DH2323 Lab2 Report

Shuang Qiu (shuangq@kth.se)

1. Objective

The objective of this lab is to implement a raytracer, which uses the following techniques:

1. Represent 3D scenes using triangular surfaces.
2. Trace the ray of each pixel in the camera image into the scene.
3. Compute ray-triangle intersections, to find out which surface a ray hit.

2. Insertion of Ray and Triangles

2.1 Calculate Intersections

First we need a function to calculate the closest intersections of a ray and a triangle.

```
//...

for(int i=0; i<triangles.size(); ++i){
    Triangle triangle = triangles[i];

    //v0 + u*e1 + v*e2 = s + t*d;
    vec3 v0 = triangle.v0;
    vec3 v1 = triangle.v1;
    vec3 v2 = triangle.v2;
    vec3 e1 = v1 - v0;
    vec3 e2 = v2 - v0;
    vec3 b = start - v0;
    mat3 A( -dir, e1, e2);

    vec3 x = glm::inverse(A) * b;   //x = (t u v);

    //within the triangle:
    // 0 < u
```

```
    // 0 < v
    // u+v < 1

    //after the beginnning of the ray
    // 0 <= t

    if((x.y > 0.0f) && (x.z > 0.0f) && (x.y + x.z < 1.0f) && (x.x >= 0.0f)){
        result = true;

        vec3 position = v0 + x.y * e1 + x.z * e2;
        dist = glm::length(position - start);

        if(dist < closestIntersection.distance){
            closestIntersection.position = position;
            closestIntersection.distance = dist;
            closestIntersection.triangleIndex = i;
        }
    }

//...
```

But there is a problem, if we do it like above, there will be no value for the edge of the triangles (Figure 1 left). Therefore I change the condition to `(x.y >= 0.0f) && (x.z >= 0.0f) && (x.y + x.z <= 1.0f) && (x.x >= 0.0f)` instead, so that the result will be better (Figure 1 right).

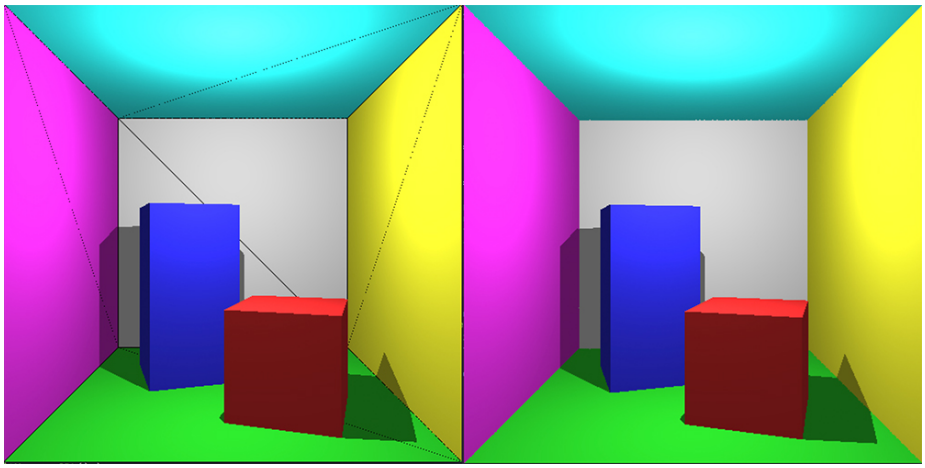

Figure 1: Step1 result using two conditions

2.2 Cramer's Rule

Instead of using `glm::inverse()`, we could use **Cramer's Rule**[1] to calculate vector x.

2.2.1 Implementation

According to the equations, I wrote a function `CramersRules()` to do the job.

```
vec3 CramersRule(vec3 col1, vec3 col2, vec3 col3, vec3 b)
{
    float A0 = det(col1, col2, col3);
    float Ax = det(b, col2, col3);
    float Ay = det(col1, b, col3);
    float Az = det(col1, col2, b);

    return vec3(Ax/A0, Ay/A0, Az/A0);
}

float det(vec3 col1, vec3 col2, vec3 col3)
{
    float plus = (col1[0] * col2[1] * col3[2])
    + (col1[1] * col2[2] * col3[0])
    + (col1[2] * col2[0] * col3[1]);

    float minus = (col1[2] * col2[1] * col3[0])
    + (col1[1] * col2[0] * col3[2])
    + (col1[0] * col2[2] * col3[1]);

    return plus - minus;
}
```

2.2.2 Runtime Comparison

As shown in Figure 2 and 3, we could see that using Cramer's rule[2] will make the program run faster (when the screen size is set to 50*50).

