COSC6372 HW3 – Transformation and Projection

Jiechang Guo UHID: 2084258

March 18, 2023

1 Problem

The problem for this assignment is to render the teapot with different camera poses, object transformations, and different projection modes.

2 Method

In this assignment, the transformation and projection are integrated into the Gz library. The object will be transformed from local space to world space by the model matrix(transMatrix), then transformed from world space to view space by the view matrix. In order to be rendered on screen, a projection matrix will apply to the vertices which transforms the object into clip space. Finally, after the viewport transformation, the vertices are represented in the screen space. The process is shown in figure 1.

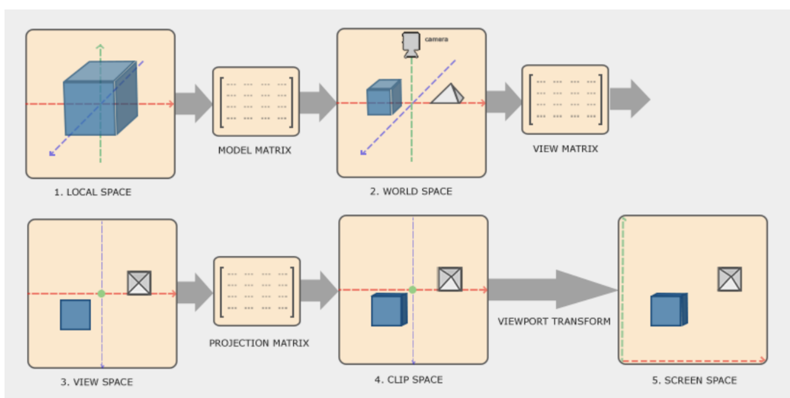

Figure 1: 2D bounding box of a triangle[5]

2.1 Model matrix

Homogeneous coordinates Before we get into the model matrix, the idea of homogeneous coordinates is important which is the w component of a vector. Only with the homogeneous coordinates we can do matrix manipulation on 3D vectors. If the 3D vector is represented as a position of a vertex, the w component of the vector should be 1.0, however, if the 3D vector is the normal vector of a face of a model which is a direction vector, the w component will be 0.0. Moreover, the perspective effect of 3D is based on the w component[6].
From vertex to homogeneous coordinates

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1.0 \end{pmatrix}$$

From homogeneous coordinates to vertex

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}$$

In the rest part of the section, the mathematical representation of translate, rotate, and scale transformation will be introduced. For the implementation of translate, rotate, and scale, please check the source code.
Translate Translation will move the object to a new position according to the translation vector $[T_x, T_y, T_z]$.

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

Rotate The rotation of the 3D object is represented by an angle and a rotation axis. Let the angle be θ, the rotation is a normalized vector $[R_x, R_y, R_z]$.

$$\begin{bmatrix} cos\theta + {R_x}^2(1-cos\theta) & R_xR_y(1-cos\theta) - R_zsin\theta & R_xR_z(1-cos\theta) + R_ysin\theta & 0 \\ R_yR_x(1-cos\theta) + R_zsin\theta & cos\theta + {R_y}^2(1-cos\theta) & R_yR_z(1-cos\theta) - R_xsin\theta & 0 \\ R_zR_x(1-cos\theta) - R_ysin\theta & R_zR_y(1-cos\theta) + R_xsin\theta & cos\theta + {R_z}^2(1-cos\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale For the scale transformation, we keep the direction of the original vector but change the length of the vector according to the scale variables as $[S_x, S_y, S_z]$.

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x * S_x \\ y * S_y \\ z * S_z \\ 1 \end{pmatrix}$$

Combining matrice The transform matrices will apply to the vertex in the order of the sequence being set to the library. The order of translate, rotate and scale transformation matters. It is recommended to scale and rotate the model first, then translate it[6]. The equation to transform the object from local space to world space is as followed. The right-most matrix is first multiplied by the vertex vector. Please also see the implementation section.

$$V\_world = T * R * S * V\_local$$

2.2 View matrix

The view matrix transforms the vertices in the world coordinates into view coordinates. To define a view space, the position of the camera/eye, its forward direction which is the direction the camera or the eye is looking at, to define the pose of the camera, right and up vectors are needed in order to create a coordinate system origin at the camera[4].

