Ray tracing

By Jorge Bárcena

1.- INTRODUCTION AND SOURCES

In this document I will explain how my program works using the ray tracer method. This document will be updated every week, so there will be changes during each week. To carry out this project, I have used several resources:

- Ray tracing in one week, to have a main idea of how ray tracing works and where I could start working. This book has helped me to create a .ppm file from the code, which can show my image.

- Fundamentals of computer graphic, this book is helping me a lot to organize my program, since it gives you a structure of how this project can be organized, which I am applying to my project. Also this book contains a lot of information about matrices, and how to use them correctly that is helping me a lot to complete the math library.

- On the internet I found this page (https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection), this page has helped me a lot , when it comes to understanding how the intersection of the ray works with the objects of the scene (plane and sphere).

1.1. How the ray tracing works

What raytracing try to do is to simulate the human way of perceiving reality. This technology is oriented to render images in 3D, in many cases approaching photorealism.

But, how does this technology work? First of all we need a point of view from which the scene will be rendered, this point would coincide in real life with the human eye, since from this point the vision rays will be launched.
In the second place we will need a point of light that illuminates the scene (If there is no light the image will be black).
Finally we need a scene with objects so that these are rendered in the generated image.
Once we have all these components we can explain how this technology works. From the point of origin (eye), a ray is launched, this ray test an intersection with an object of the scene, between all the intersected objects, the algorithm detect the nearest object to the origin, once the program have identified the object closest to the origin, the program can calculate the lights at the point of intersection. the color is obtained through the material of the object and the properties of the light. This color, in my case, is from the RGB form, this is so, because then, the image I generate is .ppm, this type of files saves the color of the RGB form in each pixel of the image starting from top to bottom , and left to right. In the case that the material of the object has translucent or reflective properties, the program will need more rays to see the more distant objects.
Regarding the perspective, this is generated automatically, since the ray tracing process tries to simulate the human seeing method. So it is not necessary to apply any method that distorts the perspective.
This process will be carried out for each pixel of the image that we want to obtain, it is a slow process but it allows us to obtain high quality images.

2.- HOW IS THE PROGRAM ORGANIZED?

Currently my program have 10 classes, plus a file called "ray_tracer.cpp", which is what is executed when the program is launched. The project in the visual studio is a console project, but at the end of the program, an image file is created (in the directory where the program is). The classes are the following:

1. Camera.
2. ImageFile.
3. Ray.
4. Light.
5. Render.
6. Scene
7. ObjectToTrace.
8. Sphere.
9. Plane.
10. Material.
11. Ray tracing (main).

2.1.- Camera Class

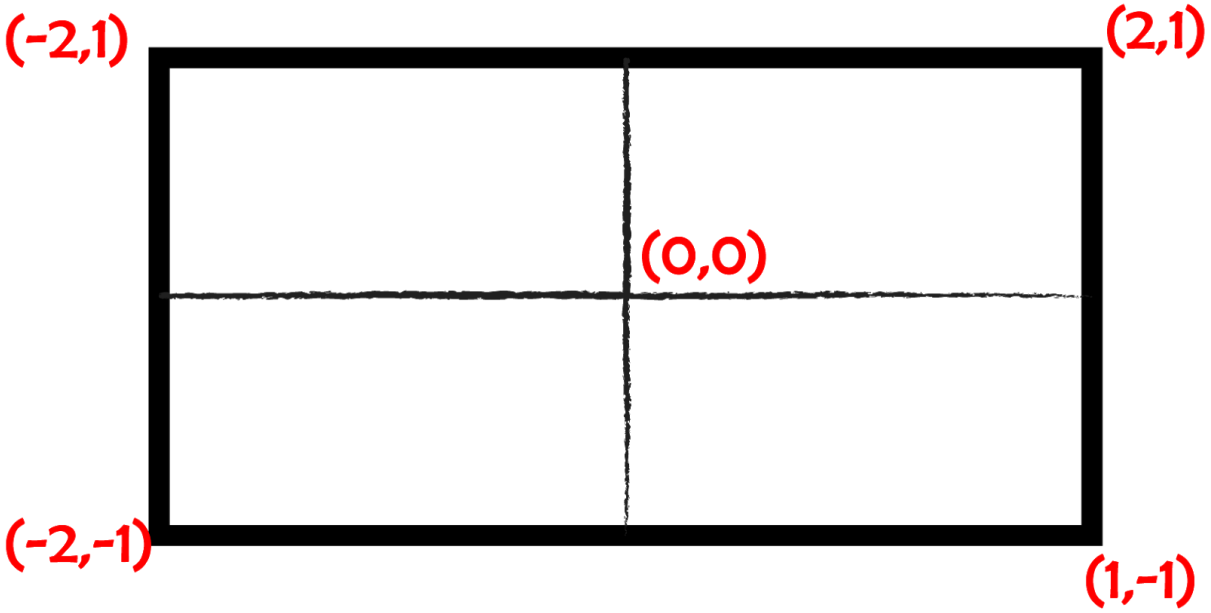
The function of this class is set the camera in one point of the world, to render all the objects that are inside the field of view that it has. This class has a header file that contains the methods and the attributes of the camera. Also, this file include the "Ray.h" and the "MathLibrary.h", because are used inside the class. Its attributes are the following:

- vec3 origin; → The origin of the camera.
- vec3 lowerLeftCorner; → The lower left corner of the camera.
- vec3 horizontal; → The width of the camera.
- vec3 vertical; → The height of the camera.

Regarding the methods are the following:

- Ray getRay(float x, float y); → Returns a ray that goes from the camera origin and with the direction obtained of the x and y coordinates.

- Camera (vec3 _origin); → It is the constructor that establishes the origin attribute and completes all the remaining attributes depending on the origin.



* That is how the camera is set in my project. The coordinates are the X and the Y.

2.2.- ImageFile Class

This class will be responsible for managing the image file that will be created what the execution of the program has finished. For this I have used a C ++ library called <fstream>, which allows you to open files. This class has a header file (ImageFile.h) that contains the definition of its attributes and its methods, also are included the libraries of "iostream" and "fstream", in addition is using the namespace "std". With this it is possible to generate a file that is interpreted as an image. The type of file that I decided to use to generate an image at the end of the execution is the ".ppm", which I chose because the .ppm files are easy to generate since they are composed of color (in RGB format) of each pixel. The .ppm files store the colors of the pixels from bottom to top and from left to right, adding a line break between each color. The format of this file would be something like this:

255 124 066
254 058 235
255 255 255
...

Its attributes are the following:

- const char* myName; → Save the name that we want the file have. Is a const char* variable, because the <fstream> library can not used with the std::string.
- fstream myFile; → Contain the file that we are modifying.

Regarding the methods are the following:

- void openImage(const char* name, string mode); → Open the file in 3 different modes (read, write, add).
- void closeImage(); → Close the file. It is necessary at the end of the program.
- void addText(const char* text); → Add the given text to the file.
- void addText(string text); → This function cast the string to a const char*, and add the text to the file.

2.3.- Ray Class

The function of this class is throw a Ray from the origin in one direction.This class has a header file (Ray.h) that contains the definition of its attributes and its methods, also are included the library of "MathLibrary.h". Its attributes are the following:

- vec3 origin; ⟶ The origin of the ray (vector3).
- vec3 direction; ⟶ The direction of the ray (vector3).

Regarding the methods are the following:

- inline vec3 point_at_parametrer(float t); ⟶ Return the vec3 point, in the ‘t’ length of the ray. This method is declared **inline**. This means that the code of this function is in the header file, not in the .cpp. The key word "inline" does that the function does not run as a normal function, when the compiler finds the word "inline", copy and paste the code of the function where it is called, this makes the program does not jump to the memory address where the function is saved and continues to execute the function without changing the memory address. This improve the performance of the program

2.4.- Light class

The function of this class is to set a point of light in the world. Also are included the libraries of "MathLibrary.h" and “Ray.h”. Its attributes are the following:

- Vec3 lightCenter; ⟶ Saves the position of the light in the world.
- float lightIntensity; ⟶ Saves the light intensity that we want to apply to our scene.

