École Centrale Lyon

Rapport

Informatique Graphique

Élèves :
Letizia Licitra

Enseignant :
Nicolas Bonneel

22 mars 2024

Table des matières

1 Introduction **2**

2 BE 1 **2**
2.1 1.0 Intersection Ray-Sphere 2
2.2 1.1 Enlightenment Lambertian model 2
2.3 1.2 Scene 3

3 BE 2 **3**
3.1 2.1 Shadows 3
3.2 2.2 Mirror Surface 4
3.3 2.2 Transparent Surface 5

4 BE 3 **5**

5 BE 4 **6**
5.1 4.1 Antialiasing filter 6
5.2 4.2 Sweet Shadows 6
5.3 4.3 Depth of field 7

6 BE 5 **7**
6.1 5.1 8

7 Feedback **9**

1 Introduction

The objective of this project is to implement path tracing, a sophisticated rendering technique utilized for generating lifelike images by replicating the intricate behavior of light within a three-dimensional environment. Each BE will incorporate distinct functionalities to enhance the rendering process. C++ has been selected as the programming language for its efficiency and versatility in handling complex computations required for path tracing.

All code modifications can be found in the project's GitHub repository linked below :
https://github.com/letizialicitra/Informatique_Graphique

2 BE 1

2.1 1.0 Intersection Ray-Sphere

Firstly, I define a function named '**intersect**' within the class 'Sphere'.I compute the parameters of the quadratic equation, denoted as a, b, and c of the intersection between the ray and the Sphere. Then, I calculate the discriminant (Δ). If it is negative, the function returns false, indicating no intersection. Otherwise, I compute both solutions, but I consider only the one closest to the origin of the ray, namely t_2.
By iterating over all pixels in the pixel grid, casting rays, and checking for intersections, we can determine if any intersections occur. If an intersection is found, setting the pixel to white will produce the desired image.

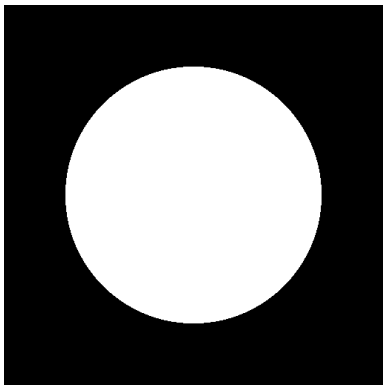

FIGURE 1

At the end of this section, we will obtain a white sphere on a black background. The color white is given cause of the rayons from the camera that have intersection with the area defined by the sphere.

2.2 1.1 Enlightenment Lambertian model

The Lambertian lighting model describes how surfaces reflect light uniformly in all directions, regardless of the observer's viewpoint.A Lambertian surface is one that exhibits diffuse reflection, scattering light equally in all directions.

To achieve this, it's crucial to compute the pixel intensity when an intersection occurs. This involves considering the albedo (representing the object's color) and the light intensity, initially set at 2×10^6. Additionally, we compute the diffuse reflection using the formula :

$$\text{std} : :\max\left(0, \frac{\langle \mathbf{N}, (\text{position\_light} - \mathbf{P}) \rangle}{\|\text{position\_light} - \mathbf{P}\|^2}\right)$$

where **N** is the surface normal, **P** is the intersection point, and position_light is the position of the light source. This formula ensures that negative values are clamped to zero, and it accounts for the angle between the surface normal and the vector from the intersection point to the light source position.

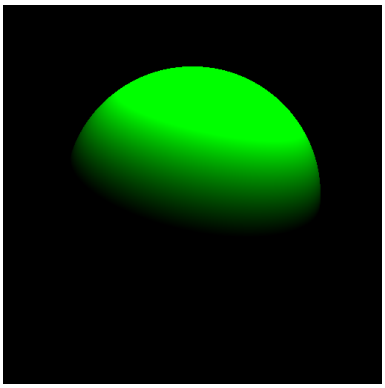

FIGURE 2

For further lighting computation, we also need to determine the unit normal **N** at P, which can be calculated as $\mathbf{N} = \frac{P - C}{\|P - C\|}$. In the **intersect** function of the `Sphere` class, I have introduced two additional vectors, **N** and **P**, passed by reference. This modification facilitates the computation of the normal vector **N**, defined as the difference between the intersection point **P** and the center of the sphere **C**.

2.3 1.2 Scene

In this section, we'll focus on constructing the **Scene**. This class contains a vector of Spheres and includes functions to add a Sphere to the Scene and to determine if there are intersections between each Sphere and the camera rays.
The function '**intersect**' of the class Scene iterates through each object in the scene using a for loop, checking for intersections with the camera rays. If an intersection is found, it stores the ID of the object if the intersection distance is less than previously found distances. This allows us to identify the object closest to the origin of the ray.
We also need to modify the main function by adding a Scene and different spheres to construct the walls of our scene. To create the walls, we utilize large spheres to ensure intersections between them.

