Bradley Erskine's ray tracing project

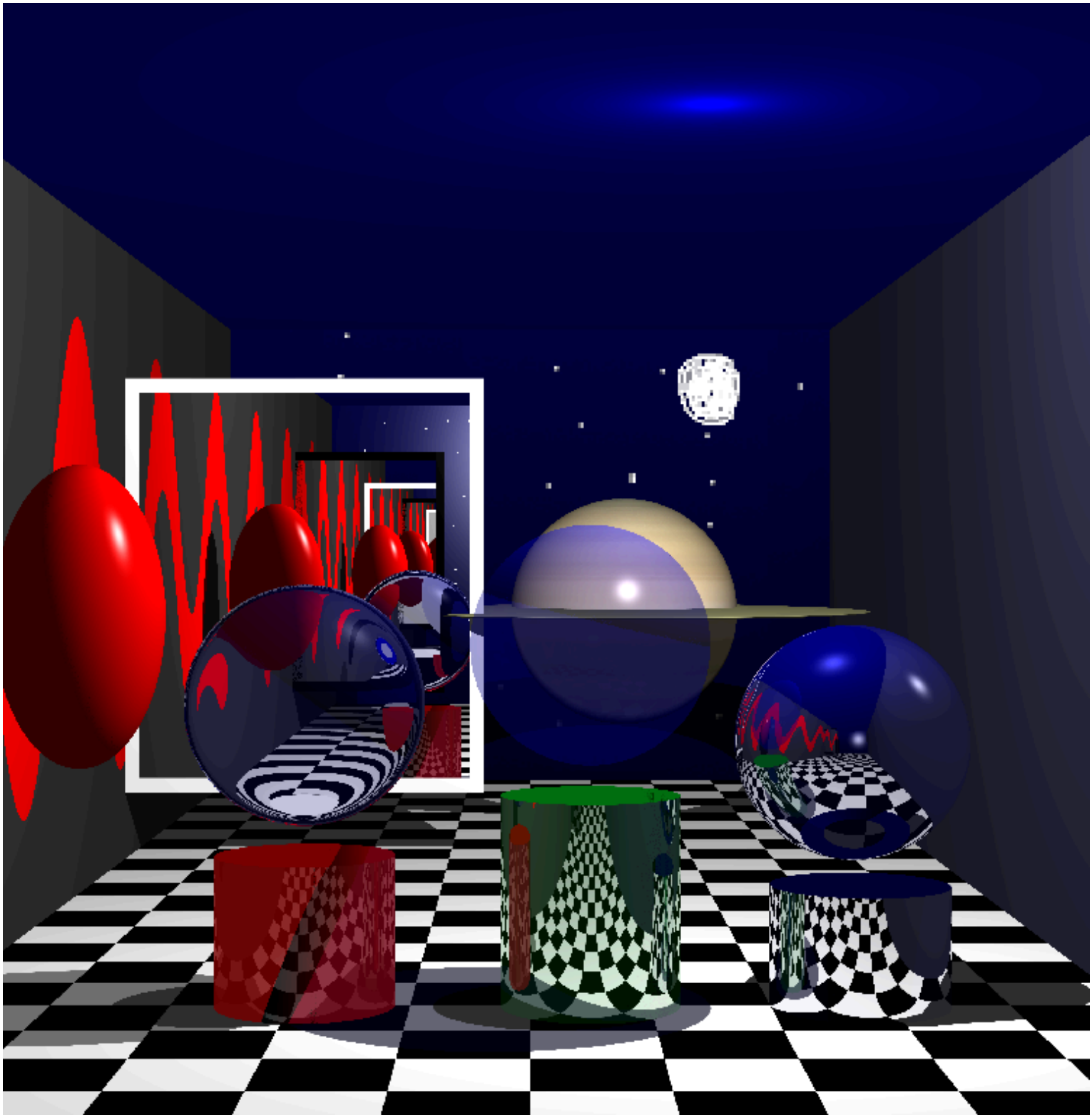


Fig1 completed scene

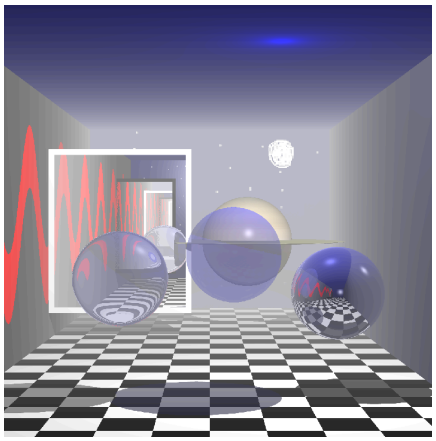


Fig 2

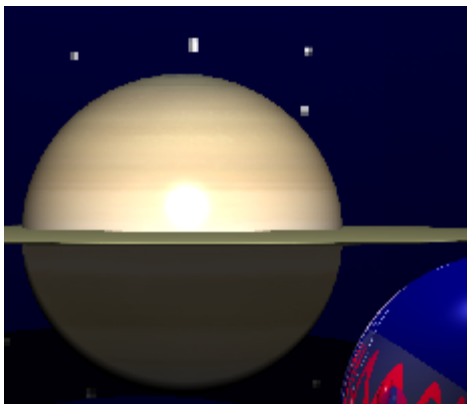


Fig 3

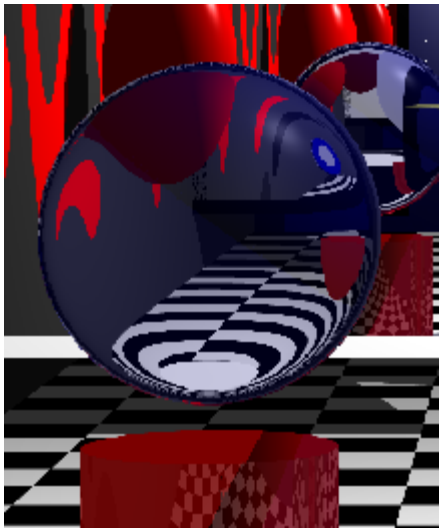


Fig 4

The Basic Ray Tracer: Origin or ray is inside a box consisting of 6 planes with different patterns' colors and textures. Transparent sphere
Objects cast shadows with lighter for Transparent and refractive. Transparent also blends its color to be more realistic. Can see clearly at bottom of fig 2.

Middle sphere in Fig1 is **transparent** with transparency set to 0.6f.
There is a mirror in front and behind the ray origin. Is also a reflective sphere and reflective cylinders.

The floor is a checkered planar.
The method I used was extending the lab-8 code

```
