DH2323 Lab Assignment3
Rasterization

Yeqi Wang yeqi@kth.se 200002221299

June 2023

Task1 - Drawing

1.1 Perspective Projection of Points from 3D to 2D & Drawing points

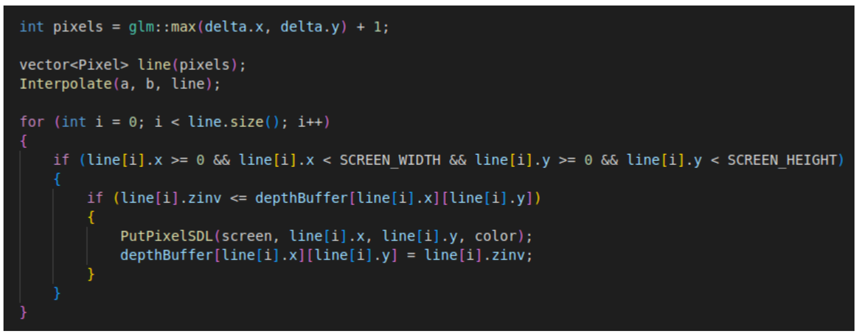
Assume we have a pinhole camera positioned at the origin looking along the z-axis of a 3D coordinate system. Let the x-axis point to the right and let the y-axis point downwards. Then the projection of an arbitrary 3D point (X, Y, Z) to a 2D point (x, y) in the camera image can be written as x = f $\frac{X}{Z}$+ $\frac{W}{2}$ and y = f $\frac{Y}{Z}$+ $\frac{H}{2}$. When we drawing points in the VertexShader() function,it takes the 3D position of a vertex v, computes its 2D image position using the equation and store it in the given 2D integer vector p.

1.2 Drawing edges

Linear interpolation is used to write the function that responsible for drawing edges. As what have been done in the Assignment 1, Linear interpolation simply calculates the step for two dimensions and do the stepping and store it in the output array. When we draw the edge, the output array is called and is checked if it is outside of the screen. If not then do the drawing these interpolated points. Once the DrawLine() function is fixed, the DrawPolygonLine() function in the instruction can be used in the code. The code snippet is provided in the DrawLine() function.

