Assignment 02

Name: An Zihang

Student ID: 121090001

0. Abstract

This assignment is about the MVP process. I set vertices, construct triangles and use Model, View and Projection matrices to transform it. With rasterizer provided, I can display triangles on the screen, and control its rotation angle around an arbitrarily given axis.

1. Implement Model Matrix

This is the matrix that "set a proper position, scaling and rotation angle" for the given objects. The given function **get_model_matrix** is divided into three steps:

```
Eigen::Matrix4f get_model_matrix(float rotation_angle, Eigen::Vector3f T,
Eigen::Vector3f S, Eigen::Vector3f P0, Eigen::Vector3f P1) {
    Eigen::Matrix4f M_trans = get_M_T(T);// get translation matrix
    Eigen::Matrix4f S_trans = get_M_S(S);// get scaling matrix
    Eigen::Matrix4f R_trans = get_M_R(P0, P1-P0, rotation_angle);// get
rotation matrix
    Eigen::Matrix4f model = R_trans * S_trans * M_trans; // mix the
transformations
    return model;
}
```

1.1 Translation matrix M_trans and Scaling matrix S_trans

Parameter **T**: The vector that represent the translation of the objects. I use the following lines to construct a Translation Matrix with **T**

```
Eigen::Matrix4f M_trans = Eigen::Matrix4f::Identity();
M_trans.col(3).head(3) = T;
return M_trans;
```

$$M\_trans = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

With similar way, we can get the Scaling Matrix with given scaling vector **S**

$$S\_trans = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

1.2 Rotation matrix R_trans

Parameter: rotation_angle (**in degree**), rotation axis u (given by $P_1 - P_0$)

I first normalize the axis and convert the rotation_angle into radian. Then by Rodrigues' Rotation Formular, I construct a matrix N such that $Nv = u \times v \, \forall v \in \{vertex \ vectors\}$

$$N = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix}$$

Then implement the Rodrigues' Rotation Formular

$$R(u, r) = \cos(r)I + (1 - \cos(r))uu^T + \sin(r)N$$

```
N << 0, -u[2], u[1],
     u[2], 0, -u[0],
     -u[1], u[0], 0;
Eigen::Matrix3f I = Eigen::Matrix3f::Identity();
R = cosf(radian) * I + (1 - cosf(radian)) * u*u.transpose() + sinf(radian) *
N;
```

2. Implement Perspective Projection Matrix

After putting objects and camera in proper positions, the next step is to project objects onto $[-1, 1]^3$

This step is partitioned into 2 parts.

2.1 Frustum to cuboid

Parameters: **N**(z coordinate of the near plane), **F**(z coordinate of the far plane)

Any point $\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$ between N and F is transformed to $\begin{pmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ z' \\ 1 \end{pmatrix}$, while $z' = z$ for$z = N$ or $z = F$. Solving this, we can get a matrix to do the transformation.

$$M_{pers \to ortho} = \begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & N - F & -NF \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

2.2 Orthographic Projection

Parameters: **n, f** (z borders), **l, r** (x borders), **t, b**(y borders)

To transform the cuboid into $[-1, 1]^3$, we need to first move its center to origin, then scale it.

$$M_{toOri} = \begin{bmatrix} 1 & 0 & 0 & -\frac{(r+l)}{2} \\ 0 & 1 & 0 & -\frac{(t+b)}{2} \\ 0 & 0 & 1 & -\frac{(n+f)}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad M_{cubic} = \begin{bmatrix} \frac{2}{(r-l)} & 0 & 0 & 0 \\ 0 & \frac{2}{(t-b)} & 0 & 0 \\ 0 & 0 & \frac{2}{(n-f)} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. Implement main() Function

3.1 Two execution modes (with GUI and without GUI)

There are two ways to generate photos of triangles.

If the program is run with parameter **-G**, it will open a window to show the triangle, and the user can press **a** or **d** on keyboard to increase or decrease the rotation angle. There are optional advanced parameters to add after **-G**, such as **angle** (in degree), and **coordinates of P0 and P1**. Caution, once the user wants to input the coordinates for P0 and P1, he/she has to input **all 6 parameters** that stands for xyz coordinates for P0 and P1.

