CSC4140 Assignment II

Computer Graphics

February 25, 2022

Transformation

This assignment is 10% of the total mark.

Strict Due Date: 11:59PM, Feb 25^{th}, 2022

Student ID: 118010202

Student Name: Zhixuan LIU

This assignment represents my own work in accordance with University regulations.

Signature: Zhixuan Liu

1 Implement *get_model_matrix()*

In this section, I will introduce how to get the 4×4 transformation matrix in the homogeneous coordinate with regard to the translation, scaling, and rotation processes.

1.1 Matrix for Translation

Suppose we have a translation vector T (3×1), we want to get the corresponding translation matrix. We can use the following codes:

```
//Step 1: Build the Translation Matrix M_trans:
Eigen::Matrix4f M_trans;
M_trans << 1,0,0,T[0],
           0,1,0,T[1],
           0,0,1,T[2],
           0,0,0,1;
```

1.2 Matrix for Scaling

Suppose we have a scaling vector S (3×1), we want to get the corresponding scaling matrix. We can use the following codes:

```
//Step 2: Build the Scale Matrix S_trans:
Eigen::Matrix4f S_trans;
S_trans << S[0],0,0,0,
           0,S[1],0,0,
           0,0,S[2],0,
           0,0,0,1;
```

1.3 Matrix for Rotation

We want to implement a rotation by θ around axis ω, which is defined by two points: $\omega = P_1 - P_0$. Here we would like to implement the Rodrigues' Rotation Formula: We can implement

$$
\begin{aligned}
\mathsf{R}_{\hat{\omega}}(\theta) &= e^{\tilde{\omega}\theta} \\
&= \mathsf{I} + \tilde{\omega}\sin\theta + \tilde{\omega}^2(1-\cos\theta) \\
&= \begin{bmatrix}
\cos\theta + \omega_x^2(1-\cos\theta) & \omega_x\omega_y(1-\cos\theta) - \omega_z\sin\theta & \omega_y\sin\theta + \omega_x\omega_z(1-\cos\theta) \\
\omega_z\sin\theta + \omega_x\omega_y(1-\cos\theta) & \cos\theta + \omega_y^2(1-\cos\theta) & -\omega_x\sin\theta + \omega_y\omega_z(1-\cos\theta) \\
-\omega_y\sin\theta + \omega_x\omega_z(1-\cos\theta) & \omega_x\sin\theta + \omega_y\omega_z(1-\cos\theta) & \cos\theta + \omega_z^2(1-\cos\theta)
\end{bmatrix},
\end{aligned}
$$

this formula using codes, before that, we should also normalize the axis:

```
    Eigen::Matrix4f P_trans;
    Eigen::Vector3f PToOrigin = P1 - P0;
    // Normalize vector
    Eigen::Vector3f P_axis;
    for(int i=0; i<3; i++){
        P_axis[i] = PToOrigin[i]/sqrt(pow(PToOrigin[0],2) + pow(PToOrigin[1],2) + pow(PToOrigin
            [2],2));
    }
    float ux = P_axis[0];
    float uy = P_axis[1];
    float uz = P_axis[2];
    float angle = rotation_angle/180 * MY_PI;
    P_trans << cos(angle) + pow(ux,2)*(1-cos(angle)), ux*uy*(1-cos(angle))-uz*sin(angle),/
              ux*uz*(1-cos(angle))+uy*sin(angle),0,
              uy*ux*(1-cos(angle))+uz*sin(angle), cos(angle) + pow(uy,2)*(1-cos(angle)),/
              uy*uz*(1-cos(angle))-ux*sin(angle),0,
              uz*ux*(1-cos(angle))-uy*sin(angle), uz*uy*(1-cos(angle))+ux*sin(angle),/
              cos(angle) + pow(uz,2)*(1-cos(angle)),0,
              0,0,0,1;

    // Combine the tree matrix into one matrix
    Eigen::Matrix4f model = S_trans*P_trans*M_trans;
    return model;
```