2.5.- Render class

The function of this class is to perform the precise calculations to determine the color of each pixel. During the execution of the program, the method of the class (vec3 getColor ()) will be called by each pixel of the image, and will return a vector 3, which will be a color in RGB format. This class has a header file (Render.h) that contains the definition of its attributes and its methods, also are included the libraries of "MathLibrary.h", “Ray.h”, “Scene.h” and “std::vector”. Its attributes are the following:

- Scene* myScene; ⟶ Saves the scene that the program is going to render. Regarding why I used pointers to store objects, it's because the objects I'm going to use have already been created before, so it would be a waste of memory, create new variables while the same objects are already in memory.

Regarding the methods are the following:

- vec3 getColor(const Ray& ray, hit_record &record, int depth = 0); → First, let's talk about the parameters requested by the function. First of all, the **Ray**, I have decided that it is constant because it will not be modified within the function, doing this that I protect the ray of some future change. The reference is because we want to use the ray that was previously created and not create another one. The second parameter is **hit_record** &, hit_record is a structure that saves the normal vector, the point of intersection between the ray and an object, and the point of the ray where the intersection occurs. We'll talk later about it. It is a reference, since we want to use the hit_record created previously, since it is a variable that we will only use once within the function, we should not do create a new variable each time we call the function.Respect for the third parameter is a default argument, this means that it is not necessary to pass by an argument the value of depth, in case the call of the function do not have any value, the compiler will set "**depth**" with the value that appears in it declaration , in this case 0 , in the case of that value, is not empty in it call of the function, the "depth" will be the corresponding value. This parameter is available to control the amount of rays that are thrown in the case of reflections, setting a maximum of 5.
  What this function do? In summary what my function does is this:

```
If (depth < 5) then
        For each object do
                If (hit()) then
                        color = object.color
                        hit_record = object.hit_record
                        If ( hit_record.point < currentPoint) then

                                If (setProyectedShadow())
                                        color = black

                                Color = Iluminate_pixel()
                                Apply material (reflection, reflexion)

                        Return color
                else
                        Return backgroundColor

else return blackgroundColor
```

- bool setProyectedShadow(const Ray & ray); → This function is used to determine if in a position that is in the argument hit_record must have a projected shadow. In case the ray "**Ray**" makes an intersection with some object, the function will return the value of true, otherwise it will return false.
- void iluminateObject(double angle, vec3 & color); → The purpose of this function is to determine the degree of illumination of the **color**, its second argument, depending on the parameter "**angle**". The argument of the color is passed by reference, because we want to change the value outside the function, in this case, the color of each pixel.
- Ray getShadowRay(const hit_record & record); → The purpose of this function is to return a ray from the point where an intersection has taken place, to the light of the scene. This ray will determine, later, if that color will be a projected shadow or not. To avoid the problem of "self Intersection" a small modification is applied to the point from which the ray comes out. We move that point in relation to the normal of that object, thus avoiding the problem of "self Intersection".

2.6.- Scene class

The function of this class is to save all the object that our class “render” is going to render. This class saves elements relative to the scene like the light. Also are included the libraries of "MathLibrary.h", “Ray.h”, “Sphere.h”, “Plane.h”, “objectToTrace.h”, “light.h” and “std::vector”. Its attributes are the following:

- std::vector <objectToTrace*> objectsToTrace; → Saves all the objects that can be drawn. I decided to use a std :: vector, because it is more useful, in this case, than an array, and I need to know the length of the vector repeatedly. I've also decided because at the beginning of the execution of the program I can add objects in a random way, with a matrix, I would be limited to its original size.Regarding why I used pointers to store objects, it's because the objects I'm going to use have already been created before, so it would be a waste of memory, create new variables while the same objects are already in memory.
- Light* light; →Save a light type object. This attribute is created to set the illumination of each object to calculate the color of each pixel, in the getColor () function. The type of pointer was chosen for the same reason as the previous attribute, so as not to waste memory and take advantage of the objects that we have already created.
- Const vec3 backgroundColor; → That is the default color, if the class “render” do not intersect with any object of the scene.