[1] Cramer's rule. (2017, May 11). In Wikipedia, The Free Encyclopedia. Retrieved 12:19, May 24, 2017, from https://en.wikipedia.org/w/index.php?title=Cramer%27s_rule&oldid=779888699.

[2] Cramer's rule. (2017, May 11). In Wikipedia, The Free Encyclopedia. Retrieved 12:19, May 24, 2017, from https://en.wikipedia.org/w/index.php?title=Cramer%27s_rule&oldid=779888699.

```
Draw Time: 130
Draw Time: 118
Draw Time: 124
Draw Time: 119
Draw Time: 115
Draw Time: 115
Draw Time: 115
Draw Time: 112
Draw Time: 112
Draw Time: 116
```

Figure 2: The time Draw() takes using glm::inverse

```
Draw Time: 93
Draw Time: 83
Draw Time: 85
Draw Time: 82
Draw Time: 85
Draw Time: 83
Draw Time: 82
Draw Time: 82
Draw Time: 78
```

Figure 3: The time Draw() takes using Cramer's rule

3. Tracing Rays

In raytracing, we cast a ray for every pixel and find the closest intersection point, which will decide the color of the pixel. The position of the camera is the start point of the ray. The ray vector can be computed as:

$$d = (x - W/2, y - H/2, f)$$

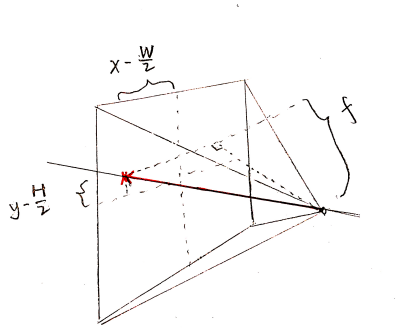

Figure 4: The camera points at the center of the world

Here we set the focal length of the camera to the value of the screen height. As mentioned in lab1, the vertical/horizontal field of view can be calculated:

$$vfv = 2 * arctan(\frac{H/2}{f})$$

$$hfv = 2 * arctan(\frac{W/2}{f})$$

When the Cornell Box is loaded, it is scaled to fill the volume -1 to 1. If the focal length of the camera is set to H, then the z value of the camera should be `-1 - 2 = -3` (Figure 5). So the position of the camera is `(0, 0, -3)`.

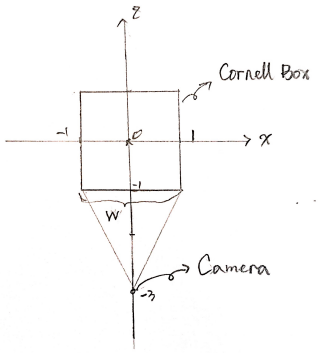

Figure 5: Camera position if focal length set to H

4. Moving the Camera

The rotation matrix about one of the axes in three dimension:

$$\mathbf{R_x}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{bmatrix}$$

$$\mathbf{R_y}(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix}$$

$$\mathbf{R_z}(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When the left arrow is pressed, increment the rotation angle `yaw` and multiply the new rotation matrix with the camera position vector to get the current camera position. In this way the camera position will rotate around the **y** axis.

```
if(keyState[SDLK_LEFT]){
    //move the camera to the left
    yaw += 0.05f;
    //camera rotate along y axis
    R = mat3(cos(yaw), 0, sin(yaw),
             0, 1, 0,
             -sin(yaw), 0, cos(yaw));
}

//...
newCameraPos = cameraPos * R;
```

Except changing the position of the camera, we also need to rotate the camera to make it always looking at the center. Here I multiply the direction vector with the rotation matrix in the `Draw()` function:

```
//...
vec3 dir(x - SCREEN_WIDTH / 2, y - SCREEN_HEIGHT / 2, focalLength);
dir = dir * R;
//...
```

5. Illumination

5.1 Direct Light and Direct Shadow

According to equation (28) in the guide book, the light that gets reflected is:

$$R = \rho * D = (\rho * Pmax(\hat{r} \cdot \hat{n}, 0))/4\pi r^2$$

ρ describes the fraction of the incoming light that gets reflected by the diffuse surface for each color component. Here it is the color we stored in each triangle.

