Introduction to Raytracing

by Nicolas Bonneel

Lecture № 1

Preliminaries

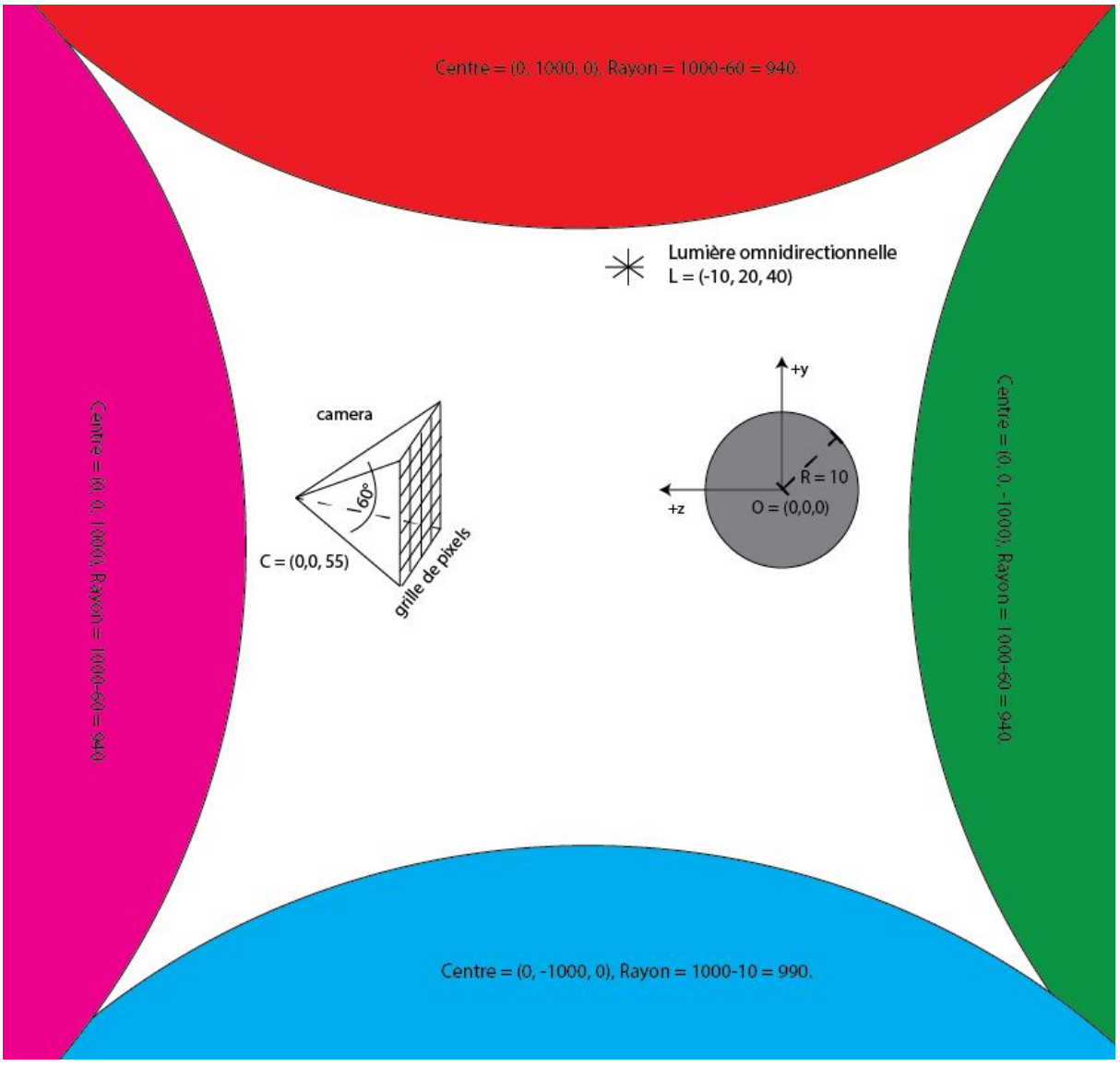
There are two major families of rendering methods. Methods based on "rasterization" involves projecting geometric shapes (e.g., triangles) onto the screen, like OpenGL. Methods based on ray tracing consist in simulating light paths (or reverse light paths) in the scene.

While rasterization methods are very effective, and often directly benefit from the graphics card, they are not very precise and physical effects (reflection, refraction, indirect lighting) are difficult to achieve. They are therefore mostly suitable for games, or interactive simulations.

