# COSC363 Raytracing Report

Liam Pribis 81326643

31st May 2020

# 1 Final Output

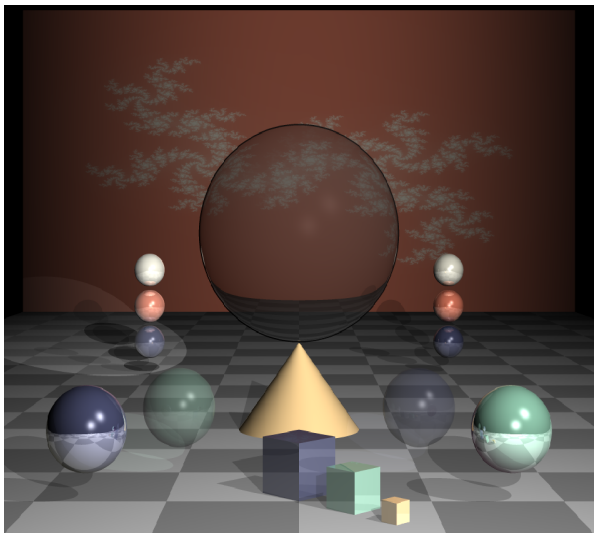

Figure 1: Raytraced output

# 2 Project architecture

The project is implemented in pure C, and the only library used is `glut`. This was done for a better understanding of the mechanics behind raytracing, because everything was rewritten from scratch. `glm` was not used. Instead, a `vec3_t` struct was implemented with the necessary operations to replace `glm`.

The raytracer uses a `scene_object_t` for every object in the scene. As inheritance is not available in C, `scene_object_t` contains a number of function pointers, which are used as a v-table. Each type of object points these function pointers to specific functions for that shape. When the tracer needs to calculate normals, for example, it will call the "`normal`" function pointer. This function pointer points to a specific implementation of the normal calculation for the shape (which will be different for spheres vs planes, etc.). The `scene_object_t` struct also includes a "data" void pointer, which can be used by any implementer of the scene object to store additional object-specific data.

Scene objects are contained in a `scene_t` struct. When new objects need to be added, the `scene_t` will automatically reallocate it memory to fit the new object and increase it's size tracker (this is similar to `vec<T>` in C++).

Most of the rendering logic is contained in `main.c`. The `trace` function calls the `get_lighting` function for each scene object to calculate the Phong lighting. The trace function also calls the `get_color` function from the scene object's v-table to obtain the ambient color of the object. This is done rather than directly acessing the objects color from the struct because it allows objects to override the `get_color` method and provide more complex implentations of retrieving the color. An example of this is the julia set plane, which provides a color based on a point in a specific julia set.

# 3 Extra features

## 3.1 Cone object

The cone object is defined by a base vector, a radius, and a height. The intersection point is calculated by solving the quadratic equation $at^2 + bt + c = 0$ using the following coefficients:

$$\alpha = (\frac{R}{H})^2$$

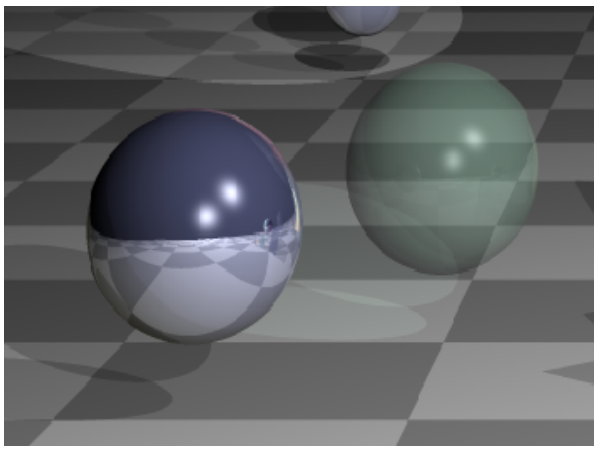

Figure 2: Tinted shadow

$$a = d_x^2 + d_z^2 - \alpha d_y^2$$
$$b = 2(d_x(p_x - x_c) + d_z(p_z - z_c) + \alpha d_y(H - p_y + y_c)$$
$$c = (p_x - x_c)^2 + (p_z - z_c)^2 - \alpha(H - p_y + y_c)^2$$

$R$ and $H$ are the radius and height of the cone. $d$ is the direction of the hit ray. $p$ is the starting point of the hit ray. Of the two solutions to the quardratic, the one with the smaller $t$ value was chosen (as this is the one the camera will see because it is closer). The hit point was then calculated from $t$ and the ray equation $p_0 + td$. If the hit point did not lie in the range from base to base+height of the cone, the hit was rejected.

