Assignment 3:
EBasic Ray Tracing

NAME: LU YUANJI
STUDENT NUMBER: 2023533014
EMAIL: LUYJ2023@SHANGHAITECH.EDU.CN

1 INTRODUCTION

In this project, we focus on the implementation of a basic ray-tracing renderer, including ray-triangle intersection, BVH tree construction and traversal, direct illumination. Also, support of rectangle area light with soft shadow and environment light via environment mapping are available.

2 IMPLEMENTATION DETAILS

2.1 Code Compilation

The developing environment of this project is vscode with its plugins. The code is compiled by cmake.

2.2 Ray-triangle Intersection

Ray-triangle intersection is implemented in `TriangleIntersect` in `accel.cpp`. To get `t, u, v`, we leverage the solution of the following linear equation:

$$(1-u-v)p_0 + up_1 + vp_2 = \mathbf{o} + t\mathbf{d}$$

where $(1-u-v, u, v)$ is the barycentric coodinate, $\mathbf{o} + t\mathbf{d}$ is the ray. The ray intersects with the triangle if and only if:

$$u \geq 0, v \geq 0, u + v \leq 1, t_{\min} \leq t \leq t_{\max}$$

2.3 Ray-AABB Intersection

Ray-AABB Intersection is implemented in `AABB::intersect` in `accel.cpp`. The ray intersects with the AABB if and only if $t_{\text{in}} \leq t_{\text{out}}$, and t_{in} is derived by the max component value of $t_{\min}$, t_{out} is derived by the min component value of $t_{\max}$. And $t_{\min}$, $t_{\min}$ are derived by $\frac{b_{\text{low}} - \mathbf{o}}{\mathbf{d}}$ and $\frac{b_{\text{upper}} - \mathbf{o}}{\mathbf{d}}$ respectively.

2.4 BVH Construction

The main implementation of construction of BVH is in `bvh_tree.h`. The alogrithm is a divide-and-conquer one, and we do two major modifications.

2.4.1 Stop Criteria. If `span_left` crossed over `span_right`, or the tree exceeds the max depth, we stop dividing the scene; else, we continue splitting.

2.4.2 Median Split. When splitting, we find the median node according to the center of AABB of each node, and make those nodes that are less than median stay in the left side, and those larger than median stay in the right side of the nodes list. Then we recursively do the construction and split at the median.

2.5 Direct Illumination Integrator

This section is implemented mainly in `integrator.cpp`. In this integrator, we do not take multi-reflection into account and only calculate the relationship between the intersection and light source directly. The next part will introduce the three different kinds of integrator implemented in this project, which serve for different purposes.

2.5.1 Point Light Integrator. To render image, we traverse the all pixels, and each pixel is sampled `spp` number of rays to determine the final color of this specific pixel. Multiple rays are sampled to reduce aliasing problem. For each ray specifically, we calculate the `Li`, which is the radiance the pixel, and then commit it to the main thread since we do this in a parallel pattern.

To calculate `Li`, we traverse the tree to see if the sampled ray intersects with any triangle. If does, then we first judge the material(BSDF) of the surface: if ideal diffuse, we calculate its color by `directLighting` straightforwardly, and if perfect refraction, since its trasparent, we cannot determine its color here and should sample another ray by its BSDF to continue this process; if doesn't, then we just break this traverse since it must not intersect with any other triangle.

To calculate `directLighting`, we first perform occlusion test, which sample a test ray from the intersection point to the point light source to see if there exists occlusion along this ray: if does, then return a black color; if not, then we get its color by render equation:

$$L_i = \int_A f_r \cdot L_o |\cos\theta_i| \mathrm{d}\omega_o$$

where f_r is evaulated by `bsdf->evaluate(interaction)`, L_o is evaluated by `point_light_flux / (4 * PI * dist_to_light * dist_to_light)`, $\cos\theta_i$ is evaluated by `std::max(Dot(light_dir, interaction.normal), 0.0f)`

2.5.2 Area Light Integrator. For render logic, there is no difference on implementation.

For `Li` part, a judgement of `hasAreaLight` is performed before evaluation of color. This is necessary because it handles the occasion of the rendering of light itself.

For `directLighting` part, we sample `light_sample_cnt` number of points in area light as point light, and iteratively do the same thing as the point light color calculation.

2.5.3 Environment Light Integrator. For render logic, there is no difference on implementation.

student number: 2023533014
email: luyj2023@shanghaitech.edu.cn

For `Li` part, there's a litte difference when the sampled ray does not intersect with any triangles–we return the color of envrionment light color, which is intuitive.

