CS350 Project Part 2: Ray Tracing Engine

Can Çağatay Sevgican, Onur Özdemir

June 2, 2024

Abstract

In this project, we built a ray tracing engine through plain C++ and only using C++ Standard Libraries. The main objective was to learn more about computer graphics, vectors, complex matrix computations, and the generation of 2D images from 3D scenes. We achieved the result by a ray-casting-based rendering system.

1 Introduction

Ray Tracing was an intriguing topic for us since the output seemed satisfying, while we had access to many sources online, covering the topic. Our ray tracer is essentially a path tracer, which simulates rays sourced from our camera. To simulate rays, different materials, and anti-aliasing we performed complex matrix calculations.

2 Project Objectives

Our ray tracing engine was designed to incorporate the following advanced features:

- **Anti-Aliasing**: Decreases the jagged edges on an image while bettering an image's general smoothness between different surfaces.
- **Depth of Field (DoF)**: It's used for mimicking optically induced phenomena that result into the defocusing of objects.
- **Material Simulation**: Simulated a variety of materials including glass, metal and plastic based on their reflectiveness, and refraction.
- **Camera Rendering**: We simulated a camera system, which can be moved around. Field of View is also adjustable.

3 Technical Approach

Our approach was depending heavily into computational techniques. We selected C++ in order to perform complex multiplications in less time. C++ also has many mathematical standard library functions which were helpful. Our leading Integrated Development Environment was Visual Studio. Our outputs are in PPM (Portable Pixmap Format) and BMP (Bitmap) formats. We selected PPM for its simplicity, and converted it into BMP since it can be opened through native apps.

3.1 Mathematical Foundations

We utilized many mathematical and geometric equations. We knew most of them already, such as discriminant in a quadratic formula, or matrix multiplication. However, we also came across some interesting equations throughout the project. Such as Schlick's approximation for reflectance.

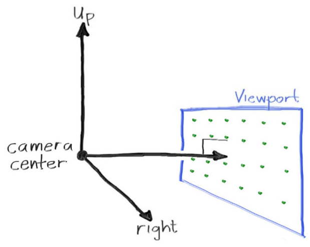
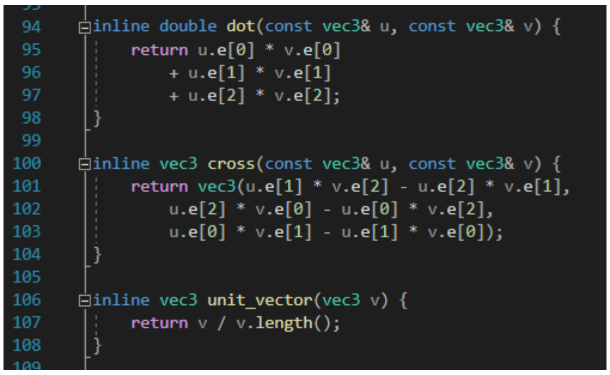


Figure 1: Vectors for camera



```
inline double dot(const vec3& u, const vec3& v) {
    return u.e[0] * v.e[0]
        + u.e[1] * v.e[1]
        + u.e[2] * v.e[2];
}

inline vec3 cross(const vec3& u, const vec3& v) {
    return vec3(u.e[1] * v.e[2] - u.e[2] * v.e[1],
        u.e[2] * v.e[0] - u.e[0] * v.e[2],
        u.e[0] * v.e[1] - u.e[1] * v.e[0]);
}

inline vec3 unit_vector(vec3 v) {
    return v / v.length();
}
```

Figure 2: Calculations for vectors

3.2 Algorithm Design

The core consists of the ray tracing algorithm, which includes casting ray from the camera, at each pixel's center, in the size of the camera. The algorithm starts scanning row by row. Then we check whether if that ray hits an hittable, or not. Here we use discriminant to understand if that solution has 0 roots, 1 root, or 2 roots. Which forms the basis of our algorithm.

4 Implementation Details

4.1 Ray Tracing Algorithm

The primary algorithm operates by casting rays into the scene and computing their interactions with various surfaces. Each ray may undergo multiple interactions, such as reflection and refraction, before contributing to the final color of a pixel.

