The Inescapable Chamber of Time

By Lucas Redding

The Scene

The room consists of walls of all different colours that contain strange objects set on pedestals lit by a single harsh fluorescent light on the ceiling. The back wall immediately draws the eyes with its strange vibrant procedural patterns. Sitting on the chequered floor are three conical pedestals each displaying a unique sphere. It's immediately apparent that these three spheres must represent the past, present and future. The refractive sphere on the right shows the uncertainty of the future through the distortion of light that passes through it. However, its pinkish hue shows that hope can be found in the unknown. The centre, transparent sphere shows the present by clearly displaying what lies within, masked only by a hint of blue representing the melancholy of existence. The final sphere is textured to show the past probably based on the last meal you ate. The three balls are reflected back and forth by two parallel, gold trimmed mirrors that represent the infinite experience of time (**Fig 0**).

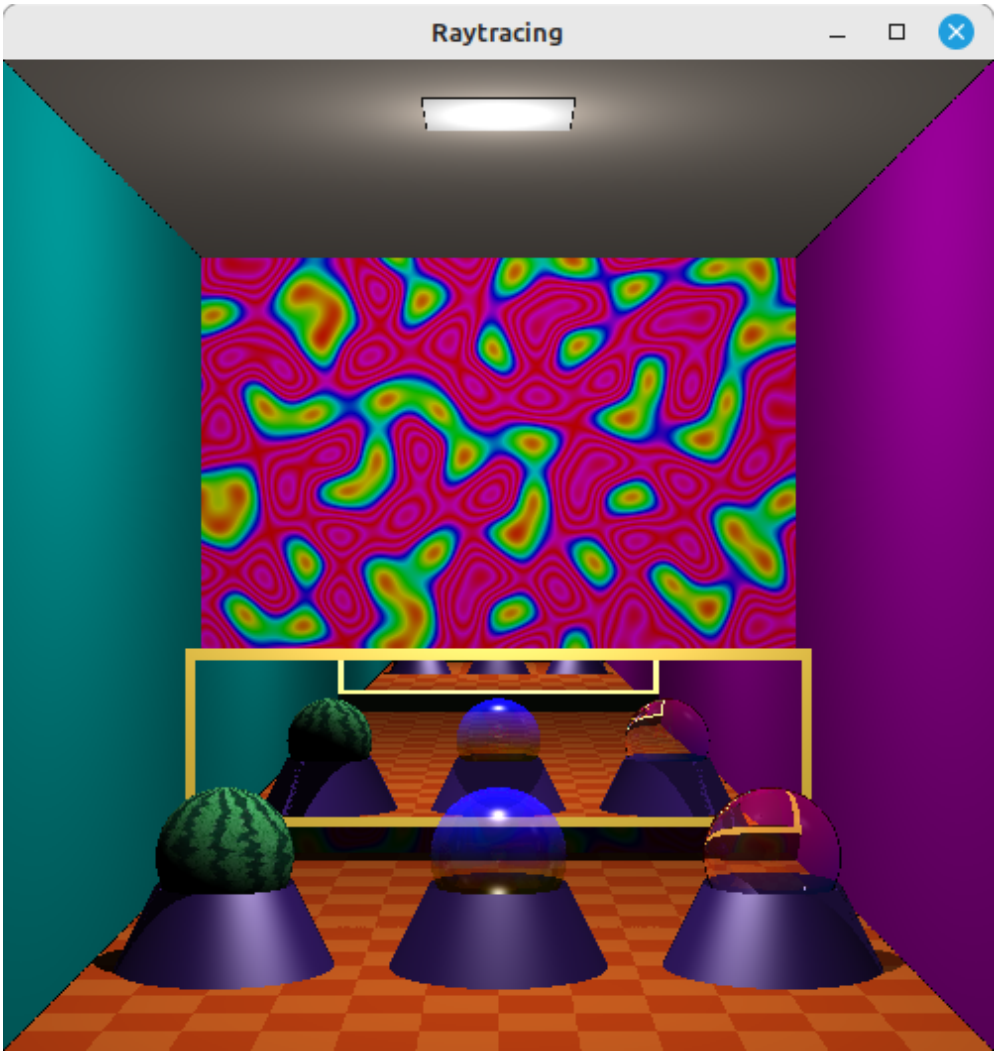


Fig 0 Full scene

The Features

Transparent Sphere

The blue tinted transparent sphere is treated differently than a special case of a transparent object. When a ray hits a transparent object, a new ray is cast from the point of collision on the object and in the direction of the original ray to ensure the light passes straight through (**fig 1**). As opposed to the usual transparent function which would involve finding the point where the ray exits the sphere to cast a new ray exiting the sphere, I have decided to trace the internal ray within the hollow sphere to get internal colours and allow for reflections within the sphere. This has the interesting result of causing the specular highlight on the outside of the sphere to be reflected within the sphere causing a 'caustic-like' effect within the sphere (**fig 2**).

```
if(obj->isTransparent() && step < MAX_STEPS) {
    float rho = obj->getTransparencyCoeff();
    Ray transparentRay(ray.hit, ray.dir);
    glm::vec3 transparentColor = trace(transparentRay, step + 1);
    color = color + (rho * transparentColor);
}
```

