CG: Rendering - Coursework 2

Neel Amonkar

26th November, 2023

1 Basic raytracer features

1.1 Feature 1 - Image writing (ppm format)

The image writing code was based off of, like many other things, the Raytracing in One Weekend book [4] - specifically, Listing 3. The first iteration of this code was copied directly into the executable C++ file (then called main.cpp) - later, it was split into the writeColor method in the Color class, and the writeToFile method in the Image class. For the latter, Github Copilot auto-filled in the function contents after I'd put down the function signature - it took the "filename" parameter as a cue to generate the code for opening the filestream, and referenced the code in the executable C++ file for the rest.

1.2 Feature 2 - Camera implementation with coordinate transformation

Again, the initial implementation of this was taken straight from [4], specifically Listing 9. Prior to this, I implemented the Vec3, Ray and Color classes by basically copy-pasting the definitions from the book. The camera code was copied into the executable C++ file for the first execution, but after that I began refactoring it into its own class. I put down all the camera parameters listed in the JSON files as private attributes, and Copilot was able to generate the boilerplate constructor functions as inline suggestions. I then wrote the signature for a private "initialise" function (in which the viewport and pixel coordinates in the world frame were calculated), and Copilot filled the rest using the ROTW code in the main file.

Later on, when extending the implementation to allow for arbitrary centres, up-vectors and look-at points, I initially used ChatGPT to generate a method for deriving the rotation matrix (represented as a 2D array of type double) between 2 vectors, and Copilot inline suggestions to convert the method to fit my implementation. However, while it performed well initially, it turned out that floating point errors would lead the result to be slightly wrong. I went back to check ROTW and it turned out that the functionality had already been accounted for, in a much less complicated way, so I used that code in my Camera class and deleted the rotationMatrix method (which is why it isn't mentioned in the LLM Responses JSON).

1.3 Feature 3 - Intersection tests (spheres, triangles, cylinders)

I began with an abstract Shape class, defined in shape.h. Copilot was really helpful in generating the boilerplate here - I was able to quickly declare an abstract intersection method taking a Ray as input and returning a boolean (later refactored to be the t-value at which the intersection occurs).

As for the individual classes, the Sphere intersection was taken from Listing 11 in [4], later manually refactored to return the smallest root of the quadratic equation defined. For the Triangle intersection, I initially followed a tutorial [8] to implement it from scratch, since I wanted to learn the principle. However, it wasn't working, so I gave up and used an implementation of the MT algorithm from Wikipedia [14], adapted line-by-line by Copilot's autocomplete tool to fit my implementation of the code (and manually extended later to return the t value). Finally, for the Cylinder intersection, I found a tutorial for the curved surface intersection [1], which I began implementing (and somehow Copilot was able to infer the rest despite the chat insisting it had no access to my browser pages). I found a separate tutorial for ray-disk intersections [9] to handle the caps (which, yet again, when I began implementing it manually, Copilot was able to offer accurate suggestions for each step).

1.4 Feature 4 - Binary image writing (intersection/no intersection)

With the boolean versions of the intersection tests implemented, I was able to test binary image writing. My initial tests involved manually instantiating Shape, Scene and Camera objects, but this obviously quickly proved to be inefficient. I therefore imported the nlohmann::json library and wrote 2 methods: the scene parsing done manually, the camera parsing done line-by-line with Copilot. The rayColor method actually returning the colour at each pixel was, again, taken from Listing 11 in [4] - it needed no modification at all for the binary case. Two examples of rendered binary scenes can be seen in Figure 1 and 2 (although the objects in the latter are largely obscured).

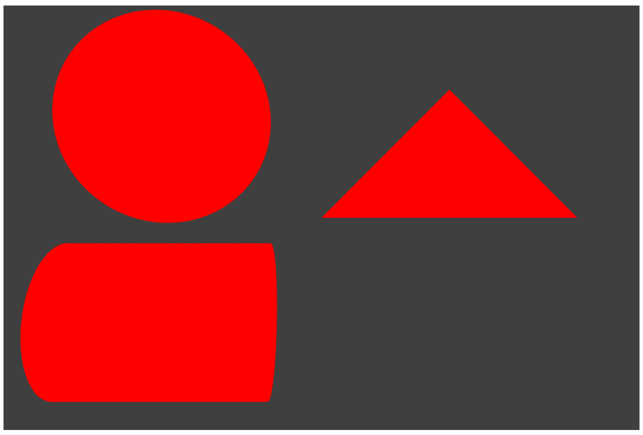

Figure 1: The binary primitives render

1.5 Feature 5 - Blinn-Phong shading

For the Blinn-Phong shading, I had to extend my code considerably. First off, since the calculation of the colour requires the coordinates of the point, the intersection methods had to be refactored to return the t-value closest to the camera (thus allowing you, in conjunction with the Ray, to calculate the point coordinates). This also required a refactoring of the code iterating through the shapes and checking intersections: this was initially just a linear search through a vector of Shape objects that returned the first recorded intersection point. However, this was obviously flawed: the order of shapes in the vector was completely arbitrary, after all. This was an easy refactor: I simply stored the point and the shape corresponding to the minimum t in temporary objects.