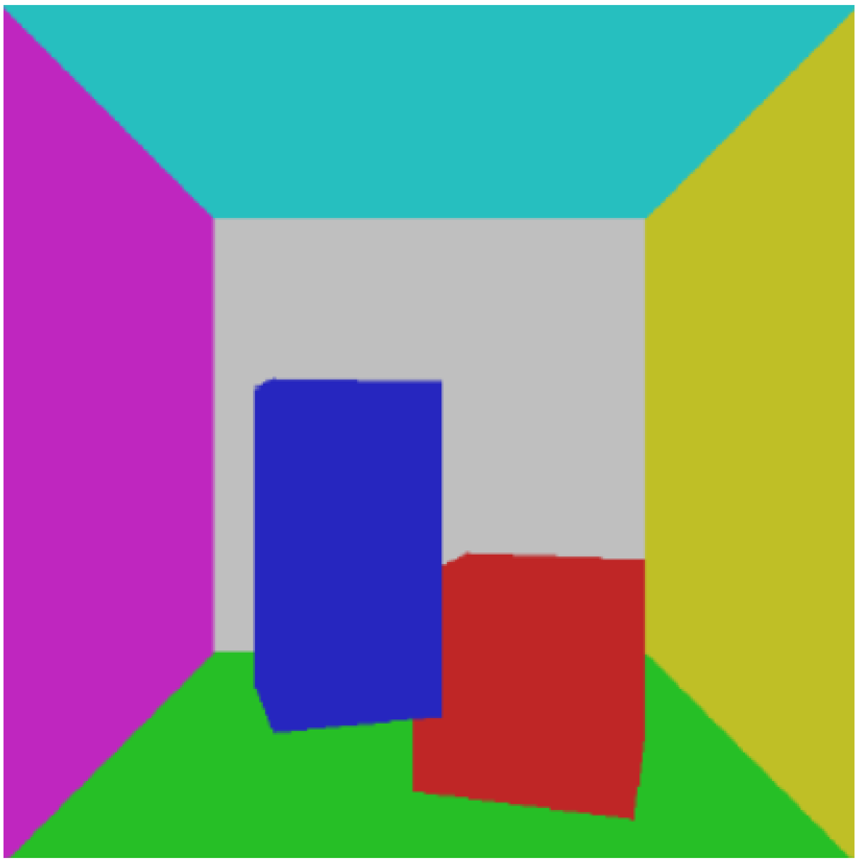
1.3 Filled triangles

The main idea is to draw the triangle row by row. Two arrays of vectors are defined that store the start position and the end position of the triangle for each row.
To fill arrays with values representing the polygon we first initialize the start arrays with really big values and the end array with really small values. Then loop through edges and compute the pixels corresponding to the line and replace the current value if the x-value in the right array is smaller and x-value in the left array is larger. In the end we will have the smallest x-value for each row in the left array and the largest value for each row in the right array.
Using the comment in the code skeleton, I get this code snippet of ComputePolygonRows() works.



```
int pixels = glm::max(delta.x, delta.y) + 1;

vector<Pixel> line(pixels);
Interpolate(a, b, line);

for (int i = 0; i < line.size(); i++)
{
    if (line[i].x >= 0 && line[i].x < SCREEN_WIDTH && line[i].y >= 0 && line[i].y < SCREEN_HEIGHT)
    {
        if (line[i].zinv <= depthBuffer[line[i].x][line[i].y])
        {
            PutPixelSDL(screen, line[i].x, line[i].y, color);
            depthBuffer[line[i].x][line[i].y] = line[i].zinv;
        }
    }
}
```

Final Image



Task Summary

Everything works well. I get trouble in understanding filling triangle, the main idea seems easy to understand for me but the implementation do make me confused a lot. But I think the code skeleton(comment) given in the instruction is really useful even though it seems hard to me at the very beginning. First update the minimum and maximum of the vertices of the triangle y value, resize and then with initialization loop through all edges to update the right pixel and left pixel.

Task 2 - Moving Camera

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

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

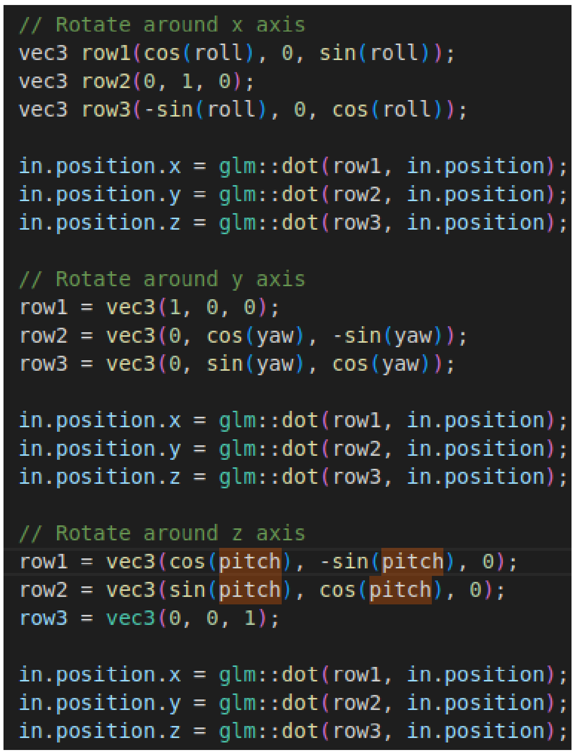
When the camera is rotated around x-axis this specified rotation matrix is used with roll:

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

When the camera is rotated around x-axis this specified rotation matrix is used with pitch:

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

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 rotation matrix functions in the Rotate() function.



