Computer Graphics: Rendering

Coursework Final Report

2900180

The following report will detail the features that I have implemented for the raytracer created in the coursework for Computer Graphics: Rendering. For the rest of this report, when I refer to coding assistance, I am referring to `Google Gemini 3 Pro`. These features include:

	Section	Status
1	Blender Exporter	Completed
2	Image Reader/Writer	Completed
3	Ray Intersection w. Acceleration	Completed
4	Whitted Style Ray Tracer	Completed, except refraction
5	Anti Aliasing	Completed
6	Textures	Completed
7	System Integration	Completed for parts that could be
8	Distributed RT	Incomplete
9	Len Effect	Incomplete
10	Timeliness	Only module 1 Submitted

Blender Exporter

The first feature that I developed was the exporter for blender. This basically took a scene as a `.blend` file and transformed it into a `.json` file that held all the essential information. This was functionality that was made primarily with AI as it understood the librarys used in the blender Python script very well. This was a relatively simple script that iterated over all the `bpy.data.objects` and extracted the key info from each of them. One of the struggles that I had was eventually when I needed to get the material information for shading, I found that it was difficult to change some of the corresponding material properties in Blender into the ones that I wanted for my C++ script. I found that getting values for k_d, k_a, and k_s needed for Blinn-Phong shading was very particular and the AI help I was getting gave very rough proxies of what values looked accurate. An example prompt here was:

```
Write me a Python script to run in the Blender scripting environment that
creates a json file representing the scene. Make sure to include point lights
(position and radiant intensity), all important details of any meshes
```

`including all transformations, and give all details of the camera in order to simulate it.`

This gave very successful results, however I had to slightly modify the script to include the 'up vector' of the camera, as this meant in my camera space rotations, it would be slightly off till I made this correction. To see examples of the output, check out ASCII/test1.json

Image Reader/Writer

The next feature developed was a PPM Image reader, updater and writer. For this, I primarily developed this by myself without any AI help. One of the prompts that I used for this was:

`What does the P3 mean at the start of a PPM file?`

In which it helped me understand the format of PPM in a much clearer sense (the answer to the above question is that P3 is the magic number representing the file type). Apart from this, this module was basically a CRUD for the images. In order to validate that I was doing this correctly, I used `gtest`, the Google testing software in order to write unit tests for this functionality, which my code passed. See the README on how to run these tests. Furthermore, all other examples of images shown in the report are examples for this feature working.

Ray Intersection with Acceleration

After importing the camera and all meshes, the next feature developed was to change the pixels of the camera into rays in the world space. This was done using the rotation matrices in the file and converting them into euler angles. This was confirmed by using a blender script (found in Blend/Ray_visualiser.py) to show the rays, see Figure 1 .An error that took a while to solve was that the camera fit (Horizontal or Vertical) would be automatically decided in Blender, and this had to be programmed into the C++.

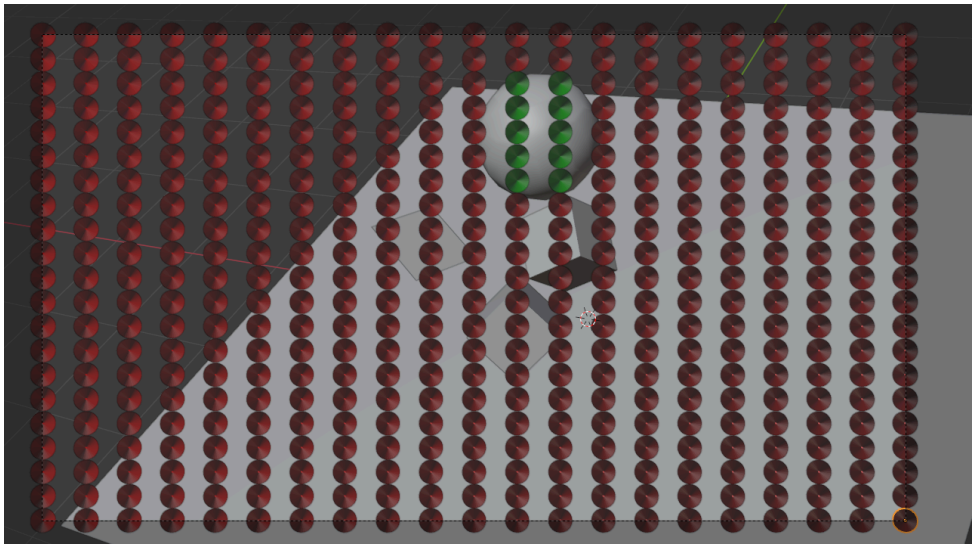
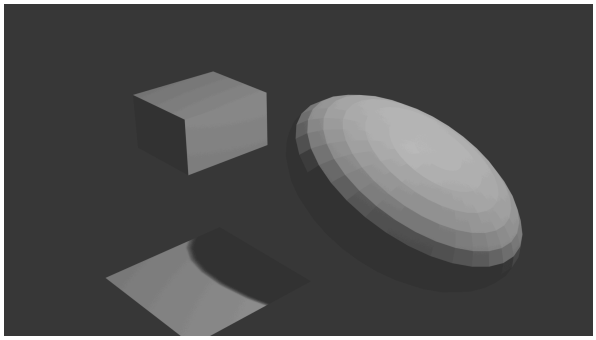


