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

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

June 21, 2021

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

Contents

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.

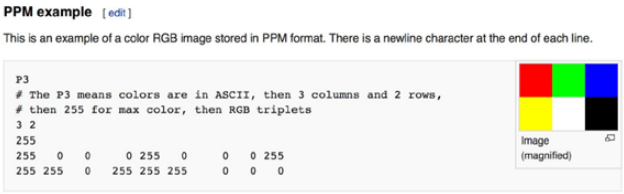


Figure 1: PPM format example.

2.2 Results

Figure 2: 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.

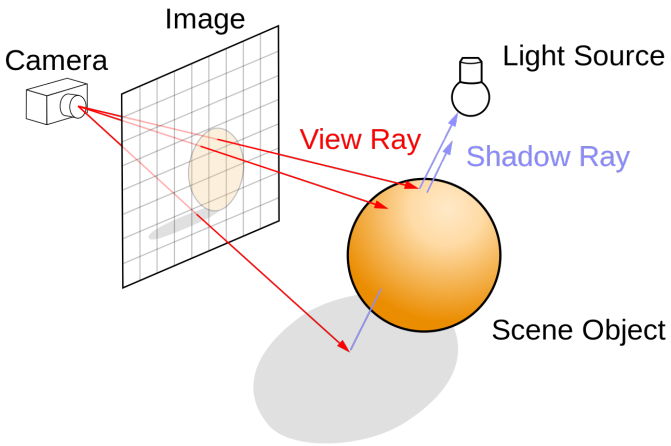


Figure 3: The ray-tracing algorithm builds an image by extending rays into a scene and bouncing them off surfaces and towards sources of light to approximate the color value of pixels. LaTeX-Tutorial.

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 Results

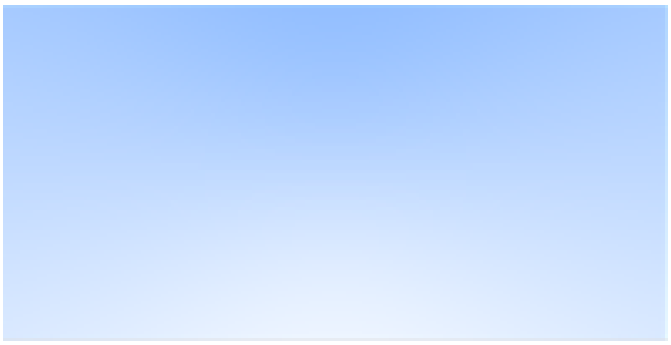


Figure 4: Output of the blank background without an object by using rays

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.

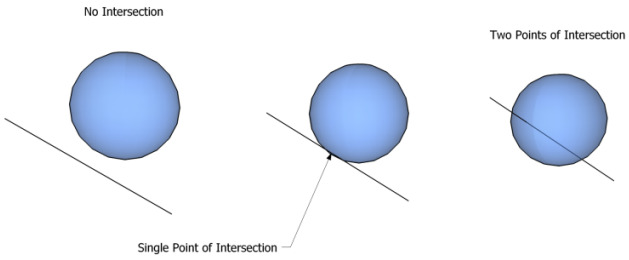


Figure 5: The three possible line-sphere intersections: 1. No intersection. 2. Point intersection. 3. Two point intersection.

LaTeX-Tutorial.

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 Results

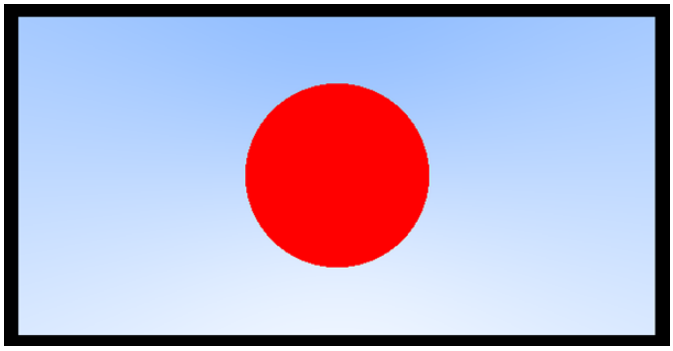


Figure 6: Rendering a simple Sphere

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.

