Implementation of a Raytracer in C++ Computer Graphics Project (Rendering Track)

Alhajras Algdairy
alhajras.algdairy@gmail.com, 4963555, aa382

June 12, 2021

University: University of Freiburg
Instructor: Dr.-Ing. Matthias Teschner

Contents

1 Objective **3**
- 1.1 Milestones . . . 3

2 Creating an image **4**
- 2.1 Concept . . . 4
- 2.2 Pseudo code . . . 4
- 2.3 Output . . . 5

3 Rays **6**
- 3.1 Concept . . . 6
- 3.2 Pseudo code . . . 6
- 3.3 Output . . . 7

4 Adding a sphere **7**
- 4.1 Concept . . . 7
- 4.2 Pseudo code . . . 8
- 4.3 Output . . . 8

5 Adding a Triangle **9**
- 5.1 Concept . . . 9
- 5.2 Pseudo code . . . 9
- 5.3 Output . . . 9

6 Shading **11**
- 6.1 Concept . . . 11
  - 6.1.1 Lambert's Cosine Law . . . 11
  - 6.1.2 Phong reflection model . . . 11

7 Software architecture **13**
- 7.1 Concept . . . 13

1 Objective

The aim of this report is to document the steps which are important to implement a basic Raytracer in C++ language. As a beginner myself into the Computer graphics world, I will be explaining the tools, concept, equations, algorithms and sources that have been used during this small project.

1.1 Milestones

- Create PPM file and show an image, this includes understanding how images are generated.
- Show a sphere because it is a trivial object.
- Show a Triangle and implement an object array so it can be used in the future for rendering more than one object.
- Show a cube and a complex object by using the triangle array or mesh renderer.
- Create a benchmark tests in order to compare the basic functionality of the raytracer and after adding more features to it in the future.
- Add transformations: Translation, Rotation, Scaling, Reflection and Shearing.
- Phong model, and benchmark tests.
- Transparency and reflection

2 Creating an image

2.1 Concept

There are so many formats, ppm file, PPM stands for Portable Pixmap Format, which was developed in the 1980s to allow image files to be transferred between different computer platforms.

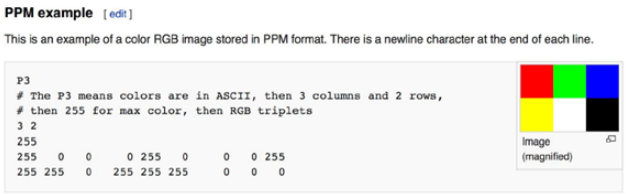

Figure 1: PPM format example.

2.2 Pseudo code

Below is pseudo code for writing out a ppm file.

```
FILE *fp;

fp = fopen("out.ppm","w");  // file name is out.ppm
int height = ImageHeight; //  assume ImageHeight is defined and initialized
int width = ImageWidth;  //   same for width
int max_ccv = 255;
fprintf(fp,"P3 %d %d %d\n", width, height, max_ccv);  // write the header

for (i=height-1; i>=0; i--)

for (j=0; j<width; j++) {

write the pixel [i,j]  s  red, green, and blue value;

}
fclose(fp);
```

LaTeX-Tutorial

2.3 Output

Figure 2: Hello world image in PPM format

Figure 3: Hello world image in PPM format

3 Rays

3.1 Concept

Optical ray tracing describes a method for producing visual images constructed in 3D computer graphics environments, with more photorealism than either ray casting or scanline rendering techniques. It works by tracing a path from an imaginary eye through each pixel in a virtual screen, and calculating the color of the object visible through it. All raytracers are using something called rays. Let's think of a ray as a function

$$p(t) = A + tB$$

Here p is a 3D position along a line in 3D. A is the ray origin and B is the ray direction.

