Ray Tracing Rendering in Fragment Shader

WANG Zeyu

November 26, 2022

1 Introduction

Different from rasterization rendering pipeline, ray tracing is better at simulating the behaviors of light and therefore is able to produce more realistic rendering result. Ray tracing rendering is typically computation demanding thus is hard to be applied to real-time rendering. However, with the rapid development of GPU, real-time ray tracing is becoming possible.

In this project, I implement the ray tracing rendering pipeline with simple geometries in the fragment shader to explore the characteristics of ray tracing. A preview of achieved result is provided in 1. The code is available at `https://github.com/zwangek/COMP5411RenderingProject`.

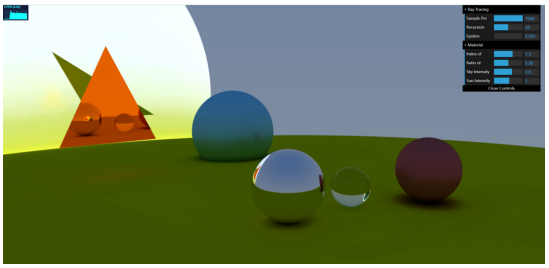

Figure 1: Preview

2 Methodology

In this section, I will describe technical details of my implementation of ray tracing rendering pipeline. I accomplish the project with WebGL and Three.js library.

2.1 Setup

To apply the fragment shader implemented with ray tracing, I create a square mesh with Three.js as my canvas. The fragment shader is attached to this square and the camera is always pointing toward this square in order to render the ray tracing scene to the screen.

2.2 Geometry Object

To reduce computation, I only implement the simplest geometries including sphere and triangle. The sphere has a vector as the coordinate of the center and a float number as the radius. The triangle has three vectors as the coordinates of three vertices and a boolean value to indicate whether the triangle is double sided. In addition to geometry, each object also records material information comprising emission, color, and type (i.e., diffusive, reflective, or refractive), which is used for computing ray scattering and shading. The scene is then a collection of arrays of geometries. These are defined as structures in the shader.

2.3 Ray Generation

Ray is defined as a structure with a vector of its origin and a vector of its direction. Since each pixel in the final image is rendered by back-tracing rays shooting from the pixel, the initial ray is generated with origin as the camera position and direction pointing towards the pixel that the shader is rendering. Moreover, to achieve anti-aliasing, multiple rays will be generated for a single pixel and they are perturbed with a random offset.

2.4 Ray-Object Intersection

I use brute force to traverse the scene with for loops to check whether a ray intersect with any object in the scene. It is essentially solving math equations to determine whether a ray hits an object. The loop will record the closest hit to the origin of the

ray and a HitRecord structure will store necessary information of the intersection, which contains, the coordinate of the hit point, the distance between the hit point and the origin of the ray, the normal of the hit point, and the material of the object.

2.5 Ray Scattering

On hitting an object, a ray will be scattered towards another direction according to the material. In my project, I implement diffusive, reflective and refractive materials.

2.5.1 Diffusive Material

I adopt Lambertian model to simulate the behavior of light on real-world diffusive material. To be specific, the new direction is calculated from the normal vector plus a random vector in a unit sphere.

2.5.2 Reflective Material

It is straightforward to implement the reflective material. The scattered light will be within the same plane with incident light and normal, with reflected angle equal to incident angle.

2.5.3 Refractive Material

The refractive material is harder to implement in comparison with previous two materials. The overall implementation has three components. First, the refracted ray must follow Snell's law which is defined as $(n_1 \sin\theta_1 = n_2 \sin\theta_2)$, where n is the index of refraction and θ is the incident or refracted angle. The second component is total internal reflection, which happens when the refracted angle is larger than the critical angle. I simply use 90 degree as the critical angle in my implementation. The third component is Schlick Approximation, which simulates the effect of varying reflectivity with viewing angle.

2.6 Shading

To get the final color of a ray, the energy of a ray is tracked during tracing. The energy starts with an RGB vector 0 and it will gain the energy when the ray hit an emitting object or the sky box. Another attenuation RGB vector starts with 1 and it will multiply with the RGB color of the material to attenuate the energy of the ray. The final color for the pixel is then multiplication result of the color and the attenuation.

3 Technical Challenge

The primary challenge for me is that I cannot obtain the geometry data of Three.js library (in addition to vertices, including information about faces). To solve this, I alternatively build the scene inside the shader. On the other hand, the pseudo random number generator I implement still have certain discernible distribution. However, this can be alleviated by clipping rays with small distance between the point of intersection and origin.

