DH2323 Lab Assignment2
Raytracing

Yeqi Wang yeqi@kth.se 200002221299

June 2023

Task1 - Tracing Rays

1.1 Intersection of Ray and Triangle

As we use triangles to represent surfaces, we use two vector e_1 and e_2 as edges of the triangle where $e_1 = v_1$ - v_0 and $e_2 = v_2$ - v_0 and v_0, v_1, v_2 are vertices of the triangle. Thus any point r in the plane of the triangle can then be described by two scalar coordinates u and v as r = v_0 + ue_1 + ve_2 where parameters u and v have limitation of 0 < u, 0 < v, u+v < 1.
Now move on to describing ray, we need a start point and direction to describe an unique ray. The ray can be represented as r = s + td wher s stands for the start point, d stands for direction and t stands for bias.
Then comes the intersection between ray and triangles. An equation is made between two mathematical expression above. According to linear algebra, parameters u, v and t are listed in the column matrix x = $(t \ \ u \ \ v)^T$ then n the solution x should be x = $A^{-1}b$ where A = $(-d \ \ e_1 \ \ e_2)$ and b = s - v_0.
All these computation are done in the closetintersection function and when all the conditions are met, we will set the maximum to the current ray position and see if there is a much closer intersection exists in the scenario and store the current intersection information into the intersection struct. The code snippet of checking condition of parameters and existence of closer intersection is shown.

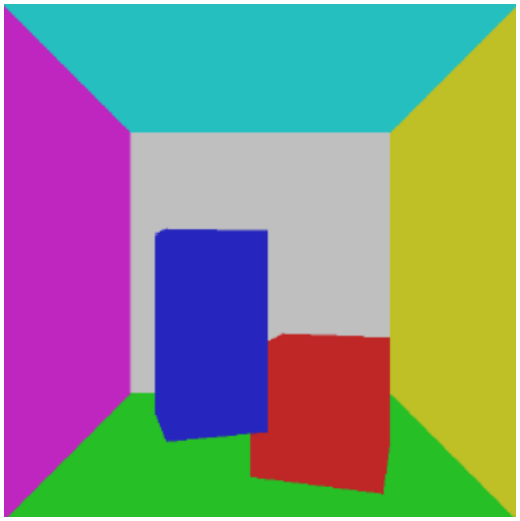
```
if (u >= 0 && v >= 0 && u + v <= 1)
{
    if (t >= 0 && t <= m)
    {
        m = t; // Set the maximum to the current ray position to see if there is much closer intersection
        closestIntersection.distance = t;
        closestIntersection.position = start + (dir * closestIntersection.distance);
        closestIntersection.triangleIndex = i;
    }
    exist = true;
}
```

1.2 Tracing Rays

Assume that the camera is aligned with the coordinate system of the model with x-axis pointing to the right and y-axis pointing down and z-axis pointing forward into the image and then the vector pointing where the light reaching pixel (x,y) comes from can be describe as d = (x-W/2, y-H/2, f) which is written in the draw function as the direction variable standing for ray direction. In the draw function, we will use closetintersection function to check if it exists the closet intersection in the scenario, if so we will draw the pixel as the color of the current triangle. The code snippet shows how I did it in the draw function.

```
for (int y = 0; y < SCREEN_HEIGHT; ++y)
{
    for (int x = 0; x < SCREEN_WIDTH; ++x)
    {
        vec3 direction((x - SCREEN_WIDTH / 2), (y - SCREEN_HEIGHT / 2), focalLength);
        Intersection intersection;
        vec3 black(0, 0, 0);
        direction = R * glm::normalize(direction);
        if (ClosestIntersection(cameraPos, direction, triangles, intersection))
        {
            vec3 colour = triangles[intersection.triangleIndex].color;
            PutPixelSDL(screen, x, y, colour);
        }
        else
        {
            PutPixelSDL(screen, x, y, black);
        }
    }
}
```

Final Image



Everything works well. Things become a little harder when I am working on doing the intersection. I ignore the update of "m" which represents the maximum to the current position of ray. As a result, the blue cube have a part of its over the red one which really confuses me before but I finally figure it out.

