Raytracing

Dina EL ZEIN dina.el-zein@ens-lyon.fr

Dec 28, 2020

1 Introduction

Raytracing is a rendering technique that can produce realistic lighting effects in applications where taking relatively long time to render can be tolerated such as computer generated images but more poorly suited for real time applications such as video games where speed is important in rendering each frame. In images, it's the determination of the correct color at each pixel. One way of producing images in 3d scene is to to shoot rays from the camera through each pixel of the image and then calculate the color at the point of intersection (which is a combination of the color of the object and the color of the camera ray). This method requires us to loop over all the pixels in the image and shoot a ray into each of these pixels. Raytracers work by tracing the path of light starting at the position of the camera (the one shooting rays) and going out until it hits an object. The point of intersection of the camera ray at the object is tested to check if it's shadow or not, if the object is reflective or transparent then we can shoot more rays to check the color of light being transmitted. The code to replicate the experiments is released at `https://github.com/dinalzein/Raytracing.git`.

2 Principle

The idea is to simulate the behavior of light using the Helmholtz reciprocity principle. The Helmholtz reciprocity principle describes how a ray of light and its reverse ray encounter matched optical adventures [3]. In different words, it says that the rays of light can be followed in an inverse order from the sensor to the light source instead from the light source to the sensor. To do that, we launch light rays from the camera through each pixel of the image and we check for the object it intersects. We can then display this pixel in the color of the object.

3 Preliminaries

To implement a simple ray tracing, we need the following:

- Save a .bmp image stored as an array of pixels with the value of red, green, and blue color at each pixel. We can use stb-image to do this and here how to access the red, green and blue components of the pixel at row i and column j:

  $pixel\_array[((height - i - 1) * width + j) * 3 + 0] = red$
  $pixel\_array[((height - i - 1) * width + j) * 3 + 1] = green$
  $pixel\_array[((height - i - 1) * width + j) * 3 + 2] = blue$

  The resolution of the image is defined by its width and height (these are defined in the main class).

- Define a Vector class which allows us to work with vectors. A vector is a position in the 3D space identified by X, Y, and Z coordinates. The vector class defines the X, Y, and Z coordinates (in double precision).

- Define a Ray class that represents the ray of light. The ray of light is identified by two vectors: one for origin and the other for direction.
- Define a Sphere class that represents a sphere object (we will deal with sphere objects with our implementation). The sphere is represented by: a vector for its origin, a double value for its radius, and a vector for its color (the x, y, and z coordinates of the vector represent the red, green, and blue components if the color of the object in RGB).

  For more information about the implementation of these methods, please check the code for these classes in the Raytracing project file.

4 The Light

Our main goal is to find the color of the light leaving at the point of intersection (between the sphere and the ray) backward. For now, we just found the point at which the ray hits the object and we color it white, but the light may continue in the opposite direction and this is what's called backward ray tracing. The image now looks flat due to the lack of lighting. For it not to look flat, we can replace the color at the pixel(i,j) when the ray intersects the object by:

$$pixel(i,j) \leftarrow max(0, \overrightarrow{l}.\overrightarrow{N}) * I/(4\pi d^2)$$

where $\overrightarrow{l}$ is the unit vector from the intersection P (between the ray and the object (in our case sphere)) towards the light source, $\overrightarrow{N}$ is the normal to the sphere toward the point of intersection, I is the intensity of the light, and d is the distance between the light and P. This is what we get with intensity =1000000000 and light position $\overrightarrow{L}$ (15, 70, -30).

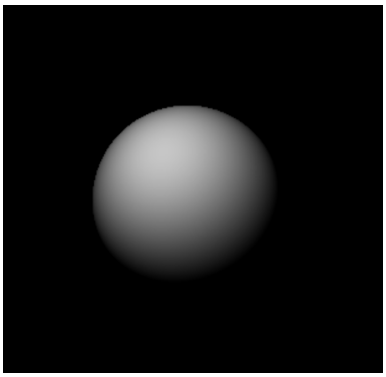

Figure 1: Light

Now we want to work with more than one sphere. To do so, we added multiple spheres to our scene object by creating a vector of Spheres. We added multiple spheres with different positions and radii. We kept the radius of the first sphere as before and we chose the radius of the other spheres to be large so that we don't have multiple clear spheres in the image. One may assume that these spheres (with large radii) behave as the background (wall, floor..) in the image. We concluded this from the image in the subject instructions as we can just see one sphere there.