4 Results

The results of three different kinds of materials are presented in this section. It is observable in Figure 2 that the bottom part of the blue ball is greenish and the shadow of the red ball is soft and reddish. This is because of the indirect lighting and soft shadow effect of ray tracing. Different from rasterization rendering pipeline, they can be achieved without extra efforts, producing realistic rendering result. The reflection is also perfectly simulated as in Figure 3. When two reflective objects are placed in front of each other, they are able to reflect each other in an infinite loop, which is typically hard for rasterization rendering. In Figure 4, the effect of refraction is shown. The sphere correctly images the world upside-down as in real world. When the index of refraction is adjusted from low to high, a varying distribution between reflection and refraction can be observed. It is also possible to illuminate the scene without ambient light using light sources only as in Figure 5.

5 Discussion

With support for triangles, the pipeline can be extended to render triangular meshed. However, the performance issue will become a large barrier. With only several simple geometries, the frame rate drops to around 50 fps due to the brute force algorithm. There is a large room for optimization.

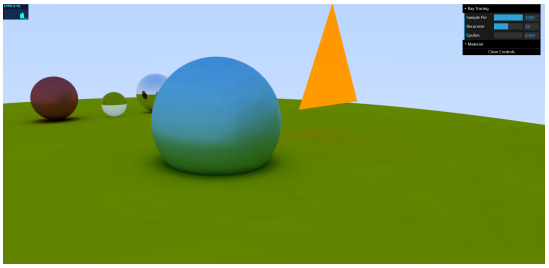

Figure 2: Diffusion

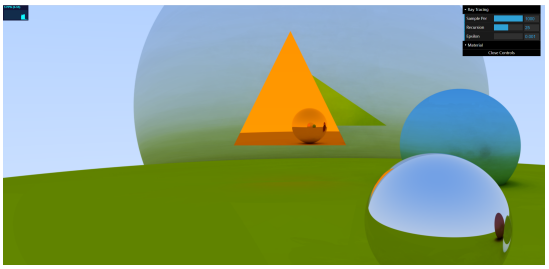

Figure 3: Reflection

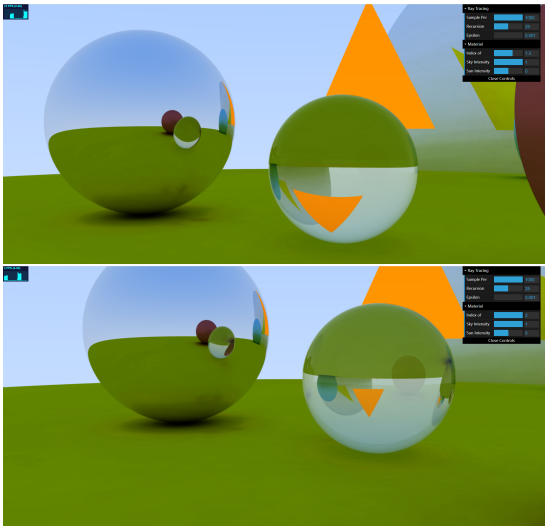

Figure 4: Refraction. Top: Low index of refraction. Bottom: High index of refraction

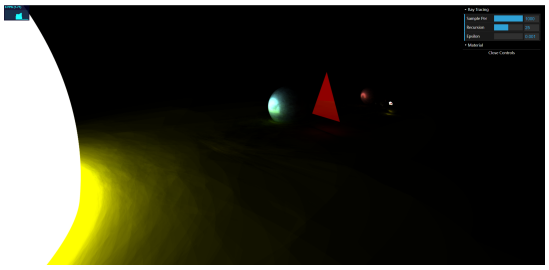

Figure 5: Illumination with light sources.

For example, BVH algorithm may significantly reduce computation complexity. Also, reducing sample per pixel is another intuitive solution. Nevertheless, this will lead to noisy result, therefore, a denoising algorithm is needed.

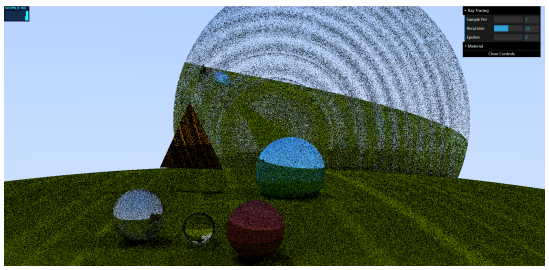

Figure 6: Low sample per pixel. Rings are caused by pseudo randomness.

6 Conclusion

In this project, I successfully implement the ray tracing rendering pipeline with fragment shader. And I explore many effects of ray tracing rendering by implementing three types of material and rendering the scene with different parameters. The implementation is naive and has a large space for future improvements. I would like to continue working on this project to study more about real-time ray tracing rendering in the future.