Normals for the point $p$ were calculated using the following equations:

$$\alpha = \arctan(\frac{p_x - x_c}{p_z - z_c})$$

$$\theta = \arctan(\frac{R}{H})$$

$$\hat{n} = (\sin\alpha\cos\theta, \sin\theta, \cos\alpha\cos\theta)$$

### 3.2 Refraction

Refraction is calculated in the `trace` function. When the primary ray hits a refractive object, a secondary ray is sent through the object based on Snell's law. The hitpoint for this ray (which will be another point on the same object) then becomes the starting point for a secondary ray that is sent out in to the scene with a recursive call of the `trace` function. The direction for this ray is again calculated using Snell's law using the `refract(vec3_t incident, vec3_t normal, double eta)` function. The color of the object is then modified based on the result of the secondary ray trace and the object's refractive coefficient.

### 3.3 Tinted shadows for transparent objects

During shadow calculations, if a shadow ray hits a transparent object, the program will tint the point in shadow based on the color of the transparent object. This has the effect of colored transparent objects tinting objects (like a colored lens), rather than shadowing them. An example of this can be seen in figure 2. The sphere on the left has a normal shadow, while the transparent sphere on the right has a green tinted shadow. Note that since this is en extension of the overall shadow system, it is also affected by multiple light sources.

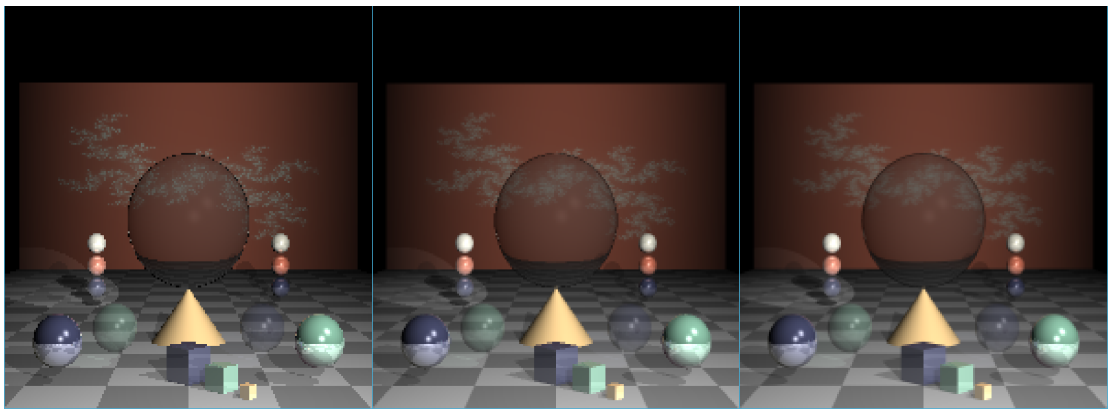

Figure 3: No anti-aliasing, $2 \times 2$ anti-aliasing, $8 \times 8$ anti-aliasing

### 3.4 Multiple light sources

The program supports an arbitrary amount of light sources. Additional light sources affect the Phong lighting model, and add extra shadows. When Phong lighting is calculated, each light is looped over, and the Phong lighting is calculated for that specific light. The resulting shading is then added to the object's ambient color. The effect of this can be seen in the render, where two specular highlights are visible for each specular object and two shadows can be seen from each object.

