s2088605 - CGR CW2

Josue Fleitas

November 2023

1 Introduction

This report will cover the detailed construction of each of the features of my C++ raytracer. The project was mainly built using ChatGPT responses with some additional help from online resources such as the scratchapixel tutorials and the Raytracing in One Weekend (RiOW) book series (Shirley, 2023; scratchapixel, 2023).

2 Basic Raytracer 1 - Image Write

Figuring out how to write an image to a .ppm file was a fairly simple exercise of prompting ChatGPT to provide me some code that achieved exactly this. The resulting loop it gave me remained relatively unchanged throughout the project (aside from some refactoring to implement tone mapping and keep my code tidy). I did also extend the code by adding a loading bar (similar to that in RioW) so I could see the progress for long renders in the console.

The code given works by outputting lines in the ppm format to an output file stream starting with the image width and height. Figure 1 shows the output of a .ppm file generated using this code (all red pixels).

3 Basic Raytracer 2 - Camera Implementation

Setting up the camera involved a few things. I first started by prompting GPT for some code to implement a utility vec3 class to represent 3D vectors and their common operations (dot products, cross products, etc.). Once I wrote down this code, I then asked ChatGPT for a very basic camera setup.

The code it gave me was very similar to the RiOW book. It consisted of a constructor that computes the camera coordinate vectors from the camera properties such as FOV and a *get_ray* method that shoots a ray out to the scene from each pixel center. It then updated the main output loop to use these generated rays to compute the colour of each pixel. At this point, I also spent some time

Figure 1: Image Write Test

refactoring the .ppm output loop into a render public method inside the Camera class in order to keep my main.cpp as clean as possible. Another change I did make to the code provided was to follow RiOW's approach of inverting the vertical basis vector for the camera coordinate system such that pixels increase as we go down the coordinate system. I decided to do this so I could refer back to the tutorial's code in case I got stuck at any future point, but either approach is equivalent if handled correctly.

4 Basic Raytracer 3 - Intersection Tests

Intersection tests involved prompting ChatGPT to provide me with some code to create the necessary primitives for my raytracer (Spheres, Cylinders and Triangles). I also asked it to provide some code to detect an intersection with the Ray class it had previously given me.

The output for these prompts were surprisingly close to working implementations. The sphere intersection code was correct and the cylinder code only required a few further prompts asking for intersection tests for the top and bottom planes. I ended up tweaking the cylinder intersection test slightly to introduce a small optimization as the LLM code was still checking for intersections when the ray of light was parallel to the planes.

Finally, the triangle intersection took a few tries of back and forth. ChatGPT kept giving me code that worked only under certain assumptions such as the triangle being parallel with one of the axes. Once I asked it for an intersection test for a triangular plane in 3D space, the code it gave me produced the correct result seen in Figure 2. As I will discuss in the section on textures, I did end up refactoring this code as I followed a section of the scratchapixel tutorial to guide me on how to correctly implement Barycentric coordinate calculations.

At this point, I also spent some time prompting GPT to give me code on how to set up my JsonParser file to read in the given scene jsons. In order to do this, I directly asked it to give me some code that would parse the JSON given (I copied in an example JSON). GPT, however, suggested to use the JSON for Modern C++ library from @nlohmann on GitHub, but I went with jsoncpp instead as I had used it in previous projects. On top of writing the parser, I also created some architectural classes to again keep my client code clean, such as a Scene class which collects the lights and Hittable objects in the render.

5 Basic Raytracer 4 - Binary Image Writing

This was one of the simpler features to get working and barely involved using the LLM. Instead, I simply added a method in my Camera class that would return a red colour when the hit method of a Hittable primitive returned true and black otherwise. I did ask GPT to provide me with some code for a colour utility and I eventually placed all of my tone-mapping logic inside this class as the author does in RiOW. As alluded to before, the result of this code can be seen in Figure 2.

