# CSE306 Raytracer

Alexis Roger

May 2020

## 1. Introduction

This project is aimed at implementing an integral ray tracer in C++ without libraries. Implementing the raytracer was done in two steps: the first was to understand the key concepts with spheres and to write most of the specificities of the code at it is faster to compute than meshes. In a second time we will then move onto rendering meshes. We will look at each of these functionalities in detail as we go.

For reference, all runtimes given here, unless indicated otherwise, where achieved running the code in parallel on a laptop with an Intel i7-4700MQ CPU (4 cores at 2.4GHz).

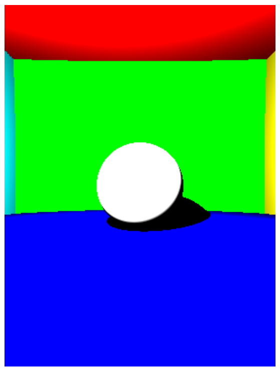
The GitHub repository with the code of the project may be found below. The 'master' branch is the code for the spheres and the 'cat' branch is the code for the meshes.
https://github.com/Alexis-BX/CSE306-Raytracer

## 2. General code structure

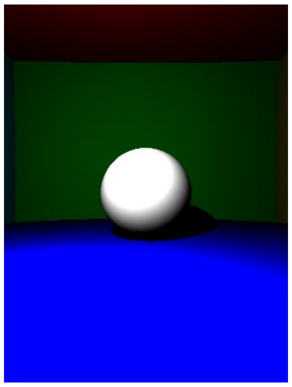
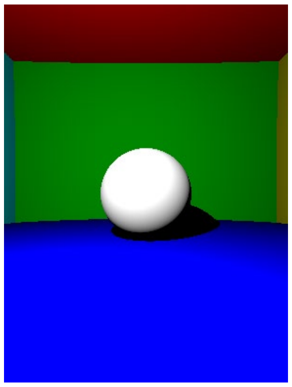
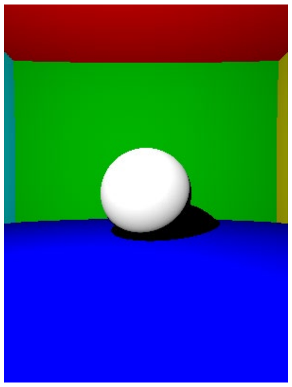
The code is organized in a linear fashion. To begin, all external imports and constants are in 'master.cpp'. This file is then imported by 'vector.cpp' which defines the Vector class and all of its overloaded operators. This is then imported by 'objects.cpp' which generates the different objects that will be in the scene, such as the Sphere, Ray, Camera, Light. The spheres store not only their color position and space but also an integer representing their material (0=opaque, 1=mirror, 2=transparent). This is then imported by 'helper.cpp' which has all of the important functions, mainly the getColor function and all it calls. Finally this is included in 'main.cpp' which initializes and launches the rendering.

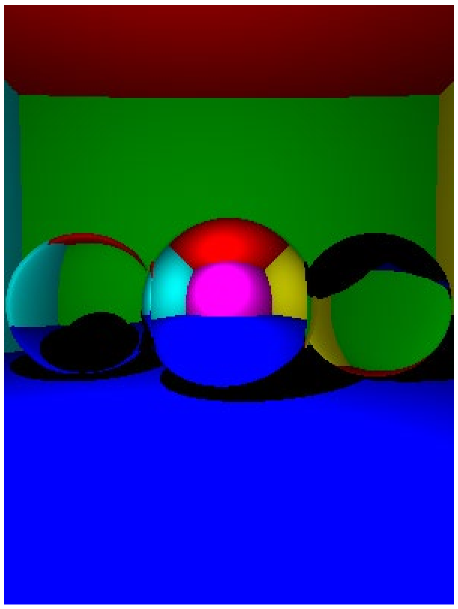
## 3. Spheres

To begin we create a world entirely with spheres. And we start with the most basic case: homogeneous spheres where the color is only visible if the light can shine directly on it. We can see that the colors are plain and that the shadows are simply pitch black circles. Compute time: instantaneous.