Now, we have to find the intersection between the ray and the multiple spheres we added. To do so, we added a method called "findIntersection" in the main class. This method loops through all the scene objects and calls the "findIntersection" method in the sphere class on each of these objects (spheres in our case). If

there are multiple spheres that intersect the ray, we take the smallest intersection (by smallest we mean the one which is closest to the origin). By doing this we get:

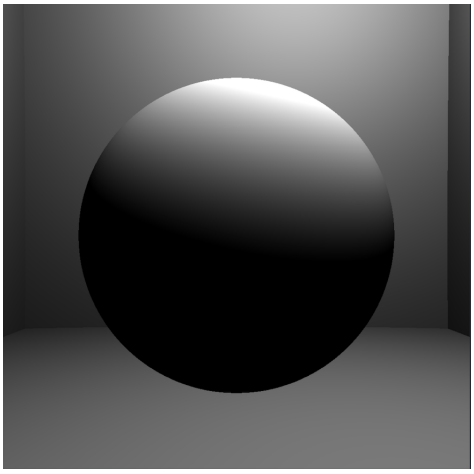

Figure 2: Light with multiple spheres

Now, it's time to use the color vector of the sphere by giving each sphere a color. The color vector has three coordinates x, y, and z each corresponds to the red, green, and blue components (RGB components) of the color respectively. We chose to adjust the values of these components to be between 0 and 1 (instead of 0 and 255) so (0,0,0) corresponds to black and (1,1,1) corresponds to white. By giving each sphere a color we get the same image as before but with different colors (as we are just changing the colors of the spheres):

Figure 3: Light with multiple spheres with different colors

For more information about the implementation of all of this part, please refer to the code of the mentioned methods in the main class.

5 Cast Shadows

In this section we want to study how to handle cast shadows. A Cast shadow is a shadow of an object on other surface caused by the blockage of the light by an object. The principle is easy, when an intersection P is found, generate a light ray from P going toward the light source L (so with direction $P - L$) and compute

the intersections again with the objects. If there is an intersection, determine the point of intersection P' and check if the distance from P to P' is less than the distance from P to L, if so the point is shadowed and its color is black. After doing this, we get:

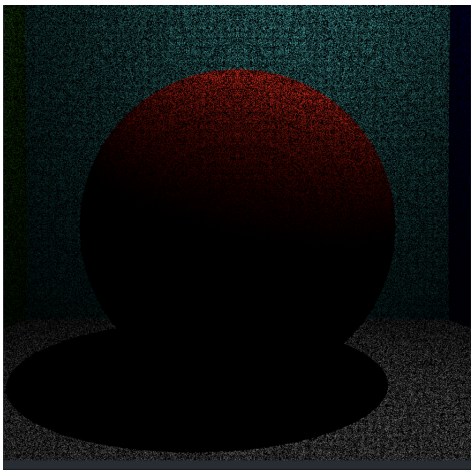

Figure 4: cast shadows

This noise is due to numerical inaccuracy, simply launch the light ray from the point $P + epsilon * N$ where N is the normal as defined before. Now, we get:

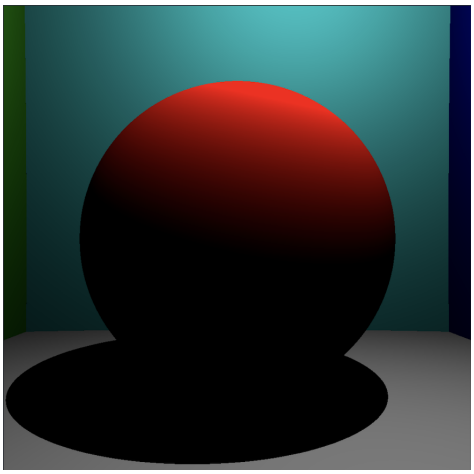

Figure 5: cast shadows without noise

Specular Surfaces

Specular reflection which is known as mirrors is the reflection of light from surface. To implement specular reflections, we reflect the incident light ray around the normal $\overrightarrow{N}$ and return the reflected color instead of sending the light ray in the direction of light position $\overrightarrow{L}$. The reflected light ray is then

