Projet 1 - Raytracer

Virgile Martin Bx23

May 7, 2023

1 Introduction

For this first project, I had to program a raytracer. A raytracer is a program that simulates the projection of light rays from a "camera" representing our observer position and calculates the interactions between the rays and the elements of the scene by applying more or less strictly the rules of physics. To do so, I based myself on the lecture notes and the code you provided during the tutorials.

2 First step: the classes

The first step of the project was to define classes for rays (Ray) as well as for spheres (Sphere), the first objects we worked with. A ray is simply represented by a half line, i.e. a point of origin O and a direction vector u. For spheres, we needed a center C and the length of the radius R. Moreover we added a vector "albedo" used to define the color of the sphere as a triplet RGB. Namely that each point is defined by a vector, the three coordinates in a plane in 3 dimensions. Once these classes have been defined even if they will be modified later, I defined the Scene class allowing to define the elements of the scene. A scene is only composed of a list of spheres, for the moment, composing our image, as well as the light intensity I and the place of the light source S. Then, I had to define a function to determine if a ray crosses a sphere. To do this we compute $\delta = \langle u, O-C\rangle^2 - \|O-C\|^2 + R^2$ and according to the value we get we can say what kind of interaction there is between the sphere and the ray. I simply followed page 20 of the reading notes for this. We also use this function to calculate the point of intersection P between the ray and the sphere, the unit normal N at P, as well as the distance t between O and P. Then we had to define a similar function but for the whole scene so that for each ray we return the sphere that has been touched by the ray, obviously the one whose point of intersection P is the closest, that is to say, among all the spheres that have been touched by the ray the one that has the lowest t. Now that we have defined our classes and we have functions to determine where the rays interact with the spheres, we need to know which color to display. To do this I created the function Color which returns the color of a pixel in the image. The first thing I did was simply to do

the following calculation:

$$L = \frac{I * albedo * max(\langle N, \omega_i \rangle, 0)}{4 * \pi^2 * d^2}$$

where $\omega_i = \frac{S-P}{\|S-P\|}$ et $d = \|S - P\|$, and return the result. We obtain Figure 1.

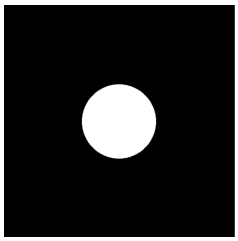

Figure 1: My first image

3 Second step: Light effects

Then we added shadows, which are based on the principle that there is a shadow ray starting from point P (with a small difference epsilon because of calculation error) and having the same direction as the light ray. It is then determined what it interacts with and this point of intersection then obtains the color black. We obtain Figure 2.

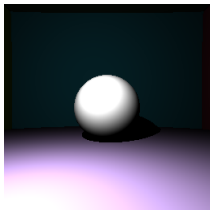

Figure 2: Sphere with shadow but no gamma correction. I = 2E7

Then, according to the reading notes, we must add what is called a gamma correction. It consists in taking the RGB value of each pixel and mounting it to the power $\frac{1}{2.2}$ while taking care that the calculated value is between 0 and 255. We obtain Figure 3.
Once this was done, we could tackle the mirror and transparent effects. To do this, I added a Boolean variable to my Sphere class, reflection and refraction, to know if the object has the property or not. We started with the mirror effect, as in the lecture notes. To do this, I modified the Color function. Instead of returning the color of the sphere, it returns the color of the object touched by a ray opposite to the light ray starting from P. Its direction vector is given by the following formula:

$$\omega_r = u–2 * \langle u, N \rangle * N$$

Thus, we return the result of the Color function when it takes as input $Ray(P, \omega_r)$. We obtain Figure 4.

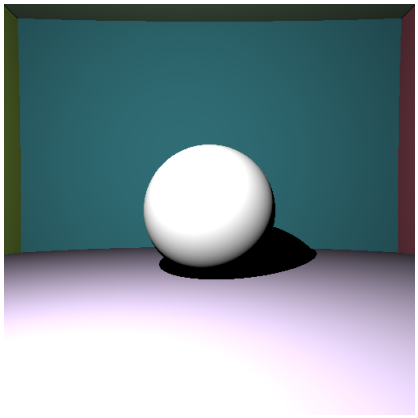

Figure 3: Sphere with shadow and gamma correction. I = 2E10

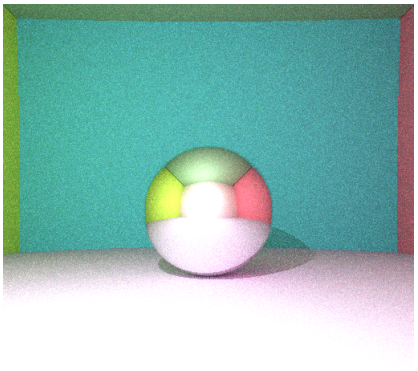

Figure 4: Sphere with reflective effects in 30ms. I = 2E10, 512×512

Concerning the transparent effect, it is a little more complicated because we have to take into account the refraction effect and therefore we have to add to all the objects a refraction index n_{index}. Moreover, I added the fact that when we compute the intersection, we keep the refractive index of the affected sphere. Initially, the refractive index is 1 (that of the air). First we check if the ray is leaving the sphere, in which case we reverse n_1 and n_2. Then, we calculate ω_t^T and ω_t^N according to the formulas of the course. If the condition mentioned on page 25 is not respected, we are in a case of reflection (the internal reflection) and not of refraction. Otherwise, we return the result of the function Color when it takes as input $Ray(P, \omega_t^T + \omega_T^N)$. We obtain Figure 5.
For more realism, we have added the Fresnel laws using a trick given in the lecture notes. We calculate the reflection coefficient for incidence R and we take a (uniform) random number a and if $a < R$ we launch a reflective ray otherwise a refractive ray. We obtain Figure 6.