2 Implement *get_projection_matrix()*

2.1 Constructing Orthogonal Matrix

We want to project a cuboid into a 1^3 cube, we first should do the translation, which aims to move the center of the cuboid to the origin. Based on the parameters we get of the cube, we can use the following formula for the translation matrix:

$$\begin{pmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Next step, we do the scaling in order to convert the cuboid to a 1^3 cube:

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Next, we will get the ortho_matrix just multiplying the above two matrices.

```
Eigen::Matrix4f ortho;
ortho <<   2 / (right - left), 0, 0, (right + left) / (right - left),
           0, 2 / (top - bottom), 0, (top + bottom) / (top - bottom),
           0, 0, 2 / (zNear - zFar), (zFar + zNear) / (zFar - zNear),
           0, 0, 0, 1;
```

2.2 Projection to Orthonormal

I used the following formula to calculate this squeezing matrix:

$$\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -n \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Here follows the code implementation:

```
Eigen::Matrix4f PerstToOrtho;
PerstToOrtho << zNear, 0, 0, 0,
                0, zNear, 0, 0,
                0, 0, zNear + zFar, -zNear * zFar,
                0, 0, 1, 0;
projection = ortho * PerstToOrtho;

// Last Step is to combine them together
Eigen::Matrix4f projection = ortho * PerstToOrtho;
```

2.3 Compile and Run

In the homework dir, open the terminal and type:

```
// for using "A" and "D" to control the rotation
$ sh compile_run.sh

```

```
// OTHER WAYS
$ rm -rf build
$ cd build
$ cmake ..
$ make
$ ./HW2 -r 45
$ ./HW2 -r 45 result1.png
```

3 Results

In the following section, the positions of the triangle and the eye_pos and so on I defined are:

```
    // Position of the triangle
    std::vector<Eigen::Vector3f> pos{{2, 0, -2}, {0, 2, -2}, {-2, 0, -2}};

    // eye_pos
    Eigen::Vector3f eye_pos = {0, 0, 5};

    // some parameters settings to see the triangle
    float eye_fov = 45;
    float aspect_ratio = 1;
    float zNear = 0.1;
    float zFar = 50;
```

Base on the above settings, we can see the triangle looks like Figure 1:

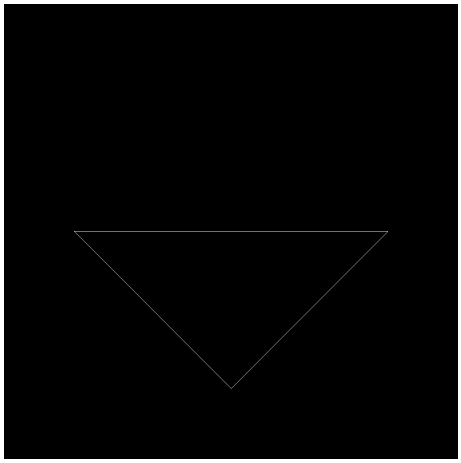
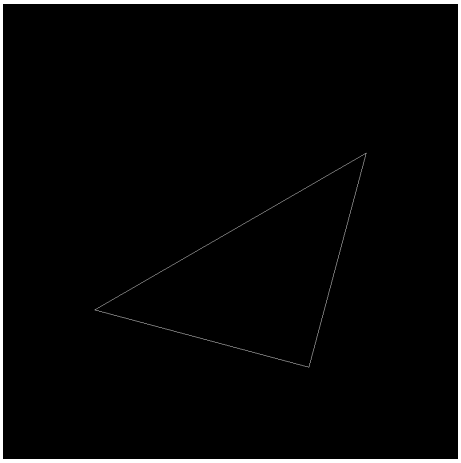


Figure 1: The initial triangle.

3.1 Rotation around z axis

First, we set the rotation axis to be z-axis, then we will get the following results in Figure 2 if we rotate 30 degree and will get the following results in Figure 3 if we rotate 90 degree.