2.7.- The ObjectToTrace Class

This is an abstract class. What is an abstract class? An abstract class is that class that works as base of others. To create an abstract class in C ++ we must place a pure virtual function, in order to create these functions in C ++ it must be done following this form:

Virtual *typeOfReturnValue* NameOfTheFunction() = 0;

Why I decided to use an abstract class? This is because in my program there are elements that are going to render (plane, cube, sphere ...), these elements have some methods and attributes equals. All the objects that are going to render in the scene should inherit from this class. The attributes that will have in common and therefore, will have this class, will be the following:

- Material myMaterial; → This attribute saves the formation of the material of each object to render in the scene, something indispensable to proceed with the rendering. The Material class will explain later, but briefly, it includes information about the properties of that object (refraction, reflection ...).

Regarding the virtual methods that has this class, there is one, and it is a method that all objects in the scene should have. This method is the following:

- virtual bool hit(const Ray& ray, hit_record& record) const = 0; → What is the function of this method? This method is responsible for determining if an object has an intersection with the ray "**ray**", and store the information of the intersection in the variable "**record**". The ray object is const because we should not modify any value of the ray, and it is called by reference because we are going to use other ray previously created. The variable record has a reference because we want to use the values of this struct, outside the function. Why is this method virtual? This is because not all objects have the same mode of intersection, that is, the plane will have another mode that the sphere to determine if there is an intersection; but all the objects in the scene must have an intersection method. For all this I have decided to use a pure virtual function.

2.7.1.- hit_record struct

This structure is created in order to store the information of an intersection between a ray and an object. This struct contains three variables:

- Double pointAtRay; →This variable is responsible for saving the point of the ray in which the intersection occurs. This variable will be used, later, in the Render class, to

determine if one object is in front of another, the smallest value of this variable will determine that the object is closer than the origin of the ray, therefore ahead.

- Vec3 pointInWorld; → This variable is responsible for saving the position of the world where the intersection between the object and the ray occurs. This will be useful to create refractions and reflections.
- Vec3 Normal; → This variable is responsible for saving the normal of the point of intersection between the ray and the object, with respect to the object. Depending of each object (plane, sphere ...) the way to obtain this value will be different, because of this, this variable will be passed by reference to the function "hit ()", and inside the function "hit ()" will be established these variables, to make use of them, later, outside the function.

This is an abstract class, it will not contain the definition of any of its functions. So this class there are only one file "ObjectToTrace.h", there is not "ObjectToTrace.cpp" because it is not necessary.

2.8.- The Sphere Class

This class is referred to all the object of type Sphere, this class inherits from the parent class ObjectToTrace. This class is responsible for storing the "sphere" objects that will be represented in our scene. This class is composed of two files, first "sphere.h", where all the attributes of the class and the methods are declared, in this file are also included the libraries "MathLibrary.h", "Ray.h" and "ObjectToTrace.h".The second file is "Sphere.cpp" where the definitions of the previously declared methods are found. This class has a series of own elements of a sphere, these elements are stored by variables. These variables are the following.

- vec3 centerSphere; → Save the center of the sphere. This point will be the center of the scene with respect to the world position.
- Float radius; → The radius of the sphere. This radius determines how big our sphere is in the scene.

