Assignment 1: Implementation of Ray Tracing in C

[1]Luisa Santo, [2]Hugo Gaspar , [3]Vasco Nunes

[1]Information Systems and Computer Engineering Department, University of Lisbon, Av. Rovisco Pais, 1049-001, Lisbon, Portugal.

March 26, 2019

In computer graphics, ray tracing is a popular technique for rendering images with computers. In this project we implemented a local illumination model such as the Blinn-Phong model illumination and a version of ray tracing in C. We conducted experiments to demonstrate rendering time with the different NFF Files.

1 Introduction

In computer graphics, ray tracing is a popular technique for rendering images with computers. It is capable of simulating a variety of light effects, e.g., reflection and refraction, generating high degree of photorealism. In consequence, its computational cost is high.

In this assignment, we learned how to use OpenGL to render simple scenes. Afterwards, we moved away from OpenGL basics and implement a basic ray tracer that can handle ambient and diffuse lighting, soft shadows, reflections and refractions. We manage to calculate the render time of each NFF testing file to know the difference. We also based on a starter code able to render the scene with OpenGL.

1.1 Algorithm Overview

Given a scene of 3D objects and light sources, the goal is to generate an image from the viewpoint of a camera or eye. A 2D image plane consisting of pixels is placed between the viewpoint and the scene. The color of a pixel is the projection of the scene onto the image plane towards the viewpoint. Ray tracing generates the color of a pixel by sending the light ray through the pixel "backwards" away from the viewpoint. When a "backward" ray hits an object, it's recursively traced back depending on the material of the surface. In complex models more factors are considered, such as scattering, chromatic aberration, etc. For simplicity, we consider only reflections and refractions. A reflected ray for reflective surface and a refracted ray for translucent surface will be further traced. Figure 1 demonstrates the tracing paths for a single pixel. The blue ball and the red ball are both reflective and refractive. There are three back-tracing paths:

1. (reflection on red) → (refraction on blue) → (refraction on blue),
2. (reflection on red) → (reflection on blue), and
3. (refraction on red) → (refraction on red).

Since the last two paths are traced back to light sources, they are the only paths contributing to the final color of the pixel.

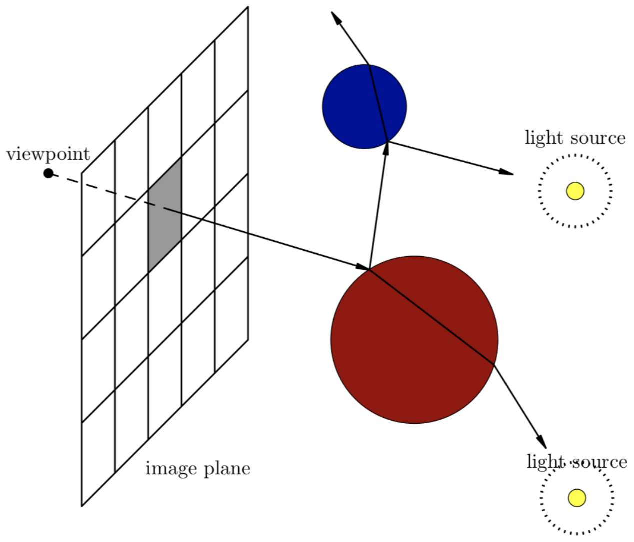

Figure 1: Ray tracing

Ray tracing has lots of advantages over the earlier rendering method. For example, the "ray casting" shoots rays from the viewpoint and finds the closest object that intersects the ray. Ray casting does not trace the rays further. The advantage of ray tracing is that it traces the lights bouncing among objects, which allows a much more realistic simulation of lighting over other rendering methods. A natural capability of ray tracing is to simulate reflections, refractions, and shadows, which are difficult using other algorithms.

The pseudo code for the ray tracing algorithm is shown in Algorithm 1. The final combined color `Combined` is a linear combination of the following three parts: the color of the object itself (`localColor`), the color from reflection (`reflectedColor`), and the color from refraction

Algorithm 1 `Ray Tracing`

```
1. for each pixel pixel in the image plane do
2.   Initialize:
     currentPoint ← the viewpoint
     ray ← the direction from the viewpoint to pixel
3.   return the color of pixel as COLOR(currentPoint, ray), which is calculated recursively as follows:
4.   COLOR(currentPoint, ray)
      1. if currentPoint is blocked by objects from all light sources then
      2.   return the ambient color
      3. else
      4.   Find the point hitPoint where ray first hits an object in the scene.
      5.   if hitPoint does not exists then
      6.     return the background color
      7.   else
      8.      Calculate the reflected ray reflectedRay
      9.      Calculate the refracted ray refractedRay
     10.      Set reflectedColor ← COLOR(hitPoint, reflectedRay)
     11.      Set refractedColor ← COLOR(hitPoint, refractedRay)
     12.     return combined color Combined(localColor, reflectedColor, refractedColor)
     13.   end if
     14.   end if
5. end for
```

(refractedColor). The local color is determined by the *Blinn-Phong* shading model.