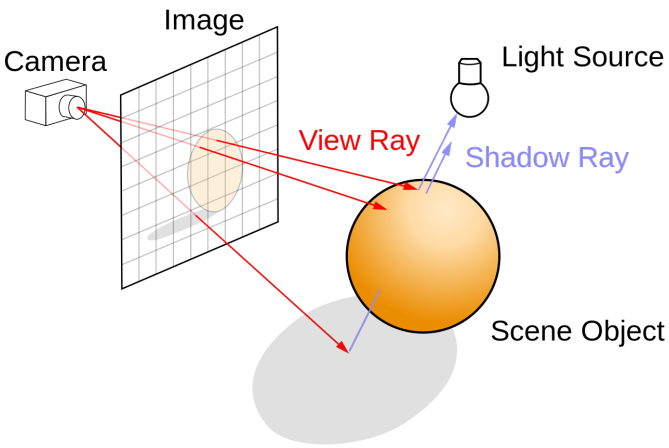

Figure 4: The ray-tracing algorithm builds an image by extending rays into a scene and bouncing them off surfaces and towards sources of light to approximate the color value of pixels. LaTeX-Tutorial.

3.2 Pseudo code

```
#ifndef RAY_H
#define RAY_H

#include "vec3.h"


class ray {
        public:
        ray() {}
        ray(const point3& origin, const vec3& direction)
        : orig(origin), dir(direction), tm(0)
```

```
        {}

        ray(const point3& origin, const vec3& direction, double time)
        : orig(origin), dir(direction), tm(time)
        {}

        point3 origin() const  { return orig; }
        vec3 direction() const { return dir; }
        double time() const    { return tm; }

        point3 at(double t) const {
                return orig + t*dir;
        }

        public:
        point3 orig;
        vec3 dir;
        double tm;
};

#endif
```

3.3 Output

Figure 5: Output of the blank background without an object by using rays

4 Adding a sphere

4.1 Concept

People often use spheres in ray tracers because calculating whether a ray hits a sphere is pretty straightforward.
The general equation of a sphere is:

$$\left|(x-c)^2\right| = r^2$$

Subsituting ray formula in spher we get:

$$(A+tB-c)^2.(A+tB-c)^2 = r^2$$

The form of a quadratic formula is now observable. (This quadratic equation is an instance of Joachimsthal's equation.[2])
By solving the equation we get three different cases as shown in figure below. LaTeX-Tutorial.

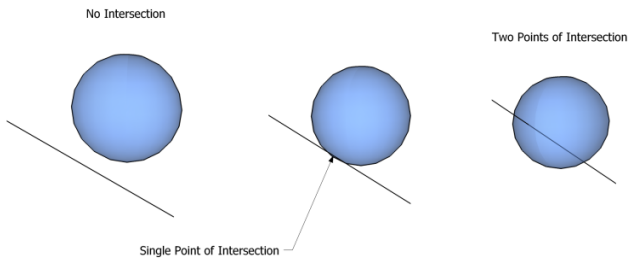

Figure 6: The three possible line-sphere intersections: 1. No intersection. 2. Point intersection. 3. Two point intersection.

4.2 Pseudo code

```
bool hit_sphere(const vec3& center, float radius, const ray& r) {
        vec3 oc = r.origin() - center;
        float a = dot(r.direction(), r.direction());
        float b = 2.0 * dot(oc, r.direction());
        float c = dot(oc, oc) - radius * radius;
        float discrimnant = b * b - 4 * a * c;
        return (discrimnant > 0);
}
```

4.3 Output

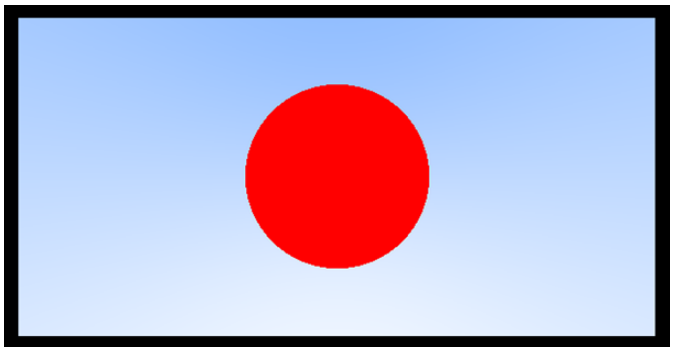

Figure 7: Rendering a simple Sphere

5 Adding a Triangle

5.1 Concept

Computing the intersection of a ray with a primitive such as a sphere, is not difficult. However, since it is difficult to model most 3D objects with spheres alone, it is necessary to use some other types of primitive to represent more complex objects (objects of arbitrary shape). Instead of working with complex primitives such as NURBS or Bezier patches, we can convert every object into a triangle mesh and compute the intersection of a ray with every triangle in this mesh. By solving the equation we get three different cases as shown in figure below. LaTeX-Tutorial

