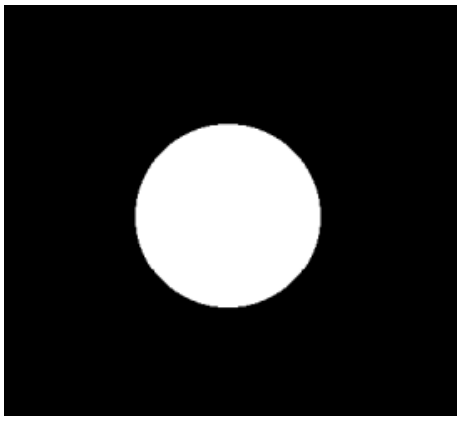
CSE 306 Project 1 report, Dimitrije Zdrale, BX 24

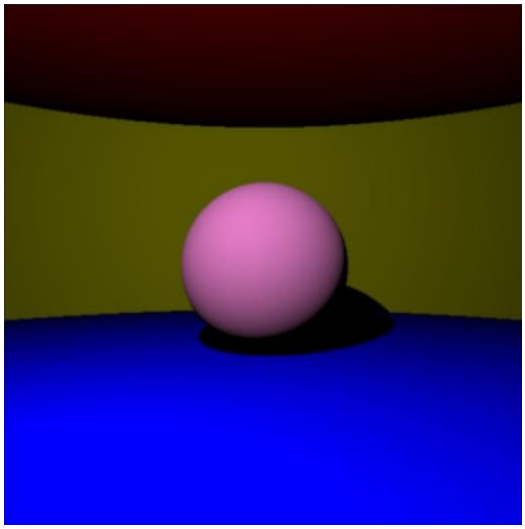
1. Initial steps

At the beginning of the project and lab work, I first had to grasp the full concept of the ray tracer. After understanding the idea, I moved on to generating an initial picture, consisting of a white circle on a black background.
First, I added the sphere object, centered on the screen. Now, to actually see it, I proceeded to read about the formulas used to detect an intersection of a ray with the sphere. This resulted in the initial draft of the "scene_intersect" function, which was, at that stage, only returning the "t" value, and then, depending on it, the program decided whether it should paint the pixel white or black. In the end, it resulted in a quick computation of the following image:



After this, I proceeded to add all the walls (6), creating a closed room in which the sphere was located. This, of course, required some modification of the coloring, namely obtaining the albedo values of the intersected spheres. This resulted in the creation of the "getColor" method, which, at the time, took a reference to the intersected sphere and returned it's color.
This, of course, had to be done in combination with direct lighting from a point-light source, as mentioned in the lecture notes.
For proper results, I had to implement shadow computation for the spheres. At this step, the shadow computation was computed after all the colors have been obtained. Even though it was a direct implementation of a fairly simple formula, it still required adjustments, namely, the gamma correction and ray launch offset. This step didn't take long, as the gamma correction operation was performed on the final pixels. As for the offset, it required some experimental observations to find which epsilon offset was close enough to the optimal.

After these steps, I had a nice picture of a solid sphere inside a colorful room.



(Before adding the side walls)

2. Mirror sphere

After obtaining the initial solid sphere, I proceeded to add another one with the goal to make it a mirror sphere.

This step was a straightforward implementation of the formulas from the lecture notes, however, I did experience some issues with the ray bounce offset, creating an "anomaly" on the side which reflects the other sphere. However, I had noticed that it is the same in the lecture notes.

3. Refraction spheres

Apart from the mirror sphere, I proceeded to implement the two refraction spheres: the regular one, and the hollow sphere. The formulae were implemented quickly, however, flipping the normal when needed created some issues. Even though I was aware of the warning, some of the experienced issues were: refractive spheres appearing as mirror surfaces, solid color surfaces, and sometimes appearing as an amalgamation of different colors. After some time, I've managed to implement the "normal flip" properly.