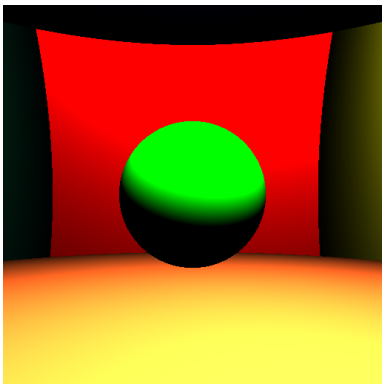

FIGURE 3

3 BE 2

3.1 2.1 Shadows

In this section, we will introduce **shadowing effects**. Specifically, when an intersection occurs between the light source and the Scene, we will cast another ray. If this ray intersects with a Sphere and the distance between the sphere and the ground floor is smaller than the distance between the light source and the ground floor, we will color that pixel black. These modifications can be observed in the main function, where I added the ray *ray_light*. If an intersection occurs and the distance *t_light* (from the ray to the light) is less than *d_light_squared* (calculated as (light − **P**).norm2()), the color will be set to black, resulting in a shadow effect.

Moreover, we can incorporate **gamma correction** to enhance the precision of color in-

tensity. This involves raising the RGB values (within a normalized range [0, 1]) to the power of $1/\gamma$, typically where $\gamma = 2.2$.
To achieve this, we utilize the function std : :pow(intensity_color$[x], 1/2.2$).

This is the result obtained : it is possible to notice the shadow beneath the central ball, extending onto the floor.

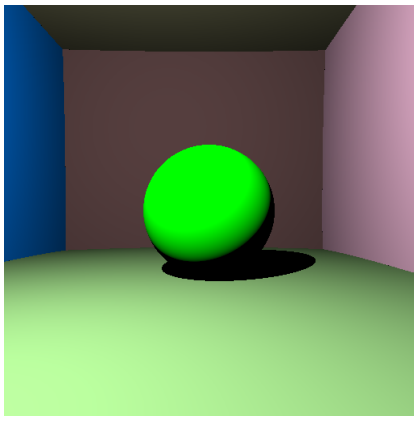

FIGURE 4

3.2 2.2 Mirror Surface

In order to implement reflection, I've developed a function called ***getColor***, which is recursively invoked in the main function. This function utilizes the parameter *number_of_rebonds* to determine the number of times the direction changes. Since it's impractical to compute this infinitely, we limit the number of rebonds.
Here's how the *getColor* function operates :

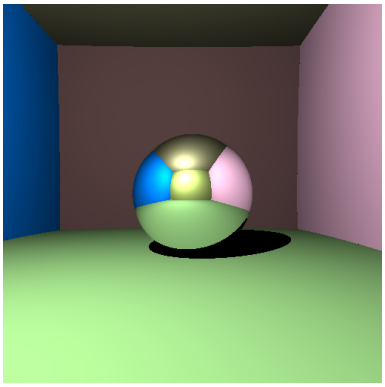

FIGURE 5

- At the beginning, there's an if statement to check if *number_of_rebonds* is equal to 0. In that case, the function returns the color black (0,0,0).
- When an intersection occurs, there's another if statement to determine if the object is a mirror.
- If the object is a mirror, the direction ω_i is computed using the formula $\omega_r - 2 \cdot \text{dot}(N, \omega_r) \cdot N$. Then, a new ray, *ray_mirror*, is created with this direction, and the *getColor* function is recursively called with the updated value of *number_of_rebonds* (decremented by 1).
- If the object is not a mirror, it falls under the case of diffuse surfaces (similar to the previous code).

The main function will be simplified : within the nested loop, the *getColor* function will be called only once with the chosen *number_of_rebonds*.

3.3 2.2 Transparent Surface

The modification to the code involves adding an if statement to handle the transparency effect inside the *getColor* function. If the object is transparent and has refractive indices n_1 and n_2, we compute the normal direction. It's important to note that if $n_1 > n_2$, the normal direction should be negated, so we take its absolute value. We then determine the tangential direction of refraction using the formula seen above. Subsequently, we create the refracted ray and call the *getColor* function again, reducing the *number_of_rebonds* parameter by 1.