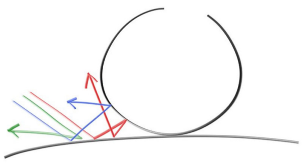


Figure 3: Bouncing light rays

4.2 Anti-Aliasing

We have used Super Sampling as a way of dealing with aliasing. The issue with point sampling is that it samples one point for each pixel hence resulting in incorrect reproductions of edges. Such as not having enough samples; this could cause an image composed faraway checkerboards for separate black squares and white squares rather than an blended grey tint. Several rays are sampled per pixel in super sampling projects they are averaged therefore it produces smoother images with more details.

Super-Sampling involves sampling multiple rays per pixel and averaging their results, leading to smoother edges and more detailed images. We used 3 techniques for this:

- **Random Sampling**: Generating random points within each pixel's area using a random number generator.
- **Averaging Colors**: Modifying the `write_color` function to average the colors of all samples.
- **Clamping Colors**: Ensuring color values remained within valid ranges using a clamping function.

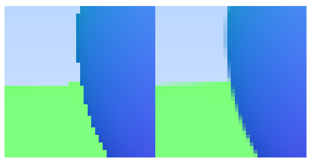


Figure 4: Anti-Alliasing before vs after

These approaches significantly reduced aliasing and improved image quality by creating smoother edges.

4.3 Depth of Field

Depth of Field was simulated by adjusting the focus distance and aperture size. Instead of simulating the whole camera lens system, we used rays through an infinitely sharp circular lens. These rays were directed towards a focus plane such that all objects present in that plane were perfectly focused. This focus plane is located at camera center at a distance that is perpendicular to the camera's viewing direction.

4.4 Material Simulation

To simulate realistic diffuse surfaces, also called matte surfaces, we used material diffusion, which spread light in different directions. The diffuse surfaces in our scene do not create light but they adopt colour that is related to the region around them, and this is influenced by their original colour. From Simple Diffuse Reflection we were able to advance into Lambertian reflection by looking at various ways of programming diffuse materials according to a given tutorial instructions. Some techniques we followed in diffusion coding included use of:

1. **Simple Diffuse Material:**

- The surface reflects light rays in completely different routes.
- Random vectors are equally distributed with respect to the unit sphere through a rejection technique in this process.
- In order to maintain that they lie on the unit sphere, the vectors are normalized subsequently.
- This random direction ensures the surface appears matte.

2. **Lambertian Reflection:**

- Reflection proportionally diffuses rays based on the angle between the reflected ray and the surface normal.
- This results in more realistic shading and is a closer approximation of real-world diffuse reflection.
- It makes sure that rays are often more scattered in directions near the surface normal, creating softer and more accurate lighting effects.

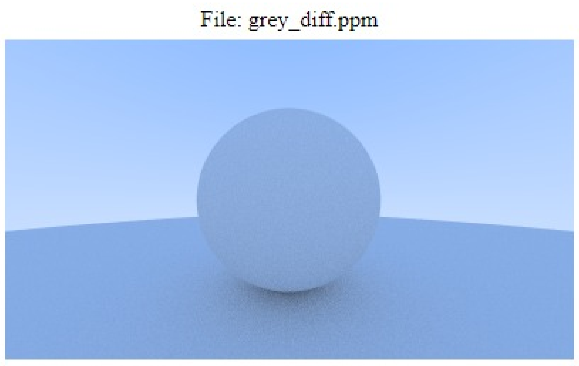
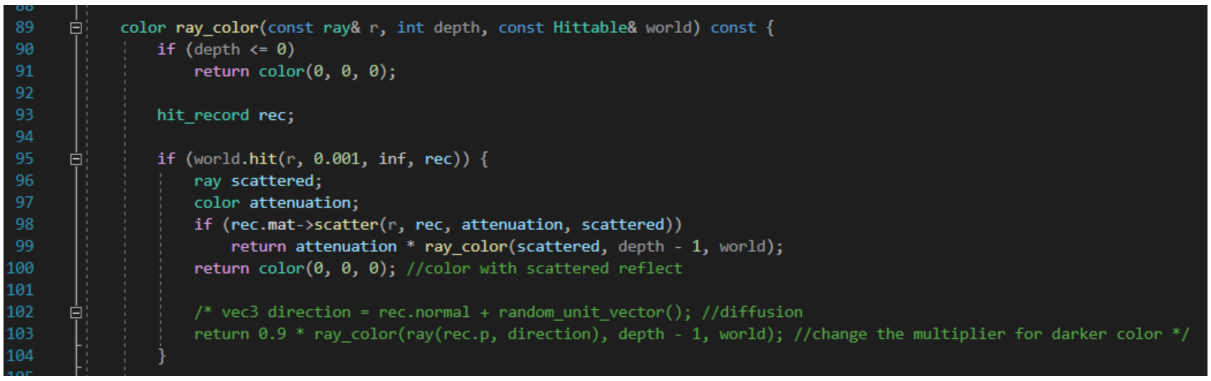


Figure 5: Diffusion

For the ray color function diffuse reflection was implemented. It recursively computes the color by scattering rays in random directions until the maximum recursion depth is reached or no more surfaces are hit.