$$P = O + tR$$
$$Ax + By + Cz + D = 0$$
$$A * P_x + B * P_y + C * P_z + D = 0$$
$$A * (O_x + tR_x) + B * (O_y + tR_y) + C * (O_z + tR_z) + D = 0$$
$$A * O_x + B * O_y + C * O_z + A * tR_x + B * tR_y + C * tR_z + D = 0$$
$$t * (A * R_x + B * R_y + C * R_z) + A * O_x + B * O_y + C * O_z + D = 0$$
$$t = -\frac{A * O_x + B * O_y + C * O_z + D}{A * R_x + B * R_y + C * R_z}$$
$$t = -\frac{N(A,B,C) \cdot O + D}{N(A,B,C) \cdot R}$$

5.2 Pseudo code

```
Vec3f edge0 = v1 - v0;
Vec3f edge1 = v2 - v1;
Vec3f edge2 = v0 - v2;
Vec3f C0 = P - v0;
Vec3f C1 = P - v1;
Vec3f C2 = P - v2;
if (dotProduct(N, crossProduct(edge0, C0)) > 0 &&
dotProduct(N, crossProduct(edge1, C1)) > 0 &&
dotProduct(N, crossProduct(edge2, C2)) > 0) return true; // P is inside the triangle
                }
```

5.3 Output

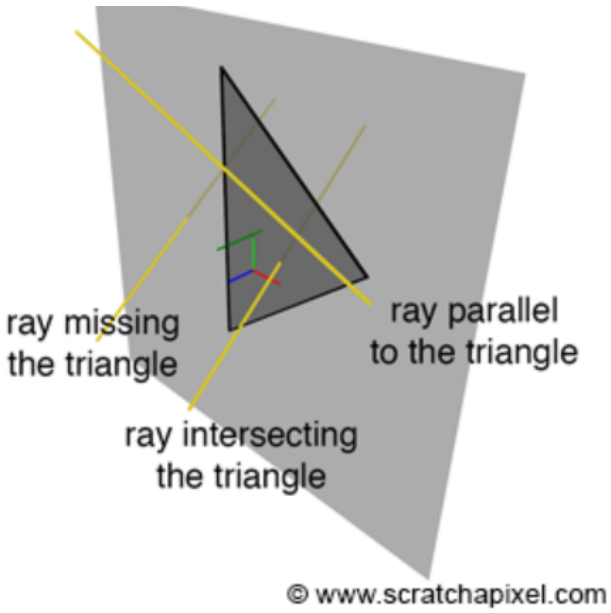

Figure 8: Rendering a simple Sphere

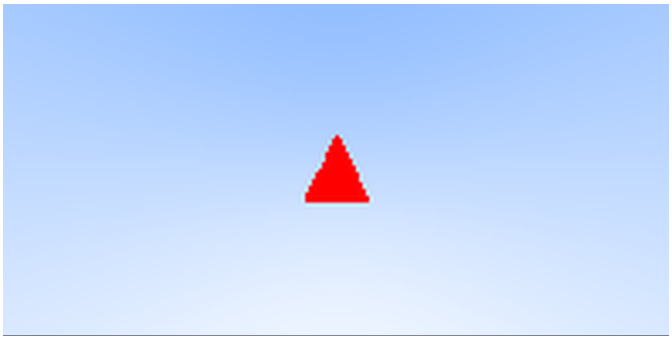

Figure 9: Rendering a simple Triangle

6 Shading

6.1 Concept

Rendering a scene needs two steps the first step is solving the visibility issue, this means which object is visible to the camera and what its shape, second step is Shading, this deals with the color of the object and its intensity. Shading include also how objects color affect each other, for example having light hits the object will make its color look brighter, on the other hand regions which light doesn't hit or reach will have dark color or shadow. In this chapter shading concepts will be discussed and implemented. The main key to shading is calculating the amount of light that hits a point lets call it P.
The computed light at a point P depends on the following:

- Light illuminated by source source L^{source} in real life usually lamp, fire or the sun, it can have any color and intensity but here we will use white color.
- Light illimunated on the surface $L^{surface}$ is the surface illumination caused by
- Light reflected from the surface $L^{reflected}$
- The observation angle / looking at angle / camera.