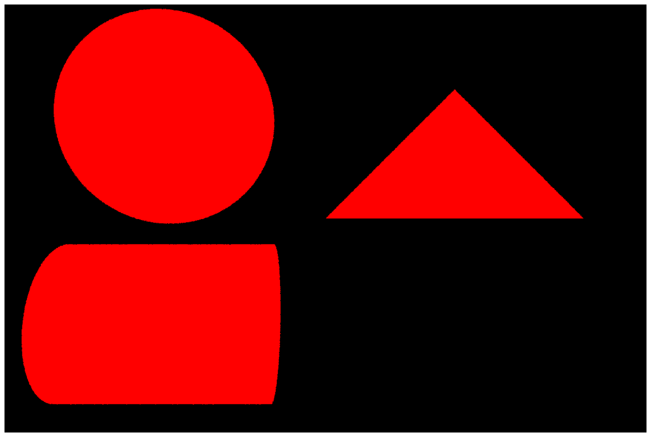


Figure 2: Intersection Test with primitives

6 Basic Raytracer 5 - Blinn-Phong Shading

Blinn-Phong shading was probably one of the hardest features for me to implement and involved many back and forth conversations with the LLM. Part of the reason why this was so hard was because prior to starting on Blinn-Phong I had decided to implement some quick tone mapping and gamma correction (as will be discussed in section 8 below) but at the time had a bug in my code that was making my renders look gray-ish. The other main challenge I encountered was in figuring out how to add the ambient term as I was misunderstanding how this was applied which was causing a white-tint over my renders.

Once I managed to iron out both of these bugs however I realised that the code ChatGPT had been giving me was virtually correct although I did have to explicitly ask it to give me an implementation that considered multiple light sources. The code then works by accumulating diffuse and specular intensities for each hit point and light and returns the sum of these and an ambient term (weighted by their respective coefficients) as the final shading.

In order to integrate the Blinn Phong material in my code, I created an abstract Material class with a getShading method that would return the resulting colour for each of my materials. I then added some parsing logic to my Json-Parser to create the Blinn Phong material with its parameters and assign it to the respective primitives. The resulting code produced renders such as the one in Figure 3.

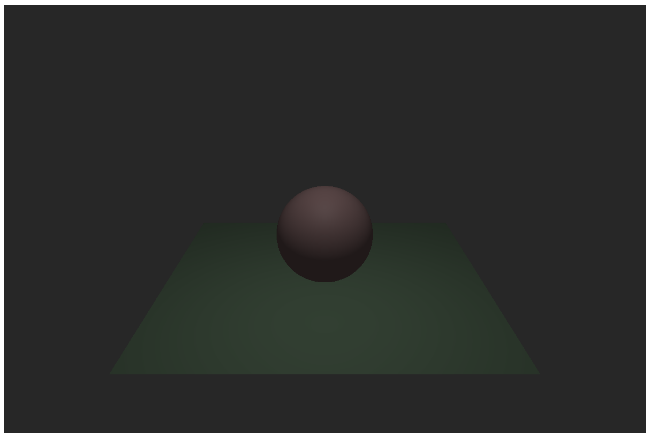


Figure 3: Blinn Phong shading without reflections or refractions

7 Basic Raytracer 6 - Shadows

Until this point, I had been setting the lights in my scene manually in the driver code. In order to implement shadows, however, I had to first refactor this and create a new PointLight class with a position and intensity parameter, which ChatGPT helped with. Once this was set up, I asked ChatGPT to give me some code to compute the illumination of a point in the scene from such a point light based on its position. This gave me a correct implementation that worked by casting a shadow ray onto the scene and checking whether it intersected with an object or not, returning an intensity of 0 in case of an intersection. I could then use this new method instead of the light intensity value to determine the multiplier in my Blinn Phong loop. In order to match the reference images, I also had to tweak the ambient term in my material slightly (to a brighter value) so shadows would not appear completely black. An example render with some hard shadows utilizing this method can be found in Figure 4 below.