int numTilesX = xWidth / tileWidth;
int numTilesZ = zLength / tileLength;
int ix = (ray.hit.x + xWidth / 2) / tileWidth; // +
int iz = (ray.hit.z - 5) / tileLength; //- 5 to avo
int k = iz % 2;
int l = ix % 2;
// Determine the color based on the indices
glm::vec3 color;
if ((k + l) % 2 == 0) color = glm::vec3(0, 0, 0);
else color = glm::vec3(0.9, 0.9, 0.9);
obj->setColor(color);
```

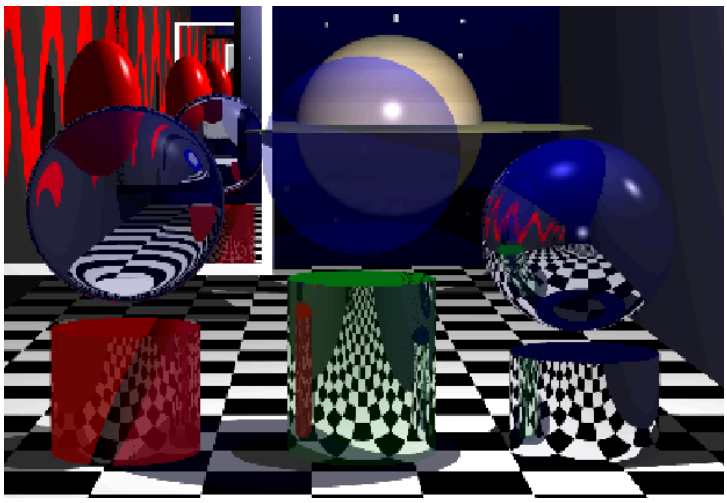
Used modular arithmetic with x and z axis so can when they both even is one color and otherwise is an other color

Extensions

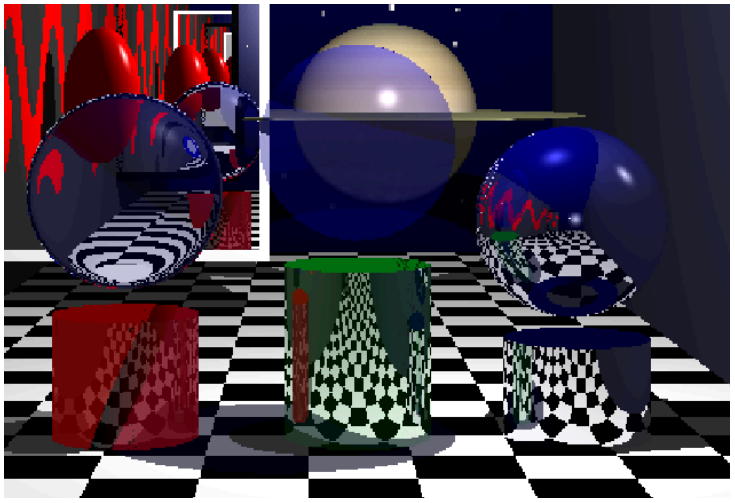
Multiple Reflections: Seen in figure 1 with a large mirror in front of the camera with white border.This is reflecting a mirror that is behind the ray origin and has a black border. Can see the transformed red sphere and sin pattern repeated.

