Assignment 2 – Ray Tracer Report

Name: Ambrose Ledbrook
ID: 79172462
User Code: ajl190

Build Commands

To run the ray tracer navigate to the root directory of the project in terminal and then run the following commands: `cmake .`, `make`, `./RayTracer.out`. Or open the project in a cpp IDE and run RayTracer.cpp from there.

Features

Primitives

Cone

I implemented a 'Cone' class which extends the SceneObjects class. This class has methods to calculate the ray intersection and normal vector for a cone.

Equation as given in lectures: $(x - x_c)^2 + (z - z_c)^2 = \left(\frac{r}{h}\right)^2 (h - y + y_c)^2$

Implementation:

```
float Cone::intersect(glm::vec3 pos, glm::vec3 dir)
{
    glm::vec3 d = pos - center;
    float yd = height - pos.y + center.y;
    float rh = radius / height;
    float tan = powf(rh, 2);

    float a = powf(dir.x, 2) + powf(dir.z, 2) - (tan*powf(dir.y, 2));
    float b = 2*(d.x * dir.x + d.z * dir.z + tan * yd * dir.y);
    float c = powf(d.x, 2) + powf(d.z, 2) - (tan * powf(yd, 2));
    float delta = powf(b, 2) - 4*(a * c);
```

Figure 1 Cone intersection implementation

```
glm::vec3 Cone::normal(glm::vec3 pnt) {
    glm::vec3 d = pnt - center;
    float r = sqrt(pow(d.x, 2) + pow(d.z, 2));
    glm::vec3 normal = glm::vec3(d.x, r * (radius / height), d.z);
    normal = glm::normalize(normal);
```

Figure 2 Cone normal implementation

Cylinder

A 'Cylinder' class was implemented in the same way as the 'Cone' class using slightly different equations to calculate the intersection point and normal vector, as shown below. Equation as given in lectures: $t^2(d_x^2 + d_z^2) + 2t\{d_x(x_0 - x_c) + d_{z(z_0 - z_c)}\} + \{(x_0 - x_c)^2 + (z_0 - z_c)^2 - R^2\} = 0$

```
float Cylinder::intersect(glm::vec3 pos, glm::vec3 dir) {
    glm::vec3 d = pos - center;

    float a = powf(dir.x, 2) + powf(dir.z, 2);
    float b = 2*(d.x * dir.x + d.z * dir.z);
    float c = powf(d.x, 2) + powf(d.z, 2) - powf(radius, 2);
    float delta = powf(b, 2) - 4*(a * c);
```

Figure 3 Cylinder intersection implementation

```
glm::vec3 Cylinder::normal(glm::vec3 pnt) {
    glm::vec3 d = pnt - center;
    glm::vec3 normal = glm::vec3(d.x, 0, d.z);
    normal = glm::normalize(normal);
    return normal;
}
```

Figure 4 Cylinder normal implementation

Tetrahedron

I implemented a 'Tetrahedron' SceneObjects subclass based off the 'Plane' class that was provided. This class creates a triangle plane which is used to construct the tetrahedron plane by plane. The equations and implementations are given below. The

intersection equation used was derived from the plane intersection equation given in labs as this class creates a triangular plane. The equation: The point p is inside the plane if and only if $(u_A \; x \; v_A) \bullet n, \; (u_B \; x \; v_B) \bullet n, \; (u_C \; x \; v_C) \bullet n,$

```
float Tetrahedron::intersect(glm::vec3 pos, glm::vec3 dir) {
    glm::vec3 norm = normal(pos);
    glm::vec3 vDif = a - pos;

    float vDotN = glm::dot(dir, norm);
    if (fabs(vDotN) < 0.001) return -1;

    float t = glm::dot(vDif, norm) / vDotN;
    if (fabs(t) < 0.001) return -1;

    glm::vec3 q = pos + dir * t;
    if (isInside(q)) return t;
    else return -1;
```

Figure 5 Tetrahedron intersection implementation

```
bool Tetrahedron::isInside(glm::vec3 point) {
    glm::vec3 norm = normal(point);

    glm::vec3 uA = b - a;
    glm::vec3 uB = c - b;
    glm::vec3 uC = a - c;

    glm::vec3 vA = point - a;
    glm::vec3 vB = point - b;
    glm::vec3 vC = point = c;

    return glm::dot(glm::cross(uA, vA), norm) > 0 &&
        glm::dot(glm::cross(uB, vB), norm) > 0 &&
        glm::dot(glm::cross(uC, vC), norm) > 0;
```

Figure 6 Tetrahedron is inside check

```
glm::vec3 Tetrahedron::normal(glm::vec3 pos)
{
    glm::vec3 n = glm::cross(b - a, c - a);
    n = glm::normalize(n);
    return n;
}
```

Figure 7 Tetrahedron normal implementation

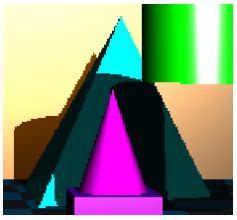

Figure 8 Tetrahedron in scene

Multiple Light Sources

Figure 9 Shadows cast from multiple light sources

There are two light sources within the scene, as seen in the above figure shadows are cast using both light sources. The same calculations used for the first light source are repeated for the second light source.