2 Intersection

2.1 Plane

Computing the intersection of a ray with a plane it's simple. Suppose we have a plane defined by a point $\boldsymbol{p0}$ and a normal $\boldsymbol{N}$ with the plane at that point. We can compute a vector if we choose another point $\boldsymbol{p}$ belonging in the same plane and subtract that point to $\boldsymbol{p0}$. Since this vector lies on the plane, the dot product with the plane's normal is 0, then we can write:

$$(p - p0) \cdot N = 0 \tag{1}$$

If we define a ray on the parametric form we get:

$$o + d * t = p \tag{2}$$

Thus, we can replace $\boldsymbol{p}$ in 1, leading to:

$$t = -\frac{(o - p0) \cdot N}{N \cdot d} \tag{3}$$

From 3 we can conclude that:

- If the dot product between the ray direction d $\boldsymbol{d}$ and the plane normal $\boldsymbol{N}$ is 0, then the ray and plane are parallel, resulting in no intersection;
- the intersection is behind the plane if t $<$ 0, therefore we can reject it.

2.2 Sphere

Considering a ray in parametric form 2, and the quadratic equation, we will describe the algorithm taught in class:

1. Get the distance square from the origin of the ray to the center of the sphere,

$$d^2 = (x_c - x_o)^2 + (y_c - y_o)^2 + (z_c - z_o)^2 \tag{4}$$

2. Calculate b

$$B = x_d(x_c - x_o) + y_d(y_c - y_o) + z_d(z_c - z_o) \tag{5}$$

   - If d^2 is greater than r^2 and B is negative, there is no intersection point, due to the fact that the direction of the ray points in the opposite direction of the sphere;

3. Calculate R

$$R = B^2 - d^2 - r^2 \tag{6}$$

   - If R is negative, there is no intersection point;

4. Get t:

   - If $d^2 > r^2$, t = B - $\sqrt{R}$
   - If $d^2 \leq r^2$, t = B - $\sqrt{R}$

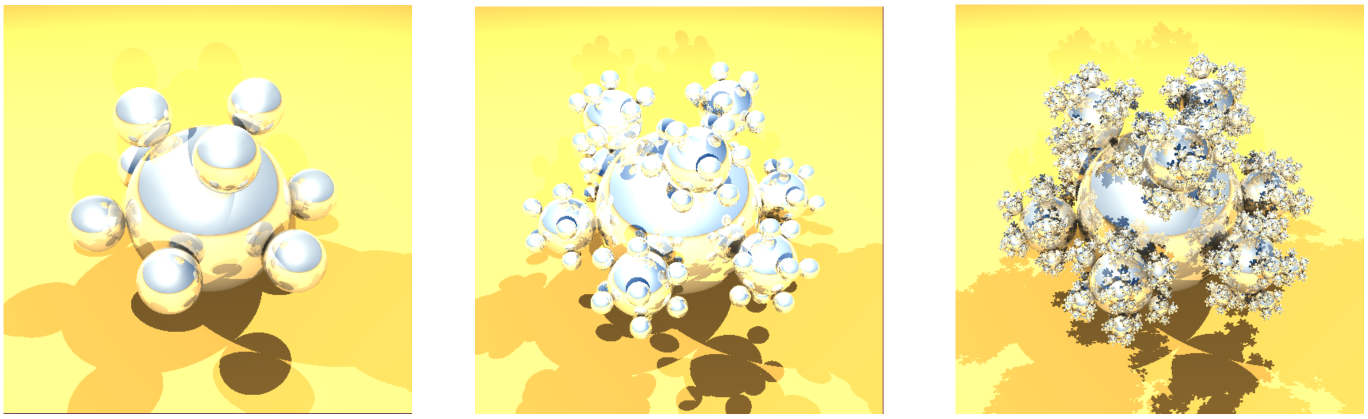

Figure 2: Final image of balls_low, balls_medium and balls_high

2.3 Triangle

In order to compute the intersection of a ray with a triangle it was used the "Fast, Minimum Storage Ray/Triangle Intersection" algorithm developed by Tomas Möller and Ben Trumbore.

A ray R(t) with origin O and normalized direction D is defined as:

$$R(t) = O + tD \tag{7}$$

A triangle is defined by 3 vertices, V0, V1, and V2. A point on a triangle is given by:

$$T(u, v) = (1 - u - v)V0 + uV1 + uV2 \tag{8}$$

where (u,v) are the barycentric coordinates and must fulfill u $\geq$ 0, v $\geq$ 0 and u + v $\leq$ 1

In order to compute the intersection between the ray R(t) and the triangle T(u,v) it is needed to solve:

$$R(t) = T(u, v) \tag{9}$$

which after some rearrangements results in:

$$\begin{bmatrix} -D & V1 - V0 & V2 - V0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V0 \tag{10}$$

Replacing:

$$E1 = V1 - V0 \tag{11}$$

$$E2 = V2 - V0 \tag{12}$$

$$T = O - V0 \tag{13}$$

And using Cramer's Rule the solution is calculated:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\begin{vmatrix} -D & E1 & E2 \end{vmatrix}} \begin{bmatrix} \begin{vmatrix} T & E1 & E2 \end{vmatrix} \\ \begin{vmatrix} -D & T & E2 \end{vmatrix} \\ \begin{vmatrix} -D & E1 & T \end{vmatrix} \end{bmatrix} \tag{14}$$

3 Experiment

3.1 Settings

In order to test the algorithms, a parser was developed to extract information regarding the camera, lights and objects from the NFF files. Figure 3 shows the work-flow of the entire project.