For `directLighting` part, there's a little difference in environment light radiance calculation, since it has nothing to do with interaction itself, unlike previous integrators.

3 RESULTS

Thanks to different integrators, we get four images that renders under different settings:

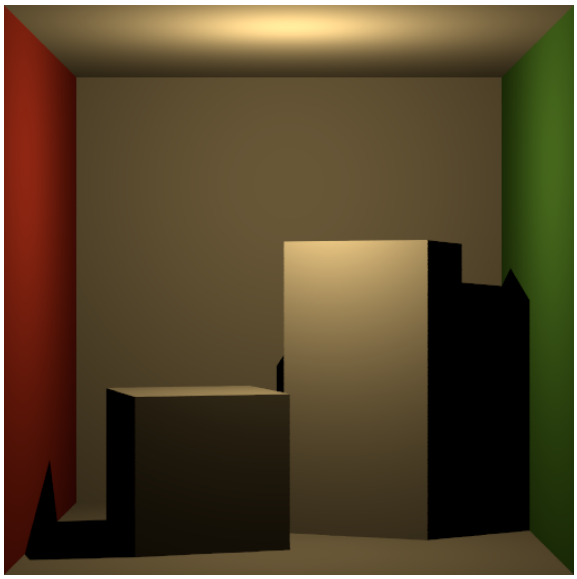

Fig. 1. Boxes with no refractive material rendering under a single point light

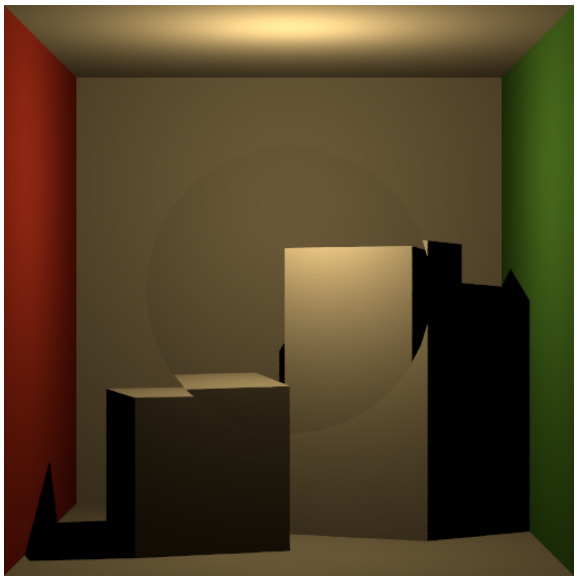

Fig. 2. Boxes with refractive material rendering under a single point light

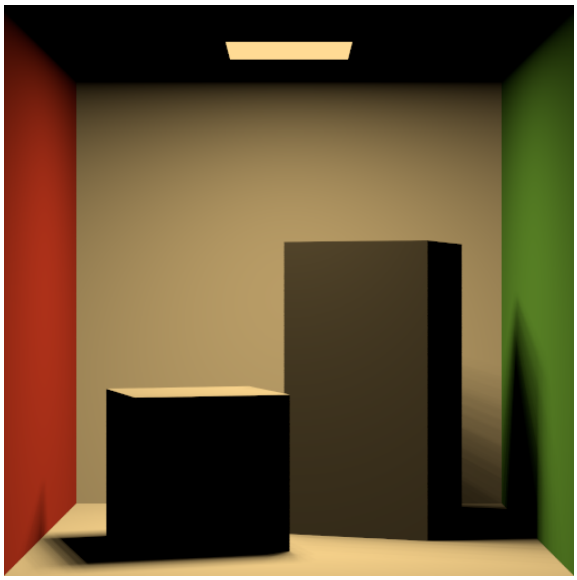

Fig. 3. Boxes rendering under a rectangle area light

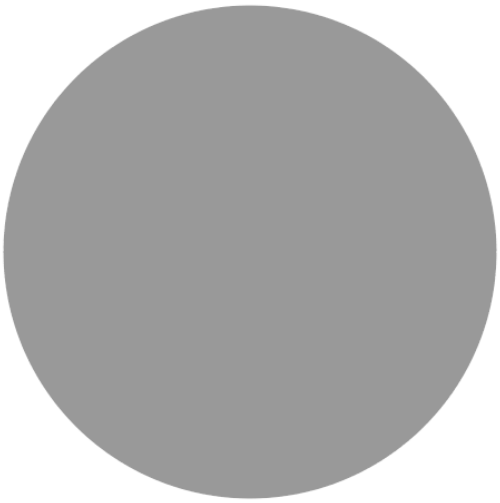

Fig. 4. Sphere rendering under a white environment light