wi);
   Float sin2ThetaI = std::max(0.f, 1.f -
       cosThetaI * cosThetaI);
   Float sin2ThetaT = eta * eta * sin2ThetaI;
   if (sin2ThetaT >= 1) return false;

   Float cosThetaT = std::sqrt(1 - sin2ThetaT);

   *wt = eta * -wi + (eta * cosThetaI - cosThetaT)
        * Vector3f(n);
   return true;
}
```

As for the "sample" function, it is similar to the specular part, but we need to consider the refract. The core part of the implementation is shown below.

```
bool refracted = Refract(-interaction.wi,
    interaction.normal, etaI / etaT, &interaction.
    wo);
if(!refracted){
  interaction.wo = (2 * cos_theta * interaction.
      normal + interaction.wi).normalized();
}
return pdf(interaction);
```

2.6 (optional)Advanced BVH with higher performance(using SAH)

Reference: I learn this part from https://medium.com/@bromanz/how-to-create-awesome-accelerators-the-surface-area-heuristic-e14b5dec6160 and https://blog.csdn.net/weixin_44176696/article/details/118655688

This part is implemented in the "geometry.cpp". In this part, SAH is used to achieve better performance. MacDonald and Booth proposed the SAH. It predicts the cost of a defined split position on a per-node basis. The model states that:

$$C(A, B) = t_{traversal} + p_A \sum_{i=1}^{N_A} t_{intersect}(a_i) + p_B \sum_{i=1}^{N_B} t_{intersect}(b_i)$$

- C(A, B) is the cost for splitting a node into volumes A and B
- t_traversal is the time to traverse an interior node
- p_A and p_B are the probabilities that the ray passes through the volumes A and B
- N_A and N_B are the number of triangles in volumes A and B
- a_i and b_i are the ith triangle in volumes A and B
- t_intersect is the cost for one ray-triangle intersection.

We can compute p_A and p_B as:

$$p(C|P) = \frac{S_C}{S_P}$$

- S_C and S_P are the surface areas of volumes C and P (We can simply compute the surface area of a node by summing all faces of a node)
- p(C|P) is the conditional probability that a random ray passing through P will also pass through C, given that C is a convex volume in another convex volume P.

With this formula, the SAH rewards many triangles in tight boxes which are not likely to get pierced by a ray. We can compute the SAH for various split positions. Then, we select the one with the lowest cost. We can receive possible split positions by taking triangle bounds into consideration. Another method is called binning. It defines a certain amount of linearly distributed positions over an axis.

In our case, for n triangle meshes, we firstly traverse all the possible cases to find the split method with least. And we should try every axis, namely the x axis, y axis and z axis. After each traversal, we update and continue to do such things recursively. There exists a small problem. We need to calculate the AABB bounding box about n times. If we traverse the interval to find the maximum value to build the bounding box every time, the efficiency of tree building will be very low. We have a little trick, using the idea of prefix to pre-calculate the maximum and minimum xyz values of the interval [l, i], [i+1, r], and then spend O(1) time to query each time.

3 RESULTS

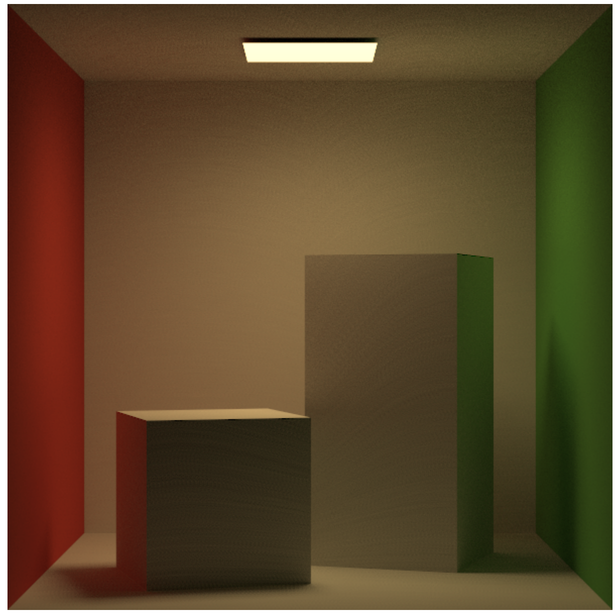

Fig. 1. simple

Fig. 2. large mesh

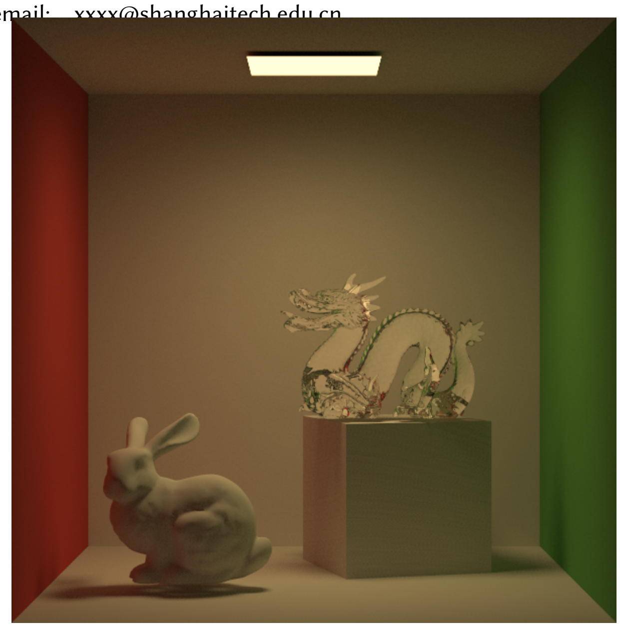

Fig. 3. translucent

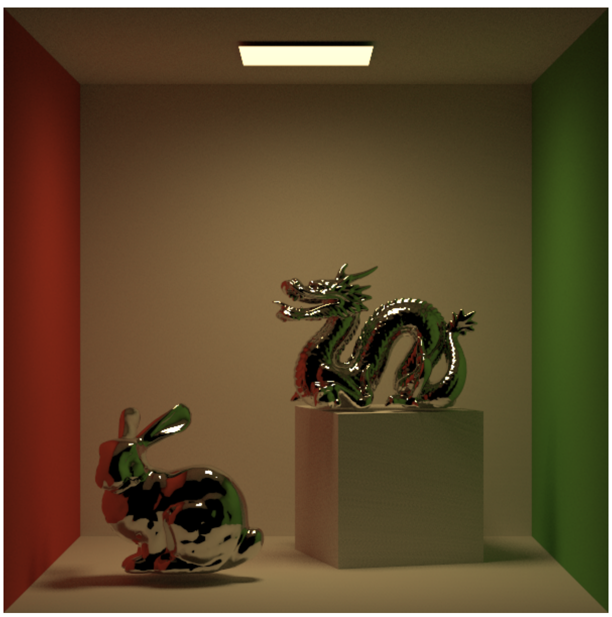

Fig. 4. specular