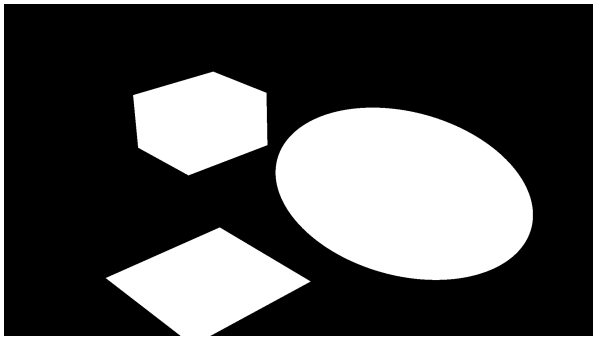
Figure 1: Pixels shown as rays, aligning with camera viewbox

After this was completed, the next feature was to write code that tested the intersection between the rays and each mesh in the scene. For cubes and spheres, this involved rotating the ray into the basis of the object, then running intersection tests with the transformed ray.

For the plane it was a lot more simple as a bounds check with the four corners of the plane were satisfactory. In order to check this was working, each pixel in the image could be transformed into a ray, and an intersection test with all objects could be run. Finally this could be written to a `.ppm` file to see if the intersections were working.



(a) Blender Render of created scene



(b) PPM file generated from module 2

In order to get the above working, I used prompts such as

```
How do I check for intersection between a ray and transformed cube? Give me a
plan for how to do this.
```

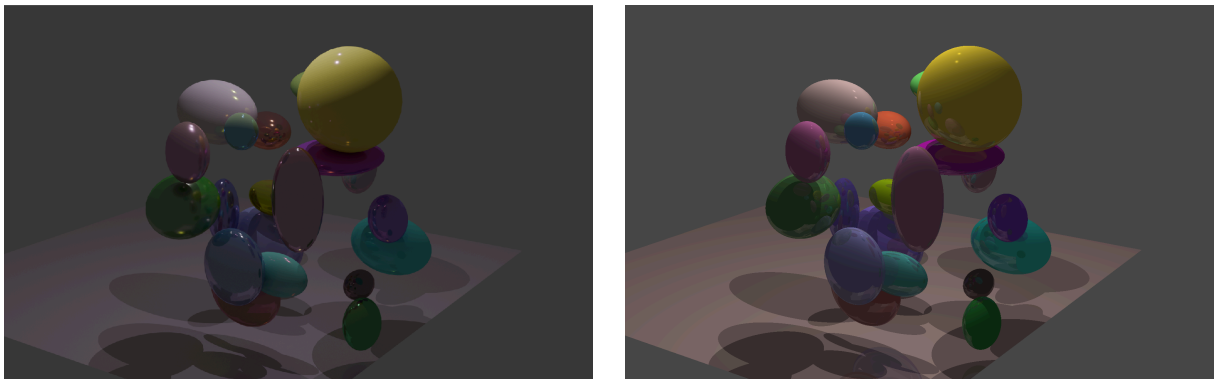
This allowed the AI to create a plan, and it also gave me articles to read in order to write the intersect functions. I also asked it to help plan the acceleration component of the function as well. This was the next step in the feature, in which I used a Bounding Box Hierarchy in order to skip checks in sections of the scene that the ray was not close to at all. To validate this, I created a file with a large amount of items and ran A/B tests, getting the following results.

```
-- TESTING NO HIERARCHY --------------
Execution time: 13354 ms
Amount of intersection tests: 103680000
---------------------------------------

-- TESTING HIERARCHY --------------
Execution time: 8121 ms
Amount of intersection tests: 110067359
---------------------------------------
```

Whitted style raytracing

The next feature that I built was a shade function, which I first had to build into the render exporter to get the material properties such as the colour, the reflectivity, transparancy, etc. This meant that after the intersection test was complete, the code would then figure out the shade based on the angle the camera was viewing it and where the light was. This function also included reflectivity and shadows, and gave the results shown in Figure 3



(a) Blender Render of test scene (b) PPM file generated

Figure 3: Comparison between blender rendering and my rendering

As you can see, this shows implementation of Blinn-Phong shading, with reflections and shadows. I think the main difference between the two is the reflected specular components, potentially having a different specular constant between the two renders.