### 3.5 Spotlight

The spotlight is defined with a position, a direction, and a cone angle. Spotlight shading is accomplished in a similary way to how shadows are calculated. For each primary ray hit, a ray is sent towards the light source. First, the angle between the spotlight's direction and the hit-to-spotlight ray is calculated. $d_hl$ is the direction from the hit to the spotlight, and $d_l$ is the direction of the light.

$$d_{hl} \cdot -d_l = \|d_{hl}\| \, \|d_l\| \cos\theta$$

$$\theta = \arccos \frac{d_{hl} \cdot -d_l}{\|d_{hl}\| \, \|d_l\|}$$

Shadow rays are also sent from the hitpoint to the spotlight. If $\theta$ is less than the spotlight's cone angle and the shadow ray returned no hits that are in front of the spotlight, then the hit point has the spotlight's color added to it, because it is illuminated by the spotlight.

### 3.6 Anti-aliasing

The raytracer supports arbitrary amounts of anti-aliasing. This is accomplished in the `display` function. For each screen pixel, a buffer of size $n \times n$ is used, where $n \times n$ is the amount of subpixel samples. $n \times n$ rays are sent out, and their colors are stored in the buffer. For each screen quad, the buffer is averaged to obtain the resulting color. The amount of anti-aliasing ($n$) can be passed in to the raytracer through a command line argument. For example if `./out/main 4` is called, $4 \times 4 = 8$ subpixels will be sampled for each pixel. If no argument is passed, or the argument is not a valid number, the default value of $2 \times 2$ anti-aliasing will be used. The build script also passes on the argument to the main function when building, so `./build run 4` will also result in $4 \times 4$ anti-aliasing. A demonstration of this is shown in figure 3, using $200 \times 200$ screen quads each.

### 3.7 Procedurally generated pattern

The procedurally generated pattern is a julia set. This is calculated using complex number multiplication and an iterative algorithm. For the plane containing the julia set, the `get_color` function pointer is overwritten so that it

points to a function that calculates the color in a julia set based on the ray hitpoint location.

For each ray hit on the julia set plane, the hit position (as a complex number $z_x + z_y i$) is normalized to fall between $-r - ri$ and $r + ri$ using a linear mapping. First the hit point is converted to $st$ coordinates, and then the $st$ coordinates are mapped to the $r$-bounded rectangle. This was done to reuse the $st$ mapping code for a plane which had already been written. For this demonstration $r$ was chosen to be 1.3, but this value can be varied for different results. A constant starting value $c_x + c_y i = 0.32 + 0.5i$ was also chosen, which determines which julia set will be rendered. Then, an iterative algorithm was applied. Each iteration, $z = z^2 + c$ was calculated. The iterations continued until $z$ escaped a circle in the complex plane with radius r or the maximum number of iterations was exceeded, and the number of iterations was recorded. The number of iterations was used to determine the color produced at the hit point.

### 3.8 Fog

Fog was rendered by measuring the distance of a ray's hit point from the eye location. A fog distance value was chosen, and no objects were visible beyond this distance. The proportion of this distance that the ray was from the eye position was used to linearly interpolate between the normal ray color and the fog color. The ray hit x position was also weighted more heavily in this calculation do give a vignette effect where the left and right edges of the scene appeared darker.

## 4 Render time

The raytracer flushes the screen for each vertical line of the window traced, giving an indication of progress in rendering the scene. Progress updates are also printed to the console. 1000 subdivisions (opengl quads) were used in the x and z direction. With a default antialiasing value of $2 \times 2$, the scene renders in approximately 40 seconds on an Intel i5-4460. With no anti-aliasing, the scene renders in approximately 10 seconds. Other values of anti-aliasing will render in a time proportional to the number of subpixels used. In a future version, the render time could be considerably increased with multithreading. This would be simple because each pixel is independent of others, and can be easily implemented on multiple threads without data sharing issues.

## 5 Build process

A simple shell script build system was used for this project. The script runs `gcc` on every .c file in `/src/` and produces an .o file in `/out/`. It then links all .o files with `main.o`, and produces an executable `/out/main`.

### 5.1 Usage

- `./build clean` - remove all files in `/out/` directory.
- `./build build` - compile all files and link to main executable, but to not run. The executable is `/out/main`.
- `./build run` - Clean output directory, compile files, and run main executable. This will also pass one argument on to the main executable. For example `./build run 4` will compile and then run `./out/main 4`. The program may take an optional argument that specifies the anti-aliasing amount. Running `./build run 4` will enable 4x4 anti-aliasing on the program. If none is specified, 2x2 anti-aliasing will be applied.