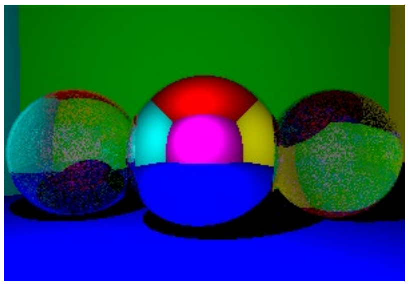
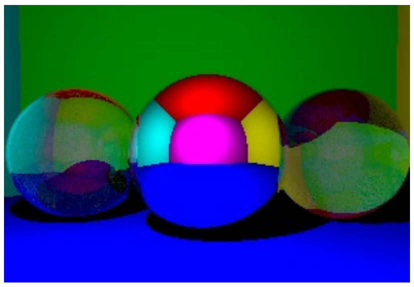
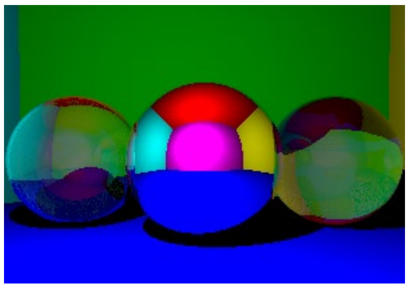
We then implemented gamma correcting to smooth out the image and to spread the light more evenly on the spectrum. Different values of gamma where tested and the results can be seen bellow. For all future images we stayed with the standard of a gamma at 2.2 with a light intensity of 100'000. From left to right we have gamma of 1, gamma 2.2 and gamma of 4. Pictures with gamma between 2 and 3 definitely seem like the most natural. Compute time: instantaneous.

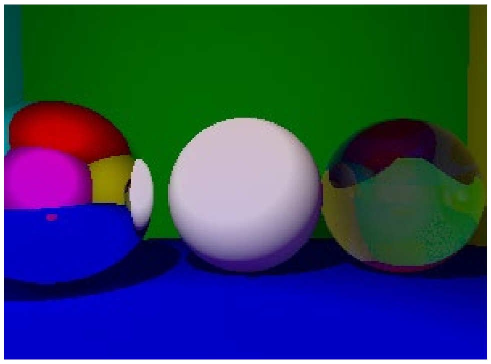






We then proceeded to adding some material diversity to our spheres. We created mirror, translucent and hollow spheres. For the mirror spheres we just reflect the incoming rays at the same angle relative to the normal. For translucent spheres we use the Snell Descartes law to calculate the refracted ray's direction. For these transparent spheres, as they have no color, we use them to store the refraction indices. We assumed the air had an index of 1 and gave the spheres the index of glass, 1.5. As they do not have any color we use the color parameter to store these indices. To create hollow spheres we created 2 transparent spheres with same center point but slightly different radius. We also flipped the refraction indices of the inner sphere. With a ray depth of 5 we get the following, with the hollow sphere on the left. Acting like a lens, the fully transparent one flips the image upside down while the shell maintains it. We see no difference of a path depth over 5 at this point and compute time: instantaneous.

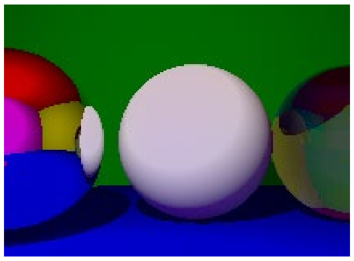
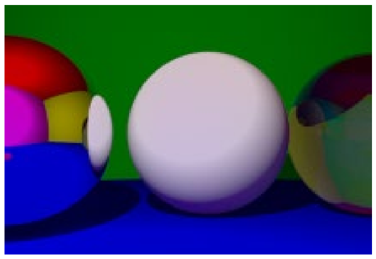
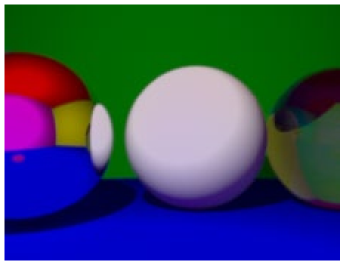
In the real world, refractions are not perfect. Part of the incoming ray is reflected on the surface of hollow spheres instead of transmitting through. It would be too costly to pursue every path so we use the Fresnel law in order to randomize the path we follow. We therefore need to start running our simulation multiple times to average out the results and get a smoother image. With this we can start seeing a faded trace of the pink background in the middle of the transparent balls.  From left to right we have 10, 100 and 1000 rays per pixel. The run time is: instantaneous, a few seconds, 20 seconds. Having more rays really gives the balls the glassy look we want.