The part that I have not implemented yet is refraction. This would involve altering the colour by the internal reflection (with changes due to changing index of refraction). I have implemented exporting the required material properties, however unfortunately the functionality within the shading function has not been developed.

Antialiasing

This was relativly simple to implement, instead of shooting one ray the code shoots multiple (decided from the command line). These shade values were then averaged to give a more precise measurement. I implemented this without the use of AI. The following code, found in src/Raytracer.cpp `render_image` function shows the use of antialiasing:

```
Eigen::Vector3f overall_shade = Eigen::Vector3f::Zero();
bool is_hit = false;
for (int sample_i = 0; sample_i < _ray_tracer_settings.amount_of_antialiasing_samples_per_pixel; sample_i++) {
    float sample_offset_x = distribution(generator);
    float sample_offset_y = distribution(generator);

    // lock into bounds of image
    float sample_px = std::min((float)px + sample_offset_x, (float)image.get_width()-1.0f);
    float sample_py = std::min((float)py + sample_offset_y, (float)image.get_height()-1.0f);

    Pixel p = image.get_pixel(sample_px, sample_py);
    Ray r = p.as_ray(_props);
    Hit h;

    if (_bbht->check_intersect(r, &h, &intersection_test_counter)) {
        Eigen::Vector3f s = shade(&h, _lights, &_props, 1.0f, _bbht, 0,
_ray_tracer_settings.max_depth_of_reflection_recursion);

        // shade operates in 0-1 shading region, convert back to rgb255
        overall_shade[0] += s[0] * 255.0f;
        overall_shade[1] += s[1] * 255.0f;
        overall_shade[2] += s[2] * 255.0f;

        is_hit = true;
    }
}

overall_shade /= _ray_tracer_settings.amount_of_antialiasing_samples_per_pixel; // finding the average
if (!is_hit) overall_shade += blender_background;
Colour new_colour{overall_shade[0], overall_shade[1], overall_shade[2]};

image.update_pixel(px, py, new_colour);
```

Textures

In term of the textures, the first step was to add the u, v coordinates of the image to the intersection test. The following prompt was used:

```
Given the current script I have for intersection <pasted script>, how would I
make it so the hit structure also returns a u,v coordinate for the texture?
```

This gave good results. I also had to update the blender script to return the absolute path to the PPM files that it was using for the textures. After this, when the shade function is running, the base colour can be selected by getting the corresponding pixel in the image and using that. This gave the following results:

(a) Blender Render of test scene (b) PPM file generated

Figure 4: Comparison between blender textures and my textures

In Figure 4, each mesh has the texture overlayed over it - comparing between the blender output and mine, it looks like mine has harsher lighting and slightly lighter colour, which I believe is a result of different material calculations between blender's code and mine (such as a different ambient coefficient). One of the difficulties that I had with this was getting the orientation of the image correct on the faces, however trial and error helped me work this out.

System Integration

I believe that the code that I implemented is very modular and all parts that I have completed have been integrated well. I think this is seen most clearly in the main file where the raytracer is created and controlled from the command line:

```
int main(int argc, char* argv[]) {
  RayTracer raytracer;

  raytracer.create_settings_from_command_args(argc, argv);
  raytracer.setup();

  raytracer.render_image();
  return 0;
}
```

In the create settings from command args function, all optional features (antialiasing, reflections, etc) have the ability to toggle on and off depending on preferences. The filenames for the input and output can also be entered this way as well.

Timeliness

In terms of the differences between what I submitted (only Module 1), they were primarily based around the change in details that I needed for the objects in the scene. In Module 1, my exporter in Blender (and thus reader in C++) only focused on the key characteristics of each item. For example with the Mesh, all material characteristics were ignored, and in the final iteration of the code this was updated. A further update was made to all the actual Mesh objects in C++ to account for this as well. Apart from this, no major changes were made.

Strengths and Weaknesses of using Coding Assistants.

The use of coding assistants for this assignments was both a positive and negative experience. In terms of the benefits, I found that using assistants meant that I could debug much quicker as it would very quickly sift through complex error messages and arrive at solutions. Furthermore, it gave suggestions that I believe helped progress my C++ skills, for example letting me know when to hand references into functions, as well as proper `const` correctness for attributes and other functions. I tried to use these suggestions as learning points for the future. In terms of negatives, I think that using multiple prompts over different days meant that it would not understand some of the assumptions that were made by times I had asked it questions. For example, in my shade function, I had the following line:

```
Eigen::Vector3f V = (P - props->location).normalized();
```

This represents the view vector, which in some cases the bot would assume were pointing *towards* the camera, and in other cases *away* from the camera. This misalignment meant that it would be assured of the following code, but it would not work. Another issue I found working with the code assistants was because of the more difficult maths involved (for example the ray intersection tests), it meant that I would rely too heavily on the help it would give, and struggle to debug any issues as I lacked the understanding of what some of the code was doing.