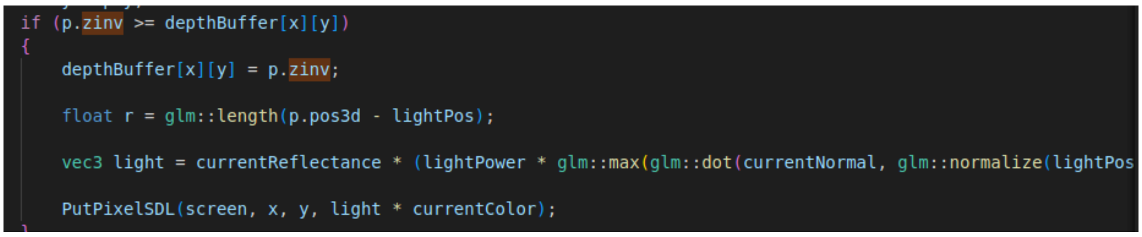
```
// Rotate around x axis
vec3 row1(cos(roll), 0, sin(roll));
vec3 row2(0, 1, 0);
vec3 row3(-sin(roll), 0, cos(roll));

in.position.x = glm::dot(row1, in.position);
in.position.y = glm::dot(row2, in.position);
in.position.z = glm::dot(row3, in.position);

// Rotate around y axis
row1 = vec3(1, 0, 0);
row2 = vec3(0, cos(yaw), -sin(yaw));
row3 = vec3(0, sin(yaw), cos(yaw));

in.position.x = glm::dot(row1, in.position);
in.position.y = glm::dot(row2, in.position);
in.position.z = glm::dot(row3, in.position);

// Rotate around z axis
row1 = vec3(cos(pitch), -sin(pitch), 0);
row2 = vec3(sin(pitch), cos(pitch), 0);
row3 = vec3(0, 0, 1);

in.position.x = glm::dot(row1, in.position);
in.position.y = glm::dot(row2, in.position);
in.position.z = glm::dot(row3, in.position);
```

Task Summary

This task is similar to the previous assignment on moving camera but this time I add two additional options of rotating. Although this time the matrix is described row by row and use different method of computation, it is easy to implement after all.