In contrast, methods based on ray tracing are much slower, but simulate very well and quite easily all the light-matter interactions. They are more suited to the movie industry or for light simulation for architecture design, for which several hours of computation per frame are common (e.g., 47h per frame for Avatar, shared on computer farms).

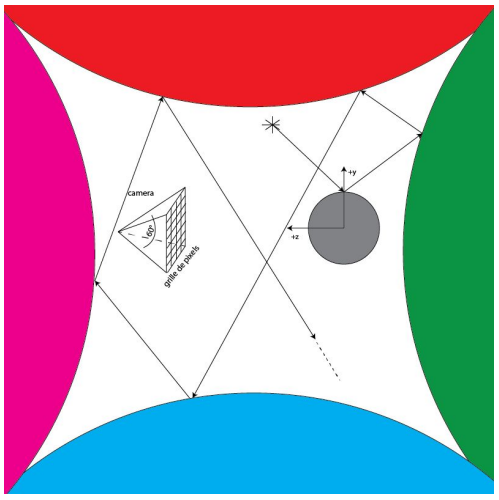
The purpose of this course is to teach you the basics in raytracing and to make you implement a simple raytracer manipulating primitives (spheres, planes, triangles, meshes). We will gradually improve this raytracer, including the effects of indirect lighting and other effects.

Note: we will consider the following scene, composed solely of spheres, which we will improve over time (several spheres, real planes or even triangulated geometry). The side walls are not shown for the sake of readability.

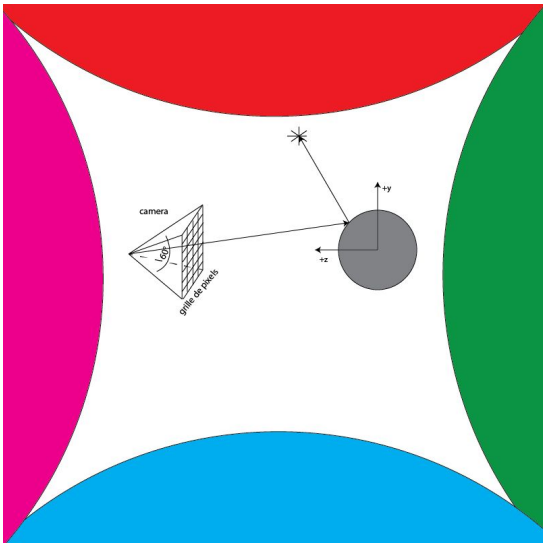

Centre = (0, 1000, 0), Rayon = 1000-60 = 940.
Lumière omnidirectionnelle
L = (-10, 20, 40)
camera
60°
C = (0,0, 55)
grille de pixels
+y
+z
R = 10
O = (0,0,0)
Centre = (0, 0, -1000), Rayon = 1000-60 = 940.
Centre = (0, 0, 1000), Rayon = 1000-60 = 940
Centre = (0, -1000, 0), Rayon = 1000-10 = 990.

Principle

The principle of raytracing is to simulate the behavior of light. If we opt for a strategy that sends light rays from all light sources, and simulate light-matter interactions, very few of these rays would reach the sensor (i.e., the camera) or only after a very long time as shown in the figure below. [Note: there are however mixed techniques such as photon mapping or bidirectional path tracing that do this; we will not consider these methods]



The (backward) raytracing approach we will study has a fundamental principle: the Helmholtz reciprocity principle. This principle says that one can follow the rays of light in the reversed direction - from the sensor to the light sources instead of from the light sources to the sensor - and get the exact same result. We'll emit light rays (half-lines), from the camera and towards each pixel of the image, and simulate light-matter interactions.



For the first course, we will consider a scene consisting of spheres, and we will assume all surfaces are diffuse (similarly to plaster): they reflect light in all directions, independently of the incident direction. Also, first, we will not consider the different light bounces within the scene, but only direct illumination (i.e., light received by a point light source directly, not the ambient light due to other interactions) and without cast shadows.

Getting Started

The only library that we will use is stb-image. In particular,stb_image_write.h is a simple C ++ header that will allow us to write images to disk (.bmp, png, tga, jpg, hdr).

