Assignment 3: Transformation and Projection

Mai Trinh

COSC 6372 Computer Graphics[*], University of Houston

March 25, 2021

1 Problem

The goal of this assignment is to integrate the transformation and projection into the *Gz* library. Given an input file containing the information of several triangles including the coordinates of 3 vertices, and its color, the application should be able to draw the image in different view. Several classes, functions, and data types are already given in the *Gz* library. We were to complete the implementation of the transformation and projection functionalities in the *Gz* class to render the output image.

2 Methods

The main challenge of this assignment is to map object coordinates to window coordinates. To do so, we need to create and update the transformation matrix and the projection matrix according to the given parameters. The transformation matrix defines how the objects are transformed (i.e. translated, rotated, and scaled) in the world coordinate frame. The projection matrix defines the properties of the camera that views the objects in the world coordinate frame (i.e. zoom factor, aspect ratio, near and far clipping frames).

Given the coordinates of the object, the algorithm must transform it into window coordinates. Let $v = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$ be a 4x1 matrix that represents the object coordinate. To compute the coordinate, we use the transformation matrix and projection matrix as follows:

$$v' = P * M * v \qquad [4]$$

where P is the current projection matrix and M is the current modelview matrix (i.e. translation matrix). The implementation of projection and transformation is discussed in detail in the next section.

To display the image on the computer screen, we must map our window to the viewport (i.e. the area on the display device to which a window is mapped). Thus, to compute the window coordinates, we must take into account the viewport coordinates (i.e. device coordinates). Finally, the window coordinates are then computed as follows [4]:

$$\begin{aligned} v'.x &= (v'.x + 1) * (w/2) + x \\ v'.y &= (v'.y + 1) * (h/2) + y \\ v'.z &= (v'.z + 1)/2 \end{aligned}$$

where *w,h* species the current viewport parameters. The viewport size is set to the size of the frame buffer. And *x,y* specifies the center of the viewport.

3 Implementation

Assume that the task of reading input from a file and storing it in appropriate vectors are correctly implemented in the main application. The coordinates *(x,y,z)* of each pixel is stored in *vertexQueue*, and the color of each pixel is stored in *colorQueue* in *Gz* library. For each triangle, we read in a set of 3 vertices and its color, update the coordinates vertex into matrix in homogeneous coordinate. The matrix is a 4x1 vector: $M[0][0] = x; M[1][0] = y, M[2][0] = z, M[3][0] = w$; w is set to 1 [5]. This conversion from vertex to matrix and vice versa is implemented in *GzMatrix* class. Since, this is not the focus of this assignment, I will not discuss the implementation of this function (look at source code for more details). The next task is to create and update the transformation matrix and projection matrix.

3.1 Transformation

Transformation matrix is a 4x4 matrix that when applied on an object can change its size, shape or orientation and position. There are several types of transformation including translate, rotate, scale.

3.1.1 Look At

The look-at function is used to set up the camera position and orientation (i.e how you see the object in the screen). Look-At function creates a viewing matrix derived from an eye point, the center of the scene, and an up vector. The eye point is the position where you looking from (or the camera position), the center point is the position where

[*]Course assignments

you looking to, and the up point determines which direction is up (i.e determines the orientation of the camera) which gives more constraint. With these 3 points, we can compute a 4x4 transformation matrix. The matrix will have the following form:

$$\begin{bmatrix} side_x & side_y & side_z & 0 \\ up_x & up_y & up_z & 0 \\ -forward_x & -forward_y & -forward_z & 0 \\ -eye_x & -eye_y & -eye_z & 1 \end{bmatrix}$$