Fog: Implemented with methods from notes with linear and exponential fog as shown in figure 3

Adaptive anti-aliasing with followed by without it both with 500 divisions



With anti anti-aliasing



With out anti-aliasing

As seen in fig 6 I check if a great enough difference between neighbors it then calls recursive anti_alais. The recursive anti_alise splits into 4 rays and computes average color between them. It repeats this recursion on each cell until the color difference is small or reaches anti-analisis recursion depth.

```
glm::vec3 dir(xp + 0.5 * cellX, yp + 0.5 * cellY, -EDIST);
Ray ray = Ray(eye, dir);
glm::vec3 col = trace(ray, 1);

if  (colorDifference(prev_colour, col) > NEIBOUR_THRESHOLD and
    colorDifference(col, prev_prev_colour) > NEIBOUR_THRESHOLD * 1.5 ) {
    col = anti_alais(eye, xp, yp, cellX, cellY, 0);
    prev_colour = col;
}
```

Fig 6

Procedural Generation

Left wall in all images above has procedural generated pattern.
Using sin with frequency thickness and amplitude.

```
float frequency = 0.3f;
float thickness = 0.5f;
float amptitude = 7.0f;
float s = sin(frequency * ray.hit.z);
glm::vec3 color;
if (ray.hit.y < amptitude * (s + thickness) && ray.hit.y > amptitude * (s -thickness))  {
    color = glm::vec3(1.0f, 0.0f, 0.0f);
} else {
    color = glm::vec3(0.2f, 0.2f, 0.2f);
}
```

Textured Sphere. I have a saturn texture on the planet in figure 3 and in the background of figure 1.
Object-Space transformations: I have used O-S transformations on the Ring around Saturn(fig 3) is a sphere scaled (1.0, 0.1, 1.0). I also used on the red (transformed sphere) in the left of fig 1.

Refraction of light: I have a refractive sphere see fig 4 (Refractive index set to 1.08 and refr_coeff set to 0.8f.)

I implemented **cylinders** and used them for a podium setup see figure 1.

Each cylinder has a different coloured and c**learly visible cap** of different colors see figure 1.

```
float Cylinder::intersect(glm::vec3 p0, glm::vec3 dir)
{
    float dx = dir.x;
        float dy = dir.y;
        float dz = dir.z;
        float x0 = p0.x;
        float y0 = p0.y;
        float z0 = p0.z;
        float xc = center.x;
        float yc = center.y;
        float zc = center.z;
        float a = dx * dx + dz * dz;
        float b = 2.0f * (dx * (x0 - xc) + dz * (z0 - zc));
        float c = (x0 - xc) * (x0 - xc) + (z0 - zc) * (z0 - zc) - radius * radius;
        // getting varables from Intersecion equaion in notes and then solving roots
        float discriminant = b * b - 4.0f * a * c;
        if (discriminant < 0)
            return -1.0f;
        float t1 = (-b - sqrt(discriminant)) / (2.0f * a);
        float t2 = (-b + sqrt(discriminant)) / (2.0f * a);
        float y1 = y0 + dy * t1;
        float y2 = y0 + dy * t2;
        if (y1 >= yc && y1 <= yc + height)
            return t1;
        else if (y2 >= yc && y2 <= yc + height)
            return t2;
        else
            return -1.0f;
}
```

Cylinder intersection math equations. Solved for roots the equation from the labs and wiki.
Time to generate output without anti_aliasing is almost 10 seconds.
Anti-aliasing is over 30 seconds. (equivalent to adaptive anti-aliasing with 3 steps)
Adaptive anti-aliasing is about 15 seconds
Adaptive with 2 steps (recursion) is 20 seconds.

References:

https://www.solarsystemscope.com/textures/

https://en.wikipedia.org/wiki/Line-cylinder_intersection

Build commands.
Was developed using Visual Studio code
Build command: /usr/bin/cmake --build /csse/users/ber30/2024/Graphics/Assignment2/build --config Debug --target RayTracer.out -j 22 --

Name: Bradley Erskine

Date: 27-3-24