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

LaTeX-Tutorial

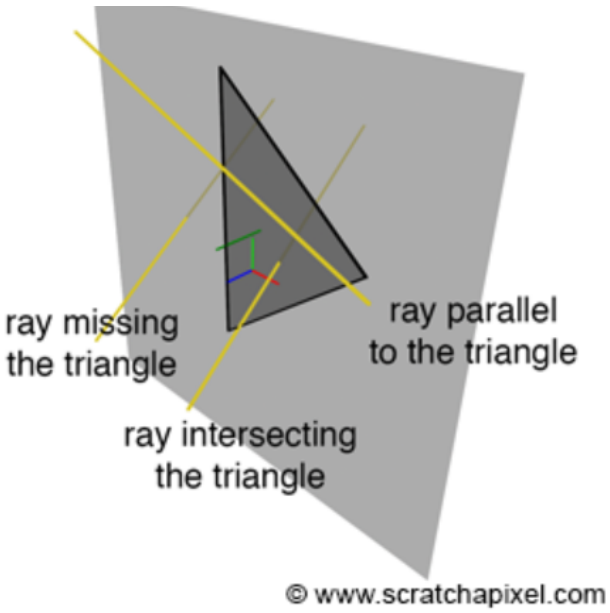


Figure 7: Rendering a simple Sphere

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 Results

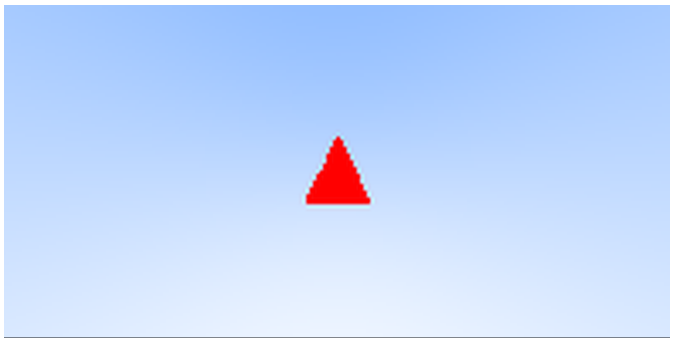


Figure 8: Rendering a simple Triangle

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.
- Surface illumination $L^{surface}$.
- Light reflected from the surface $L^{reflected}$.
- The observation angle / looking at angle / camera.

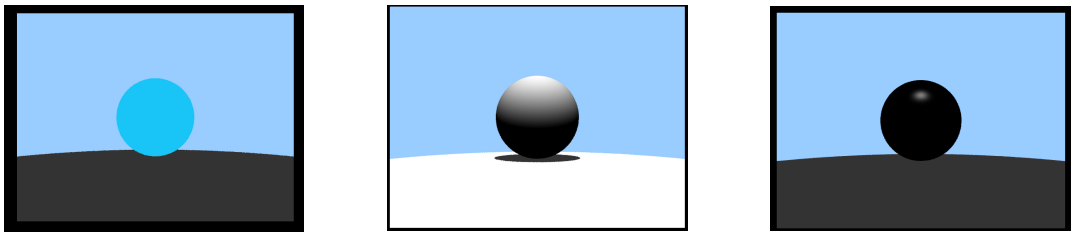
6.1.1 Lambert's Cosine Law

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.



(a) (b) (c)

Figure 9: (a). Ambient .(b). Diffuse. (c). Specular (Glossy)

- **Ambient reflection**

$$L^{amb} = \rho \otimes L^{indirect} \tag{1}$$

- ρ, is the surface color
- $L^{indirect}$, is the light reflected from other surfaces and objects, excluded the direct light (L^{source})

- **Diffuse reflection**

$$L^{diff} = L^{source}.(n.l) \otimes \rho \tag{2}$$

- 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.

