

University of Edinburgh

## **CW2 Report**

### **Abstract**

In this project, I built a C++ raytracer with Blinn-Phong using the assistance of ChatGPT and GitHub Copilot. This raytracer performs ray-intersection tests on spheres, cylinders, and triangular planes, and computes specular, ambient, and diffuse lighting conditions for these shapes. It also handles shadow and reflection ray calculations, and provides texture mapping for the geometric primitives, providing a thorough application of basic ray tracing techniques.

Esmé Puzio  
Computer Graphics: Rendering  
Professor Kartic Subr  
23/11/2023

## **Contents**

### **Features**

1. Image write
2. Camera Implementation  
*Coordinate Transformation*
3. Intersection Tests  
*Sphere, Cylinder, Triangle*
4. Binary Image Writing
5. Blinn-Phong Shading
6. Shadows
7. Textures  
*Sphere, Cylinder, Triangle*
8. Tone mapping
9. Reflection
10. Refraction
11. Bounding volume hierarchy

### **Video**

1. Conception and Screenplay
2. Modelling
3. Rendering

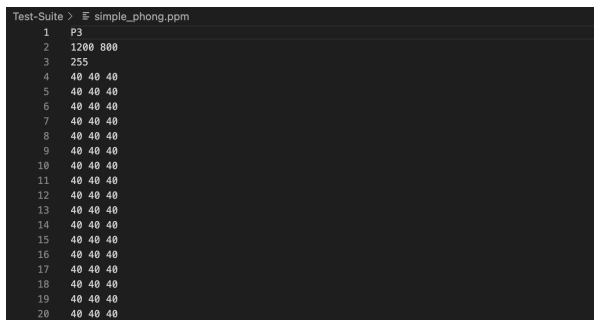
### **Additional Comments**

## Features

### 1. Image Write

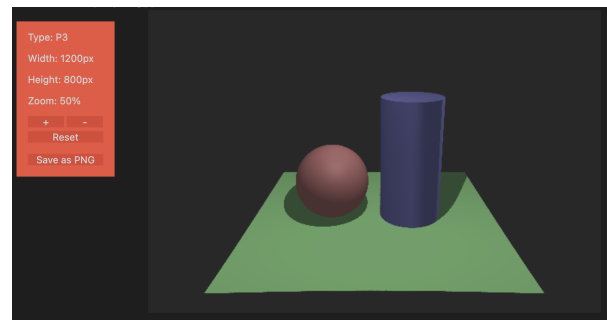
#### Implementation

To write to a ppm file, I used the ofstream library within my camera.h class file. At the beginning of the render function in this class, I specified the width, height, and the maximum color value (255) to output to the ppm file. Elsewhere in the code, I hardcoded the number of samples needed to prevent antialiasing as the integer *numSamples*. Then, in the rendering loop, for *numSamples* rays cast out from each pixel, I called the rayColor function to return the value of the pixel color for that ray as a vector of three doubles. After the total pixel color is calculated, the writeColor function is called to average these samples, scale the final pixel color to an integer ranging from 0 to 255, and output each integer to the output file specified by the ofstream object.



```
Test-Suite > F simple_phong.ppm
1 P3
2 1200 800
3 255
4 40 40 40
5 40 40 40
6 40 40 40
7 40 40 40
8 40 40 40
9 40 40 40
10 40 40 40
11 40 40 40
12 40 40 40
13 40 40 40
14 40 40 40
15 40 40 40
16 40 40 40
17 40 40 40
18 40 40 40
19 40 40 40
20 40 40 40
```

(a) Raw PPM output



(b) PPM output visualised with ngtystr's PBM/PPM/PGM viewer in Visual Studio Code

## LLM Output

I did not use an LLM in this section. Instead, I implemented image writing using part 3.1 of the “Learn Ray Tracing in One Weekend” tutorial. For the remainder of this paper I will refer to the “Learn Ray Tracing in One Weekend” tutorial as [LRTOW].