If the program is run with parameter **-I**, it will not use GUI. Instead, it will write the result into a **1024 * 1024** image called **filename**. The optional advanced parameters after **filename** are the same as the **-G** instruction.

If there is no advanced parameters, the corresponding variables will be set as default (see custom settings below).

To see more explicit syntax for execution, go to README.md.

3.2 Custom settings

Parameter	Default value
eye_pos (position of camera)	(0,0,10)
pos (the set of points)	(-1,0,-5) (1,0,-5) (0,2,-5)
ind (the 3-vertex index group of each triangle)	(0,1,2)
T (translation)	(0,0,0)
S (scaling)	(1,1,1)
P0, P1 (two points on the axis)	(0,0,0) and (0,0,-1)
eye_fov, **aspect_ratio**	$\frac{\pi}{2}$ and 1
zNear, zFar (z coordinates of the frustum planes)	-1 and -7
key (listens to the user input)	no default value

3.3 Multiple triangles

The set of points are recorded by pushing 3d vectors into a vector called **pos**, and triangles are defined by pushing 3d vectors containing 3 point indices into a vector called **ind**. The vectors are loaded onto the rasterizer **r**, then the triangles can be drawn.

As mentioned above, the triangles are defined by a list of 3 points. So we can draw multiple triangles by loading many points and pushing multiple index groups to the **ind** vector. These will be shown in the **4th** part of this report.