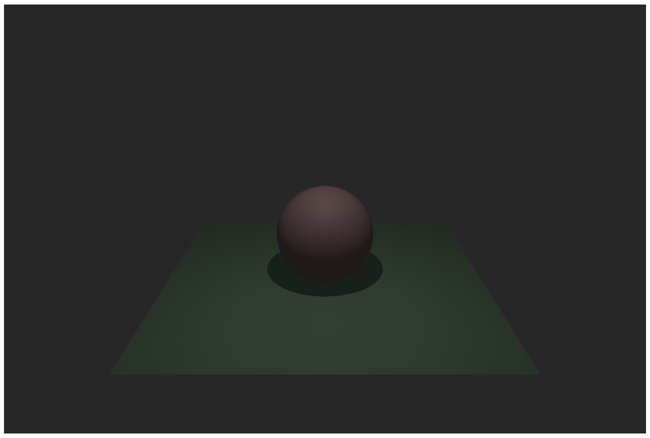


Figure 4: Hard Shadows from Point Light

8 Basic Raytracer 7 - Textures

Textures also took a bit of back and forth with the LLM and some further researching in order to get them working properly. I began this feature by asking GPT for some code that was able to read in a .PPM file and populate a Texture object from it. The code it gave me back was close to working and I only had to ask it to tweak it so it would read a binary .PPM file (P6) instead as this was the output format of the online JPEG-PPM converter I was using, but it was able to handle that correctly as well.

UV mapping a texture onto a sphere was quite simple and the LLM output was close to perfect for me. I only had to invert the v vector as the texture was being mapped upside down and divide by 255 to produce correct colours. The cylinder took a little more work as I did not quite understand how to map around the azimuthal angle. The code that GPT gave me was working only partially as it was not properly mapping on the vertical axis and was assuming that my cylinder was aligned with the y-axis. I had to do some online reading where I found some GameDevExchange posts that were quite helpful and made me realise what was wrong (John, 2016). The problem I was encountering was due to GPT assuming that my texture coordinates spanned from [0, 0] -¿ [1, 1] when this was not the case. As such, I tweaked my texture class to preprocess the texture in this way which fixed the x-axis mapping on the cylinder. I then arbitrarily picked the top of the cylinder to be v coordinate 0 and 1 by dividing by the total height of the cylinder which produced the final results.

Finally, mapping textures onto triangles was by far the hardest one to implement as it involved computing the Barycentric coordinates of the triangle. After some back and forth with GPT, I once again took to online resources to understand the problem. It was then that I came across the scratchapixel website and their extremely helpful tutorial on computing barycentric coordinates. Once I was sure about the coordinate calculations, I then arbitrarily picked the [0, 0], [0, 1] and [1, 1] coordinates of my texture and used the resulting coordinates to interpolate my texture accordingly. This ultimately produced the final results seen in Figure 5 below.