## Modifications

I moved the write\_color function from the color.h header file to my camera class, as it seemed redundant to have a header file containing only one function. When I reached later parts of the “Learn Ray Tracing in One Weekend” tutorial, I added a clamp for intense values of my output, and averaged the final pixel colour before scaling the pixel colour from the interval [0,1] to the interval [0, 256]. Further explanation for these modifications are touched on in the Tone Mapping section.

### 2. Camera Implementation

## Implementation

To implement the camera, I roughly followed the camera tutorial from [LRTOW]. I started by creating a camera class with fields for each of the inputs listed in the JSON file under “camera”. Additionally, I added extra fields to store the bias in calculating ray/object intersect tests, the basis vectors for the camera position, and the number of samples needed to prevent aliasing. Then, I added three functions to the camera class.

1. “render” writes the width and height parameters to the P3 output file, and calculates the viewport dimensions for the camera given the camera position and FOV. Then, this function calculates the location of the upper left pixel in the viewport, and generates vectors to determine the centre of the next pixel in the viewport. This function also begins the rendering loop for the raytracer, and prints out the current progress of the raytracer and time of completion to the console.
2. “rayColor” computes the colour of each pixel given a starting ray, depth, and a list of iterable objects that can be hit in the scene. This function was used to calculate Blinn-Phong and binary rendering outputs for the scene.
3. “writeColor” writes the pixel colour for each ray generated in “render” to the output file as a list of r, g, b values.

## LLM Output

To generate a basic structure for the camera class, I asked ChatGPT to build a camera class that generates rays for each pixel using the given JSON inputs. I used Copilot to rewrite the calculations for the camera transformation, as the shapes were not appearing in my scene based on the original transformation provided by ChatGPT. Copilot also helped implement code to prevent antialiasing by shooting several rays from the same centre point towards slightly different directions within the bounds of the pixel, averaging the colours returned.

ChatGPT was also used to construct a `vec3` class based on the Eigen library, as the original code it generated included a comment stating that I would need the Eigen library to perform vector operations when setting up the image plane. To be sure I was on the right track, I compared ChatGPT’s output of the `vec3` class to the `vec3` class declared in [LRTOW], the `vec3` class in the Eigen library, and the `vec3` class in the glm library, adding and modifying functions as needed. I also used Copilot to include inline functions for `vec3` based on its implementation in [LRTOW].

## Modifications

At first, the coordinate system for the sample scenes wasn't lining up right. To fix this, I tweaked how the horizontal and vertical viewport vectors were calculated. Instead of using the regular right and up basis vectors for the camera, I switched to inverse right and inverse up. This switch meant I had to redo the calculation for the top left pixel’s

position. With Copilot's help, I recalculated the `topLeftPixel` until my viewport looked like the example renders in the coursework specs.

For the antialiasing jitter calculations (`jitterpx`, `jitterpy`, `u`, and `v`), I had to make another change. The initial code from Copilot didn't use the values `'halfWidth'` or `'halfHeight'`. So, I updated the code to create random values between `[-0.5, 0.5]`, then scaled them to the image size and the vectors pointing from the top left of the image plane to the current pixel. This made it so rays were being calculated between the bounds of the pixel square, instead of only firing through the centre of the pixel. Also, I added a console output to show the percentage of lines that were done, and a clock to see how long the image took to render.

### **3. Intersection Tests**

Following the [LRTOW] tutorial, I developed three classes, which are implemented through the `"scene.h"` and `"shape.h"` header files. The first, `'Hit_Record'`, stores the intersection point, normal, texture coordinate, distance to origin point of ray, and material of the object closest to the camera that's hit by a ray. The second, `'Shape'`, acts as a generic type for spheres, triangles, and cylinders. Lastly, `'Scene'`—inheriting from `'Shape'`—holds lights and objects in the world and performs hit calculations for each object.

I then created specific classes for spheres, triangles, and cylinders, each inheriting from the basic `'Shape'` object. In these, I customised the hit check, originally declared in `'Shape'`, to handle intersections for each shape based on its geometry.