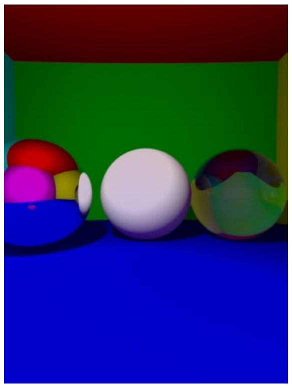
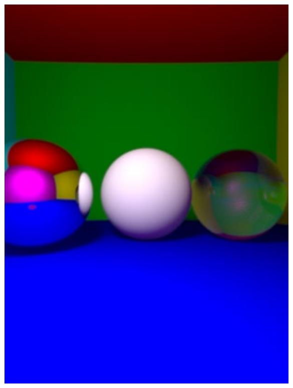
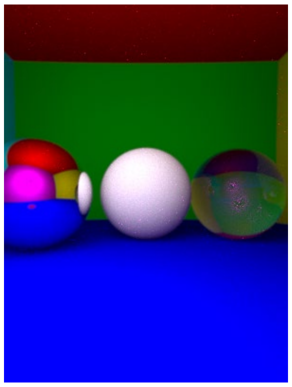
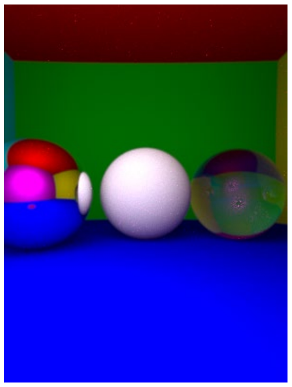


In our environment, any surface not directly exposed to light will be black. This is not very realistic as every objects reflects light of its color all around it. We will therefore implement this in the form of indirect lighting, this adjusts the shadows and immediately makes the image look less simulated and more photographed. As the spheres indirectly reflect their color, we can see a slight pink shade on the center ball, originally white. 1000 rays, computed in 30 seconds.

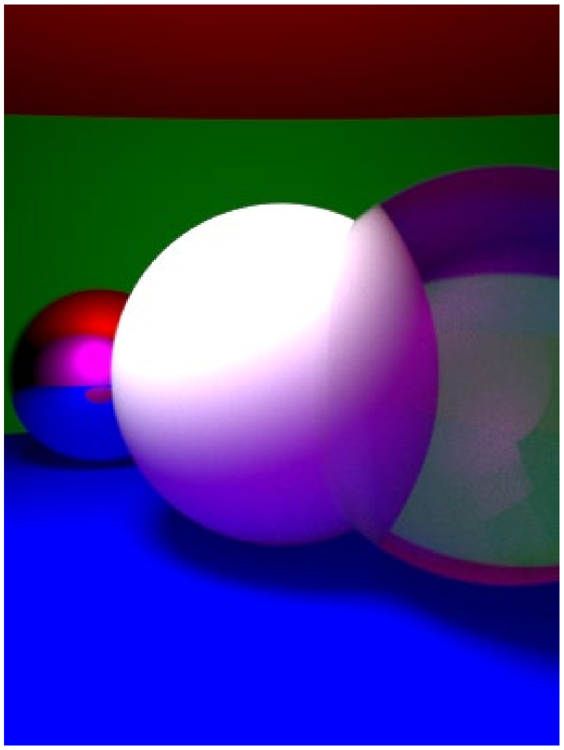
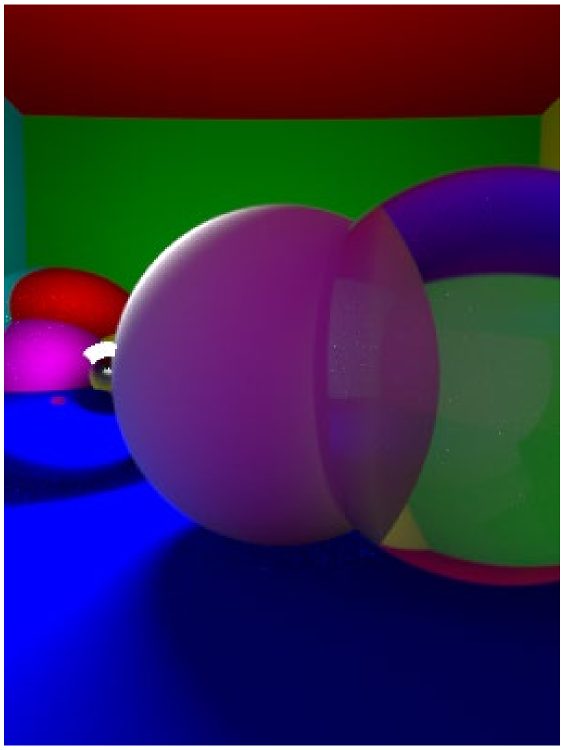
We then move on to antialiasing. This is done with the Box-Muller method and smooths out the edges of the shapes, incorporating them better in the background. We can vary the standard deviation of the function to modify the sharpness of the spheres. With 1000 rays in 30 seconds, here we have a standard deviation of 0, 0.3 and 0.6. We will stick with 0.3 as it eases the transitions seen in the case of 0 without the haziness seen in the case of 0.6.