Refraction

Refraction of light through and object is model on a cone object using an ETA of 1.05. the refraction of the cone is done in a recursive manner where two normal vectors and refraction rays are calculated at each step.

```
glm::vec3 g = glm::refract(ray.dir, normalVector, ETA);
Ray refractedRay(ray.xpt, g);
refractedRay.closestPt(sceneObjects);
if (refractedRay.xindex == -1) return backgroundCol;

glm::vec3 m = sceneObjects[refractedRay.xindex] -> normal(refractedRay.xpt);
glm::vec3 h = glm::refract(g, -m, 1.0f/ETA);
Ray refractedRay2(ray.xpt, g);
refractedRay2.closestPt(sceneObjects);
if (refractedRay2.xindex == -1) return backgroundCol;

glm::vec3 refColor = trace(refractedRay2, step+1);
colorSum *= transVal;
colorSum += refColor * (1 - transVal);
```

Figure 10 Cone refraction implementation

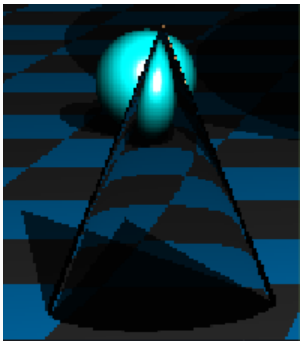

Figure 11 Cone refraction in scene

Anti-Aliasing

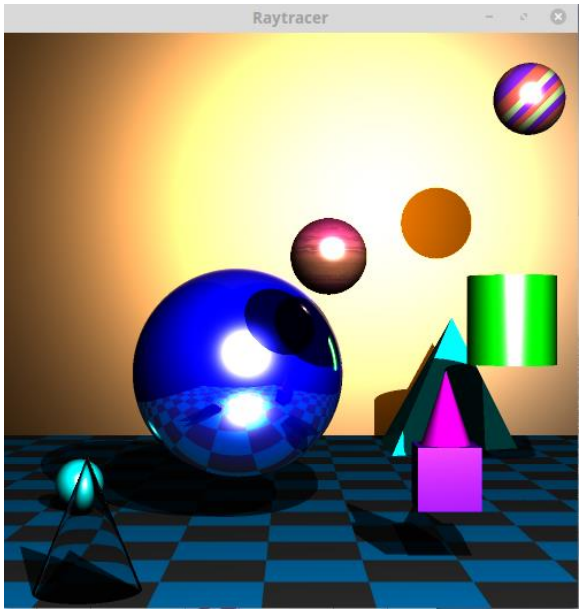

Figure 12 Scene with anti-aliasing

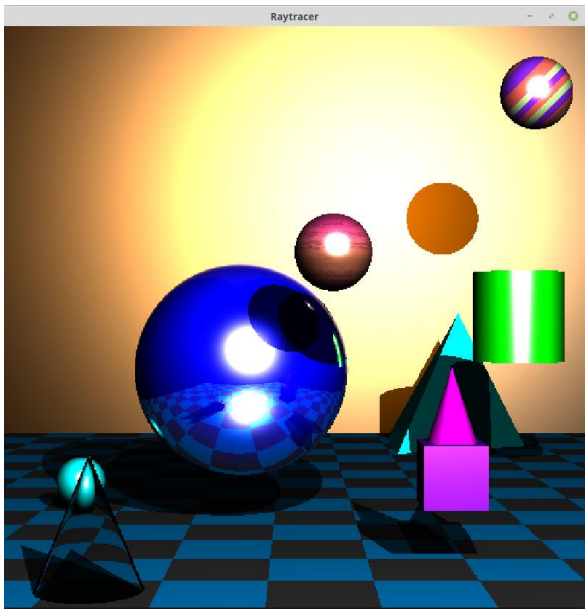

Figure 13 Scene without anti-aliasing

I implemented anti-aliasing in the scene by dividing each pixel into four sub-pixels. For each sub-pixel a ray is produced and traced returning a colour value, this colour value is then averaged which is the colour value given to the original pixel. This has helped to clean up the jaggedness of the edges of the objects and shadows in the scene. However, this did increase the load time of the application by 4 meaning to now takes ~20 seconds to load, where without anti-aliasing enabled it takes ~5 seconds. The two figures above show the differences between anti-aliasing being disabled and enabled. The implementation details of the anti-aliasing are shown below.

```
glm::vec3 antiAlias(float x, float y, glm::vec3 eye, float pixelSize) {
    float quarterLower = pixelSize * 0.25;
    float quarterUpper = pixelSize * 0.75;
    glm::vec2 quarters[4] {
        glm::vec2(quarterLower, quarterLower),
        glm::vec2(quarterLower, quarterUpper),
        glm::vec2(quarterUpper, quarterLower),
        glm::vec2(quarterUpper, quarterUpper),
    };

    glm::vec3 colorSum = glm::vec3(0, 0, 0);

    for (int i = 0; i < 4; i++) {
        Ray ray = Ray(eye, glm::vec3(x + quarters[0].x, y + quarters[0].y, -EDIST));
        ray.normalize();
        colorSum += trace(ray, 1);
    }

    colorSum *= glm::vec3(0.25);

    return colorSum;
}
```

Figure 14 Anti-aliasing implementation

Non-planar object Texturing

The texture file Gaseous4.bmp is mapped to sphere as scene in the above figure. This was done by finding the spherical coordinates of the ray on the sphere and using the provided getColorAt method to map these points to the texture. The equation for these points was sourced from online and is referenced below.

```
glm::vec3 center(3, 3, -90.0);
glm::vec3 dirTex = glm::normalize(ray.xpt - center);
float s = (0.5 - atan2(dirTex.z, dirTex.x) + M_PI) / (2 * M_PI);