The forward axis is the line along the center points to the eye points. The side axis is the cross product of the up vector and the forward vector. The side vector is perpendicular to the forward vector and can be used to construct our Cartesian coordinate system. To compute the new up vector, we use the cross product between the side vector and the forward vector, since the cross product of 2 orthogonal vectors will give us the third vector. We then normalized all vectors. The current matrix is multiplied by the matrix generated by look-at function. The implementation of the look-at function is based off of gluLookAt function [3]. The code snippet of the look-at function is shown below (assume that the function to normalize a vector and perform cross product is implemeted correctely):

```
void Gz::lookAt(GzReal eyeX, GzReal eyeY, GzReal eyeZ, GzReal centerX, GzReal centerY, GzReal centerZ,
                GzReal upX, GzReal upY, GzReal upZ) {
    GzReal forward[] = {centerX - eyeX, centerY - eyeY, centerZ - eyeZ};
    normalizeVector(forward);
    GzReal up[] = { upX, upY, upZ };
    normalizeVector(up);
    GzReal side[3];
    crossProduct(side, forward, up); // side = forward x up
    normalizeVector(side);

    /* Recompute up as: up = side x forward */
    GzReal newUp[3];
    crossProduct(newUp, side, forward);

    /* Setting transformation matrix M
    * M = {side[0],      side[1],      side[2],      0.0
    *        up[0],        up[1],        up[2],        0.0
    *       -forward[0], -forward[1], -forward[2], 0.0
    *        0.0,          0.0,          0.0,          1.0}
    */
    multMatrix(M);
    translate(-eyeX, -eyeY, -eyeZ);}
```

3.1.2 Translate

A translation process moves every point a constant distance in a specified direction. Using homogeneous coordinates, we can represent a translation of a vector space with matrix multiplication. To translate an object by a vector v, we multiply the current matrix by this translation matrix:

$$\begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad [8]$$

A snippet of code for translate function is shown below:

```
void Gz::translate(GzReal x, GzReal y, GzReal z) {}
    GzMatrix M;
   M.at(0)[0] = 1; M.at(0)[1] = 0; M.at(0)[2] = 0; M.at(0)[3] = x;
   M.at(1)[0] = 0; M.at(1)[1] = 1; M.at(1)[2] = 0; M.at(1)[3] = y;
   M.at(2)[0] = 0; M.at(2)[1] = 0; M.at(2)[2] = 1; M.at(2)[3] = z;
   M.at(3)[0] = 0; M.at(3)[1] = 0; M.at(3)[2] = 0; M.at(3)[3] = 1;
    multMatrix(M);}
```

3.1.3 Rotate

Rotation is the motion of object around a fixed point. The rotation matrix produces a rotation of angle degrees around a vector. The current matrix is then multiplied by this rotation matrix to get a new transformation matrix. The implementation of rotation, based on glRotate [2], is shown below:

```
void Gz::rotate(GzReal angle, GzReal x, GzReal y, GzReal z) {
    angle = angle * M_PI / 180;
    GzReal c = cos(angle);
    GzReal s = sin(angle);
    GzReal v[] = { x,y,z };
    normalizeVector(v);
    /* M = {v[X]*v[X]*(1-c)+c, v[X]*v[Y]*(1-c)-v[Z]*s, v[X]*v[Z]*(1-c)+v[Y]*s, 0
    *       v[Y]*v[X]*(1-c)+v[Z]*s, v[Y]*v[Y]*(1-c)+c, v[Y]*v[Z]*(1-c)-v[X]*s, 0
    *       v[X]*v[Z]*(1-c)-v[Y]*s, v[Y]*v[Z]*(1-c)+v[X]*s, v[Z]*v[Z]*(1-c)+c, 0
    *       0, 0, 0, 1}
    */
    multMatrix(M);}
```

3.1.4 Scale

Scaling is a linear transformation that enlarges or shrinks an object by a scale factor. To scale an object by a vector v, the current matrix would need to be multiplied by a scaling matrix in the form:

$$\begin{bmatrix} v_x & 0 & 0 & 0 \\ 0 & v_y & 0 & 0 \\ 0 & 0 & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad [7]$$

The implementation of scaling is shown below:

```
void Gz::scale(GzReal x, GzReal y, GzReal z) {
    GzMatrix M;
   M.at(0)[0] = x; M.at(0)[1] = 0; M.at(0)[2] = 0; M.at(0)[3] = 0;
   M.at(1)[0] = 0; M.at(1)[1] = y; M.at(1)[2] = 0; M.at(1)[3] = 0;
   M.at(2)[0] = 0; M.at(2)[1] = 0; M.at(2)[2] = z; M.at(2)[3] = 0;
   M.at(3)[0] = 0; M.at(3)[1] = 0; M.at(3)[2] = 0; M.at(3)[3] = 1;
    multMatrix(M);}
```

3.2 Projection

Projection matrix is a 4x4 matrix that used to display a 3D object on a 2D screen. By multiplying a 3D point in camera space by the projection matrix, we get the 2D coordinates of this point on the screen. There are 2 graphical projection categories: parallel projection (orthographic projection) and perspective projection [1].