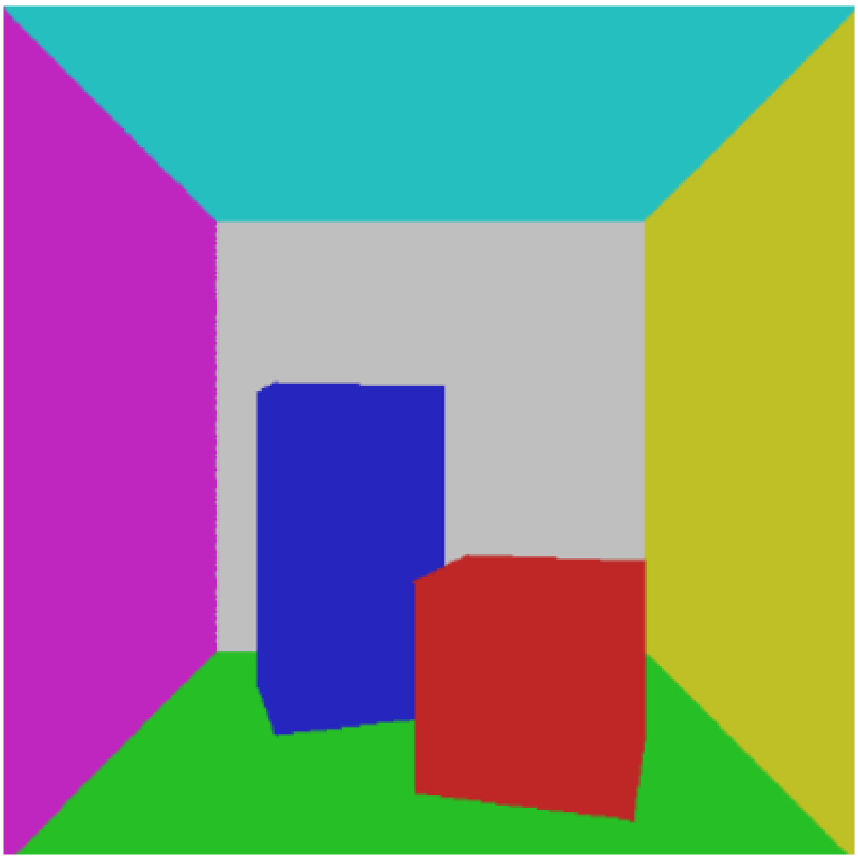
Depth Buffer

Depth buffer is introduced to deal with the problem of triangles might be drawn on top of each other in arbitrary order by treating every triangle independently. The depth value represents the depth of the closest surface point drawn at that pixel so far. When rendering a new pixel, its depth is compared to the value stored in the depth buffer for that pixel. If the new depth is smaller (indicating that the pixel is closer to the camera), the pixel is drawn, and its depth value replaces the previous one in the depth buffer.
In practice, Another float variable is introduced in the pixel struct replacing the original ivec2 variable. In functions including Interpolate() and ComputePolygonRows() etc. we need to include some operations on this new variable. The code snippet shows how the zinv is updated.



```
if (p.zinv >= depthBuffer[x][y])
{
    depthBuffer[x][y] = p.zinv;

    float r = glm::length(p.pos3d - lightPos);

    vec3 light = currentReflectance * (lightPower * glm::max(glm::dot(currentNormal, glm::normalize(lightPos

    PutPixelSDL(screen, x, y, light * currentColor);
```

Final Image



Task Summary

Everything works well. The image output in the previous task is just the same as I did in the first assignment which has one part of blue cube is drawn on top of the red one. In this task, the instruction introduce depth buffer as a solution to this problem which is easy to understand for me.

Illumination

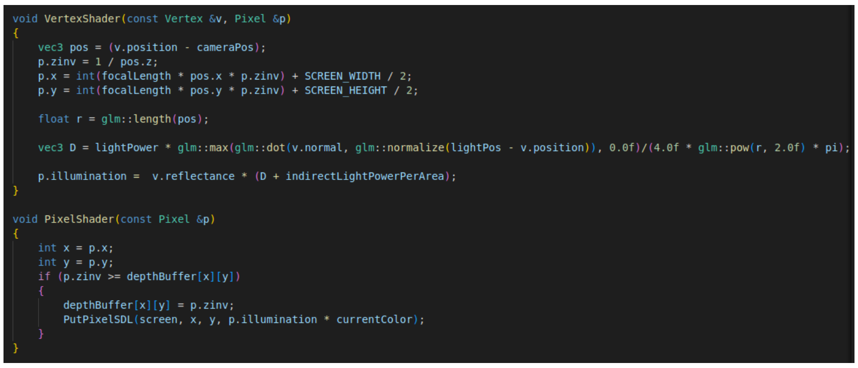
Per pixel Illumination

This illumination computation is implemented for every vertex and then interpolate these values across the polygon. In practice a new variable is added to the struct of Pixel and we compute the illumination for each vertex using the following equations:

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

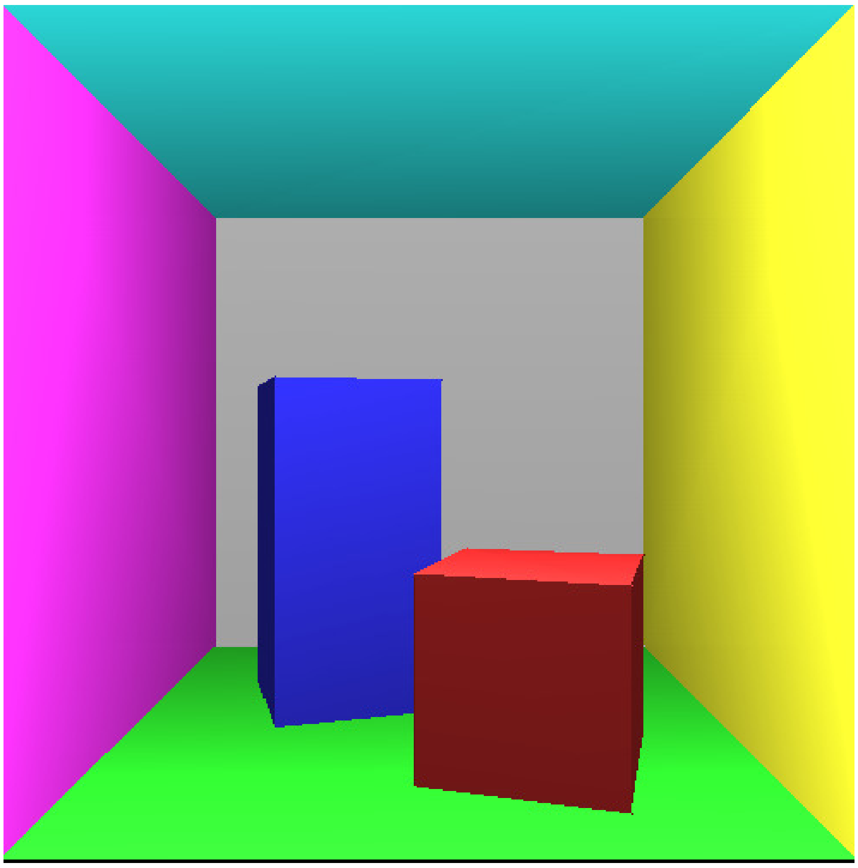
$$R = \rho * (D + N)$$

where D is the power of the incoming direct light per surface area, N is the incoming indirect illumination reflected from other surfaces, P is the power of the light source, r is a vector from the surface point to the light source and $\hat{n}$ is the normal of the surface.
To do this, new struct of vertex is introduced with its position, normal and reflectance to accomplish the variable needed to compute the illumination. The code snippet of VertexShader() and PixelShader() shows how this computation is done.



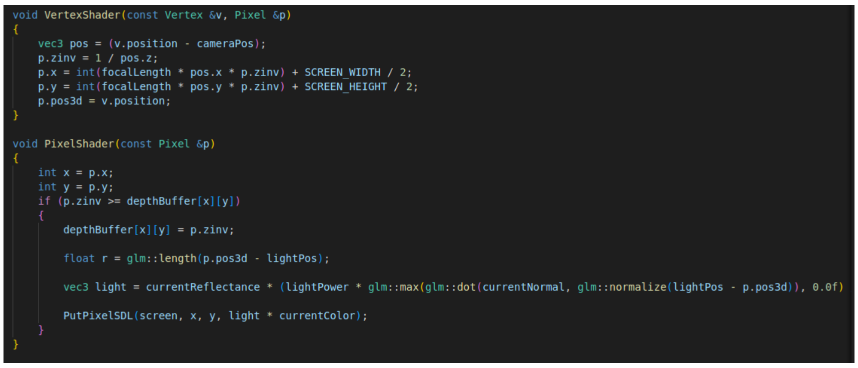
```
void VertexShader(const Vertex &v, Pixel &p)
{
    vec3 pos = (v.position - cameraPos);
    p.zinv = 1 / pos.z;
    p.x = int(focalLength * pos.x * p.zinv) + SCREEN_WIDTH / 2;
    p.y = int(focalLength * pos.y * p.zinv) + SCREEN_HEIGHT / 2;

    float r = glm::length(pos);

    vec3 D = lightPower * glm::max(glm::dot(v.normal, glm::normalize(lightPos - v.position)), 0.0f)/(4.0f * glm::pow(r, 2.0f) * pi);

    p.illumination =  v.reflectance * (D + indirectLightPowerPerArea);
}

void PixelShader(const Pixel &p)
{
    int x = p.x;
    int y = p.y;
    if (p.zinv >= depthBuffer[x][y])
    {
        depthBuffer[x][y] = p.zinv;
        PutPixelSDL(screen, x, y, p.illumination * currentColor);
    }
}
```