6.1.1 Lambert's Cosine Law

Lambert's cosine law says that the amount of light energy arriving at a surface is proportional to the cosine of the angle between the light direction and the surface normal. Illumination strength at a surface is proportional to the cosine of the angle between l and n, the angel will be denoted as θ, the next three cases illustrate the relationship between the L^{source} and $L^{surface}$:
The $L^{surface}$,L^{source} relation is: $L^{surface} = L^{source}.\cos\theta$

- $L^{surface} = L^{source}$, if $\theta = 0°$.
- $L^{surface} = 0$, if $\theta = 90°$.
- $0 < L^{surface} < L^{source}$, if $0° < \theta < 90°$.

6.1.2 Phong reflection model

Phong reflection is an empirical model of local illumination. It describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces. It is based on Phong's informal observation that shiny surfaces have small intense specular highlights, while dull surfaces have large highlights that fall off more gradually. The model also includes an ambient term to account for the small amount of light that is scattered about the entire scene.

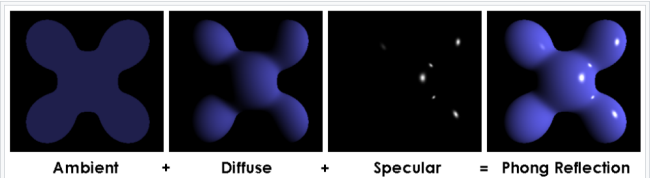

Figure 10: Visual illustration of the Phong equation: here the light is white, the ambient and diffuse colors are both blue, and the specular color is white, reflecting a small part of the light hitting the surface, but only in very narrow highlights. The intensity of the diffuse component varies with the direction of the surface, and the ambient component is uniform (independent of direction).

- **Ambient reflection**
  - $L^{amb} = \rho \otimes L^{indirect}$
  - ρ, is the surface color
  - $L^{indirect}$, is the light reflected from other surfaces and objects, excluded the direct light ($L^{surface}$)
- **Diffuse reflection**
  - $L^{diff} = L^{source}.(n.l) \otimes \rho$
  - L^{source}, is the light source color and intensity which usually white.
  - n and l , are the representation of the Lambert's cosine law, where n is the normal surface vector and l is the indecent light coming from the light source.
- **Specular reflection**
  - $L^{spec} = L^{source}.(n.l).(r.v)^m \otimes \rho^{white}$
  - r, which is the direction that a perfectly reflected ray of light would take from this point on the surface.
  - v, which is the direction pointing towards the viewer (such as a virtual camera).
  - m, which is a shininess constant for this material, which is larger for surfaces that are smoother and more mirror-like. When this constant is large the specular highlight is small.

The overall illumination on the surface can be computed by summing up the three components that make up Phong model:

$$\mathrm{L}^{surface} = L^{amb} + \sum_{n=1}^{lights}(L_n^{diff} + L_n^{spec})$$