$$\overrightarrow{r} = \overrightarrow{i} - 2 < \overrightarrow{i}, \overrightarrow{N} > \overrightarrow{N}$$

where $\overrightarrow{i}$ is the direction of the camera ray and $\overrightarrow{N}$ is the normal. In order to get the color of the reflected ray we added a recursive getColor method that returns the reflected color as a Vector with its three components : red, green, and blue components (in RGB).

In the implementation process for this part, we keep the choice for the user to decide whether they want to specular surface by adding a double parameter called mirror in the Sphere class that can take values either -1 or 1, 1 for specular surface and -1 otherwise. We then obtain the following image:

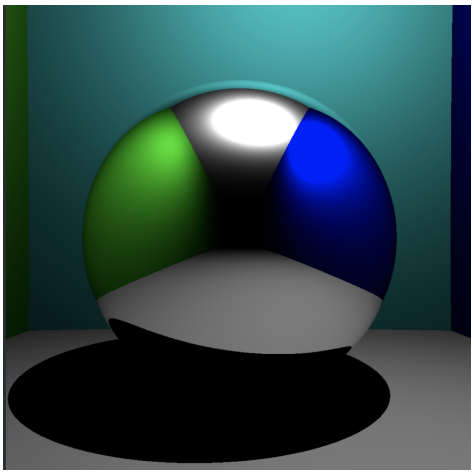

Figure 6: specular surface

6 Transparent Surfaces

Just as there is a direction from which the light can be reflected into the direction of the incident ray, there is a direction from which the light can be transmitted (or refracted) into the direction of incident ray, this ray is called the transparent ray [2]. For the refracted ray, the formula is

$$n\_airsin\theta_i = n\_spheresin\theta_o$$

where the n's represent the refraction indices ($n\_air = 1.0003 \approx 1 \text{and} n\_sphere = 1.333 \approx 1.3$), θ_i and θ_o the incident and refracted angles respectively.

The refracted ray will have this tangent component:

$$\frac{n\_air}{n\_sphere}(\overrightarrow{i} - < \overrightarrow{i}, \overrightarrow{n} > \overrightarrow{n})$$

where $\overrightarrow{n}$ is the normal vector (same value as N).

and this normal component:

$$-\sqrt{1 - sin^2\theta\_o}\overrightarrow{n}$$

and so the refracted ray is:

$$\overrightarrow{r} = \frac{n\_air}{n\_sphere}(\overrightarrow{i} - < \overrightarrow{i}, \overrightarrow{n} > \overrightarrow{n}) - \sqrt{1 - (\frac{n\_air}{n\_sphere})^2(1 - < \overrightarrow{i}, \overrightarrow{n} >^2)}\overrightarrow{n}$$

Now, we have to check if the normal is in the same direction as the incident ray i.e. $< \overrightarrow{i}, \overrightarrow{n} >< 0$ meaning that the ray enters the sphere, otherwise, the roles of n_air and n_sphere are reversed ($n\_air = 1.3$ and $n\_sphere = 1$) and the normal is reversed ($= -\overrightarrow{N}$). We then obtain the following image:

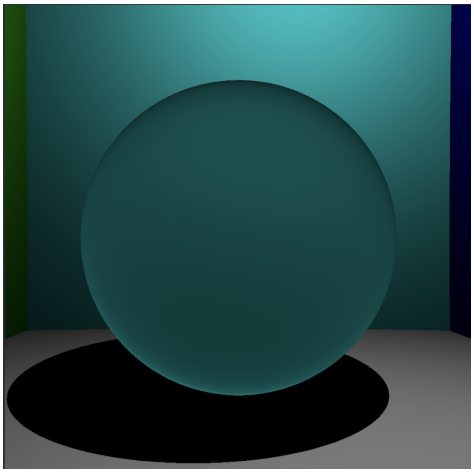

Figure 7: transparent surface

7 Gamma Correction

Some images look either bleached out or too dark. These images can be corrected by a gamma correction. Gamma correction controls the brightness of an image by a power function $(1./2.2)$ for the red, green, and blue components of the color. After applying gamma correction we get:

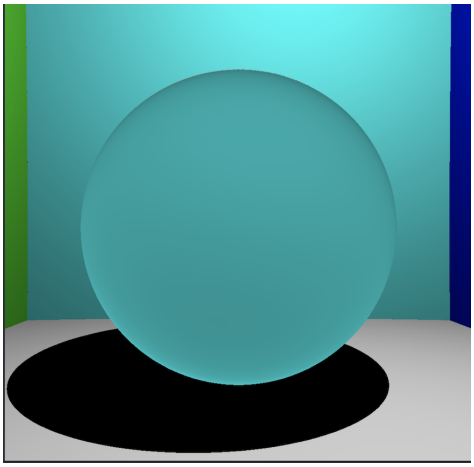

Figure 8: transparent surface with gamma correction

8 Indirect Lighting

Working with indirect lighting is very similar to working with specular surfaces. We have to recursively bounce light rays for the image but with random reflected rays.

Bidirectional Reflectance Distribution Functions (BRDFs)

BRDFs are functions that describe how light is reflected. We presented the specular surfaces one which are a BRDF while the transparent surfaces are not BRDF because they represent transmitted lights and not reflected ones.

For specular, the BRDF is:

$$f(\overrightarrow{i}, \overrightarrow{o}) = \frac{1}{cos\theta\_i}\delta(2 < \overrightarrow{i}, \overrightarrow{n} > \overrightarrow{n} - \overrightarrow{i} - \overrightarrow{o})$$

where $\overrightarrow{n}$ is the normal, $\overrightarrow{i}$ is the incident direction, $\overrightarrow{o}$ is the direction to be reflected in, and δ is the Dirac distribution.

where as for diffuse, the BRDF is defined as:

$$f(\overrightarrow{i}, \overrightarrow{o}) = \frac{constant}{\pi}$$

which is a constant function.

Now, we will introduce the rendering equation, the one which gives the color of a certain pixel.

$$L_o(x, \overrightarrow{o}) = E(x, \overrightarrow{o}) + \int f(\overrightarrow{i}, \overrightarrow{o}) L_i(x, \overrightarrow{i}) cos\theta_i d_{\overrightarrow{i}}$$

where $E(x, \overrightarrow{o})$ is the emissivity of the surface, $f(\overrightarrow{i}, \overrightarrow{o})$ is the BRDF, $L_i(x, \overrightarrow{i})$ is the light intensity arriving at the point of intersection x, $L_o(x, \overrightarrow{o})$ is the intensity of the outgoing light, and $cos\theta_i$ is the angle between the incident ray $\overrightarrow{i}$ and the surface normal.

This equation says that the light outgoing in a particular direction at a point is the sum of all incident light (which includes indirect light sources) multiplied by the BRDF and cosine.

Assuming $E = 0$ for specular surfaces, we get the rendering equation to be

$$L_o(x, \overrightarrow{o}) = L_i(x, 2 < \overrightarrow{o}, \overrightarrow{n} > \overrightarrow{n} - \overrightarrow{o})$$

The Monte Carlo Method

It is a computational algorithm for calculating integrals that rely on random sampling. Given $x_0, x_1, ..., x_n$ to be random numbers that follow a probability distribution $p(x)$. Using the Monte Carlo method, one can show that

$$\int f(x)dx = \frac{1}{n}\sum i = 0^n \frac{f(x_i)}{p(x_i)} + O(\frac{1}{\sqrt{n}})$$

Now, we need to compute the integral of the rendering equation, to do so, one can generate random directions with a probability distribution close to $f(\overrightarrow{i}, \overrightarrow{o})cos\theta_i$. Thus the integral will be

$$\int f(\overrightarrow{i}, \overrightarrow{o}) L_i(x, \overrightarrow{i}) cos\theta_i d_{\overrightarrow{i}} \approx \frac{1}{n}\sum_{i=0}^{n} f(\overrightarrow{i_i}, \overrightarrow{o}) L_i(x, \overrightarrow{i_i}) cos\theta_i / p(\overrightarrow{i_i})$$

