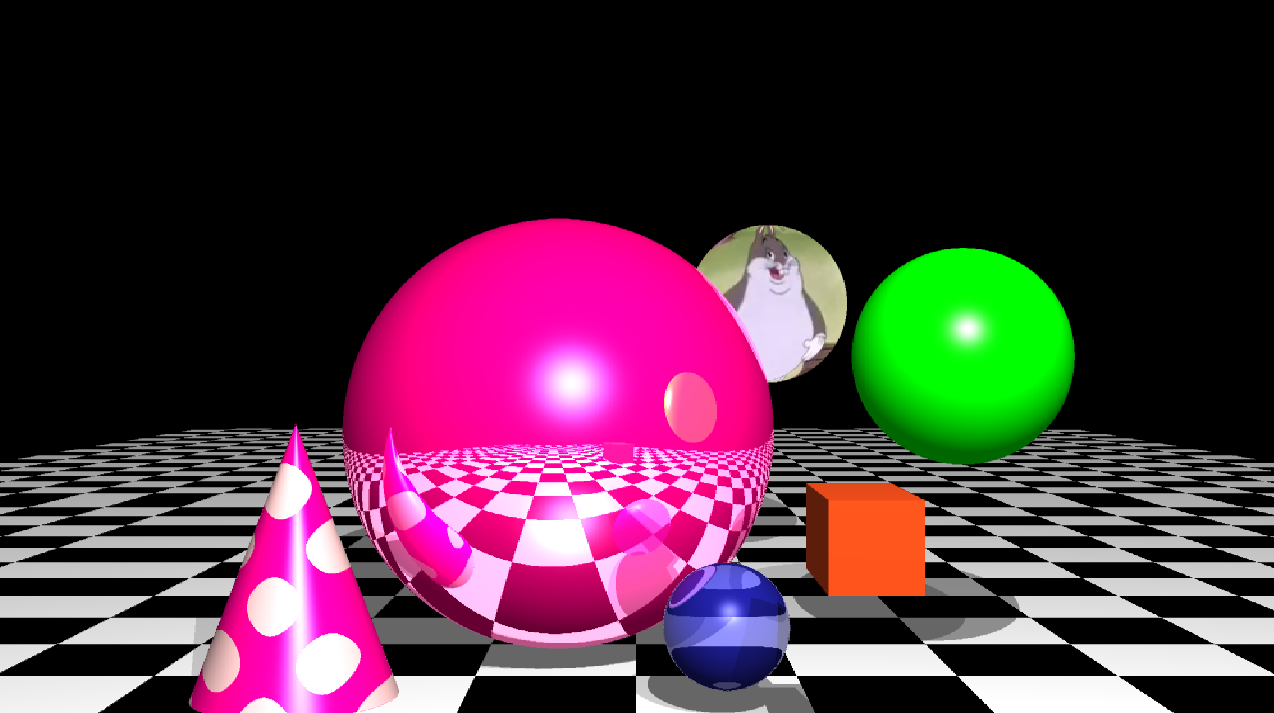
Ray Tracer

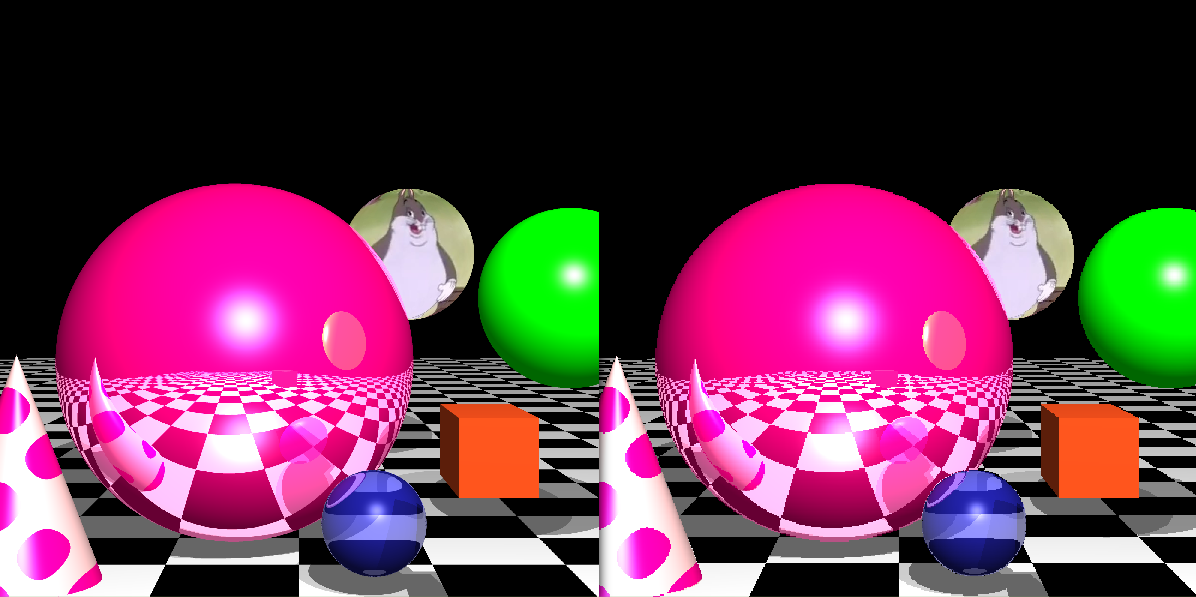


**Build command:** g++ -Wall -o "%e" "%f" Ray.cpp Sphere.cpp shape.cpp SceneObject.cpp Plane.cpp TextureBMP.cpp -lm -lGL -lGLU -lglut

**Extra Features**

**Anti Aliasing**

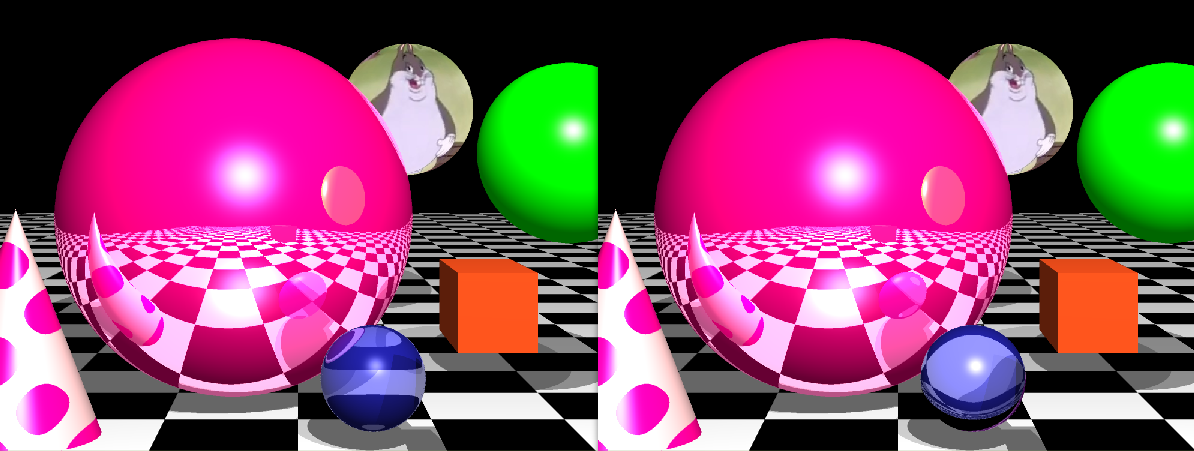
Anti-aliasing works by getting the average of the 4 cells for the ray that is being calculated this can make the image a lot smoother however takes significantly longer as 4 times as many calculations per ray the result of ray tracing is quite significant as without it things can seem quite pixelated.