Fig 1.a Code for transparency

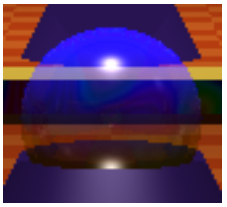


Fig 1.b Transparent sphere

Shadows

Shadows in the scene are computed by casting a shadow ray from the point that the original ray hit towards the light source. It then finds the closest point on an object that the shadow ray hits and if this point is closer than the light source then the colour of the original object is darkened. The object that cast the shadow (the shadow object) is also checked to see if it is transparent or reflective so that the original point can be darkened less if the shadow object lets some light pass through. This can be seen in the difference between the shadow cast by the opaque sphere vs the refractive sphere (**fig 2.a and fig 2.b**).

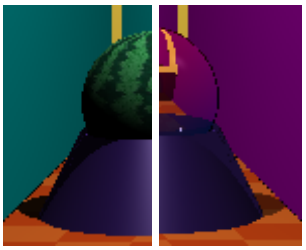


Fig 2.a and Fig 2.b Shadows cast by opaque and refractive spheres

Chequered Floor

The floor of the room is a plane coloured in two different shades of orange in a chequered pattern. The pattern is formed by dividing the plane in its x and z directions by a stripe width value to form horizontal and vertical stripes. Each point on the plane falls on a particular column and row which are each bound by modulo two as every second row will be the same and every second column will be the same. The colour is then set by whether the row and column values are different or whether they are the same causing the chequered pattern seen. The column value is inverted for hit values below 0 as otherwise the centre checkers will line up forming rectangles.

Truncated Cones

The three purple pedestals holding up the spheres are truncated cones. The equation for rendering a cone was built by combining the equation for a cone with the ray equation and solving for ‘t’ which is the distance along the ray’s direction vector, giving the hit point (**Appendix A**). As the ray can pass through the cone twice there can be multiple solutions for ‘t’, therefore the closer point to the camera is chosen as the hit point. The normal vector from the hit point on the cone is calculated through the equation given in the lecture slides. The cone is truncated by only returning a valid value for ‘t’ if the height of the hit point on the cone is below a certain value.

Refractive Sphere

The refractive sphere was displayed by changing the angle of the rays going through the sphere. A refracted vector is found by passing the initial ray’s direction, the normal vector of the object’s surface at the hit point and the refractive index of the material to the ‘glm’ library’s refract method. A new ray is cast within the refracted object using this refracted vector and the initial ray’s hit point to find where the ray exits the refractive object. The refract method is called once again to refract the internal ray out of the object using the outer surface normal of the object and one over the refractive index as the ray is now leaving the object. Finally an exit ray is cast from the object using the vector refracted out of the object which is then traced to find the desired colour value of the pixel. Using this system gives the effect of light bending through the object distorting the objects behind the refracted sphere.

Parallel Mirrors

In the room there are two parallel mirrors which each have a gold trim. One is placed at the very end of the room and the other is placed behind the camera such that there are multiple reflections between the two mirrors. Reflections are generated in a similar way to refraction by finding the normal vector of the surface and using the ‘glm’ reflect method to cast a new ray reflected off of the surface.

Anti-aliasing

In a basic ray tracing program each pixel is looped through and a ray is cast through the centre of each pixel to find a colour value for that pixel. This can lead to jagged edges where adjacent pixels have vastly different colours because the rays are hitting different objects or shadowed regions. Anti-aliasing removes these jagged edges by casting multiple rays through each pixel and finding the average colour value of that pixel which leads to smoother transitions between objects of different colours (**fig 3.a and fig 3.b**). This is implemented by dividing each pixel into four equal regions and casting a ray through the centre of each of these regions and then finding the average of the four colours by adding them and dividing by four.