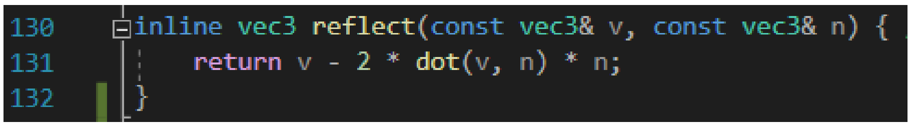


```
color ray_color(const ray& r, int depth, const Hittable& world) const {
    if (depth <= 0)
        return color(0, 0, 0);

    hit_record rec;

    if (world.hit(r, 0.001, inf, rec)) {
        ray scattered;
        color attenuation;
        if (rec.mat->scatter(r, rec, attenuation, scattered))
            return attenuation * ray_color(scattered, depth - 1, world);
        return color(0, 0, 0); //color with scattered reflect

        /* vec3 direction = rec.normal + random_unit_vector(); //diffusion
        return 0.9 * ray_color(ray(rec.p, direction), depth - 1, world); //change the multiplier for darker color */
    }
```

Figure 6: ray$_{c}olor function$

A vast variety of materials included in shaders were subjected to simulation. Lighting through glass used refraction indices and partial reflections, while metals had reflective properties and rough surfaces. The abstract class for materials, specific implementations for Lambertian (diffuse) and metal materials as well as the process for handling light scattering and reflection

1. **Metal Material:**
In the case of polished metal materials, the rays do not scatter but are directed back. They either scatter and lose their intensity versus oscillate if we consider their reflectance as R (Lamber-tian diffuse reflection) or sometimes maintain their intensity (with probability 1-R) while moving in the new direction without losses (a ray that has moved straight into the material). Sometimes it might act as a mix between the two strategies. We choosed always scattering since implementing Lambertian materials is already a very heavy task to do. For the reflection equations we used the below function:



```
inline vec3 reflect(const vec3& v, const vec3& n) {
    return v - 2 * dot(v, n) * n;
}
```

Figure 7: Reflection function

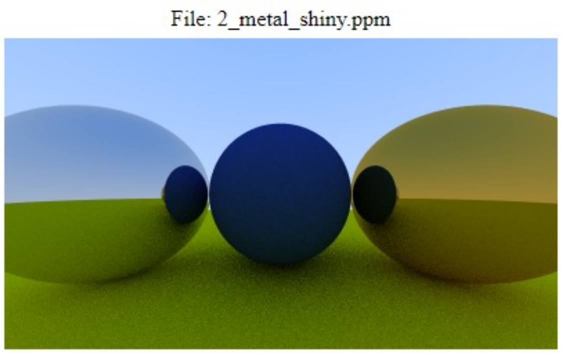


Figure 8: Shiny metal material

Also we can play around with our materials to simulate different types of environments or materials. For example below we can change our metal materials **fuzziness**:

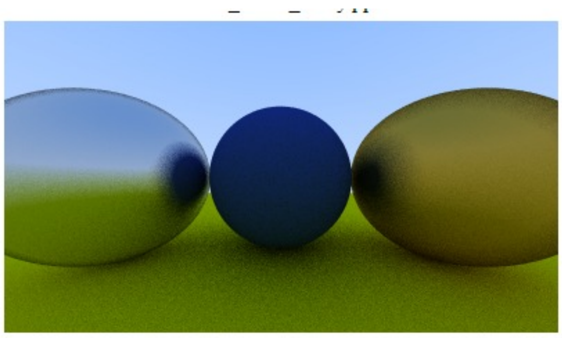


Figure 9: Fuzzy metal material

2. **Dielectrics - Glass and Air Material:** Dielectrics, which are clear, include water, glass and diamond. Whenever light traveling encounters either of the above materials, it gets divided into a transmitted and reflected light. We should treat models this way by compromising on the ultimate decision whether to have reflection or refraction; but randomly obtaining each of them for its purpose, hence generating only one scattered ray during such instances. Below there is a **glass material** we created which always refracts the rays:

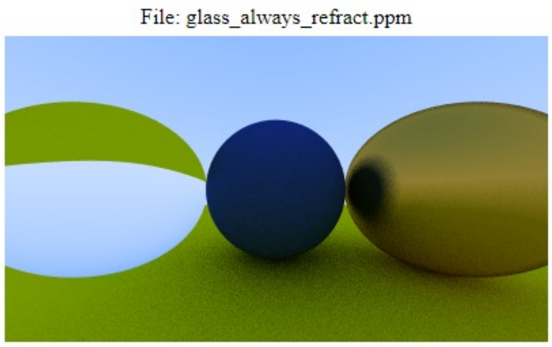


Figure 10: Always refracting glass (left one)

Refraction and Snell's Law:

Refraction arises when a beam of light curves as it changes from one substance to another. This shifting is affected by the medium's refractive index, or the amount of light that deflects when it shifts from a nothingness into that medium. For instance, we applied a dielectric material with a refractive index equal to **1.5** in order to introduce glass spheres in our scene. This is our **hollow glass material** simulation:

File: hollow_glass_sphere.ppm

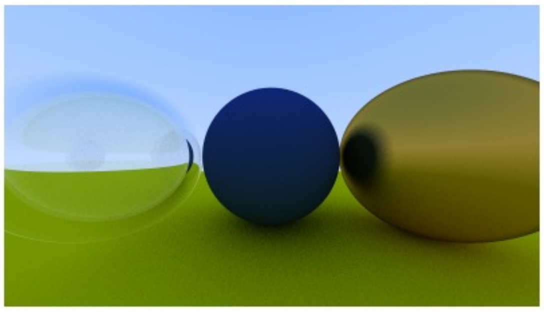


Figure 11: Hollow glass sphere (left one)

4.5 Camera Rendering

Our engine has a camera module that can be configured in multiple ways. In general, this implies different types of camera lenses and angles of view. The ability to configure the camera in this regard enables us to produce a wide range of visual effects and various perspectives. In the first place, we have changed the field of view (FOV) parameter. This parameter is a view from edge to edge of the final image. A rectangular FOV value is used throughout the image-generation process because the resulting imagery is not square.

5 Results

Our implementation of Ray Tracer, rendered scenes with different materials like Lambertian, metal, and dielectric (glass) surfaces effectively. Simulating how light rays behave after interacting with those materials has let us have real-like images portraying reflection, refraction as well as shading effects.

The ray tracer's ability to handle complex scenes with multiple objects and light sources has been demonstrated. We have achieved smooth shading (anti-aliasing), accurate reflections, and realistic transparency effects, enhancing the visual appeal of our rendered images.

6 Discussion

Our ray tracing engine successfully integrates advanced rendering techniques, achieving visually appealing results. The implementation of anti-aliasing, depth of field, and material simulation has greatly increased the realism of rendered images. Additionally, our approach is characterized by creative freedom when exploring various viewing angles due to use of a universal camera model.

7 Conclusion

In summary, our ray tracing engine has afforded us practical experience, rich in computer graphics field knowledge which has led to a deeper comprehension in regard to certain crucial principles like ray-object intersection testing, material properties description and different light models application. We are aware that this has enhanced our understanding of how complicated it really is to generate life-like pictures and the significance it carries regarding how faster running programs or even hardware-accelerated rendering itself can be developed.

Overall, the project has been very educative, allowing us to explore the field of ray tracing and its applications in creating different types of images.

8 References

Peter Shirley's *Ray Tracing in One Weekend* Series:

- *https://raytracing.github.io/books/RayTracingInOneWeekend.html*

The Cherno's Ray Tracing GitHub Repository :

- *https://github.com/TheCherno/RayTracing*

Sebastian Lague's *Coding Adventure: Ray Tracing* :

- *https://www.youtube.com/watch?v=Qz0KTGYJtUkt=88s*

Leslie Lai's *Ray Tracing in One Weekend* :

- *https://www.youtube.com/watch?v=RIfsLzCB7Mklist=PLlw1FcLpWd41aG18PMtlakvT3MQEQi*