3.2.1 Perspective

Perspective projections is a linear projection where 3D objected are projected on a picture plane and make it appears like a view from a real-world camera [1]. Objects further from the camera appear smaller than objects that are nearer to the camera. All lines appear to project toward vanishing points which skew parallel lines. To compute a perspective projection matrix, we need 4 parameters: *fovy, aspect, near, far. fovy* is the field of view y-axis parameter determines the vertical angle of the camera's lens. The bigger the angle is, the more you can see the world, and the objects become smaller. The *aspect* ratio is the viewport's aspect ratio determining the field of view in the x-direction. The *near* and *far* parameters are the distance (z-coordinate) between the camera position to the near clipping plane and the far clipping plane, respectively. We make the assumption that both values are positive, and *near* is always less than *far*. The perspective project matrix is in the form:

$$\begin{bmatrix} \frac{1}{aspect*tan\left(\frac{fovy}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{tan\left(\frac{fovy}{2}\right)} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2*far*near}{far-near} \\ 0 & 0 & -1 & 1 \end{bmatrix} \quad [7]$$

```
void Gz::perspective(GzReal fovy, GzReal aspect, GzReal zNear, GzReal zFar) {
    // convert fovy angle from degree to radian
    fovy = fovy * M_PI / 180;
    GzReal f = tan(fovy / 2);
    GzMatrix M;
   M[0][0]=1/(aspect*f); M[0][1] = 0.0; M[0][2] = 0.0; M[0][3] = 0.0;
   M[1][0]=0.0; M[1][1] = 1 / f; M[1][2] = 0.0; M[1][3] = 0.0;
   M[2][0]=0.0; M[2][1] = 0.0; M[2][2] = (zFar+zNear)/(zFar-zNear); M[2][3] = (2*zFar*zNear)/(zFar-zNear);
   M[3][0]=0.0; M[3][1] = 0.0; M[3][2] = -1; M[3][3] = 0.0;
    prjMatrix = M;}
```

3.2.2 Orthographic

In orthographic projection, all the projection lines are orthogonal to the projection plane. No matter how far away vertexs are in the z-direction, they will not recede into the distance. To compute the projection matrix for orthographic projection, we need the coordinates of the screen (left, right, top, bottom). We also need to specify the distances to the nearer and farther depth screen. Now, we need to remap the left and right screen coordinates (x-coordinate), the top and bottom (i.e. y-coordinates) and the near and far plane (i.e. z-coordinates) from -1 to 1. And finally putting it together will give us the orthographic matrix:

$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad [6]$$

The implementation of the orthographic matrix is shown below:

```
void Gz::orthographic(GzReal left, GzReal right, GzReal bottom, GzReal top, GzReal nearVal, GzReal farVal){
    GzReal tx = -(right + left) / (right - left);
    GzReal ty = -(top + bottom) / (top - bottom);
    GzReal tz = -(farVal + nearVal) / (farVal - nearVal);
    GzMatrix M;
   M.at(0)[0] = 2 / (right - left); M.at(0)[1] = 0.0; M.at(0)[2] = 0.0; M.at(0)[3] = tx;
   M.at(1)[0] = 0.0; M.at(1)[1] = 2 / (top - bottom); M.at(1)[2] = 0.0; M.at(1)[3] = ty;
   M.at(2)[0] = 0.0; M.at(2)[1] = 0.0; M.at(2)[2] = 2 / (farVal - nearVal); M.at(2)[3] =-tz;
   M.at(3)[0] = 0.0; M.at(3)[1] = 0.0; M.at(3)[2] = 0.0; M.at(3)[3] = 1.0;
    prjMatrix = M;}
```

4 Results

The results of the implementation are shown below. Each image displays a different set of parameters for transformation and projection. For example, the third image and the fifth image have the same look-at parameters and applied the same translation function; but one with orthographic projection and the other uses perspective projection. The latter shows that the teapot seems to be further away from the camera because it recedes back which is the result of using perspective projection. Note that the parameters of the teapot remains the same (because we use the same input file), we change the parameters of transformation and projection matrix which in turns changing the way we look at the object (i.e. changing the position and orientation of the camera).