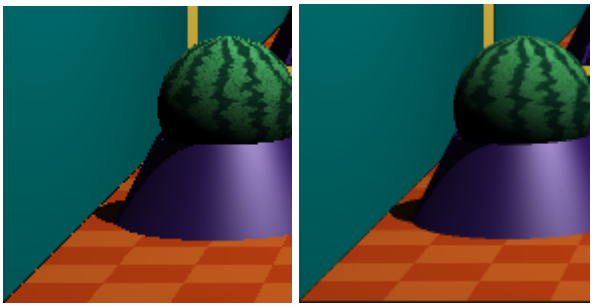


Fig 3.a No anti-aliasing **Fig 3.b** With anti-aliasing

Textured Sphere

The opaque sphere on the pedestal to the right of the room is textured with a watermelon texture. In order to put a 2d texture on a sphere the points on the sphere need to be mapped to coordinates on the texture. The texture coordinate ‘s’, which is related to the point on the sphere’s longitude, is found by first calculating the angle between the normal vector’s x and z components at the hit point of the ray which yields a value between -pi and pi. This is then divided by 2pi so that the range is between -0.5 and 0.5 and finally adding 0.5 to ensure the range is from 0 to 1 which is required for the texture mapping. The texture coordinate ‘t’ is calculated using the y component of the normal vector multiplied by 0.5 and then adding 0.5 to ensure the range is from 0 to 1 (**fig 4**). Then it is as simple as setting the colour of the object at the hit point to the value obtained from the texture at the given s,t coordinate.

```
glm::vec3 normal = obj->normal(ray.hit);
float texcoords = atan2(normal.x, normal.z) / (2*PI) + 0.5;
float texcoordt = normal.y * 0.5 + 0.5;
```

Fig 4 Equations for sphere texture mapping

Procedural Pattern

On the far wall of the scene is a procedural pattern formed using a noise function that controls a hue value. To generate the noise values a library called FastNoiseLite is used and set to use two dimensional openSimplex2 noise. The noise can then be queried using coordinates such as the x and y values of the ray hit point on the wall plane (scaled by some factor). The noise function returns a normalised value from 0 to 1. It is difficult to match a single value to an rgb colour so instead the hue factor from hsl is used. This allows a single number to represent a colour without information about the saturation or brightness so we can assume these to be set to full. The inverse of the hue is then multiplied by six such that the result is within the range of 0 to 6 (h) which is helpful as rgb can be thought of as a cube of colour values. We can then check which side of the cube the colour will be on using if statements setting one of the rgb components to 1, another to 0 and the last to x which is an expression in the hue to rgb calculation that determines the last rgb component using the hue.

Fog

Fog is rendered to the scene using a fog equation. First the lambda value is found by taking the z value of the ray hit point minus the start of the fog and dividing it by the difference between the start of the fog and the end of the fog. Lambda serves as the fraction of the

colour of the hit point that will be fog. Therefore, the colour of the point becomes one minus lamba of the object colour plus lambda of the fog colour. This gives a visualisation of fog gradually becoming more dense over distance from the camera (**fig 5**).