The hit method, present in each object type, receives the current ray, a range to check for intersections, and a `hit_record` object (by reference). This method is consistently invoked within the `rayColor` loop for every object in the scene. In the `'Scene'` class, the hit method initiates a `hit_record` object, iterates through all objects in the scene, and calls their individual hit methods. If a hit is detected, the object's hit method updates the `hit_record` with the normal, intersection point, the ray's intersection distance, and the intersected object's material, returning `true`.

In the `Scene`'s hit function, the ray's maximum distance is then updated to the nearest intersection point. This occurs so that subsequent shapes in the list are not counted as intersections if they're behind the current object. The process continues until the closest intersection point is found or if no intersections occur in the scene. When the `Scene`'s hit function returns `true` for a ray, the normal and material details are stored in the `hit_record` for use in the Blinn-Phong shader.

#### ***Sphere***

In developing the sphere class, I started by using the `sphere.h` class from [LRTOW] to declare its functions and fields. After declaring these, I turned to ChatGPT

for assistance in coding the sphere-ray intersection logic. Specifically, I needed code to determine if a ray intersected with a sphere and, if so, to set the hit\_record fields (like t, normal, material pointer, position) based on this intersection check.

The logic provided by ChatGPT for calculating the sphere-ray intersection involved several steps. First, it computed the vector from the ray's origin to the sphere's centre. Next, it used this vector and the ray's properties to calculate the quadratic coefficients. These coefficients were then used to construct the determinant. If this determinant turned out to be greater than zero, and if the intersection point was closer to the camera than any previously recorded intersection, the sphere was considered as hit by the ray.

### ***Triangle***

Initially, ChatGPT implemented the triangle plane intersections using the Möller-Trumbore algorithm. However, while debugging this code, I discovered that the initial implementation of the Möller-Trumbore algorithm was prone to self-collisions, especially when calculating rays coming from the surface of the triangle, which was how I computed shadow rays. So, I asked ChatGPT to regenerate the code for this algorithm, and the alternative solution provided worked well and did not require significant modifications.

The Möller-Trumbore algorithm implemented by ChatGPT calculated barycentric coordinates by finding the determinant of two edges of the triangle. It incorporated backface culling to eliminate rays that were parallel to the triangle's plane. Afterward, it used the barycentric coordinates to determine whether the intersection point was located on the triangle's surface.

### ***Cylinder***

The cylinder intersections were harder than those for the sphere and triangle, as a cylinder technically consists of three parts: the body, top cap, and bottom cap. In creating the cylinder class, I first set up basic fields and function definitions, drawing from the JSON input and the structure used in the sphere class. Next, I asked ChatGPT to generate the hit function for the cylinder class, but the initial code it provided was for an infinite cylinder without end caps. After realising this, I asked ChatGPT to revise the code to check intersection for a finite cylinder with spherical caps at the ends of the cylinder.

However, in the original implementation for intersectCaps, both caps were rendering over the body of the cylinder, but only one cap should be visible at a time. To correct this, I implemented a check to render a cap only if the dot product between the cap normal and the camera ray was positive. This check was done before calculating the intersection for the cylindrical body.

For intersecting with the cylindrical body, ChatGPT's approach involved calculating the quadratic coefficients of the cylinder and solving for the ray equation. The body was marked as 'hit' if there were real roots between the ray's origin and the nearest intersection point. For the cylinder caps, the method was to simulate circular planes at the top and bottom, solve the plane equation for the ray, and check if the intersection point was closer to the camera than any previous intersections.



(a) Intersection tests in binary.ppm

#### 4. Binary Image Writing

To implement Binary Image Writing, I used the rayColor method within the render loop of the render method. When the rendermode in the camera class was set to binary, a hit\_record object named "rec" was added to the scene, and the rayColor method triggered a call to the hit function of the scene class. The hit function was designed to iterate through each object in the scene and invoke their respective hit functions.

If a hit was detected on any object, "rec" saved the point of intersection, the normal at that point, and the material of the object that was closest to the camera and hit by the ray. If a hit was detected, the rayColor function would return red. In cases where no objects were hit by the ray, the returned colour for the pixel was solid black.