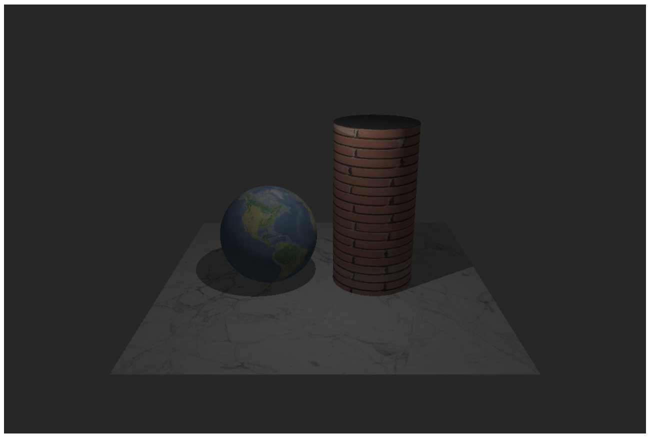


Figure 5: Texture Mapping onto Primitives

9 Basic Raytracer 8 - Tone Mapping

Tone mapping was another aspect of my code that I had some issue with due to some very early bugs. I naively decided quite early on to prompt GPT for some tone mapping code that went beyond the normal linear tone mapping. The first answer it gave me and the one I implemented was Reinhard Tone Mapping which works by taking the luminance of the image and scaling the RGB values accordingly. Even though the code that GPT returned was perfectly fine, the issue boiled down to me not gamma correcting my render which resulted in exceptionally dark images. This led me to wrongly believe that the dark images must have been a bug with my Blinn Phong shading as I was working on this at the time. Once I figured out I had to gamma correct my tone mapped image, the renders suddenly looked much closer to the reference images. I ended up further prompting GPT to also give me some code that would implement exponential

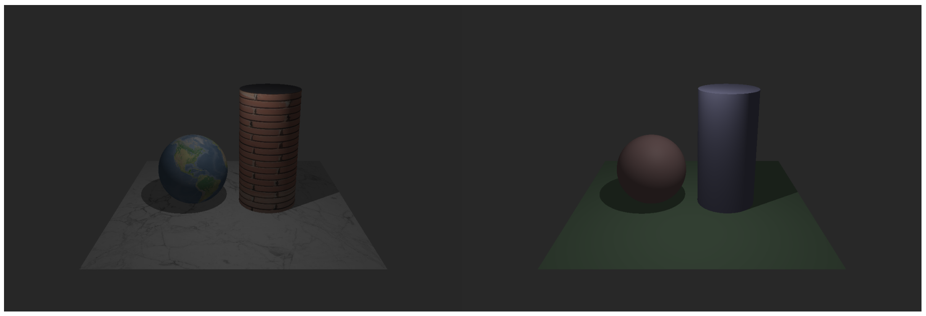

(a) Reinhard Tone Mapping (b) Exponential Tone Mapping

Figure 6: Comparison of Reinhard (L) and Exponential (R) tone mapping renders.

tone mapping, but I ended up preferring the colours produced by Reinhard. A comparison of the two tone mapping methods can be seen in Figure 6.

10 Basic Raytracer 9 - Reflections

Reflections were another relatively simple feature to implement. GPT gave me some code that was relatively close to correct. I simply asked it to extend my Blinn Phong code to accommodate for reflections. The only thing I changed apart from some variable naming for readability was the location of the reflect method as I knew I would need it for some other materials in my code. As such, I move it over to the vec3 class and prompted GPT to give me a similar method that could handle refractions. Figure 7 shows a render with a correctly implemented Blinn Phong reflection.

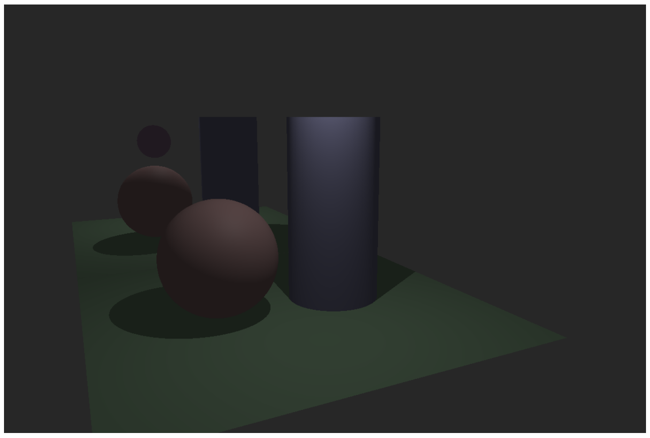


Figure 7: Blinn Phong Reflections

11 Basic Raytracer 10 - Refraction

Refractions were slightly harder to implement as it involved handling total internal reflections as well as the refraction itself. However, the code that GPT returned in its first prompt was pretty close to a working solution. I did extend it by separating the refraction direction and possibility of refraction computations as a small optimization (motivated by RiOW's approach). The combination of the two answers gave me my final refraction code which produced the render in Figure 8.