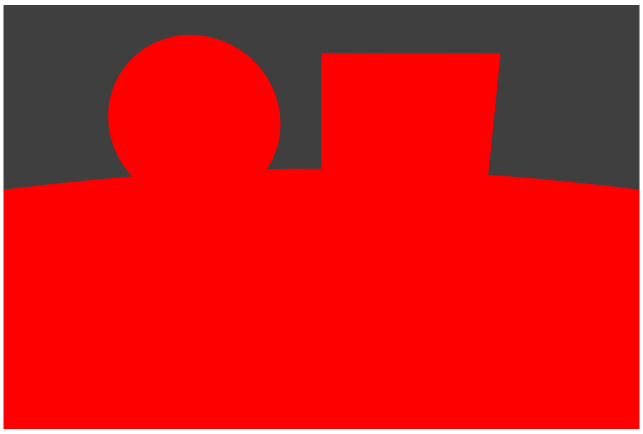
Figure 2: The binary scene render

Secondly, each shape needed a method to return the normal vector at a point on its surface. As such, Copilot was able to generate the boilerplate abstract method getNormal in the Shape class, as well as the simple implementations in Sphere and Triangle. As for the Cylinder implementation, I found a StackOverflow answer [12] covering points both at the top and bottom caps as well as on the curved surface - this, too, was somehow implemented line-by-line by Copilot.

Next, a Material class had to be defined to store the parameters from the provided JSON files. It was largely simply a matter of defining the appropriate attributes and creating getters and setters, which was done easily by Copilot. The scene-parsing method in renderer.cpp was manually extended to account for the potential existence of a material attribute for each primitive in the JSON. The same applies for the PointLight class - there's not much to be executed there, the object just serves as a repository of information.

Finally, the actual Blinn-Phong shading method had to be implemented in the rayColor method in renderer.cpp. To start with, I used the Wikipedia pages for Phong [15] and Blinn-Phong [13] shading to understand the form of the equation. The diffuse and specular terms were easily implemented - Copilot autocomplete came in handy for line-by-line completions, but it was never able to generate all the code at once, so the Wikipedia articles did come in handy as a reference. I was initially confused as to why my test renders were mostly in shadow, until I realised it was because I hadn't included an ambient light term. Since this wasn't defined in the file, I settled on using the diffuse color multiplied by an arbitrary scalar 0.4 - obviously, this isn't how ambient light works in the real world, and the illusion falls apart as soon as you consider a light that isn't a shade of white, but it worked well enough for replicating the sample renders.

I primarily used the scene shown in Figure 3 to test my implementation - the render here does additionally have shadows implemented.

1.6 Feature 6 - Shadows

This was primarily implemented using slides from the Ohio State University [10]. The algorithm presented there is simple: draw a ray from the point to all light sources, if said ray intersects any objects, the point is in shadow. Subsequently, the diffuse and specular sections of the BP method are skipped entirely. The initial boolean implementation was straightforward - to avoid shadow acne, instead of biasing the ray origin towards the light source, I simply passed the shape index of the source object to the method, so that the object never checks if its shadow ray is intersecting itself.

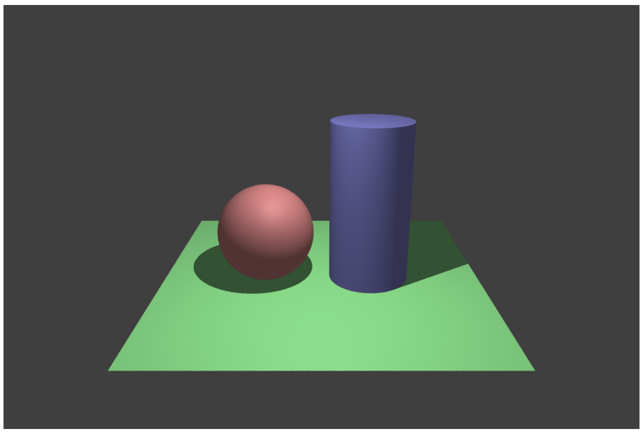

Figure 3: The simple phong scene with shadows

To account for shadows formed by multiple light sources, I altered the method to check for shadows from only **one** light source, passed as a parameter, rather than iterating through all the light sources. Then, in the light source loop in the rayColor phong code, if the point was in shadow from multiple light sources, I would multiply the diffuse colour by another arbitrary scalar for each additional light source, such that that region of shadow would get darker and darker. This was largely manually done, though of course Copilot autocompletes came in handy.