(a) Binary image writing for `binary_scene.ppm`

## LLM Output

I built upon the `rayColor` feature I had written earlier as part of the camera class to include a check for maximum depth and a call to the hit function for the scene object. Once I'd declared the check for `rendermode == binary`, GitHub copilot automatically filled in the rest of my implementation. To check that the code Copilot returned was correct, I referenced step 6.7 of the [LRTOW] tutorial.

## Modifications

I added a check for the number of bounces to the `rayColor` function, and added extra logic in the `setCameraParameters` function in `raytracer.cpp` to set the number of bounces for binary rendermode scenes to 1.

## 5. Blinn-Phong Shading

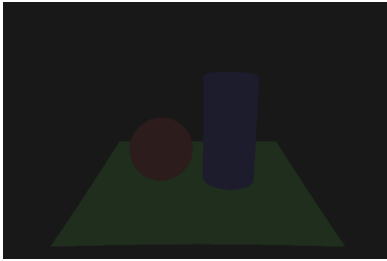
Just like with Binary Image writing, inside the render loop of the render method, I used the `rayColor` method for the rays sent out by the camera for each pixel. When a ray hit an object in the scene, I stored the hit point, the normal at that point, the distance of that point along the ray, and a pointer to the object's material in a `hit_record` object. Next, I added the ambient light contribution to the scene. I then checked if the hit point was in shadow; if it wasn't, I calculated and added both the diffuse and specular colours for that point. If the ray didn't hit anything, I returned the world's background colour.

For images lacking a texture, I set the ambient colour to 0.6 times the diffuse colour. This was because the ambient colour wasn't provided as a parameter in the JSON, and the value of 0.6 provided ambient lighting values close to those in the

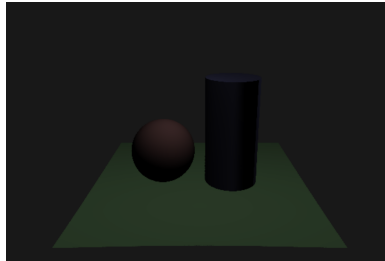


coursework specs. Then, I went through each light source in the scene. For points not in shadow, I calculated the diffuse contribution using the Phong shading formula  $k_d(L_m \cdot N)i_{m,d}$ , scaled by 1.1 to make the final render resemble the coursework specs.

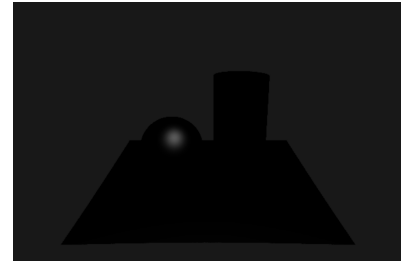
The specular contribution was a bit different. I first calculated the halfway vector between the light direction and the direction of viewing. Then, I figured out the specular intensity using the dot product of this halfway vector and the normal at the object's surface. The specular contribution was computed using the Blinn-Phong formula. To spread the specular highlight more evenly across my object's surface, I scaled down the final specular intensity by 0.1. This adjusted intensity was then added to my final color, but only when the dot product of the normal and halfway vectors was positive. Throughout this process, I made sure to clamp the ambient, diffuse, and specular values before they were added to the pixel's final colour.



(a) simple\_phong.json with ambient contribution



(b) simple\_phong.json with diffuse contribution



(c) simple\_phong.json with unscaled specular contribution

## LLM Input

I gave ChatGPT a general query for implementing Phong shading within my code, and in doing so, ChatGPT generated code that calculated the specular contribution using Phong shading as opposed to Blinn-Phong shading. Thus, I regenerated the code for specular contribution to use Blinn-Phong shading instead of Phong shading.

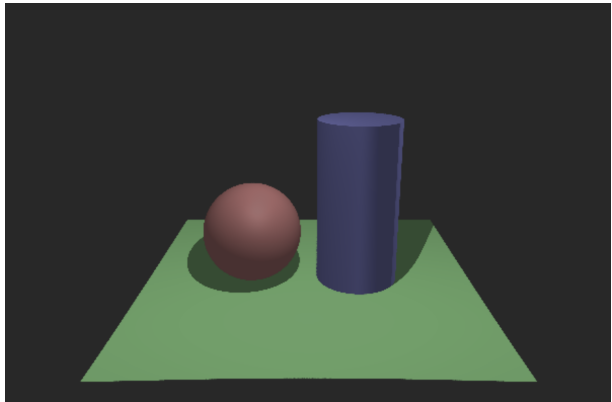
## Modifications

I added tone mapping to each of the components calculated for Blinn-Phong shading, and clamped the final result as well before computing the pixel colour.