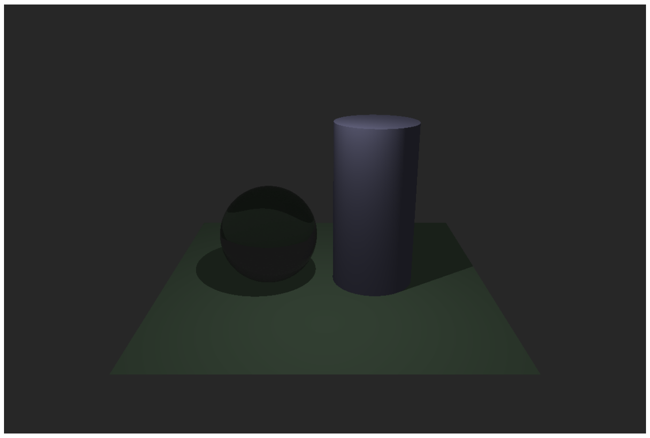
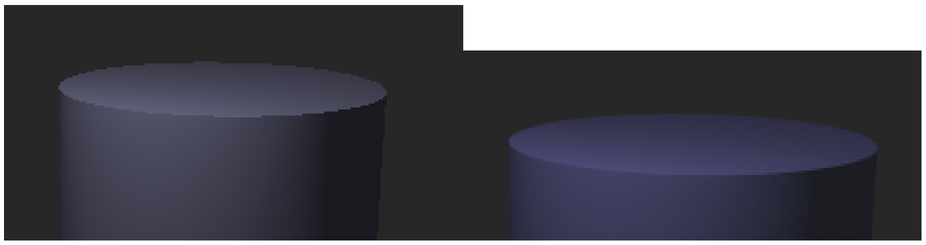


Figure 8: Blinn Phong Refractions

12 Basic Raytracer 11 - Bounding Volume Hierarchy

I started working on BVH by asking GPT the steps I should take to implement it. Not fully understanding all of the details, I then took online to fill in some of the gaps that their answer was missing. At this point, I was ready to go back to the LLM and ask for some code that would implement an axis-aligned bounding box (AABB). Once it gave me this, I asked it to give me code to compute AABBs for all of my primitives, which it did quite well (although I tweaked its answer to fit my codebase).

The harder part of coding the BVH data structure involved a lot more online referencing (particularly from the RiOW second book). My final code is a combination of the two answers as I kept going back and forth between them. The general structure of the two responses was the same but they did exhibit some differences in design such as using a random axis vs sorting along the longest axis. I also used RiOW's interval utility to clean up some of GPT's code and to help me integrate the BVH into my existing codebase. The final result



(a) No Antialiasing (b) Random Antialiasing

Figure 9: Comparison of no antialiasing (L) vs random antialiasing (R)

worked quite well, noticeably reducing the rendering time of the scene.json file provided from 50 seconds to about 40 seconds using the phong model.

13 Pathtracer 1 - Antialiasing via Multi-Sampling Pixels

Antialiasing was arguably a really simple feature to implement as GPT was able to give me a correct implementation of random jittering after a single prompt. I did however make an incorrect assumption about the range that the jitter deltas should take which resulted in incorrect renders. After clarifying this with the LLM, however, I was able to get AA working as can be seen in Figure 9.

14 Pathtracer 2 - Defocus in Finite-Aperture Cameras

Similarly to anti-aliasing, this feature was quite simple to implement and only required a few prompts to get right. As it also works by introducing some randomness in the ray sampling procedure, the hardest part was simply to realise how to sample within the aperture radius which ChatGPT was able to help with quite easily. Once implemented, my raytracer was capable of renders such as 10.