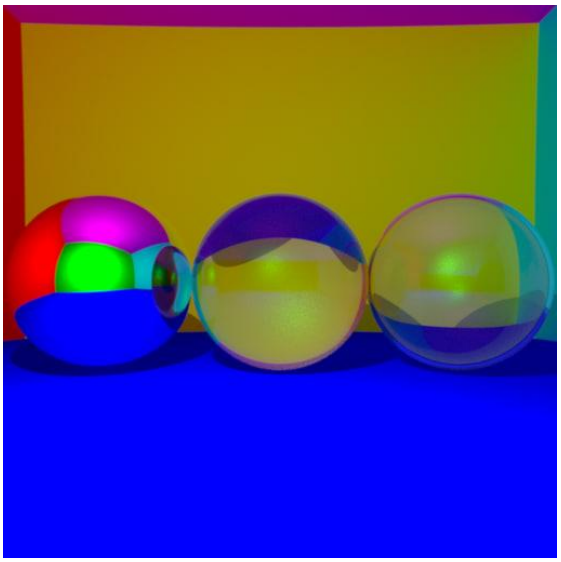
At this point, I've realized that it is better to implement all these features within the "getColor" method, so I had restructured the code.

4. Fresnel's law implementation

After successfully implementing the refractive spheres, I had decided to endow them with Fresnel's law. Even though the formula isn't complicated, I had encountered my first major issue. The spheres were a mess, and I didn't know why.

After carefully examining the entire process, I had realized that the order of my operations was wrong, specifically, the shading computation was in the wrong place. At that point, what I did was the following: launch multiple rays from the same point, run the "getColor" method, average out the results, and then apply the shading. The shading method was implemented directly in the main function, so it was applied after averaging the results. Thus, I had to restructure my code once again and move

the shadow computation into the “getColor” method. After doing this, I finally obtained the desired result:



(After indirect lighting and anti-aliasing as well)

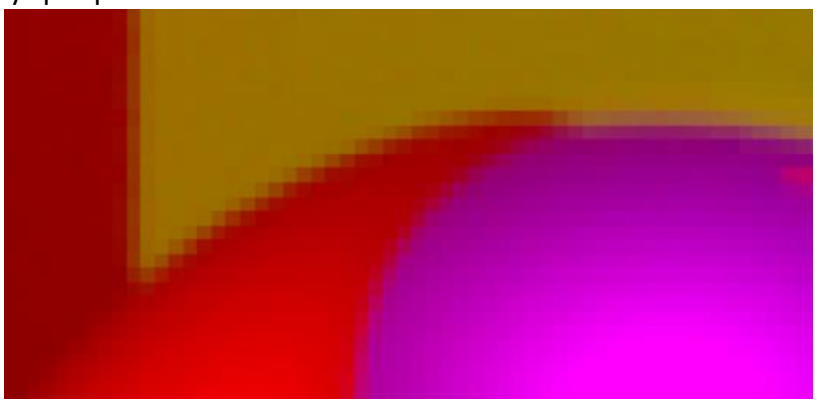
5. Clean-up

As I had reached my desired goal for the spheres at that point, I decided it was time to revisit all the functions and clean the code. This helped me remove around 170 unnecessary lines of code.

6. Indirect lighting and anti-aliasing.

After the clean-up, I proceeded to implement the indirect lighting. I’ve inserted the indirect lighting computation at the end of the “getColor” method, and I didn’t experience any issues with it.

As for anti-aliasing, it was also a fairly simple implementation. Initially, it caused me some confusion when I ran the code with a small number of rays per pixel, but everything turned out to be working properly after running the program with 100 and 1000 rays per pixel.



(Anti-aliasing on a sphere of radius 5, sample taken from the final result)

Completing these steps marked the end of my project work on the spheres.

7. Ray-mesh intersection, bounding boxes and the final wrap-up.

Moving on, I had to edit some parts of my code so everything would fit the whole "Geometry" class approach. This meant, of course, implementing the Geometry class, as well as moving some parameters from the sphere to the Geometry, namely the albedo, shape center Vector, and the Boolean flags indicating whether something is a mirror or refractive surface, as well as the parameter "inside" which was used to indicate whether a the sphere was used as an inside sphere for creating a hollow sphere (I used it to flip the normal).

After completing this, I proceeded with implementing the ray-mesh intersection with bounding boxed directly. As a solid part of the code was provided with the lecture notes, and the rest was very well explained with examples and equations, I didn't experience many issues during the initial works. The "smoothing" of the model was also not a major problem. However, there was one huge issue: when adding the cat model, the entire image appeared red (the chosen color of the cat). This issue had perplexed me for quite a while. In the end, I was able to detect the cause of this malfunction:

In the scene intersection function, I was passing the current shape's P and N parameters by reference, and modifying them there. I had completely forgotten about this, as it was implemented at the initial stages of the project. Upon realizing this, I understood that my Intersection class, which was returned by any geometry's "intersect()" method, had to return/modify the P and N values as well.