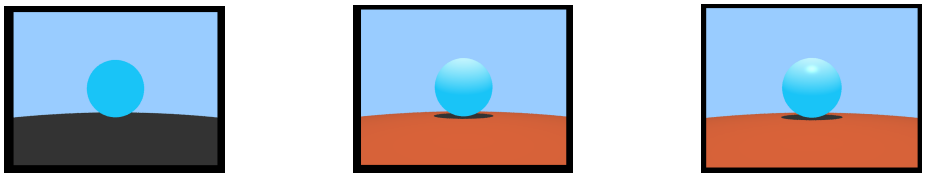

Figure 11: Some images

$$pd <= 1$$

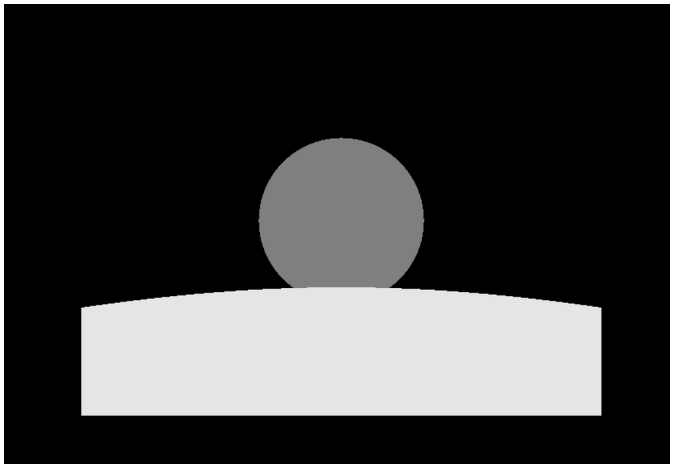

Figure 12: No light

7 Software architecture

7.1 Concept

In this chapter I will explain the software architecture aspect.

References

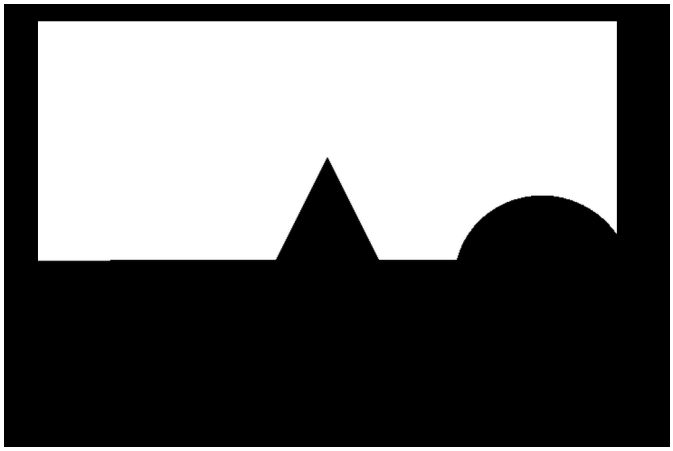

Figure 13: Dark

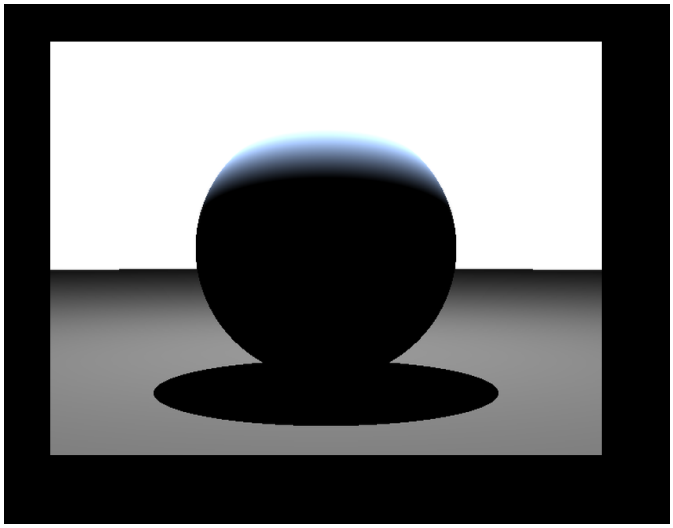

Figure 14: Dark

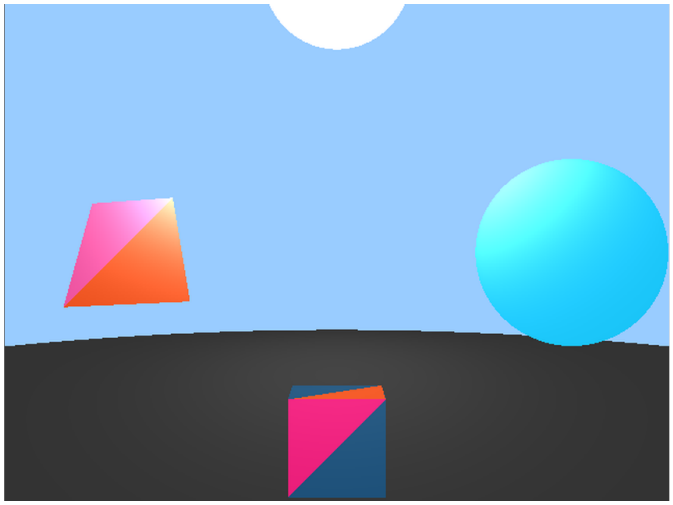

Figure 15: Full