15 Pathtracer 3 - Render Materials with BRDFs

Out of all of the advanced pathtracing features, this was the one I found the most challenging. In order to achieve it, I first asked GPT to give me some options of BRDFs I could code. After it suggested Lambertian BRDF, I asked it for some code to implement it which worked quite well on my first try. I had a similar experience with the Schlick BRDF which it was also able to help me implement quite painlessly. I did get stuck at one point with my Schlick implementation and had to consult the RiOW book for reference, but the material

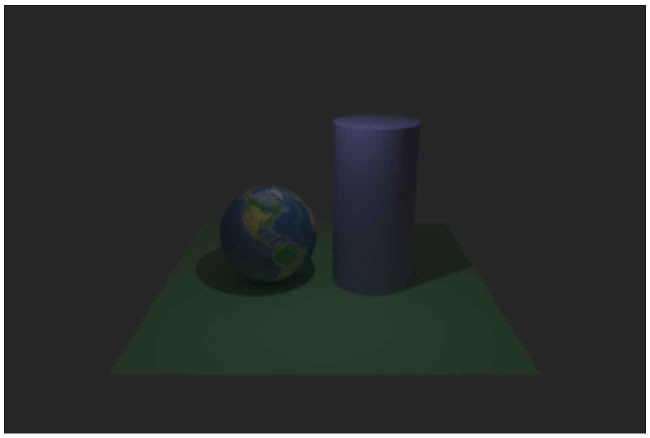


Figure 10: Aperture Defocus

behaved correctly once I fixed the reflectance colour.

The problem came when I tried to add in a micro-facet BRDF (specifically Cook-Torrance) according to the LLM output. The code given to me did not specify what some of the parameters were or how to implement them and after further prompting, GPT started giving me wrong code. Even though the algorithm I ended up with after some chatting with GPT was close to other pseudocode I found online, I was still unsure about some of my parameters which resulted in my material looking completely black. Due to the time constraint for this coursework, I never managed to fix this but left the commented out code in my files for reference. Figure 11 shows a render with the two other materials in pathtracing mode.

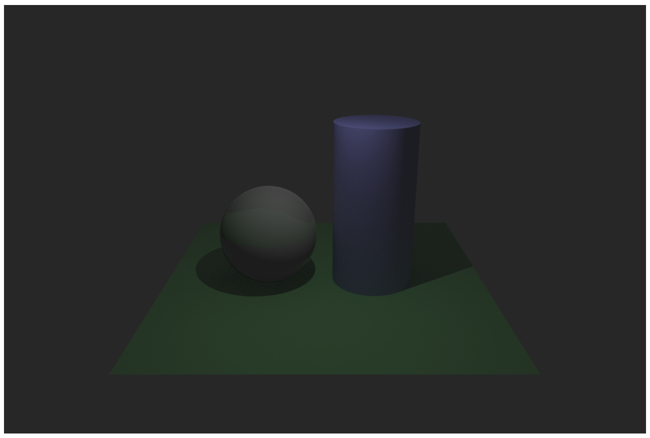


Figure 11: BRDF Materials (Lambertian and Schlick)

16 Pathtracer 4 - Soft Shadows via Sampling Area Lights

In order to add soft shadows to my raytracer, I asked ChatGPT to help me create an Area Light class which it did fairly well. I did change the way it generated random floats in order to repurpose some of my utility functions but kept the rest of the code relatively unchanged. I then asked it how I would compute soft shadows in the same class and it returned a method with a loop that accumulates total illumination at a given point. I then repurposed this method and changed the color calculation in my pathtracing loop to use this instead of the fixed light intensity value. Results showcasing these soft shadows are presented in Figure 12.