This required a lot of work, as I had to essentially re-do a lot of things. After quite some time, I had managed to edit everything as needed, as my "intersect()" method was conflicting with the needs of the triangle mesh class.

This was my final major restructuring of the code.

As soon as I had fixed the modification of the P parameter specifically, I saw the cat model appear on the screen.

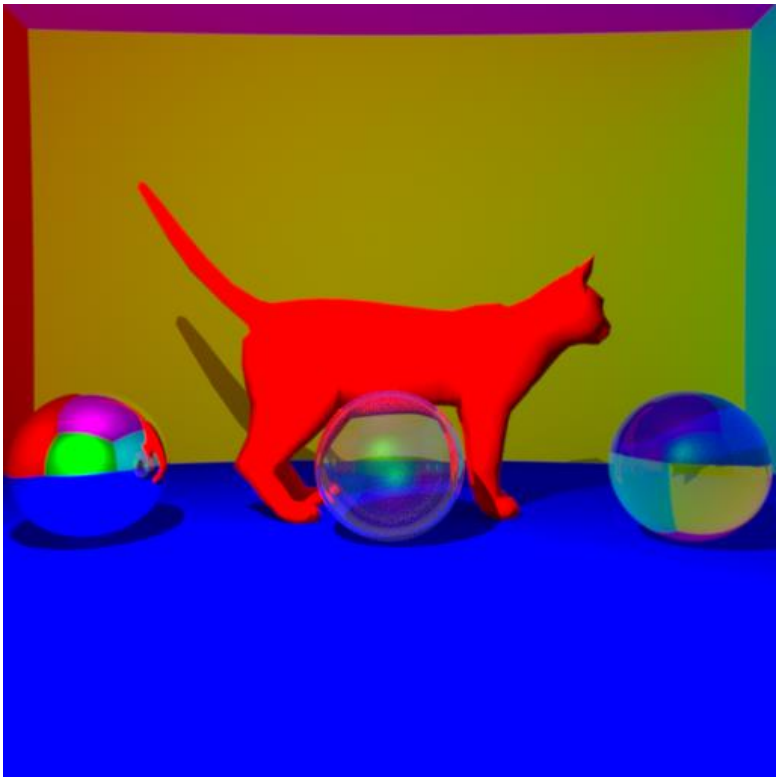
As many of these methods intertwined with the functions implemented during the "sphere phase" of the project, namely, the "getColor" and "scene_intersect" were changed significantly, I had to make sure that everything was in order in the case where I had both spheres and meshes in the scene. I was happy to see that I have not caused many issues.

8. Minor optimization and final result.

The final image took around 950 seconds on average to render, with 1000 rays per pixel. I wasn't too happy with that, so I decided to see what I could do with that.

The biggest issue with this computation time was the "ray_depth" I used in the "getColor" method. Since I had a hollow sphere, that mandated 4 intersections, so 4 rays, thus, I had to go with 5 ray_depth for each ray computation. However, I had realized that I only need that many rays for the hollow sphere. Thus, I had reduced the ray depth to 2, and, if the intersected sphere was an "inside" sphere (indicating that we are working with a hollow sphere), I would just add 2 to the "ray_depth"

parameter. This allowed me to reduce the total computation time to 650-790 seconds for 1000 rays per pixel.
This was my final modification of the code, and I was happy with the result obtained:



(Cat mesh, mirror sphere, hollow sphere and a refraction sphere. Walls are solid-color diffuse spheres.)

9. Conclusion

This project was very interesting to me, as I have always wondered how computer game graphics were made. I coded a simple game before, but had no idea how to do graphics, so the models were just 2-D gifs I made so everything appears animated. Thus, as a big fan of video games, I have really appreciated this hands-on experience.

As for the issues, most of them were caused by myself. Whenever I had to do a major rework of my code, it was either due to getting the operation order wrong, or misunderstanding the requirements of some mathematical operations, resulting in problems that were hard to explain, like seeing a completely red screen when adding a small red cat to a big colorful room.

However, I am glad that I had managed to overcome these issues and produce a great result!