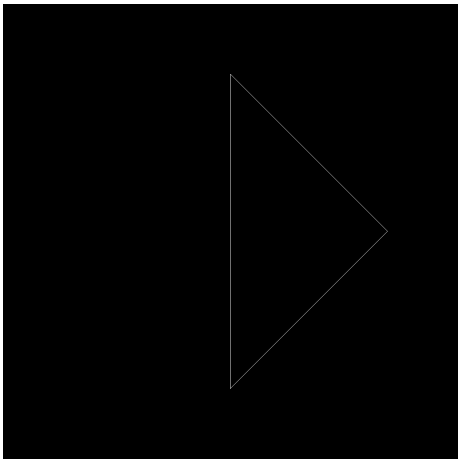


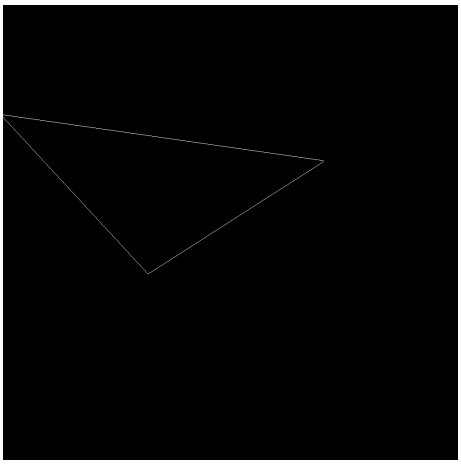
Figure 2: Rotation around z axis by 30 degree. Figure 3: Rotation around z axis by 90 degree.

3.2 Rotation around arbitrary axis

In this section, I defined my P0 and P1 so that the rotation axis is arbitrary instead of the xyz axis.

```
1     Eigen::Vector3f P1 = {3,0,1};
2     Eigen::Vector3f P0 = {0,-2,0};
```

Now Let's see the results of the rotation:



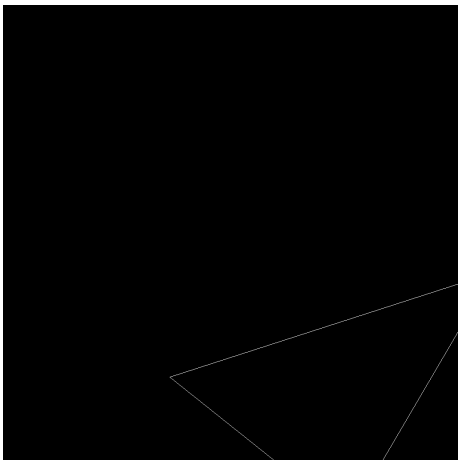


Figure 4: Rotation around axis by -40 degree. Figure 5: Rotation around axis by 50 degree.

4 Appendix

I also used the Eigen library to check whether the transformation matrices are calculated correctly:

```
1 frame count: 31
2 angle: -60
3 my model is
```

```
 0.765007 0.533135 -0.361291      0
0.0308479 0.530014 0.847427       0
 0.643283 -0.659433 0.389019      0
        0        0        0       1
Eigen model is
 0.765007 0.533135 -0.361291
0.0308479 0.530014 0.847427
 0.643283 -0.659433 0.389019
frame count: 32
angle: -70
my model is
 0.704874 0.617352 -0.349326      0
0.0909498 0.409749 0.907653       0
 0.703477 -0.671553 0.232673      0
        0        0        0       1
Eigen model is
 0.704874 0.617352 -0.349326
0.0909498 0.409749 0.907653
 0.703477 -0.671553 0.232673
frame count: 33
angle: -80
my model is
 0.642857 0.695833 -0.320237      0
  0.16131 0.285714 0.944641       0
 0.748808 -0.658927 0.0714286     0
        0        0        0       1
Eigen model is
 0.642857 0.695833 -0.320237
  0.16131 0.285714 0.944641
 0.748808 -0.658927 0.0714286
frame count: 34
angle: -90
my model is
   0.58084  0.766193 -0.274906        0
  0.239791   0.16168  0.957267        0
  0.777898 -0.621939 -0.0898162       0
         0         0         0        1
Eigen model is
   0.58084  0.766193 -0.274906
  0.239791   0.16168  0.957267
  0.777898 -0.621939 -0.0898162
```