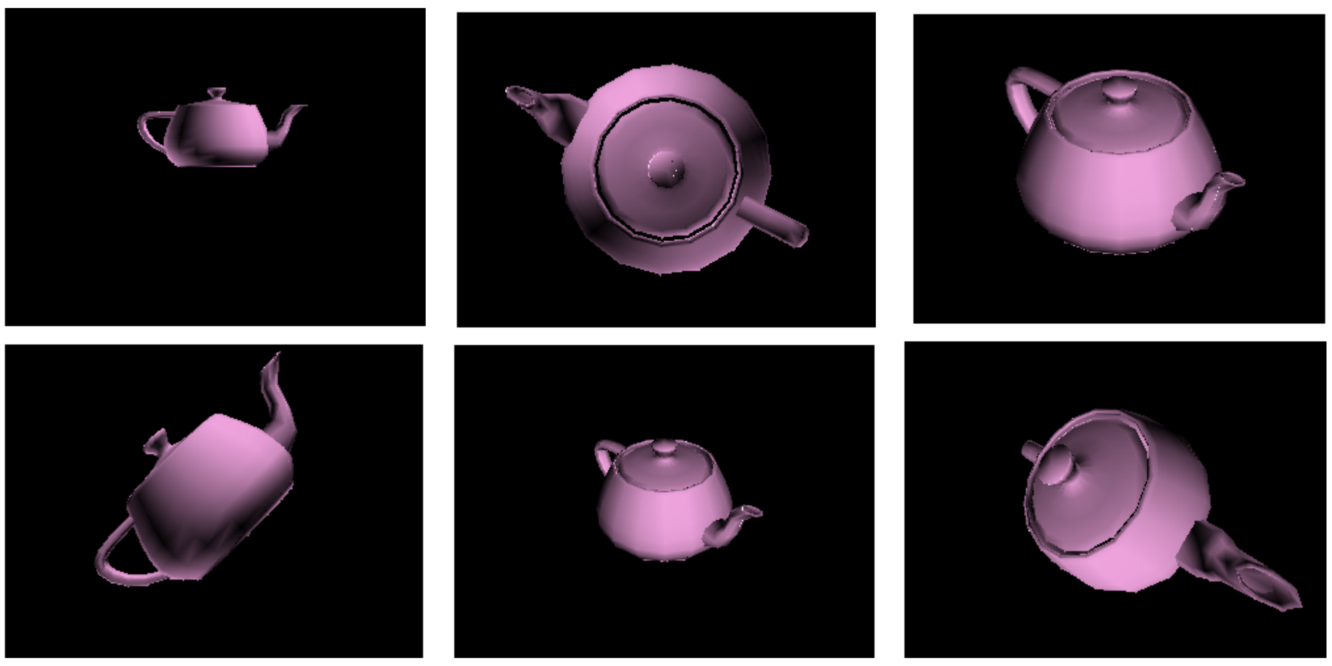

Figure 1: Result images

References

[1] *3D Projection.* URL: `https://en.wikipedia.org/wiki/3D_projection`.

[2] *glRotate.* URL: `http://docs.gl/gl3/glRotate`.

[3] *gluLookAt.* URL: `http://code.nabla.net/doc/OpenGL/api/OpenGL/man/gluLookAt.html`.

[4] *gluProject.* URL: `https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/gluProject.xml`.

[5] *Homogeneous coordinates.* URL: `https://en.wikipedia.org/wiki/Homogeneous_coordinates#Use_in_computer_graphics`.

[6] *Orthographic projection.* URL: `https://en.wikipedia.org/wiki/Orthographic_projection`.

[7] *Scaling (geometry).* URL: `https://en.wikipedia.org/wiki/Scaling_(geometry)`.

[8] *Translation (geometry).* URL: `https://en.wikipedia.org/wiki/Translation_(geometry)`.