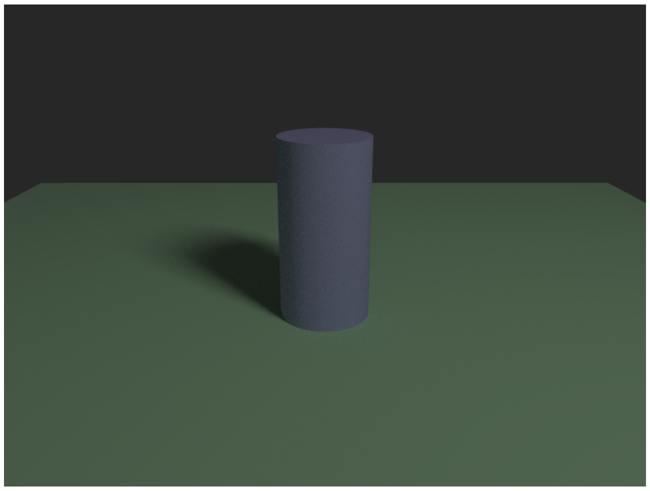


Figure 12: Soft Shadows via Area Light

17 Pathtracer 5 - Multi-Bounce Path Tracing

Multi-bounce path tracing involved adding a recursive call in my pathtracing Camera class similar to what I had in my Blinn Phong method. I therefore added this but then had some issue with incorporating direct lighting into my scenes for which I once again recruited GPT's help. With the right prompt, I was able to have it fix my code by prepending my colour computation with a direct lighting contribution algorithm that cycles through the lights in the scene and samples them checking for intersections in each case. Once this was added, my code was producing physically accurate light bounces in the pathtracing mode.

18 Video

For my video, I wanted to showcase as many of the features I had implemented as possible, particularly the wide range of materials and textures I had managed to get working. For this, I constructed a simple scene with two planes, the bottom one with a marble texture and the back one with a reflective surface (as if a mirror). I then laid down 3 equidistant spheres on the marble plane and gave them a Lambertian BRDF, an Earth image texture and a Schlick BRDF with Refractions for their materials. I then created a point light to illuminate the scene and added some cylinders with the same marble texture as columns on either side of the spheres to make the render a bit more interesting. For the video, I simply interpolated the camera's position from left to right and the light position from right to left to add some dynamic shadows and reflections. I would have liked to animate my Earth sphere so it could rotate but was unable to get this working due to time constraints.

The final render included in the submission has a resolution of 600x400 pixels, a frame rate of 15FPS and was rendered using the pathtracing engine with all of the advanced features and 8 light bounces. I originally tried to have my light source be an area light to achieve some soft shadows but even with BVH enabled this resulted in a really slow render (4+ hours). I suspect that other optimizations are required to bring this rendering time down. Modifying the code so it could utilize a GPU would also be extremely helpful. Overall, however, the video still showcases a simple but visually appealing scene with most of the features required for this assignment.

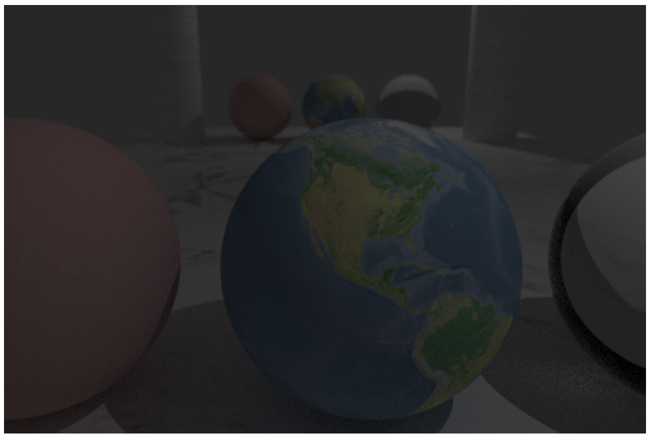


Figure 13: Still From Final Render

References

John John. How to map square texture to triangle?, January 2016. URL `https://gamedev.stackexchange.com/q/114320`.

scratchapixel. Ray-Tracing: Rendering a Triangle, 2023. URL `https://scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle`.

Peter Shirley. Ray Tracing in One Weekend Series, 2023. URL `https://raytracing.github.io/`.