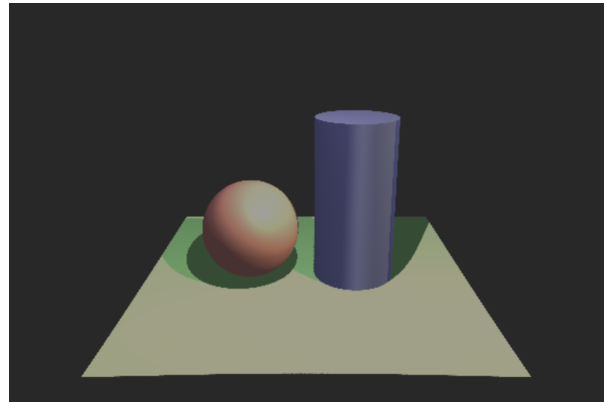
## 6. Shadows

In the Blinn-Phong shader, for each light source, I made sure to exclude the specular and diffuse components from the pixel's colour if the intersection point was in shadow. To determine this, I cast a ray from the point of intersection towards each light

source. This ray was checked using the scene's hit function to see if it intersected with any other objects in the scene before reaching the light source. To avoid the ray colliding with other points along the object it originated from, I added a bias of 0.001 along the normal at the ray's origin point. If this shadow ray hit other objects in the scene, I skipped over calculating the diffuse or specular components for the current light source.



(a) simple\_phong.json with one light source



(b) simple\_phong.json with an added light source at [1.5, 1.0, 0.25]

## LLM Output

I asked ChatGPT for a generic method to calculate shadows within a Blinn-Phong rendering loop, which provided a basic framework for a shadow caster. Once I had this outline, Copilot helped me fill in the specific calculations for specular and diffuse components based on the instructions left by ChatGPT.

## Modifications

I repurposed the hit function from the scene class for my shadow ray caster to iterate through all objects in the scene and check if there were any obstructing the path to the light source. Instead of checking for intersections along the entire length of the ray, I limited the maximum distance to the length of the vector from the intersection point to the light source. This way, I didn't count objects behind the light source as casting a shadow. As mentioned, I also added a bias of 0.001 to the starting point of the shadow ray, which helped avoid self-intersections on the surface of the triangular plane.

## 7. Textures

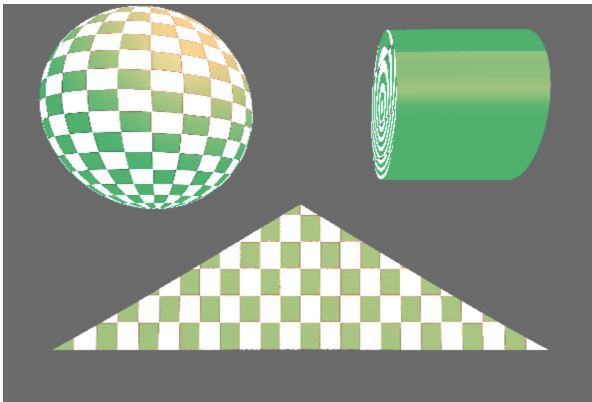
In each object's material input, I included two additional fields: a boolean `hastexture` and a string `texturepath` that holds the relative path to the image texture. To convert JPG images into PPM textures, I used [Convertio's JPG to PPM converter](#) to change JPG files into P6 PPM format, then used [Thomas EB Smith's PPM Converter](#) to

transform these P6 PPM files into P3 PPM format. In the raytracer.cpp file, I built a readPPM function using fstream to read the P3 image and write it to a 2D array, where each index in the array stored a vec3 corresponding to its position on the image texture.