Task 2 - Moving Camera

A rotation matrix is used to do the rotation of the camera. As the camera is rotated around y-axis the specified rotation matrix is used with yaw:

$$\begin{pmatrix} cos(yaw) & 0 & sin(yaw) \\ 0 & 1 & 0 \\ -sin(yaw) & 0 & cos(yaw) \end{pmatrix}$$

When the specified key is pressed yaw is updated and sequentially the rotation matrix. The rotation matrix will be used in the draw function with the direction of the camera is pointing. The code snippet shows how I do the update of rotation matrix where the matrix is included in the Rotation() function and include weight to adjust the extent of rotation.

```
// Rotate Camera
if (keystate[SDLK_q])
{
    // Move camera to the right
    yaw -= weight * 1;
    Rotate();
}

if (keystate[SDLK_e])
{
    // Move camera to the right
    yaw += weight * 1;
    Rotate();
}
```

As for translation of camera position, I simply do the addition and subtraction to the camera position.

Task Summary

Everything works well. At the beginning, I cross timed the rotation Matrix with camera position and the result looks weird but I finally figure out that the rotation should be done on the direction the camera is pointing at. After all I am confident that I now know how it works.

Illumination

3.1 Direct Illumination

Light position and light color are simple added as global variable used in the draw function and DirectLight function. In the DirectLight function, the distance from the light position to the intersection position is measured and cast another ray from the intersection to the light to check if distance to the closet intersection is smaller than to the light position and if so it should be the shadow.
Following the instruction, Light that get reflected should be:

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

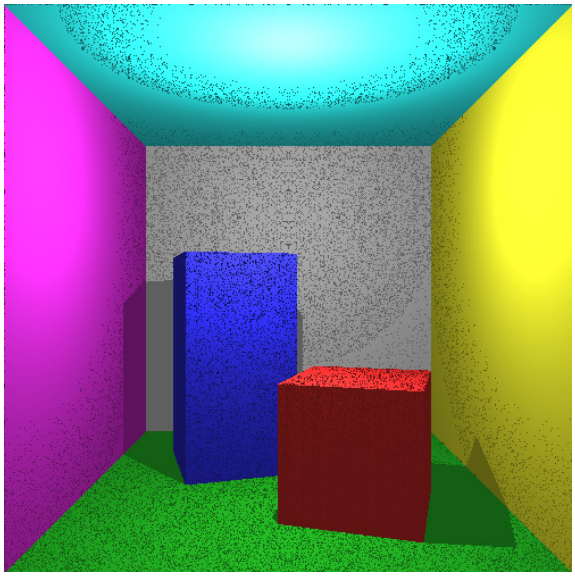
The code snippet of DirectLight() function is shown.

```
vec3 DirectLight(const Intersection &i)
{
    // Let n̂ be a unit vector describing the normal pointing out from the surface
    // Let r̂ be a unit vector describing the direction from the surface point to the light source.

    vec3 n = triangles[i.triangleIndex].normal;
    vec3 r = glm::normalize(i.position - lightPos);
    vec3 D;

    float r_length = glm::distance(lightPos, i.position);

    Intersection intersection;

    // Cast another ray from surface to the light source using ClosetIntersection
    // Check if distance to the closest intersecting surface for surface point is closer than to the light sourc
    if (ClosestIntersection(lightPos, r, triangles, intersection))
    {
        if (intersection.distance < r_length)
        {
            return vec3(0, 0, 0);
        }
    }

    // lightColor describes the power P
    // i.e. energy per time unit E/t of the emitted light for each color component

    // D = B max(r̂ . n̂ , 0) = (P max (r̂ . n̂ , 0))/4πr^2
    D = vec3((lightColor * glm::max(glm::dot(-r, n), 0.0f)) / (4.0f * glm::pow(r_length, 2.0f) * pi));

    return D;
};
```

3.2 Indirect Illumination

Just simply add the indirectLight which we defined as a constant variable to the direct light in the equation of the pixel value in the draw function.

Final Image



Task Summary

Everything works well. With experience of faults in closetintersection() function, I have not much trouble working on this. The final image seems noisy but I suppose there should be no error in my implementation.