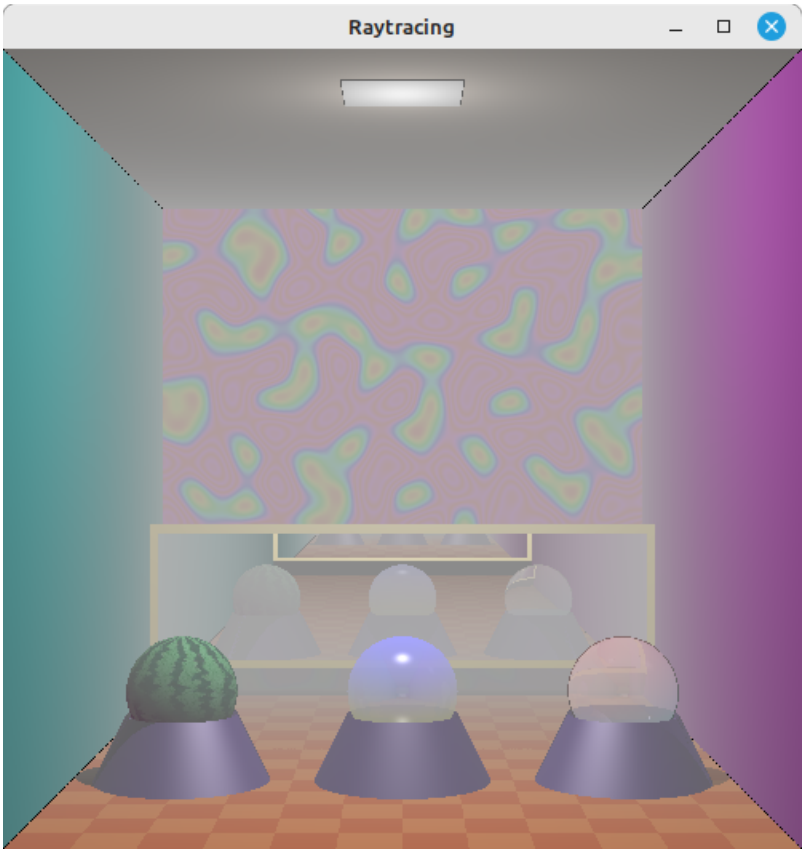


Fig 5 Scene with fog

Build Instructions

- Open QtCreator (preinstalled on lab computers.)
- Go to File -> Open File or Project -> navigate to project file -> select CMakeLists.txt
- Select the Desktop kit ->Configure Project
- Ensure the images/textures are placed in the build directory
- Click the monitor in the bottom left of QtCreator just above the green arrow
- Select Debug and then RayTracer.out
- Press the green arrow to build and run the project
- If there are errors in building and running then navigate to the projects tab on the left
- Then change the build directory to the source directory of the project
- Another solution is to move the image files into the build directory
- Running the program without fog and anti-aliasing on the lab computers should take roughly 4 seconds

References

https://learn.canterbury.ac.nz/course/view.php?id=19070&section=3

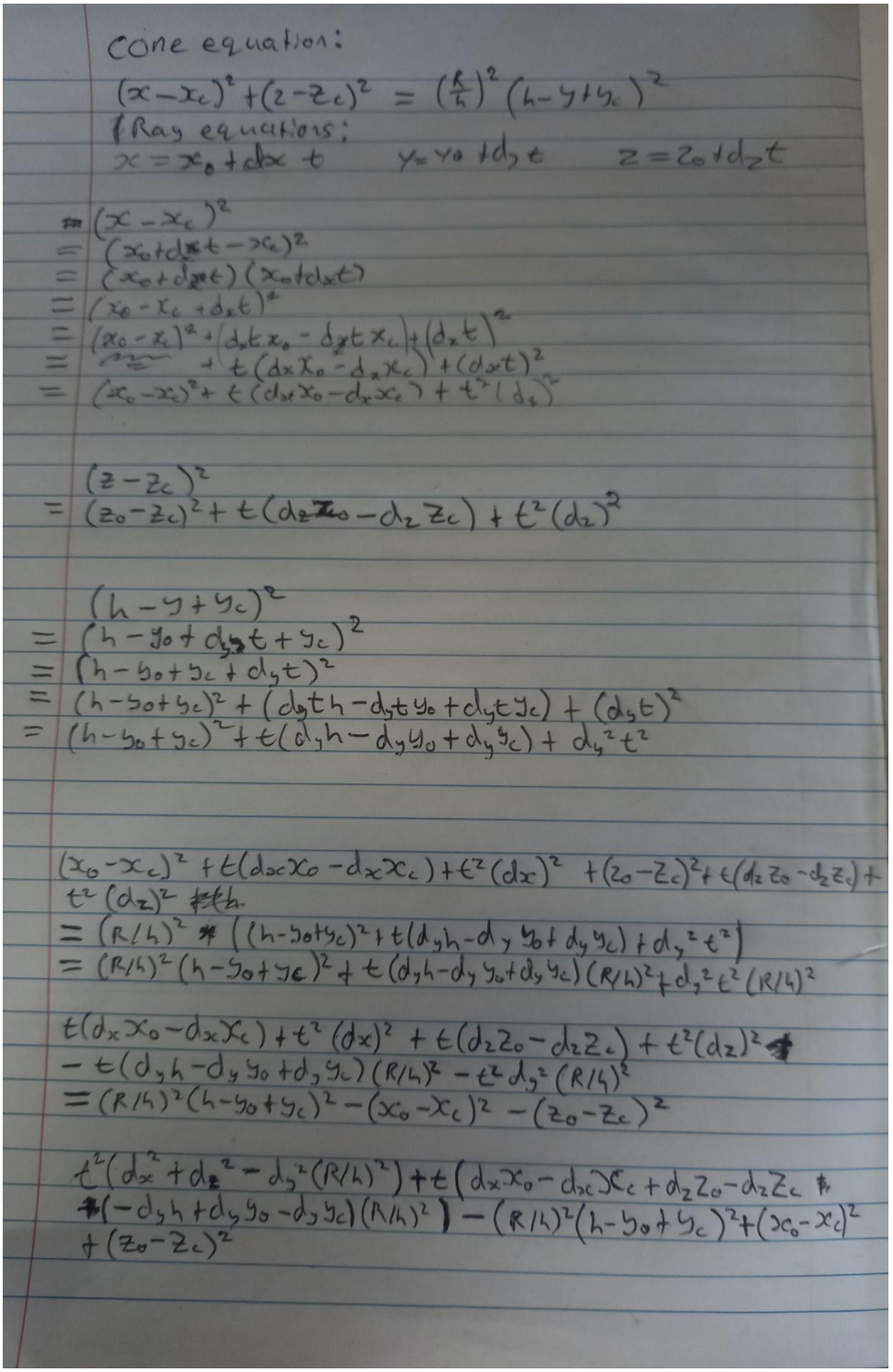
https://www.texturecan.com/details/111/
https://jonoshields.com/post/procedural-patterns
https://www.baeldung.com/cs/convert-color-hsl-rgb
https://github.com/Auburn/FastNoiseLite