Using stb-image, saving a .bmp image stored as a one-dimensional array of pixels is written:

```
  #define STB_IMAGE_WRITE_IMPLEMENTATION
  #define STBI_MSC_SECURE_CRT
  #include “stb_image_write.h”
…..
  stbi_write_bmp("fichier.bmp", width, height, 3, &pixel_array[0]);
```

stb-image expects to find in this 1D table a list of values corresponding to red, green, and blue components of each line of the picture interleaved. It also reverses the vertical direction. Access the red, green and blue components of the pixel in row i and column j is written:

```
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;
```

We will first define a Vector class, which is fundamental for 3D operations, which will allow to manipulate 3D coordinates easily. This class will contain the coordinates x, y and z in double precision, overloading the multiplication operator by a scalar, addition / subtraction of two vectors, the opposite vector (the "-" unary), and contain dot product routines, cross product and normalizations (as well as the norm, and the squared norm). If you are not familiar with operator overloading, implement these operations as functions.

We define a class Ray, which will represent a ray of light: an origin and a direction.
We define a class Sphere: an origin, and a radius.

Rays and intersections

The basic operations in a ray tracer are the generation of rays, and the calculation of intersections between a ray and a geometric primitive.

Given the configuration of our camera centered at C, visual fields "fov" 60 degrees, and a camera looking toward the negative z axis, generating a ray towards pixel (i, j) corresponds to generating a vector of direction:
V = (j-width / 2 + 0.5, i-height / 2 + 0.5, -height / (2 * tan (FOV / 2.)))
and then to normalize V ← V / norm(V), and take an origin which is the point C.
Some college-level trigonometry will give you why there is a factor -1 / (2 * tan (FOV / 2.)).
Make sure to convert values to radians to use the tan() function!

After generating a ray, one can then test if this ray intersects a sphere (start with a single sphere, and ignore the walls and floor).

For this, we see that the points of intersection between a ray and a sphere, if any, solve both the equation of the sphere, and the equation of the ray.

The set of points P on the sphere with center O and radius R is given by:

$$||P - O||^2 = R^2$$

The set of points P described by the ray of light is written as:

$$P = C + t.V$$

where V is the direction vector of the light beam, C its origin, and t a parameter along the ray. Although this is not strictly necessary, we always assume that the direction vectors (as well as normal objects) are unit vectors (of norm = 1).

The points satisfying these two equations are given by the set of t such that:

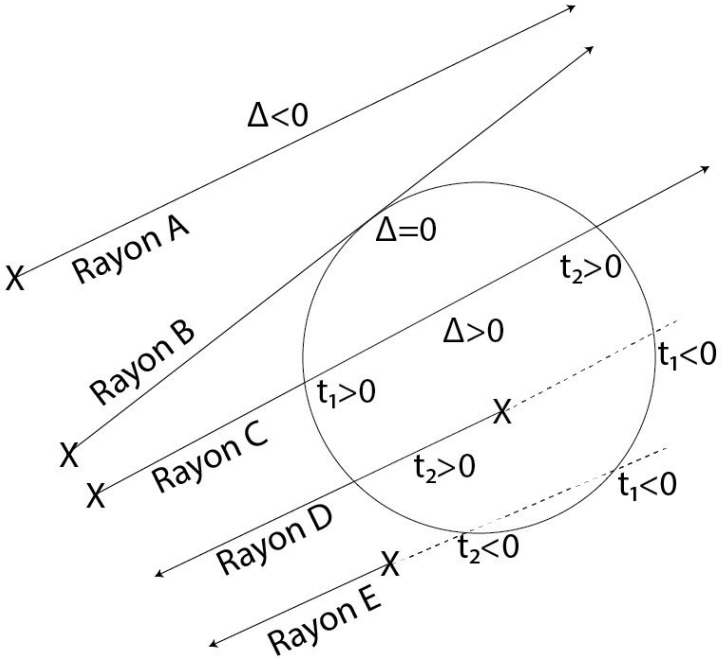
$$||C + t.V - O||^2 = R^2$$

Using the identity $||a + b||^2 = < a + b, a + b > = ||a||^2 + 2 < a, b > + ||b||^2$ we obtain the following polynomial in t :

$$t^2 + 2 < V, C - O > t + ||C - O||^2 - R^2 = 0$$

where I've considered that $||V||^2 = 1$

This is a polynomial of degree two, which may have 0 or two solutions (or two degenerate solutions in one), depending on its discriminant. Still considering the solution t_1 lower than the solution t_2 (e.g $t_1 = (-b - \sqrt{\Delta})/(2a)$ and $t_2 = (-b + \sqrt{\Delta})/(2a)$) we obtain the following cases:

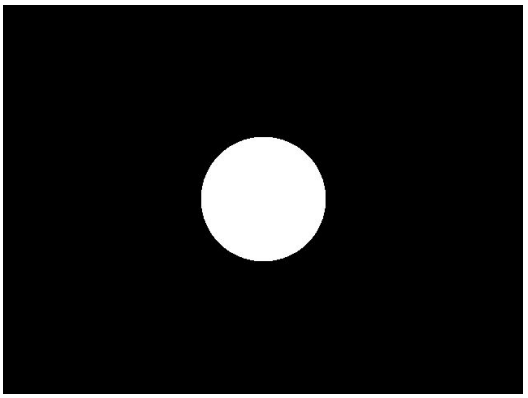


In this figure, ray A does not intersect the sphere because its discriminant is negative. Ray B intersects the sphere at one point because its discriminant is zero. Rays C, D and E intersect twice the sphere, and thus must return one of the intersection points. The origin of the ray C is outside of the sphere, and the two solutions are positive: the nearest intersection (t_1) is the one that interests us. However, ray D starts from the inside of the sphere: only t_2 is positive and of interest (the other intersection is "behind" the ray). The ray E starts outside the sphere and the two solutions are negative: this means that the two intersections are behind the camera, and so there is no intersection which are interesting to us: we consider that there is no intersection at all.
When there is one, we recovered the point of intersection $P = C + t.V$

Given these calculations, we are able to produce our first program whose algorithm is:

```
Define a sphere
For each pixel (i, j)
        Generate a ray toward pixel (i, j)
        If the ray intersects the sphere
                pixel (i, j) ← white
        else
                pixel (i, j) ← black
        End if
End
```

We obtain the following picture:



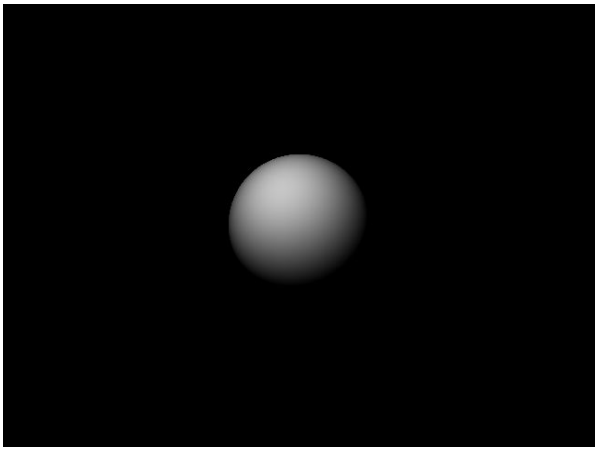
The light

This image looks flat. This is because of the lack of lighting. Considering the material as diffuse (i.e., like plaster), the intensity of the pixel is calculated as:

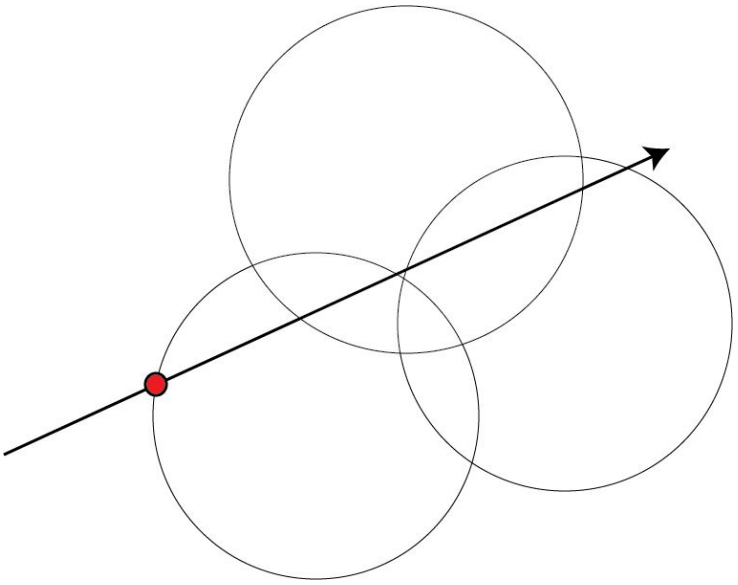
$$\text{pixel (i, j)} \leftarrow \max(0, \vec{l}.\vec{n})*I/(4\pi d^2)$$

where $\vec{l}$ is a unit vector from the intersection P and towards the light source L, $\vec{n}$ is normal to the sphere at the point intersection, I is the intensity of the light and d is the distance between the light and the intersection point.