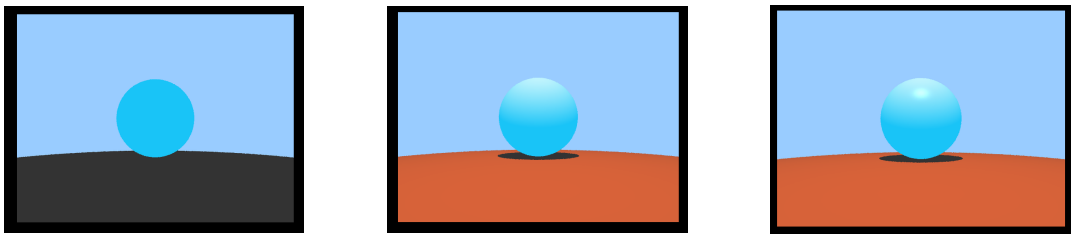
- **Specular reflection**

$$L^{spec} = L^{source}.(n.l).(r.v)^m \otimes \rho^{white} \tag{3}$$

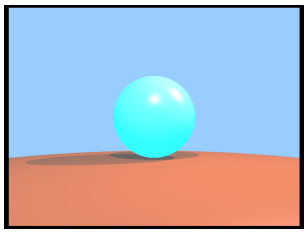
- r, which is the direction that a perfectly reflected ray of light would take from this point on the surface.
- v, which is the direction pointing towards the viewer (such as a virtual camera).
- m, which is a shininess constant for this material, which is larger for surfaces that are smoother and more 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*:

$$L^{surface} = L^{amb} + \sum_{n=1}^{lights} (L_n^{diff} + L_n^{spec}) \tag{4}$$



(a) (b) (c)



(d)

Figure 10: (a). Ambient .(b). Diffuse + Ambient. (c). Diffuse + Ambient + Specular (Glossy). (d). Multiable lights that generates brighter surface and multiple shadows.

7 Materials

7.1 Concept

In raytracing one of the important topics is to give the object a material type, this can be: Glossy, Diffuse, Transparent and Subsurface scattering, each material has different surface reflection and refraction properties, some surfaces reflect the light equally such as diffuse surfaces, some are reflecting light into a dominant direction as Glossy surfaces, some material such as water will reflect some light but also refract some.
In the previous chapter we discussed diffuse and specular (Glossy) surfaces, in this chapter we will implement a refraction surface because it has an interesting properties.

7.2 Refraction

Refraction phenomenon happens when the light pass from one medium to a different medium. Figure 12 illustrates this phenomenon, I is the incident light ray, R is the reflected light where N is the normal vector to the surface which is water (in blue), the reflected angle $\theta 2$ is equal to the incident angle $\theta 1$, in addition to the reflected light there is a refracted light T, the direction of T depends on the $\theta 1$ and *refractive index,* η (Describes how fast light travels through the material).
Refraction is described by the *Snell's law*, which states that for a given pair of media, the ratio of the sines of the angle of incidence $\theta 1$ and angle of refraction $\theta 2$ is equivalent to the opposite ratio of the indices of refraction:

$$\frac{sin(\theta_1)}{sin(\theta_2)} = \frac{\eta_2}{\eta_1} \tag{5}$$

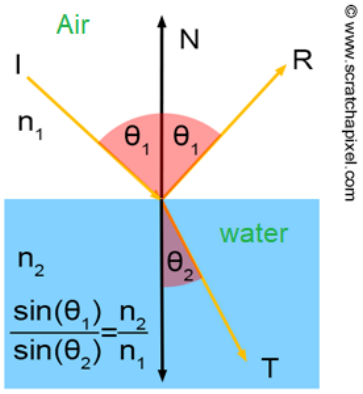


Figure 11: Incident light I in the air hitting water surface, R is reflected light and T is the transmitted and refracted light, n is *refractive index,* η Resource: scratchapixel.com.]

7.3 Fresnel

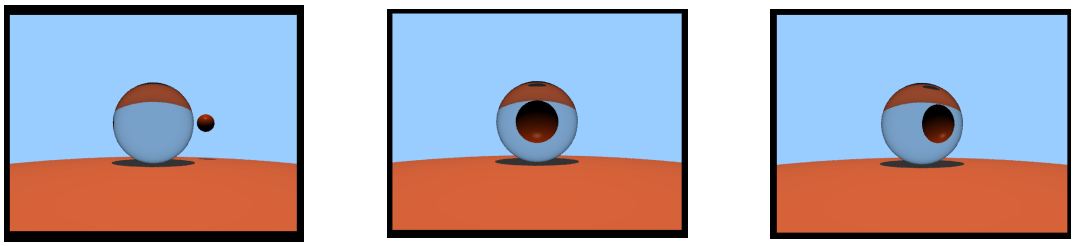
The amount of reflected vs. refracted light can be computed using what we call the *Fresnel equations*, where F_R is reflected light portion and F_T is the portion transmitted through the material.