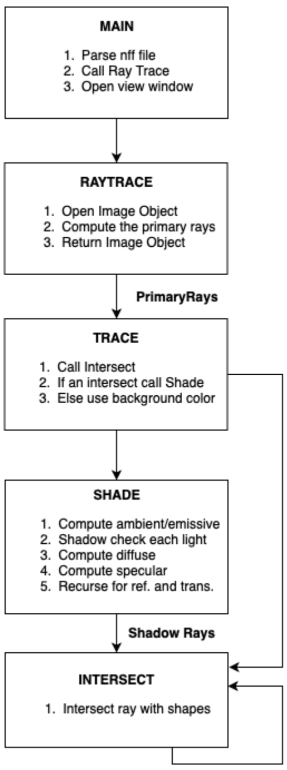

Figure 3: Project work-flow

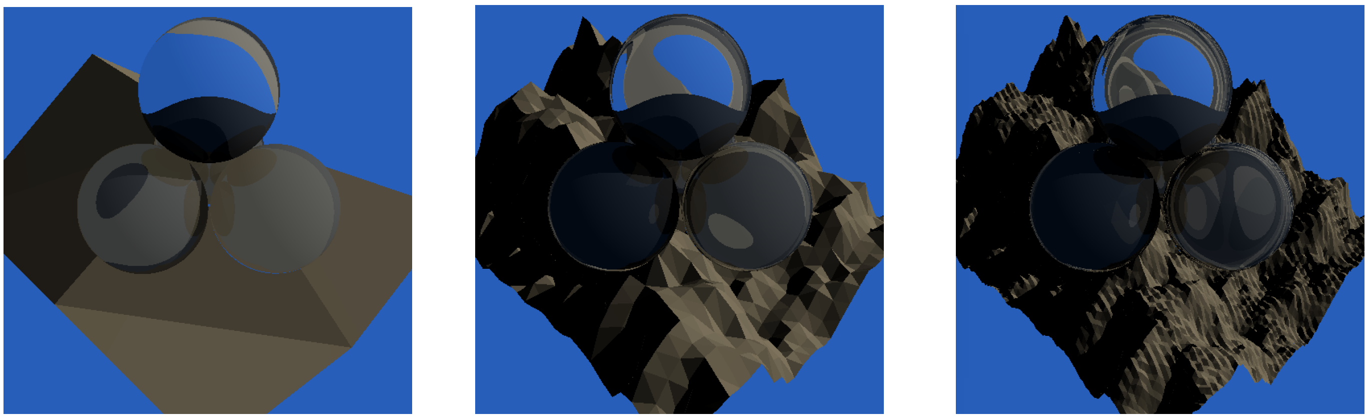
Figure 4: Final image of mount_low, mount_high and mount_very_high

3.2 Results

Our experimental results are derived from 2 simulations across six scenes. Depending on the complexity, each simulation can require from a few minutes to a few hours to complete. In this section we present some of our experimental results and the conclusions we draw based on them.

We therefore consider the time to render a frame and track how the render time is spent. In particular, we track the average time each scene that we assume is the sum of these events:

- Compute Execution: the time spent executing instructions,
- Compute Data Stall: stalls from waiting for the results of previous instructions,
- Memory Data Stall: stalls from waiting for data from the memory hierarchy, including all caches and DRAM, and
- Other: all other stalls caused by contentions on execution units and local store operations.

Table 1 shows the distribution of time used to render each six scenes shown in figures 2 and 4.

	Time(seconds)
balls_low	3
balls_medium	14
balls_high	1005
mount_low	2
mount_high	272
mount_very_high	4183

Table 1: Mean time to render each scene

4 Conclusion

In conclusion by using the ray tracing technique and the development of several algorithms previously explained we can render scenes with 3D images.

We also conclude that we could have calculated the average time spent per ray at different ray bounces: generation (ray generation kernel), **trace** (ray traversal kernel for non-shadow rays, including intersections), **Trace Shadows** (shadow ray traversal kernel) and **Shade** (shading kernel). We also think that we could have consider the **energy per ray** used by the kernel at different ray bounces by investigating the average energy spent to execute each scene.

5 Bibliography

[1] MOLLER, T., AND TRUMBORE, B. 1997. Fast, minimum storage ray-triangle intersection. Journal of graphics tools 2, 1, 21–28.

[2] https://www.scratchapixel.com/