Final Image



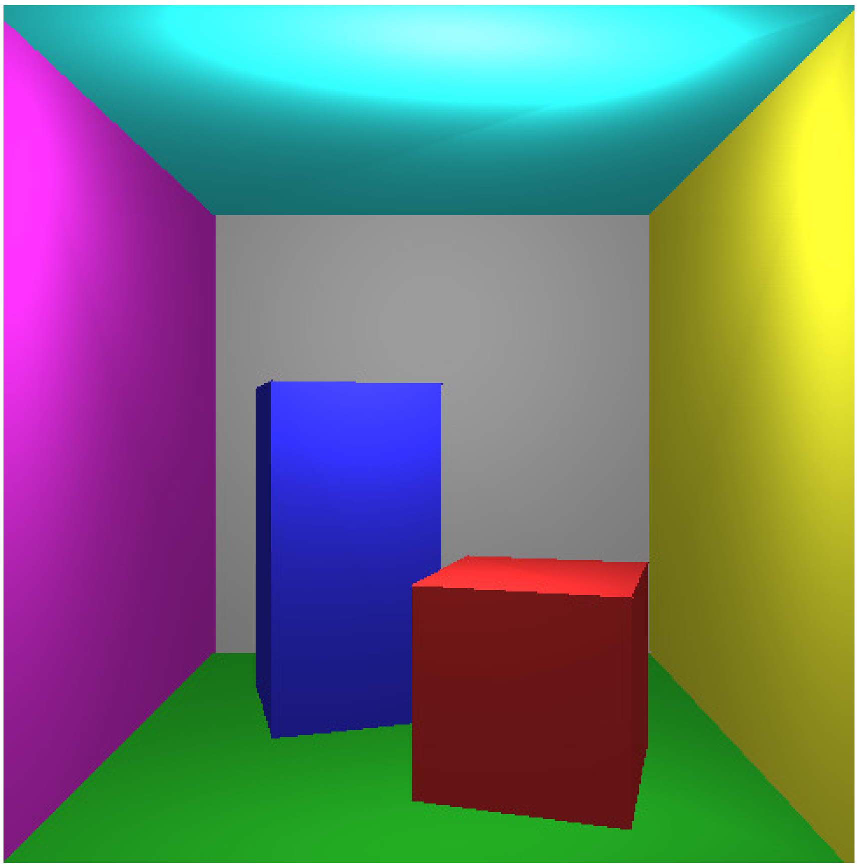
Per Vertex Illumination

This illumination computation is implemented for every pixel so it will be more detailed. The difference from the Per Vertex Illumination is that another new variable pos3D to the struct of Pixel representing 3D position that the pixel corresponds since to we do not interpolate illumination and we instead interpolate 3D position of different pixels in the triangles. The computation of illumination is done in the function PixelShader() with use of global constant variable of currentNormal and currentReflectance. The code snippet shows how it works.



```
void VertexShader(const Vertex &v, Pixel &p)
{
    vec3 pos = (v.position - cameraPos);
    p.zinv = 1 / pos.z;
    p.x = int(focalLength * pos.x * p.zinv) + SCREEN_WIDTH / 2;
    p.y = int(focalLength * pos.y * p.zinv) + SCREEN_HEIGHT / 2;
    p.pos3d = v.position;
}

void PixelShader(const Pixel &p)
{
    int x = p.x;
    int y = p.y;
    if (p.zinv >= depthBuffer[x][y])
    {
        depthBuffer[x][y] = p.zinv;

        float r = glm::length(p.pos3d - lightPos);

        vec3 light = currentReflectance * (lightPower * glm::max(glm::dot(currentNormal, glm::normalize(lightPos - p.pos3d)), 0.0f)

        PutPixelSDL(screen, x, y, light * currentColor);
    }
}
```

Final Image



Task Summary

Everything works well. The difficulty of two solution for illumination for me seems to be the same as it is still using the equation we used in the raytracing assignment. The result of per pixel illumination is better than the per vertex one as expected and how fast the rendering can be done in this part comparing to ractracing one really surprises me.