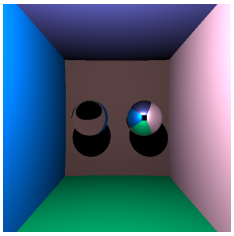

FIGURE 6

4 BE 3

In this section, we delve into the implementation of **indirect illumination** using the rendering equation and Monte Carlo integration alongside important sampling strategies.
Important modifications in the code :

In the function *getColor()*, I've added how to handle indirect illumination in this way :

- created two random numbers *r1* and *r2* ;
- generated a random direction in local coordinates and a random vector in space ;
- calculated two tangent vectors based on the normal and the random vector ;
- combined the random local direction with the tangent vectors to obtain a random direction ;
- created a ray with a slight offset from the intersection point along the normal direction ;
- accumulated the color contribution from the random ray by recursively calling *getColor* function.

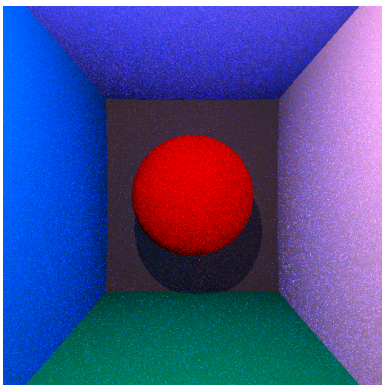

FIGURE 7 – A more diffuse and less sharp image.

In the main, I've specified the *number_of_rays*. Inside the loop over *number_of_rays*, multiple rays are cast from the same pixel position r, each with a slightly different direction. This introduces randomness in the sampling process, which is beneficial for reducing aliasing and noise in the final rendered image.

The color contribution from each of these rays is accumulated in the *color* variable. After casting all *number_of_rays* rays, the accumulated color is averaged by dividing by *number_of_rays*, effectively computing the average color value for that pixel.

By increasing the value of *number_of_rays*, you increase the number of samples taken per pixel, which generally leads to smoother and higher-quality rendered images. However, it also increases computational overhead, so there's often a trade-off between rendering speed and image quality.

5 BE 4

In this section we'll see how to add the antialiasing filter, soft shadows and depth-of-field.

5.1 4.1 Antialiasing filter

In order to address the aliasing issue and enhance the precision of the rendered images, an anti-aliasing filter has been implemented. This method involves adjusting the sampling process for each pixel by assigning weights to neighboring pixels.
The Box-Muller method is employed to generate random deviations (dx and dy) for each pixel. This method utilizes uniformly distributed random numbers to generate pairs of normally distributed random numbers, which are then used to displace the pixel position.
The generated deviations (dx and dy) are applied to adjust the pixel position (u) before casting the ray. This adjustment introduces random variations in the direction of the ray, effectively mitigating aliasing effects.

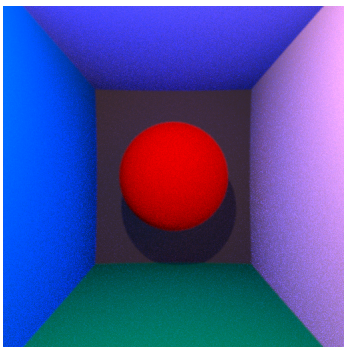

FIGURE 8

5.2 4.2 Sweet Shadows

First of all, we consider the light as a sphere that is part of the Scene. Inside the *getColor* function, if the sphere is the sphere of light, than we will return the color of the light divided by the area of the sphere.The result is an image full of noise (fig 9).
Using a bigger sphere for the light, it is possible to notice less noise in the image (fig 10) .