The shadow implementation can be seen in Figure 3, while the case for multiple light sources is shown in Figure 4.

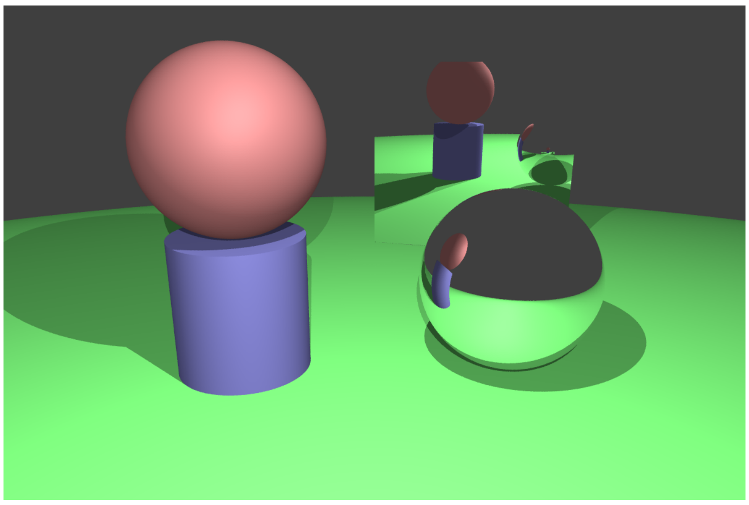

Figure 4: A more complex scene shaded with Blinn-Phong

1.7 Feature 7 - Textures (on spheres, triangles, cylinders)

This section was where I needed the most assistance from ChatGPT.

To start with, I wanted to define an Image class that could be used for both reading in image files (for textures) and outputting renders. The output function had already been defined beforehand, but reading in P3 PPM files was quite the challenge. Copilot autocomplete was incredibly helpful - I was able to define the phases of reading the file (max colour value, then resolution, then pixel data) either implicitly with variable names or explicitly with comments, and then Copilot was able to generate the actual code to perform the file input with minimal changes required.

I then extended the Material class to potentially include a texture Image in the constructor. Finally, I implemented an abstract method getDiffuseColor in the Shape class, overridden in the child classes. The idea here was that if the material contained a texture, it would override any diffuse colour that was separately defined. The method would take a point on the surface as input and calculate UV coordinates in the texture space (both being defined in the interval [0-1]), returning the colour of the pixel at that position in the image.

For the sphere, I found a tutorial online [11] that could be easily implemented (and slightly modified to stop the image being projected upside down). The Triangle was a bit more difficult: I consulted ChatGPT to understand the method, and found that you could find the UV coordinates by representing the point in barycentric coordinates (i.e. in terms of the 3 vertices) and using those as multipliers for the (assumed to be known) UV coordinates of the 3 vertices. It didn't quite seem to grasp in that conversation that I was considering a triangle in 3D, so a separate query to ChatGPT yielded an algorithm for finding the barycentric coordinates that was easily converted by Copilot to fit the implementation. I did have to make assumptions about the UV coordinates of the three vertices - since the function for the normal seemed to assume a clockwise order for the vertices, I went the same way, defining vertex 1 as the top-left corner of the texture.

Finally, the cylinder. ChatGPT generated code that returned UV values outside the interval, so I used Copilot Chat to correct it and modified that so that the U coordinate of 0 corresponded to the top edge, rather than the bottom. The code that is currently in place probably won't work if the axis isn't vertically-aligned - however, this has not been tested.

During testing, I started out with fairly high-resolution textures (500x500). However, this ended up slowing down renders GREATLY (potentially down to the sluggish Image vector implementation in the Image class) - even a 129x193 image (as seen in Figure 5) slowed it down to the point that 1) iteratively testing and 2) rendering for the video would have taken considerably longer. As such, for testing, I used 32x32 pixel art images from [3]. For the video, I used 64x64 textures from [2], as can be seen in Figure 6.