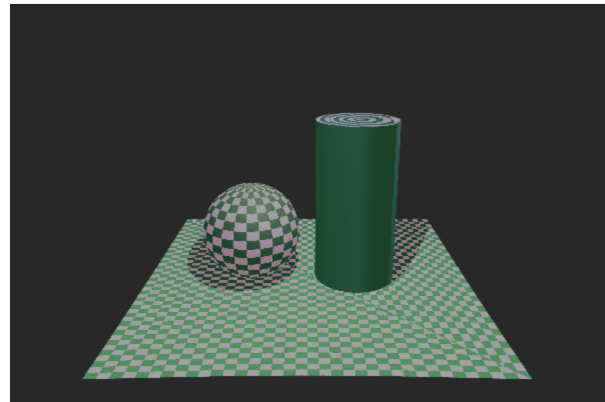
When setting material parameters for an object, if hastexture was true, I called readPPM in the setMaterialParameters function. Then, I added the output of readPPM to the texture field in the material class.

I also added a uvmap function to my sphere, cylinder and triangle classes. This function converts the 3D intersection point on each object into a 2D point on the image texture. Copilot helped implement code for UV mapping on triangles and spheres with minimal adjustments, but the UV mapping code for the body of the cylinders proved difficult to debug.

If an object was hit, I set the hit\_record's texturecoordinates to the vec3 output from the uvmap function. Then, in the Blinn-Phong shading calculations for the intersection point, I used these texturecoordinates to get the pixel colour for the point of intersection from the image texture, adding the image texture as the ambient colour for the shape, and continuing with the rest of the Blinn-Phong pipeline.



(a) Test image with checkerboard texture and phong shading applied



(b) simple\_phong.json with added checkerboard texture

## LLM Output

I used ChatGPT to generate code for the readPPM function, which did not require adjustment. I used ChatGPT to generate the initial header for the uvmap function, and alternated between ChatGPT and Copilot to generate outputs for the uvmap functions for the triangular planes, sphere, cylinder body and cylindrical caps, until I found a function that produced the most accurate mapping.

## Modifications

Initially, I had errors with the texture-mapped coordinates exceeding the bounds of the texture image, so I added modulo operators to make sure that the image texture stayed in bounds.

## **8. Tone Mapping**

I clamped the values of specular, ambient and diffuse outputs before adding them to the total pixel colour for the ray, and clamped the total pixel colour again after averaging the colour of all samples. In the writeColor function, I scaled the RGB values from the [0,1] range to the [0,255] range - implementing a linear tone mapper.

### **LLM Input:**

None - I wrote the clamp function based on the clamp function in the [LRTOW] tutorial.

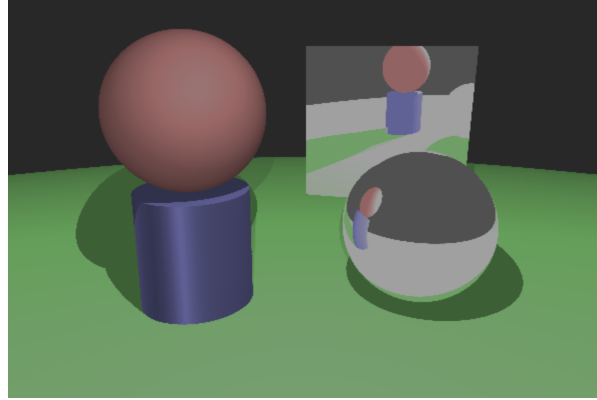
### **Modifications:**

I clamped the diffuse, specular and ambient components within the Blinn-Phong shader. I also made clamp an inline function in the vec3 class, as opposed to the color.h header where it was initially placed in the [LRTOW] tutorial.

## **9. Reflection**

After calculating the ambient light contribution in the Blinn-Phong shader rendering loop, I calculated the reflective contribution while iterating through the light sources in the scene. If the material at the point of intersection was reflective, I used the reflect function in vec3 to calculate the direction the reflected ray would travel. Then, I created a new reflection ray using the point of intersection as the origin and the direction of reflection as the direction, and called the rayColor function for my scene again, this time starting with this new outward ray.