To give the scene a bit more realism we add spherical lighting to it, making it more interesting to watch. This gives a little more contrast to the light on objects but if to big can over saturate part of the image. For 45 seconds of computation time and 1000 rays per pixel, we get the following results for a point light, a radius of 5 and a radius of 20. Increasing the light radius demands increasing the amount of rays per pixel to avoid noise so the last image is a light of radius 5 but with 2000 rays (1min30).

Finally we added depth of field to give a really camera style artistic picture. And with all the modifications we obtain the following image (left):

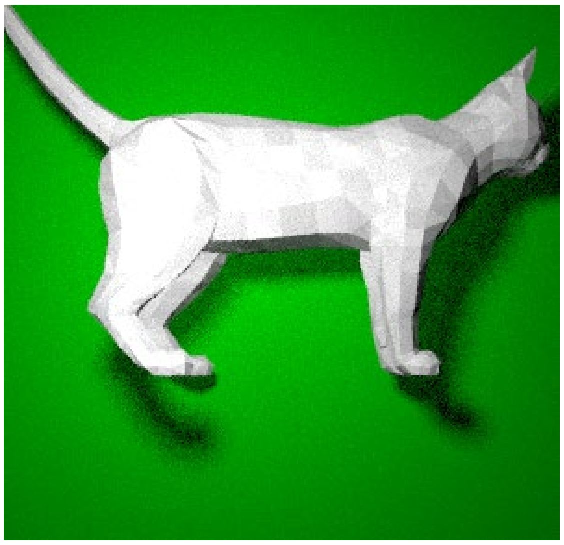
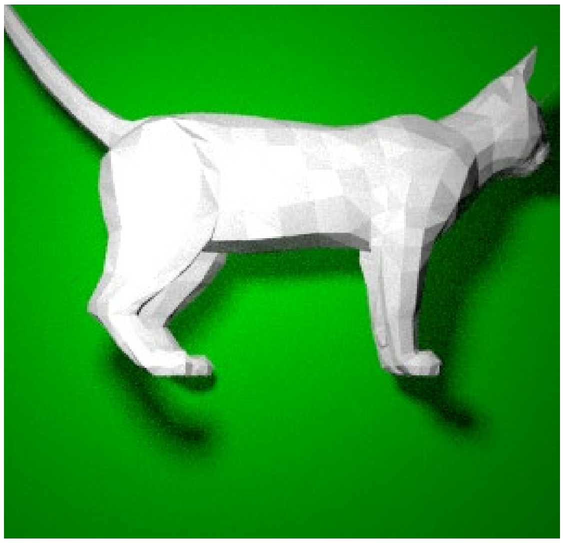



For fun it was also run on a 64 cores machine which computed 10’000 rays per pixel in under an hour but over 2000 the difference was becoming hard to sport. The light was also adjusted, slightly lower, to give a more dramatic effect (right image).

Motion blur required moving the spheres to see. With the program running in parallel this did not seem like a risk worth taking. So this optional was not done.

## 4. Mesh intersections

To do more complex objects and texture, we had to transition from spheres to meshes. To do this we implemented a ray-mesh intersection and even though we also implemented a bounding box this program was still considerably longer to run. Surounding spheres where removed to focus mainly on the cat in the center but it still took 1 minute 30 to run 20 rays per pixel (left) and 5 minutes 50 rays per pixel (right). We can still see everything implemented previously (the shadows and antialiasing mainly) but are quite granny due to the low amount of rays.

Sadly, I was not able to implement the BVH optimization but even though they take forever to render, transparent and mirror cats are fun (the quality was far from good enough to be included here).