float t = 0.5 + asin(dirTex.y) / M_PI;
materialCol = textureSphere.getColorAt(s, t);
```

Figure 15 Sphere texturing implementation

Figure 16 Textured sphere in scene

Non-planar object procedural pattern

I procedurally generated a varying pattern on a sphere as shown above. The pattern has three different colours as a value is found by finding the difference between the x and y values of the ray and then modulating this value by 3. A different colour value is then shown dependent on the three possible values of the value found. The implementation of this pattern is shown below.

```
int val = ((int) (ray.xpt.x - ray.xpt.y)) % 3;
if (val == 0) {
    materialCol = glm::vec3(1, 0.4, 0.35);
} else if (val == 1) {
    materialCol = glm::vec3(0.8, 1, 0.6);
} else {
    materialCol = glm::vec3(0.55, 0.15, 1);
}
```

Figure 17 Procedural pattern on sphere implementation

Figure 18 Procedural pattern on sphere in scene

Camera Motion

I implemented camera motion throughout the scene. The user can move the camera backwards and forwards using the up and down arrow keys, the user is also able to rotate the scene side to side using the left and right arrow keys. Bounding checks are also performed to stop the user going out the front or back of the scene.

```
void special(int key, int x, int y)
{
    //--------------------------
    // Camera motion
    if (key == GLUT_KEY_UP) EDIST += 5;
    else if (key == GLUT_KEY_DOWN) EDIST -= 5;
    else if (key == GLUT_KEY_LEFT) eye_x -= 5;
    else if (key == GLUT_KEY_RIGHT) eye_x += 5;

    // Bounding checks for camera motion
    if (EDIST <= -200) EDIST == -200;
    if (EDIST >= 0) EDIST == 0;
    //--------------------------

    glutPostRedisplay();
}
```

Figure 19 Camera motion implementation

Successes and Failures

Successes

- A major success I had over the assignment was texturing non-planar objects, through both image and procedural texturing.
- Implementing anti-aliasing for me was another success as this was very rewarding and satisfying to complete and see.

Failures

- The first failure was my attempt to perform a rotation transformation on an object. I wrote an implementation that would apply a rotation matrix to a passed point, but when trying to integrate this method into the display of an object I did not get the desired results and could not find a fix that produced the desired transformation.
- Another failure I had over the assignment was showing transparency that was not a special case on refraction. I did implement a refraction that models' transparency as shown above in the refraction feature, but I never managed to find a way to show transparency without using a special case of refraction.

References

- Planet Texture File: http://www.texturesforplanets.com
- Spherical Coordinates for Texture Mapping: https://stackoverflow.com/questions/22420778/texture-mapping-in-a-ray-tracing-for-sphere-in-c
- Ray Intersection Algorithms: Course Lecture Notes