1.8 Feature 8 - Tone mapping (linear)

This was trivially done - the large majority of the code was already included in ROTW in the image-writing method (mapping values in [0-1] to [0-255]) and I manually implemented a clamping function to deal with colours that happened to step outside the bounds somehow.

1.9 Feature 9 - Reflection

For both this and refraction, I used [7] as a reference. The reflection implementation was straightforward - I added an int parameter to the rayColor function representing the depth of the recursive call, so that the nBounces constraint could be satisfied, and the calculation of the reflection vector was straightforwardly given in the slides and easily generated by Copilot. There is another design choice here: a material being reflective means that the diffuse and specular shading is completely skipped, regardless of the reflectivity value.

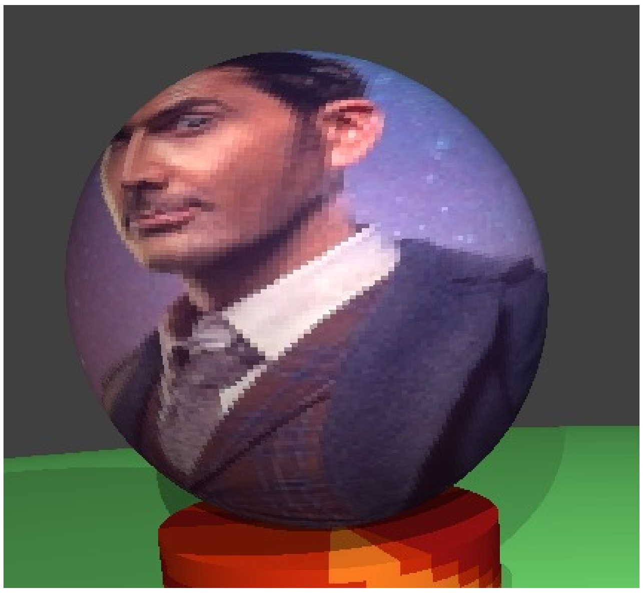

Figure 5: Noted Scottish actor, David ~~Fourteennant~~ Tennant, projected onto a 3D sphere

1.10 Feature 10 - Refraction

This was another implementation from the same source as above (helped by Copilot suggestions) with one augmentation - a boolean parameter added representing whether the origin of the ray is a refractive material or not, to account for the differing calculations for the case where the ray is entering a refractive object versus where it is exiting it. Here, again, an object being refractive means all other colour attributes are disregarded while shading. Additionally, the implementation here only accounts for either total internal reflection or the ray completely refracting: the physically-accurate case where some portion is reflected and some refracted is disregarded. I tried implementing a version of the Schlick approximation using listing 72 from [4] and Copilot's inline suggestions, but it didn't work, so I left it out.

Both reflection and refraction bias the origin of the ray towards its direction to avoid self-intersection. Reflective objects can be seen in Figure 4 and Figure 6 (the plane and the sphere), while a refractive sphere is demonstrated in the latter figure.

1.11 Feature 11 - Bounding volume hierarchy as an acceleration structure

I was unable to implement this fully in time for the submission. I began by defining a BoundingBox class representing the axis-aligned bounding box (AABB). Initially, I took inspiration from the implementation in Raytracing: The Next Week [5] - however, I decided to simplify it such that it was only the bottom left and top right points defining the box. The hit method for determining whether a ray intersects the box was derived using an answer on StackExchange [16], which was pasted into boundingbox.h, commented out, and converted to fit the implementation by Copilot.

The next task was finding ways to generate the bounding box for each primitive type. The sphere was straightforward enough, being implemented in the RTNW book. I consulted ChatGPT for the method of the triangle AABB, and began implementing it manually before Copilot cottoned on and auto-generated the rest. For the cylinder, I used a reference [6] - however, its implementation for the basic Vec3 and Cylinder classes was quite different, so I trusted Copilot to be able to fill in the gaps.