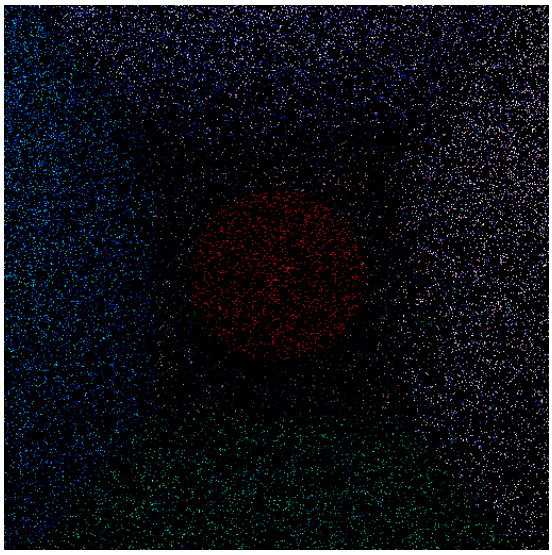

FIGURE 9

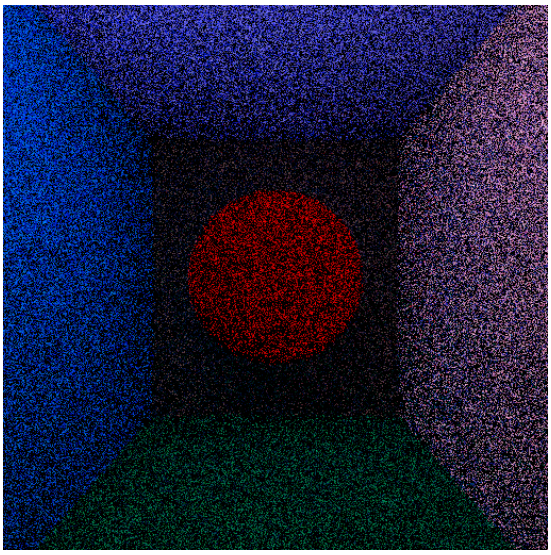

FIGURE 10

After that, we introduce the concept of soft shadows. We separate the emisphere of integration in two parts : the direct contribution is the one of the light source project, the rest of the hemisphere is for indirect. We re-parametrize the rendering equation via a

change of variable. Thus, we have to compute the determinant and the normal of the area patch. We create a new function called random_cos, that returns a random Vector that has more chances of being sampled around N than orthogonally to it. Inside the function *getColor*, we do other changes, mostly for the direct enlightement. Result : With respect to the previous one, it is possible to see less noise.

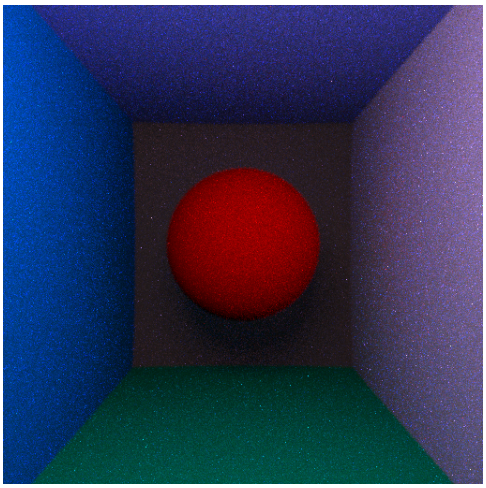

FIGURE 11

5.3 4.3 Depth of field

Due to the fact that the generated images are sharp at all distances, we will implement the depth of field : in this way all the points at the focus distance should describe a plane where points project to points on the sensor and remain sharp, and light passes through the aperture before reaching the lens. It is possible to notice that there is more deep in the image, that gives a certain sense of perspective.

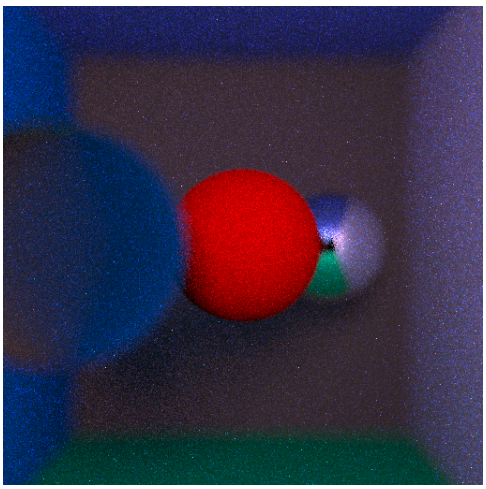

FIGURE 12

6 BE 5

In this section we will see the following steps :

- intersection ray-plan, ray triangle

6.1 5.1

In this first section our aim is to introduce the class 'Triangle' and perform the intersection ray-triangle. In order to make our objects as general as possible, we create also another Class, called 'Object', that contains the parameters that Triangle and Sphere have in common, in other words the parameter is_transparent, is_mirror, albedo. In order to compute the intersection between the traingle and the ray, due to the fact that the traingle is made of 3 points, it is necessary to compute the parameters of the matrices that represents the different distances between the points of the triangle and the intersection point. For this reason, we use Cramer methode, so we compute the determinant of the matrices to solve the system equations. At the end, in the main part, we also create and add the object t1 that represent our first Triangle.

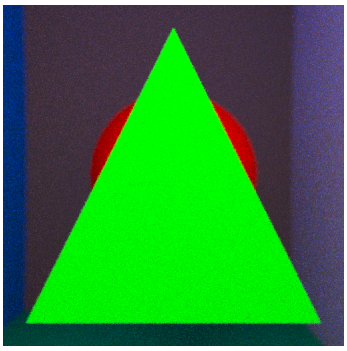

FIGURE 13

7 Feedback

The course is well organized into sections and contains many interesting topics, both in terms of programming and content (rendering). However, it can be quite complex at first to program in C++ using even complex geometry concepts. It might be useful to provide at the beginning guides to refer to for programming in C++ and also guides that can better explain geometry concepts. Also, one of the major problems is that if you have problems during one of the first BEs, it is challenging to align with the others and therefore it is necessary to rely on the videos you uploaded to the site.

Translated with DeepL.com (free version)