Regarding the methods declared in this class, are the following:

- bool hit(const Ray& ray, hit_record& record) const; → This method overwrites the hit () method of the parent class. Before going on to explain that this method we are going to make a series of comments about the function. As we can see the function contains the keyword "**const**", at the end of the declaration of the function. The function of "const" here is to protect that inside the function the values of the attributes of the class not change, because in this function they should not be changed. In this case the keyword "const" have to be used, because in his parent class, we defined the hit() function as "const" too. How the function determine if there is an intersection between the ray and the object? This is done by the analytical solution.
  The equation of a sphere is the following:

$$x^2 + y^2 + z^2 = R^2$$
$$x^2 + y^2 + z^2 = P^2$$
$$P^2 - R^2 = 0$$
$$P^2 = (Origin + t * direction)^2$$
$$(Origin + t * direction)^2 - R^2 = 0$$
$$Origin^2 + (t * direction)^2 + 2 * Origin * Direction * t - R^2 * Origin = 0$$

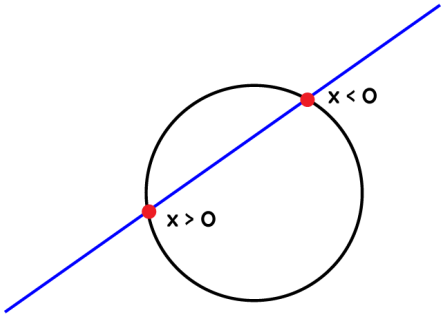
  We will have to attend to the discriminant of the second degree equation. A second degree equation is written like this:

$$x = (-b +/- \sqrt{b^2 - 4ac})/ 2a$$

  The parameters that define if the value of x is positive or negative will be:

$$b^2 - 4ac$$

  This is what we have called discriminant and what will define the point of intersection.
  This is the model of a second degree equation, the resolution of this equation will determine if there is a collision between the ray and the sphere. But there may be two solutions, one positive and one negative. We will take the positive into account, because the negative determines what is a point on the hidden face of the sphere.



2.9.- Plane Class

This class is referred to all the object of type Plane, this class inherits from the parent class ObjectToTrace. This class is responsible for storing the "plane" objects that will be represented in our scene. This class is composed of two files, first "plane.h", where all the attributes of the class and the methods are declared, in this file are also included the libraries "MathLibrary.h", "Ray.h" and "ObjectToTrace.h".The second file is "plane.cpp" where the definitions of the previously declared methods are found. This class has a series of own elements of a plane, these elements are stored by variables. These variables are the following.

- vec3 normal; ⟶ Save the normal vector of the plane. This attribute is necessary because to calculate the intersection of the plane with the ray, we need to know it.
- Float offset; ⟶ Save a point of the plane. To determine the intersection with a plane, we must know a point of the plane, to find this point, what I do in the program is multiply the normal by this variable. So this variable determines where the plane located on the scene will be.

Regarding the methods are the following:

- bool hit(const Ray& ray, hit_record& record) const; ⟶ This method overwrites the hit () method of the parent class. The function of the keyword “const” is explained before, in the sphere section. How the function determine if there is an intersection between the ray and the plane? This is done by the analytical solution.
  We know (by geometry), that the dot of 2 vectors which are perpendicular to each other is equal to 0.

$$A * B = 0$$

A plane can be defined as a point representing how far the plane is from the world origin (p0) and the normal (orientation). To obtain a vector that lives inside the plane we must subtract this point, and that point must be perpendicular to normal.

$$(p - p0) * N = 0$$
$$p = rayOrigin + rayDirection * t$$
$$(rayOrigin + t * rayDirection - p0) * N = 0$$
$$t = ((p0 - rayOrigin) * N) \div rayDirection * N$$

In case "t" is greater than 0, it means that there is an intersection with the plane, otherwise this intersection did not occur.