$$F_R = \frac{1}{2}\left(\left(\frac{\eta_2 \cos\theta_1 - \eta_1 \cos\theta_2}{\eta_2 \cos\theta_1 + \eta_1 \cos\theta_2}\right)^2 + \left(\frac{\eta_1 \cos\theta_2 - \eta_2 \cos\theta_1}{\eta_1 \cos\theta_2 + \eta_2 \cos\theta_1}\right)^2\right) \tag{6}$$

$$F_T = 1 - F_R \tag{7}$$

7.4 Results

Figure 13 shows the refraction phenomenon where the settings had a red glossy sphere passing behind a transparent sphere with an refractive index, η not equal to the air we are assuming it is any kind of liqued with big η that it refract the light with a big angle. As it can be noticed the red sphere looks larger than it should be and also inverted, moreover the ground is inverted as well,

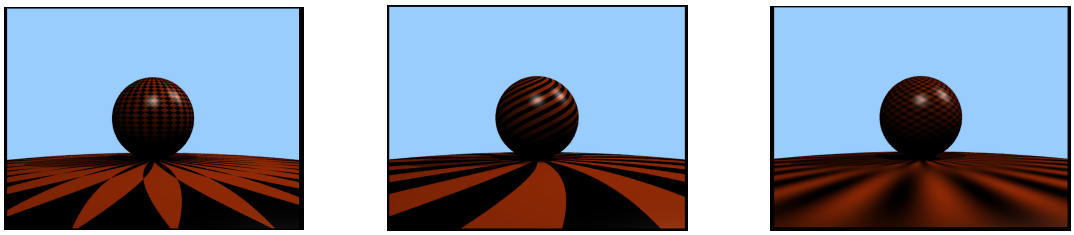


(a) (b) (c)

Figure 12: (a). Glass sphere refract the ground inverted upside down. (b). Glass sphere refract the ground and red sphere directly behind it inverted. (c). Glass sphere refract the ground and red sphere behind it inverted with an angle

8 Texture

In computer graphics, the application of a type of surface to a 3D image. A texture can be uniform, such as a brick wall, or irregular, such as wood grain or marble. The common method is to create a 2D bitmapped image of the texture, called a "texture map," which is then "wrapped around" the 3D object. An alternate method is to compute the texture entirely via mathematics instead of bitmaps. Textures are so useful, they reduce the geometric complexity of a scene by mapping a bit directly to an image or having a mathematical equation that can easily represent the surface color, they also reduce the number of vertices, and it reduces the modeling and rendering time.
We will only consider some mathematical patterns and not image mapping, procedural texturing define how to use the texture value, e.g. replace the original surface color with the texture color linearly combine the original surface color with the texture multiply, add, subtract surface color and texture color.

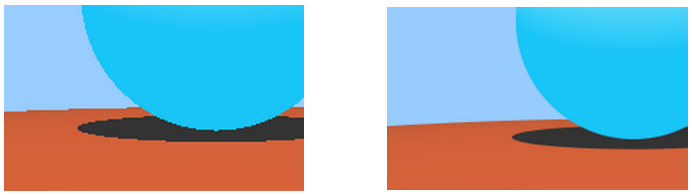


(a) (b) (c)

Figure 13: (a), (b) and (c) show different pattern for textures that can be generated by mathematical equations to map x and y value to different color value.

8.1 Anti-aliasing

The computer display is composed of pixels. This is the smallest element of every digital image, and while modern computer monitors boast high resolutions that feature millions of pixels, these pixels are still rectangular in shape. What this means is that, when round shapes are shown on screen, you're almost guaranteed to see some jagged edges, that is, aliasing. Here we will use a basic MSAA which stands for "multisample anti-aliasing," and it is among the most common types of anti-aliasing. It generally strikes the best balance between visual fidelity and performance. This can be implemented by averaging abunch of samples inside each pixel.
For a given pixel we have several samples within that pixel and send rays through each of the samples.The colors of these rays are then averaged.



(a) (b)

Figure 14: (a)Alising, (b) Anti-aliasing with 100 samples per pixel.

Figure 14 shows the refinement and how the objects edges are been smoothed by using the anti-aliasing technique. The issue is performance because the more smoothed edges get the more samples had to be taken around each pixel, for example if the image size is 200 x 200, and the samples are used for anti-aliasing are equal to 100, then we will be measuring the color for 200 x 200 x 100 pixels rathar than 200 x 200. A trade-off has to be done here in order to have a smoothed edges but with no huge performance coset, this depends on application.

References