For the direct shadow, we cast a ray from the light source to the Intersection point. If there the ray intersects with any surface in between, the color of that point should set to black.

Implementing the equation:

```
//calculate direct light
vec3 DirectLight( const Intersection& i){
    vec3 n = triangles[i.triangleIndex].normal;
    vec3 r = lightPos - i.position;
    float distance = glm::length(r);
```

```
    //if > 90 degree then receive no direct light
    float product = glm::dot(n, r) > 0 ? glm::dot(n, r) : 0;

    vec3 d = lightColor * product / float(4 * M_PI * distance * distance);

    //direct shadow
    Intersection shadowIn;
    bool ifIntersect = ClosestIntersection(lightPos, -r, triangles, shadowIn);
    if(ifIntersect && (shadowIn.distance < distance - 0.00001f)){
            d = vec3(0,0,0);
    }

    return d;
}
```

5.2 Indirect Light

In real life there is not only one bounce of direct light D on the surface. We simulated the indirect illumination by using a constant value N. The total illumination on a surface should be:

$$T = D + N$$

Therefore the reflected light should be implemented as:

```
vec3 dirLight = DirectLight(closestIntersection);
vec3 totalLight = dirLight + indirectLight;
color = totalLight * color;
```

6. Final Output

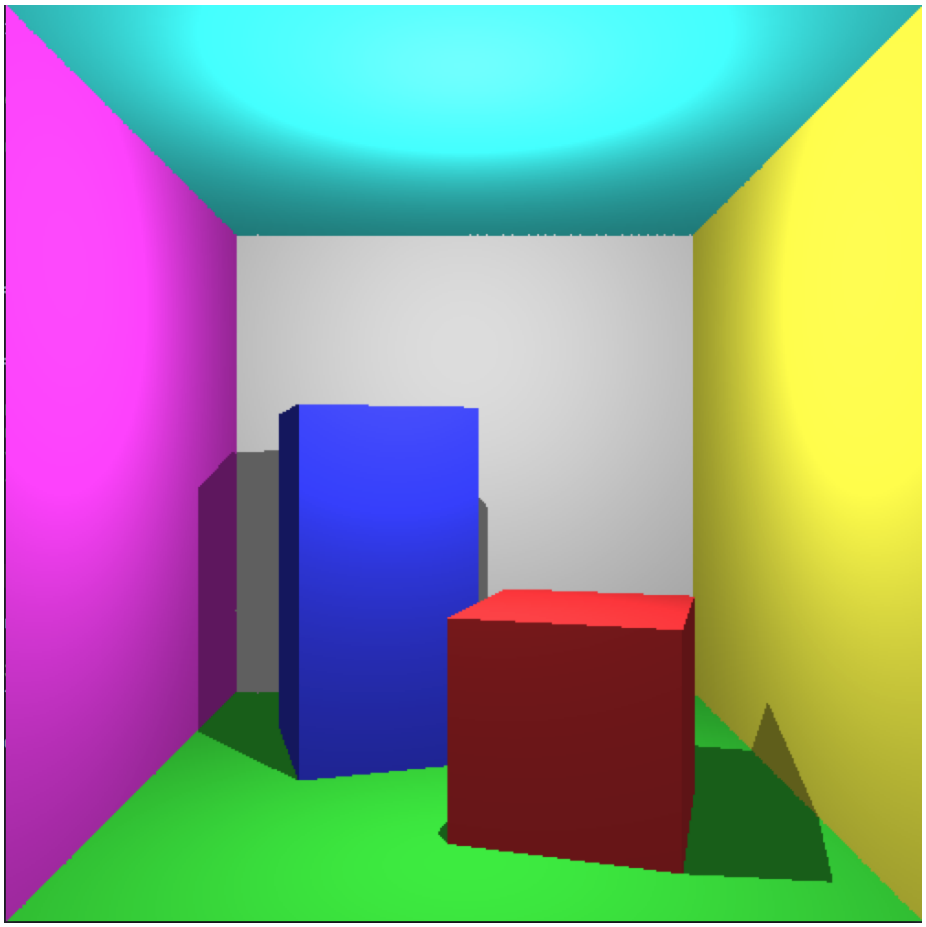

Figure 6: Final Output of lab 2