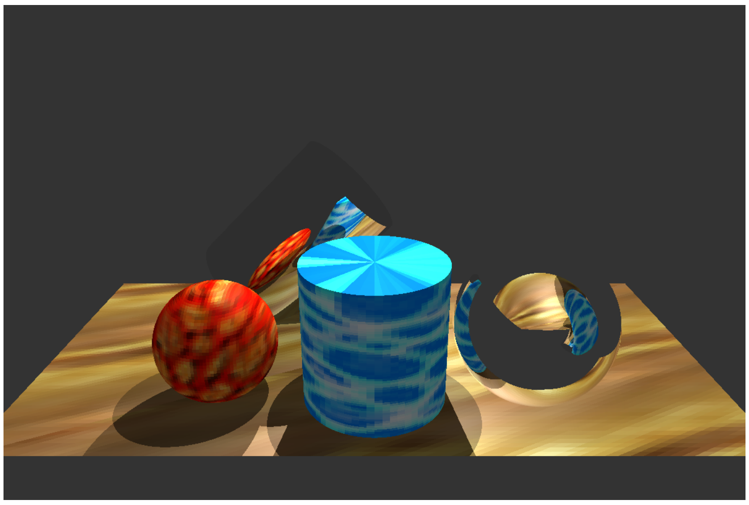

Figure 6: A frame from the video, demonstrating (among other things) texture mapping for all 3 primitive types

However, the bounding box definition is buggy - the X and Z values for the bottom left point are, for some reason, NaNs. I was unable to find an alternative method that worked in time.

For the bounding volume hierarchy itself, I prompted Copilot Chat to take a list of type BoundingBox and generate a BVHNode class - it generated the whole class plus an intersection method, which I altered to return the t-value and a pointer to the shape which registered the intersection, as both would be needed for Blinn-Phong. I also wrote functions to debug the tree, which seemed to work as intended - however, for some reason, the BVH would always log an intersection on the top-level AABB encompassing the whole scene, but then fail to intersect with either the left or right box entirely, meaning nothing in the scene would be rendered. While this could be caused by the buggy definition of the cylinder AABB, the sphere and triangle AABBs seemed to be working as expected. I was unable to figure out what the problem was in time for the submission.

2 Pathtracer features

2.1 Feature 1 - Antialiasing via multi-sampling pixels

This was achieved using the random number utility functions defined in listing 36 and the subpixel sampling functions defined in listing 40 in [4] - no intervention from Copilot or any other AI tools was required here. The feature is demonstrated in Figures 1, 2, 4 and 3 - the number of samples is set to 10.

(A note: the video does not have anti-aliasing turned on because the extra samples noticeably increased the render time of the standard test suite - adding textures to the mix could slow things down to a crawl.)

References

[1] Chris Dragan. *Raytracing shapes*. URL: https://hugi.scene.org/online/hugi24/coding%20graphics%20chris%20dragan%20raytracing%20shapes.htm.

[2] FacadeGaikan. *64X textures an overlays*. Nov. 2014. URL: https://opengameart.org/content/64x-textures-an-overlays.

[3] Jestan. *Pixel texture pack*. June 2019. URL: https://opengameart.org/content/pixel-texture-pack.

[4] Steve Hollasch Peter Shirley Trevor David Black. *Ray Tracing in One Weekend*. Aug. 2023. URL: https://raytracing.github.io/books/RayTracingInOneWeekend.html.

[5] Steve Hollasch Peter Shirley Trevor David Black. *Ray Tracing: The Next Week*. Aug. 2023. URL: https://raytracing.github.io/books/RayTracingTheNextWeek.html.

[6] Inigo Quilez. *Disk and Cylinder Bounding Box*. URL: https://iquilezles.org/articles/diskbbox/.

[7] *Reflection 2 - Department of Computer Science and Engineering - Ohio State University*. URL: https://web.cse.ohio-state.edu/~shen.94/681/Site/Slides_files/reflection_refraction.pdf.

[8] *Scratchapixel - Moller-Trumbore Ray-Triangle Intersection Tutorial*. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection.html.

[9] *Scratchapixel - Ray-Plane and Ray-Disk Intersection*. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-plane-and-ray-disk-intersection.html.

[10] *Shadows - Ohio State University*. URL: https://web.cse.ohio-state.edu/~shen.94/681/Site/Slides_files/shadow.pdf.

[11] *Spherical texture mapping*. URL: https://www.mvps.org/directx/articles/spheremap.htm.

[12] tyuan. *How can I compute normal on the surface of a cylinder?* URL: https://stackoverflow.com/a/65392348.

[13] Wikipedia contributors. *Blinn–Phong reflection model — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-November-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Blinn%E2%80%93Phong_reflection_model&oldid=1183441410.

[14] Wikipedia contributors. *Möller–Trumbore intersection algorithm — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-November-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm&oldid=1185403264.

[15] Wikipedia contributors. *Phong reflection model — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-November-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Phong_reflection_model&oldid=1133079828.

[16] zacharmarz. *Most efficient Aabb vs ray collision algorithms*. URL: https://gamedev.stackexchange.com/a/18459.