The rayColor function is recursive, continuing to process the ray for each reflective surface it hits. It stops when the ray lands on a non-reflective surface or when the ray has exceeded more than the “nbounces” limit set in the JSON input. At every point of intersection, the rayColor function computes the specular, ambient, diffuse, reflective and refractive components of the surface it reaches.



(b) mirror\_image.json rendered with 8 bounces

## LLM Input

I used GitHub Copilot to fill in the function definition provided the header checking for the reflective component inside the Blinn-Phong rendering loop.

## Modifications

Initially, I had placed the code to calculate reflectivity and refractivity after the code to compute the specular and reflective components for each pixel. This was an issue because the computation for the shadow skipped calculating the specular or diffuse components of the ray if the point of intersection was in shadow. Moving the calculations for reflectivity before shadows were calculated for the object helped resolve this issue. I also added a double called `localContribution` to calculate how much the ambient colour should contribute to the object render, so that objects could be reflective and have an ambient base colour.

## 10. Refraction

After calculating the reflective component, I checked whether the material at the point of intersection was refractive. If so, I used Snell's Law to calculate the refracted direction of a ray beginning at the point of intersection. Like the reflection calculation, I created a new transmission ray using the point of intersection as the origin and the direction of refraction as the direction, and called the `rayColor` function for my scene on the new outward ray. Thus, the Blinn-Phong shader would recursively compute refraction until a non-refractive surface was hit or the depth of the ray exceeded "nbounces". After the refraction ray returned a colour value, I clamped the final `pixelColor` based on the reflectivity of the ray, as the transmittance value of the ray must be equal to  $1.0 - \text{reflectivity of the ray}$ .

## LLM Input/Modifications

Initially, I used ChatGPT to generate code to calculate reflection inside the Blinn-Phong rendering loop, but the response it returned was too bulky. So, I asked ChatGPT to generate function code for Snell's law, which I then moved outside the camera.h class into vec3.h for easier legibility. Once I had done so, I used Copilot to regenerate code for computing refraction using Snell's law in the vec3 header file.

## **11. Bounding Volume Hierarchy as Acceleration Structure**

Unlike the other features where I was able to reference the "Learn Ray Tracing in One Weekend" tutorial, I implemented the bounding volume hierarchy almost entirely based on conversations with ChatGPT. I started by asking ChatGPT general questions about the implementation of the bounding volume hierarchy, and determined that it would be best to create two new classes for the acceleration structure.

First, I created a BoundingBox class that stored a vector of triangles, could run hit checks on all triangles it encapsulated, and could compute the minimum dimensions needed of the bounding box for it to encapsulate all triangles stored inside the vector for the class. Then, I created a BVHNode class as a child of the abstract Shape class. I wanted BVHNode to be a Shape object so that it could be pushed back into the list of objects in the scene and iterated over via the hit function inside the Scene class.

Like the triangle, circle and cylinder shapes, BVHNode implemented a hit function, which calculated if the current ray was intersecting the axis-aligned bounding box declared from the minimum and maximum points of the BVHNode. If the ray intersected the root bounding box, BVHNode's hit function would recurse through its left and right subtrees until either no leaf nodes were hit, or one or more leaf nodes were hit. If the leaf node was hit, the hit function for BVHNode would call the hit function for the boundingBox corresponding to that leaf node.

### **LLM Input:**

I heavily relied on LLM outputs to generate code for this functionality. After ChatGPT generated headers for the boundingbox class and BVHNode struct, I provided ChatGPT another round of queries to flesh out the functionalities for the functions declared within each class. Because I relied heavily on LLM output to write this functionality, it was difficult to debug, and I was not able to integrate a bounding volume hierarchy with my code before the coursework deadline.

### **Modifications**

Because I wanted to treat the BVHNode as a hittable object within my scene, I declared BVHNode as a class inheriting from the default shape rather than as a struct. I also declared the boundingBox class as an inheritance of the default shape class so that I could implement the hit function to iterate through all triangles stored in the triangles field.