Declaration

I declare that this assignment submisison represents my own work (except for allowed material provided in the course), and that ideas or extracts from other sources are properly acknowledged in the report. I have not allowed anyone to copy my work with the intention of passing it off as their own work.
Lucas Redding
51088903
01/06/23

Appendix

Appendix A - Cone/ray intersection equation rearranging



Cone equation:

$$(x-x_c)^2+(z-z_c)^2 = \left(\frac{r}{h}\right)^2 (h-y+y_c)^2$$

Ray equations:

$$x = x_0 + d_x t \qquad y = y_0 + d_y t \qquad z = z_0 + d_z t$$

$$\begin{aligned}
&(x-x_c)^2 \\
&= (x_0 + d_x t - x_c)^2 \\
&= (x_0 + d_x t)(x_0 + d_x t) \\
&= (x_0 - x_c + d_x t)^2 \\
&= (x_0 - x_c)^2 + (d_x t x_0 - d_x t x_c) + (d_x t)^2 \\
&= \quad + t(d_x x_0 - d_x x_c) + (d_x t)^2 \\
&= (x_0 - x_c)^2 + t(d_x x_0 - d_x x_c) + t^2 (d_x)^2
\end{aligned}$$

$$\begin{aligned}
&(z-z_c)^2 \\
&= (z_0 - z_c)^2 + t(d_z z_0 - d_z z_c) + t^2 (d_z)^2
\end{aligned}$$

$$\begin{aligned}
&(h-y+y_c)^2 \\
&= (h - y_0 + d_y t + y_c)^2 \\
&= (h - y_0 + y_c + d_y t)^2 \\
&= (h - y_0 + y_c)^2 + (d_y t h - d_y t y_0 + d_y t y_c) + (d_y t)^2 \\
&= (h - y_0 + y_c)^2 + t(d_y h - d_y y_0 + d_y y_c) + d_y^2 t^2
\end{aligned}$$

$$(x_0 - x_c)^2 + t(d_x x_0 - d_x x_c) + t^2 (d_x)^2 + (z_0 - z_c)^2 + t(d_z z_0 - d_z z_c) + t^2 (d_z)^2$$

$$\begin{aligned}
&= (R/h)^2 \left[(h - y_0 + y_c)^2 + t(d_y h - d_y y_0 + d_y y_c) + d_y^2 t^2 \right] \\
&= (R/h)^2 (h - y_0 + y_c)^2 + t(d_y h - d_y y_0 + d_y y_c)(R/h)^2 + d_y^2 t^2 (R/h)^2
\end{aligned}$$

$$\begin{aligned}
&t(d_x x_0 - d_x x_c) + t^2 (d_x)^2 + t(d_z z_0 - d_z z_c) + t^2 (d_z)^2 \\
&- t(d_y h - d_y y_0 + d_y y_c)(R/h)^2 - t^2 d_y^2 (R/h)^2 \\
&= (R/h)^2 (h - y_0 + y_c)^2 - (x_0 - x_c)^2 - (z_0 - z_c)^2
\end{aligned}$$

$$\begin{aligned}
&t^2 \left(d_x^2 + d_z^2 - d_y^2 (R/h)^2 \right) + t\left(d_x x_0 - d_x x_c + d_z z_0 - d_z z_c \right. \\
&\left. (-d_y h + d_y y_0 - d_y y_c)(R/h)^2 \right) - (R/h)^2 (h - y_0 + y_c)^2 + (x_0 - x_c)^2 \\
&+ (z_0 - z_c)^2
\end{aligned}$$