For diffuse surfaces, we know that the BRDF is constant so it's enough to sample $cos\theta_i$ and so the sampled vector $\overrightarrow{T}$ on the sphere proportional to $cos\theta_i$ has coordinates $x = cos(2\pi r_1)\sqrt{1 - r_2}$, $y = sin(2\pi r_1)\sqrt{1 - r_2}$, and $z = \sqrt{r_2}$ where r_1 and r_2 are uniform random numbers between 0 and 1. Then we generate two vectors which are perpendicular to the normal i.e. the first can be the cross product of the normal vector with a random one, and the second is cross product of the normal vector with the first one. The below images are with diffuse sphere with number of rays =1, 10, 100, and 1000:

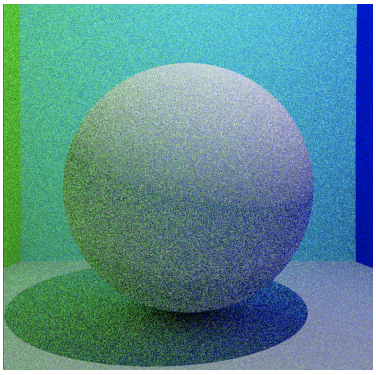

Figure 9: with n=1

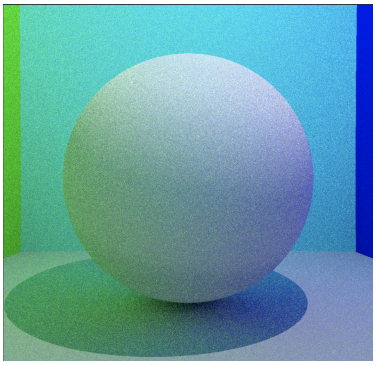

Figure 10: with n=10

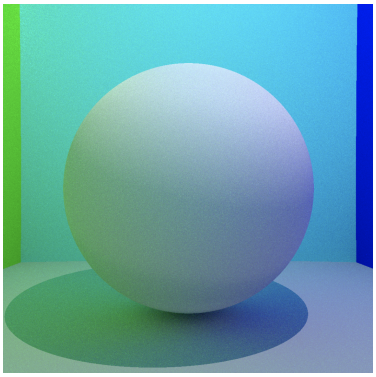

Figure 11: with n=100

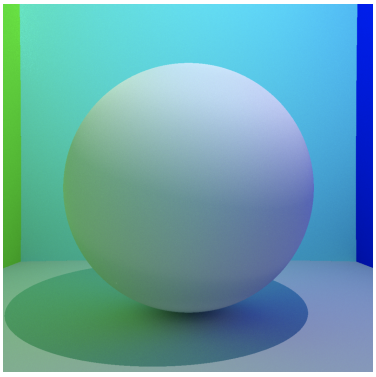

Figure 12: with n=1000

Obviously, as we are changing the number of rays (n) used, the execution time of the program will change. The execution time increases significantly as n increases, this is because we are repeating the same method (getColor) n times on each pixel and adding the values of the resulted color up so that we can color the pixel properly with less noise. As the images tell, increasing n leads to less noise and better images. The time it gets to give us the image with n=1, 10, and 100 is around 19.3407, 201.74, and 1982.81 seconds.

9 Conclusion

We presented the implementation of raytracing with the effects of shadow, reflection, refraction, and indirect lightning. All the information presented in this report are taken from the lectures given by [1].

References

[1] Nicolas Bonneel. "Introduction to Raytracing". In: (). URL: https://docs.google.com/document/d/1j6xgVlWWl_vhxOty8oMgWVnPam032EfYK77DayxXy_U/edit#.

[2] ANDREW S GLASSNER. "An Introduction to Ray Tracing". In: *The Morgan Kaufmann Series in Computer-Graphics* (2011). URL: https://www.realtimerendering.com/raytracing/An-Introduction-to-Ray-Tracing-The-Morgan-Kaufmann-Series-in-Computer-Graphics-.pdf.

[3] Eric Veach. "Reciprocity and Conservation Laws for General BSDF's". In: (Dec. 1997). URL: https://graphics.stanford.edu/papers/veach_thesis/chapter6.ps.