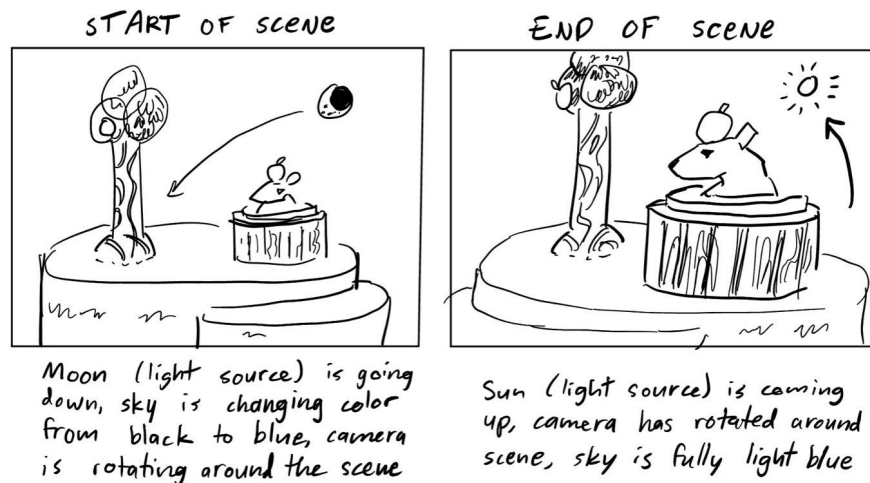
## Video

### Conception and Screenplay

For my video, I wanted to model a simple scene with textures and objects similar to those in Animal Crossing: New Leaf. I was also inspired by videos of capybara in onsen, and thought it would be straightforward to create a low-poly model of a capybara similar to one I saw created by “sneepsnorp3d” on TikTok.

My original idea for the scene included a basic orange tree modelled from 5 spheres and a cylinder, a ground plane modelled from two cylinders, a low-poly capybara modelled from triangular polygons, a moon modelled from a sphere, and a bathtub for the capybara modelled from two cylinders - a reflective cylinder for the water, and an opaque cylinder for the bucket containing the water.

I chose this scene because it would let me display several features of the raytracer - textures on a sphere, cylinder and triangle, reflection and refraction on spheres and circles, and the ability to render more complex polygonal shapes (despite being low-poly, the capybara head was modelled with 79 polygons). By adding a point light to the moon's position, I could also model moving lights within the scene.

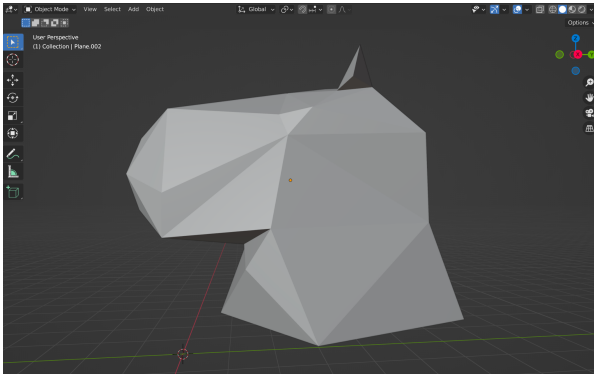


(a) Initial sketch of the scene showing camera movement

## Modelling

I modelled a simple capybara character using 3D polygons in Blender, and converted my model to a JSON input using the Blender Mesh to Triangle Python script written by my classmate Shuyuan Zhang. I designed a simple, cartoonish fur texture for the scene using Procreate, converted this texture to a PPM P3 image with the online

converters mentioned in the Textures section, and included this image texture in my scene file.



(a) Low-poly capybara head in Blender



(b) Fur texture for capybara

## Rendering

Unfortunately, because I was unable to debug the bounding-volume hierarchy for the capybara model before the coursework deadline, I was unable to create the final rendered scene for this project that I had envisioned. Instead of debugging the bounding-volume hierarchy necessary to generate this scene, I chose to spend my time on improving my image texture calculations and fleshing out the report before the coursework deadline.



## **Additional Comments**

Based on a suggestion from Sean Memery on Piazza, I used the `nlohmann/json` JSON parser library in `raytracer.cpp` to read the provided json scenes into the raytracer.