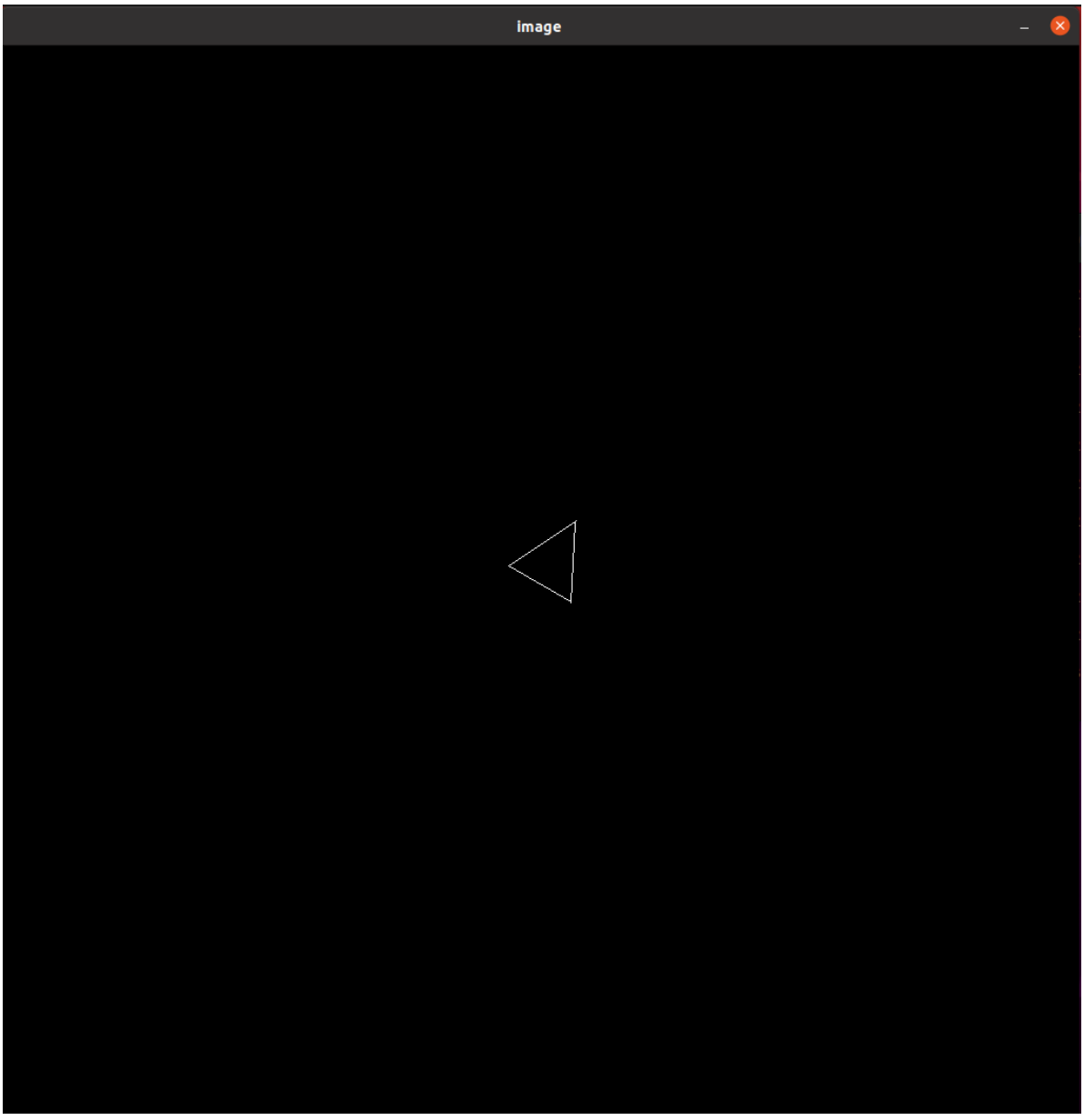
4. Draw, Transform and Show Triangles

Here I will show some execution results.

In section 4.1 and 4.2, the original triangle is (-1,0,-5)-(1,0,-5)-(0,2,-5), which is the default triangle.

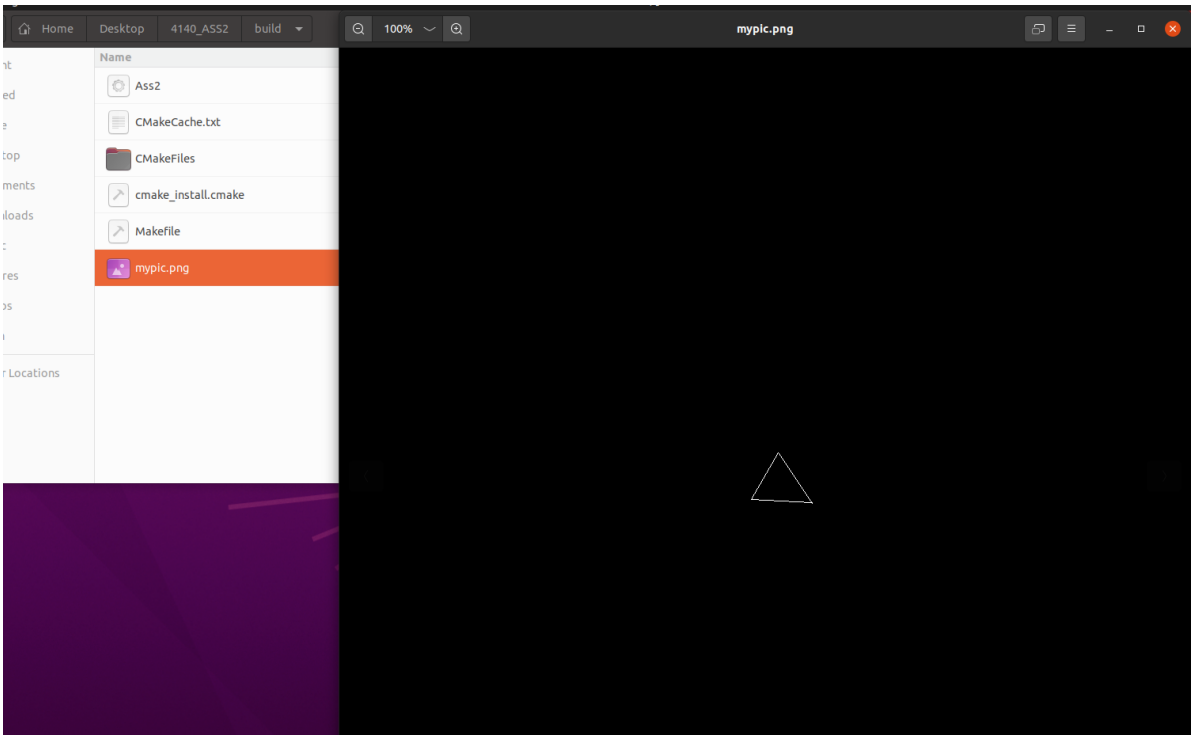
4.1 With GUI

```
# run with GUI, rotate 30 degree around axis determined by (0,0,0) and (0,0,-1)
./Ass2 -G 30 0 0 0 0 0 -1
```