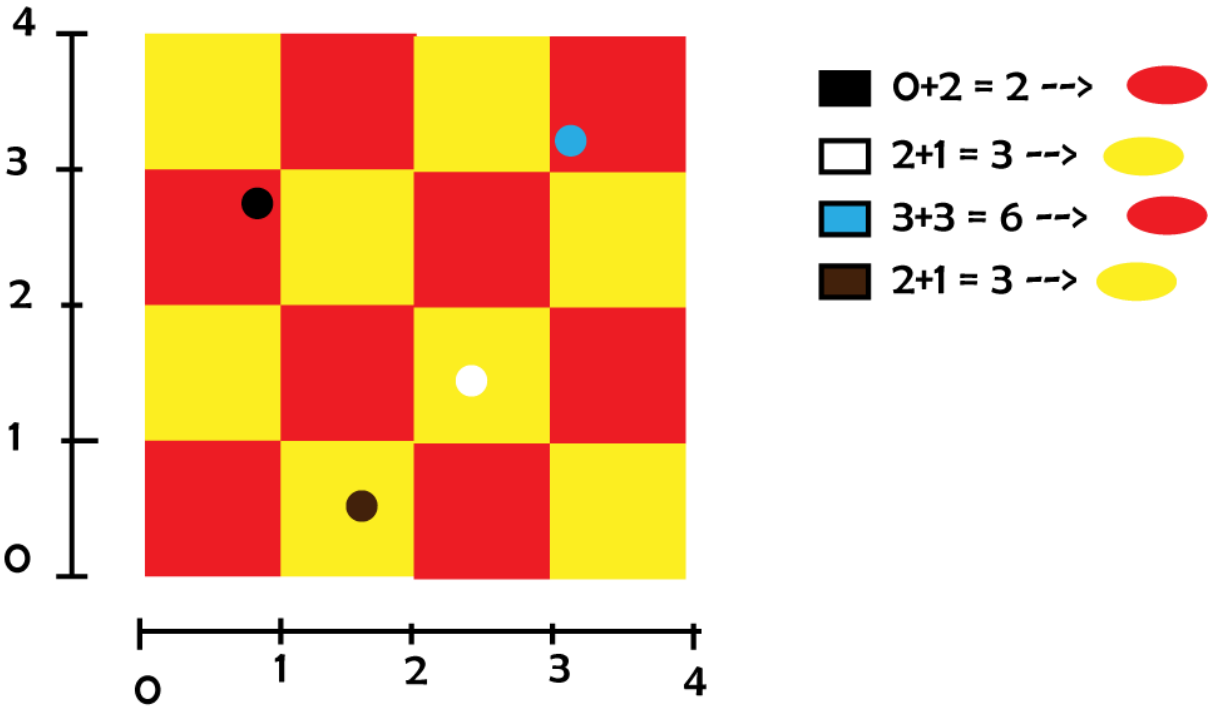
2.10.- Material Class

Esta clase está utilizada en la clase “objectToTrace”. The function of this class is to establish the physical characteristics of each object, regardless of its form. This class has a header file (Material.h) where all the attributes and methods of the class are declared, the library "mathLibrary.h" is also included. In the file "material.cpp", the getColor () function is defined. Currently there are several attributes that make up the final material of each object.

- bool texture; ⟶ A boolean that determines if the material has any texture or not..
- vec3 color; ⟶ The base color of the texture.
- float reflection = 0.0f; ⟶ Amount of reflection.
- float refraction = 0.0f; ⟶ Amount of refraction.

Regarding the methods are the following:

- vec3 getColor(double x, double z) ⟶ This function determines the texture that we have put in the plane (grid). To make this function work, the characteristics of type conversion of variables in C++ were used. First, I convert the variables x and z (double type) to int, this conversion eliminates the decimal part of the variable and only keeps the whole, no approximations are made. By this method it can be said, while the sum of the two positions is not divisible by two, one color is painted and in the opposite case of another. In this image you can see how the algorithm works.



2.11.- Ray tracing (main)

This is where the main structure of the program runs. Next I will make a summary of how this program works, which is very similar to how the ray tracing algorithm works.

```
for (int i = 0; i < yResolution; i++)
        for(int j = 0; j < xResolution; j++)
                for(int p = 0; p < antialsing; p++)
                        Hit_record info
                        Ray = camera->getRay(i,j)
                        currentColor = render.getColor(ray,info)

                currentColor /= antialsing;
                My output file += currentColor

Close file
```

This file, contains two functions too. The first one is to create a basic scene, a basic scene is the plane with three spheres. The other function is to create a random scene with spheres along the plane,this function has a Default parameter “seed”, this argument determine if the seed, to create the randoms spheres, is random, or is chosen by the user.

2.12.- UML diagram