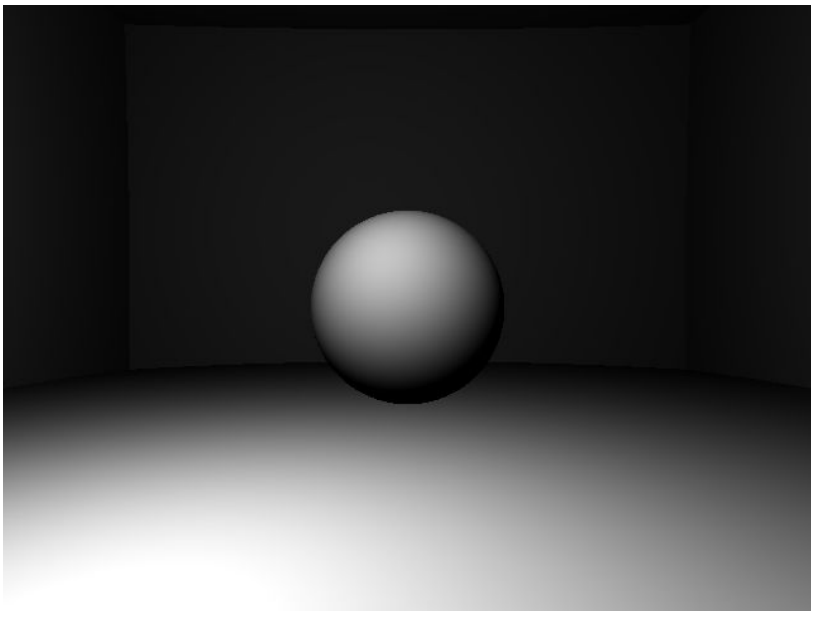
We then obtain:



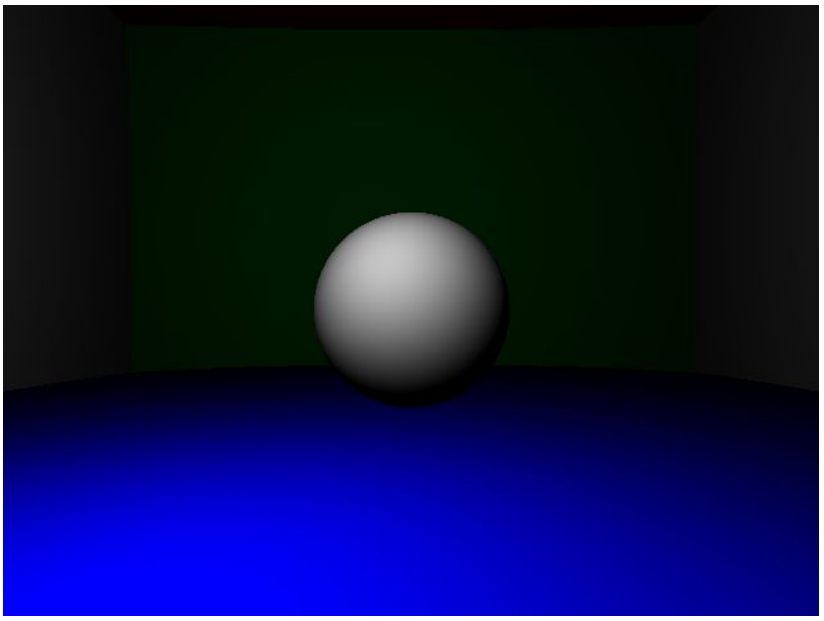
We will then try to handle multiple spheres. To do this, we will create a class "Scene", which contains an array of spheres (std :: vector <Sphere>). In the same way that the Sphere class had a method to intersect a ray of light, the Scene class will also have a similar method. The method to find the intersection between a ray and a Scene is defined as the intersection closest to the origin of the radius among all the intersections found with all surfaces of the scene. This intersection is shown in red in the following picture:



By doing this and using the introductory scene (using giant spheres to model the walls) we get the picture:



Also, we can adjust the red, green and blue components of each sphere (it is a simple multiplicative factor between 0 and 1 for each component: the "diffuse color"), we can easily color our scene. For this, a Material class will be defined, which will contain a diffuse color (a Vector, for example), and that will be assigned to each sphere. We thus obtain:



... And that concludes the first class.