4.2 Without GUI

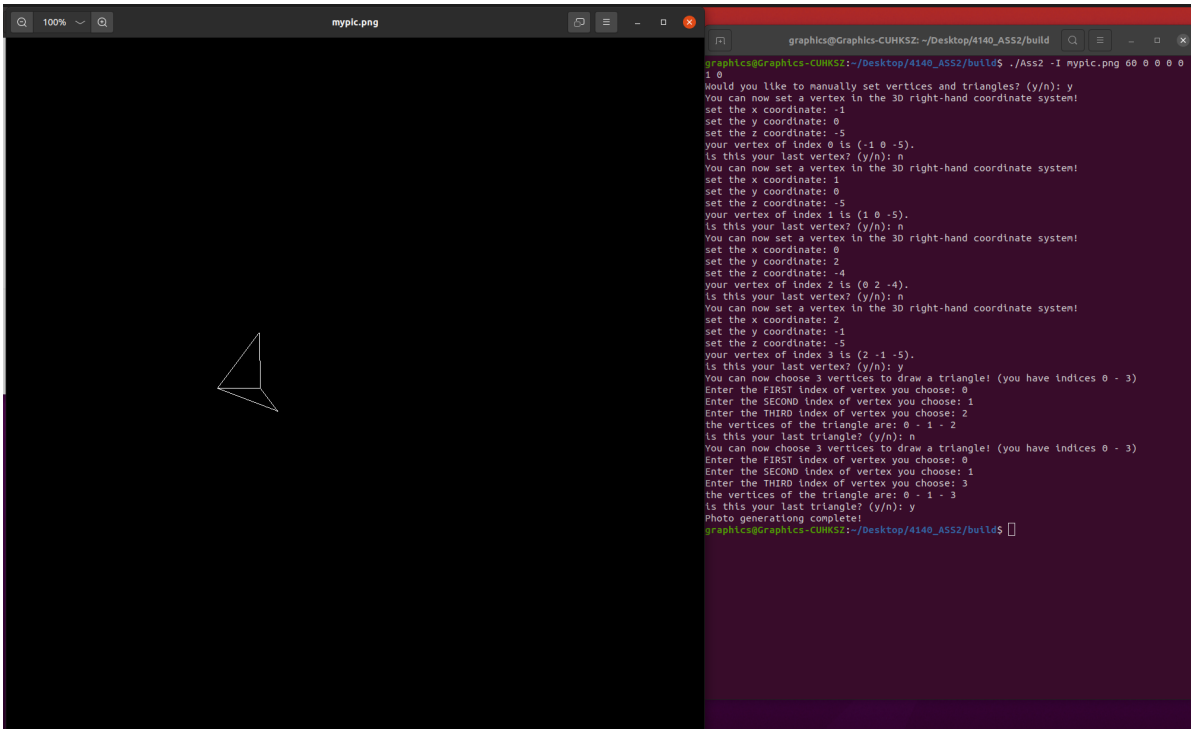
```
# output the image "mypic.png", rotate 120 degree around axis determined by (0,0,0) and (0,0,-1)
./Ass2 -I mypic.png 120 0 0 0 0 0 -1
```



4.3 Multiple triangles

Here I generated a image. There are 4 vertices (input manually) and 2 triangles, they rotate 60 degree around the axis determined by (0,0,0) and (0,1,0)

```
./Ass2 -I mypic.png 60 0 0 0 0 1 0
```