The last light effect I added to obtain a realistic image was the indirect light. As mentioned in the lecture notes (page 32), I created a function random_cos which returns a vector V according to the Box-Muller formula. Then we compute a ray ray_ind starting from P and having a direction vector V. Then we add to the existing color the result of Color(ray_ind) times the albedo of the sphere.
Figure 7. is the combination of all the light effects we have implemented.

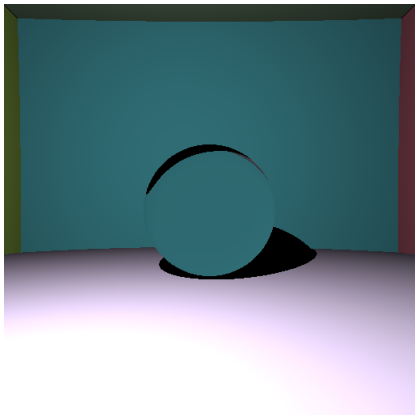

Figure 5: Sphere with refractive effects but no Fresnel law in 30ms. I = 2E10, 512 × 512

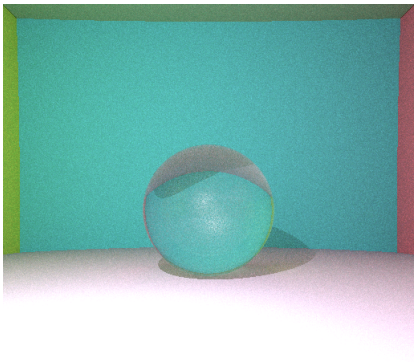

Figure 6: Sphere with refractive effects and Fresnel law in 30ms. I = 2E10, depth = 12, repetition = 100, 512 × 512

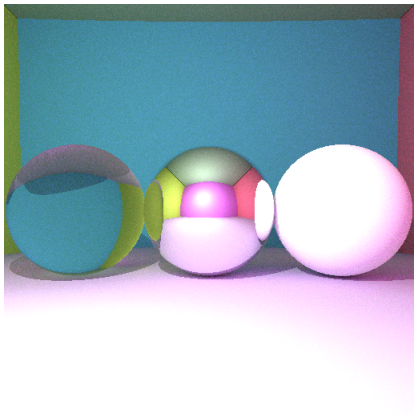

Figure 7: Three spheres with every effects in 2min. I = 2E10, 512 × 512

4 Improvement

Unfortunately I didn't manage to parallelize my program because I have a Mac and although I followed what is written in the lecture notes and followed tutorials on the internet, it didn't work.
Then, I did what is called antialiasing as indicated in the lecture notes so that the image would be smoother. All I had to do was change three lines of code.
To make our program more interesting to use, we had to add a way to add objects in our scene, other than spheres. For that, I implemented what we call Meshes simply by copying the code that the lecture notes give us. This code allows to import any shape with a ".obj" file. The imported objects are in the form of a multitude of triangles stuck

together.
As for the spheres, I had to implement a function to know if the light ray crosses a mesh. For that we first created a function to determine if the light ray passes through a triangle. Then, as for the function of intersection of the scene, we keep only the triangle whose ray crosses it and which is the closest to the light source. Thus we obtain the function to determine if a ray passes through a mesh. To finish with the meshes, I added (as mentioned in the lecture notes) a Geometry class, from which the Sphere and TriangleMesh classes inherit. And then in the Scene class, instead of having a list of Spheres, we have a list of pointers to Geometry. By modifying some slight details in the code so that there are no errors, we obtain a code that can model a scene with any kind of object. We obtain the Figure 8 and 9.

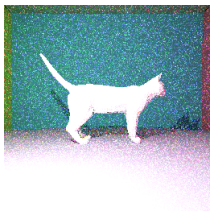

Figure 8: Cat in 15min. I = 2E10, depth = 1, repetition = 6, 256 × 256

Figure 9: Cat in 15min. I = 2E10, depth = 2, repetition = 6, 256 × 256

Finally, the code is long to be executed. To speed up the process, I implemented a BoundingBox class containing the minimum and maximum points of a box as well as an intersection function between a box and a radius. Finally I added a function to translate and scale our model by following a tutorial on the internet to understand what it was all about (the link is in the code), and I check if the ray passes through the box in the interaction function between the ray and the meshes. This makes the code faster. We get exactly the same image than in Figure 9. but more or less 3 times faster. But since it was faster I could increase the depth and we get Figure 10.

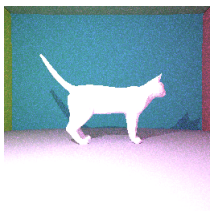

Figure 10: Cat in 1H10. I = 2E10, depth = 16, repetition = 10, 256 × 256