*Fig 1: left AA on right AA of*

|  |
| --- |
| glm::vec3 colorSum(0);  glm::vec3 avg(0.25);    Ray ray = Ray(eye, glm::vec3(xp + pixel\_size \* 0.25, yp + pixel\_size \* 0.25, -EDIST));  ray.normalize();  colorSum+=trace(ray,1);    ray = Ray(eye, glm::vec3(xp + pixel\_size \* 0.25, yp + pixel\_size \* 0.75, -EDIST));  ray.normalize();  colorSum+=trace(ray,1);    ray = Ray(eye, glm::vec3(xp + pixel\_size \* 0.75, yp + pixel\_size \* 0.25, -EDIST));  ray.normalize();  colorSum+=trace(ray,1);    ray = Ray(eye, glm::vec3(xp + pixel\_size \* 0.75, yp + pixel\_size \* 0.75, -EDIST));  ray.normalize();  colorSum+=trace(ray,1);  colorSum\*= avg;  return colorSum; |
| *Code1: Code for anti Aliasing* |

**Refraction**

The refracting object is a small sphere at the bottom of the scene where a ray is traced through the object and a new one is created the glm refract function is called so not much code was written for this function as it was done. The ETA was set to 1/1.01 however when compared to an eta of 1/1.1 the effect can be exaggerated.



*Fig 2: right eta=1/1.01 and left eta=1/1.1*

**Cone**

The cone was created an extra file named shape.cpp

|  |
| --- |
| float r = sqrt((p.x-center.x)\*(p.x-center.x) + (p.z-center.z)\*(p.z-center.z));  glm::vec3 n = glm::vec3 (p.x-center.x, r\*(radius/height), p.z-center.z);  n = glm::normalize(n);  return n; |
| *Code2: ‘the equation’ to create the cone* |

**Image texturing**

A sphere was used to map a bmp image

****

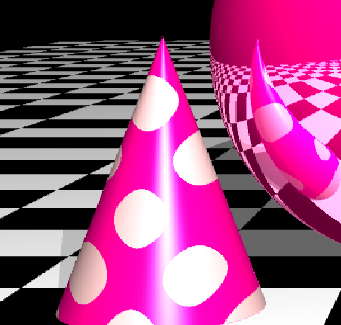
*Fig 3: an image mapped onto a sphere*

|  |
| --- |
| glm::vec3 center(10.0, 4.0, -110.0);  glm::vec3 diff = glm::normalize(ray.xpt-center);  float y=0.5+asin(diff.y)/M\_PI;  float x=(0.5-atan2(diff.z,diff.x)+M\_PI)/(2\*M\_PI);  colorSum = texture1.getColorAt(x, y); |
| *Code3: Shows how the image is mapped to the sphere (eddited for readablility)* |

**Procedural texturing**

The cone has a polka dot texture on it which has been generated using a procedure

|  |
| --- |
| if((int(sin(ray.xpt.x)+sin(ray.xpt.y))-10)%2 == 0){  sceneObjects[5]->setColor(glm::vec3(1,0,0.5625));}  else{  sceneObjects[5]->setColor(glm::vec3(0.957,0.762,0.761));} |
| *Code4: Shows the code for the texture generated on the cone* |



*Fig4: Procedural texture*

**Successes and Failures**

Everything worked well and straightforwardly with some small issues that were solved after a few hours. However a bigger problem I cam across was the procedural generation of textures. Initially I looked into Perlin noise however I noticed that is very complex and did not work even after trying days to implement it. I moved to a more basic Trigonometric way of doing things:

|  |
| --- |
| *int modx = (int)((ray.xpt.x + 50) /10) % 2;*  *int modz = (int)((ray.xpt.z + 200) /10) % 2;*  *int mody = (int)((ray.xpt.y + 50) /10) % 2;*    *if((sin(modx) && sin(modz)) || (!modx && !modz)){*  *sceneObjects[4]->setColor(glm::vec3(sin(modx),1,1));}*  *else{*  *sceneObjects[4]->setColor(glm::vec3(0,0,sin(mody)));}*  *}* |

*Although this did make an interesting pattern I moved away from it as it did seem messy and I ended up using the above method for creating the patter.*

**Bibliography**

Open GL mathematics: <https://glm.g-truc.net/0.9.4/glm-0.9.4.pdf>

Various mathematical formula: Stack overflow

Anti Aliasing: [https://www.khronos.org](https://www.khronos.org/)