5. Additional Features

5.1 Arbitrary axis (not necessarily pass the origin)

Since the axis is defined by two points rather than a vector, it doesn't necessarily pass the origin. So I pass the point **P0** to the **get_M_R()** function. When doing rotation, I first move the objects by <$-P0$>, to make the axis pass the origin, then rotate, then move back. Mathematically, the Rotation Matrix will be:

$$R\_trans = \begin{bmatrix} 1 & 0 & 0 & P0_x \\ 0 & 1 & 0 & P0_y \\ 0 & 0 & 1 & P0_z \\ 0 & 0 & 0 & 1 \end{bmatrix} R \begin{bmatrix} 1 & 0 & 0 & -P0_x \\ 0 & 1 & 0 & -P0_y \\ 0 & 0 & 1 & -P0_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where R is the rotation matrix from Rodrigues Rotation Formular.

5.2 Support of user-input vertices and triangles (using terminal)

Besides the in-code settings of the points and triangles, I also implement the module that allow the user to input points and triangles via terminal.

```
int vcnt = -1;
char willing = '0';
while (willing != 'y' && willing != 'n') {
    std::cout << "Would you like to manually set vertices and triangles? (y/n): ";
    std::cin >> willing;
}
if (willing == 'y') {
    vcnt = 0;
    float* p_input = prompt_for_vertex(0);//container for vertices
    while (p_input[3] != 1) {
        pos.push_back(Eigen::Vector3f(p_input[0], p_input[1], p_input[2]));
```

```
        vcnt++;
        delete[] p_input;
        p_input = prompt_for_vertex(vcnt);
    }
    pos.push_back(Eigen::Vector3f(p_input[0], p_input[1], p_input[2]));
    delete[] p_input;
    if (vcnt < 2) {
        std::cout << "Error: too few vertices, expected at least 3" <<
std::endl;
        return 0;
    }
    int* i_input = prompt_for_index(vcnt);//container for index groups
    while (i_input[3] != 1) {
        ind.push_back(Eigen::Vector3i(i_input[0], i_input[1], i_input[2]));
        delete[] i_input;
        i_input = prompt_for_index(vcnt);
    }
    ind.push_back(Eigen::Vector3i(i_input[0], i_input[1], i_input[2]));
    delete[] i_input;
}
```

There are prompts for users to input correctly

```
float* prompt_for_vertex(int ind) {
    float px, py, pz, *res = new float[4];
    char t = '0';
    std::cout << "You can now set a vertex in the 3D right-hand coordinate
system!" << std::endl;
    std::cout << "set the x coordinate: ";
    std::cin >> px;
    std::cout << "set the y coordinate: ";
    std::cin >> py;
    std::cout << "set the z coordinate: ";
    std::cin >> pz;
    std::cout << "your vertex of index " << ind << " is (" << px << " " <<
py << " " << pz << ")." << std::endl;
    while (t != 'y' && t != 'n') {
        std::cout << "is this your last vertex? (y/n): ";
        std::cin >> t;
    }
    res[0]=px, res[1]=py, res[2] = pz, res[3] = (t == 'y' ? 1 : 0);
    return res; // the last element stands for "whether the input ends".
}
int* prompt_for_index(int maxind) {
    int p1=-1, p2=-1, p3=-1, *res = new int[4];
    char t = '0';
    std::cout << "You can now choose 3 vertices to draw a triangle! (you
have indices " << 0 << " - " << maxind << ")" << std::endl;
    while (p1 < 0 || p1 > maxind) {
        std::cout << "Enter the FIRST index of vertex you choose: ";
        std::cin >> p1;
    }
    while (p2 < 0 || p2 > maxind) {
        std::cout << "Enter the SECOND index of vertex you choose: ";
        std::cin >> p2;
```

```
    }
    while (p3 < 0 || p3 > maxind) {
        std::cout << "Enter the THIRD index of vertex you choose: ";
        std::cin >> p3;
    }
    std::cout << "the vertices of the triangle are: " << p1 << " - " << p2
<< " - " << p3 << std::endl;
    while (t != 'y' && t != 'n') {
        std::cout << "is this your last triangle? (y/n): ";
        std::cin >> t;
    }
    res[0]=p1, res[1]=p2, res[2] = p3, res[3] = (t == 'y' ? 1 : 0);
    return res; // the last element stands for "whether the input ends".
}
```