Look At

Let $eye = [eyeX, eyeY, eyeZ]$ be the position of the camera, $center = [centerX, centerY, centerZ]$ be the target position, $up = [upX, upY, upZ]$ be the direction of the up vector. D is the forward axis of the camera, R is the right axis of the camera, U is the up axis of the camera.

$$D = \begin{pmatrix} D_x \\ D_y \\ D_z \end{pmatrix} = \begin{pmatrix} centerX \\ centerY \\ centerZ \end{pmatrix} - \begin{pmatrix} eyeX \\ eyeY \\ eyeZ \end{pmatrix}$$

The up and D vectors will be normalized.

$$R = crossproduct(D, up)$$

$$U = crossproduct(R, D)$$

$$\begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.3 Projection matrix

Orthographic projection A orthographic projection matrix is defined by the coordinates of the left and right vertical clipping planes, the bottom and top horizontal clipping planes, and also the distance from the viewer to the near and far clipping planes[2].

$$tx = -\frac{right + left}{right - left}$$

$$ty = -\frac{top + bottom}{top - bottom}$$

$$tz = -\frac{falVal + nearVal}{falVal - nearVal}$$

$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & tx \\ 0 & \frac{2}{top-bottom} & 0 & ty \\ 0 & 0 & \frac{2}{falVal-nearVal} & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective projection A perspective projection matrix is defined by the field of view angle, the aspect ratio, and the near and far clipping plane[3].

$$f = cotangent(\frac{fovy}{2})$$

$$\begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zFar+zNear}{zNear-zFar} & \frac{2*zFar*zNear}{zNear-zFar} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

2.4 Viewport transformation

Until now, we get the model in the clip place, to get the vertices in screen space, a viewport transformation, also called affine transformation will be the final step. The lower left corner of the viewport rectangle is defined as (x, y), the width and height of the viewport are needed too. Let(x_nd, y_nd) be normalized device coordinates, and the screen space coordinates (x_w, y_w) will be computed using the equation below[1].

$$x_w = (x_nd + 1)(\frac{width}{2} + x)$$

$$y_w = (y_nd + 1)(\frac{height}{2} + y)$$

$$z_w = \frac{z_nd + 1}{2}$$

3 Implementation

Combining transform matrice I changed the source code of the Gz::mulMatrix function, in order to keept the order of the transformation.

```
void Gz::multMatrix(GzMatrix mat) {
  //Multiply transMatrix by the matrix mat

  //transMatrix=mat*transMatrix;

  //the order of the matrix multiply do matters. The matrix being set first should be applied to
    the vertex first.
  //the oder of the transformation should be scale, rotation, translate.
  //V_world = T*R*S*V_local
  transMatrix= transMatrix * mat;
}
```

View matrix Although the model matrix and view matrix are usually considered as a single matrix called model-view matrix. In this assignment, I implemented it separately because this is more intuitive.

```
    void Gz::lookAt(GzReal eyeX, GzReal eyeY, GzReal eyeZ, GzReal centerX, GzReal centerY, GzReal
     centerZ, GzReal upX, GzReal upY, GzReal upZ) {
  //Define viewing transformation
  //See http://www.opengl.org/sdk/docs/man/xhtml/gluLookAt.xml
  //Or google: gluLookAt

  //set transMatrix and prjMatrix to default
  transMatrix = Identity(4);
  prjMatrix = Identity(4);

  GzMatrix center_v;
  center_v.resize(3,1);
  center_v[0][0] = centerX;
  center_v[1][0] = centerY;
  center_v[2][0] = centerZ;

  GzMatrix eye_v;
  eye_v.resize(3,1);
  eye_v[0][0] = eyeX;
  eye_v[1][0] = eyeY;
```

```
  eye_v[2][0] = eyeZ;

  GzMatrix F;
  F.resize(3,1);
  F = center_v - eye_v;
  F.Normalize();

  GzMatrix up_v;
  up_v.resize(3,1);
  up_v[0][0] = upX;
  up_v[1][0] = upY;
  up_v[2][0] = upZ;
  up_v.Normalize();

  GzMatrix s;
  s.resize(3,1);
  s = crossProduct(F,up_v);
  s.Normalize();

  //GzMatrix s_normlized = s;
  //s_normlized.Normalize();

  GzMatrix u;
  u.resize(3,1);
  u = crossProduct(s,F);

  viewMatrix = Identity(4);
  viewMatrix[0][0] = s[0][0];
  viewMatrix[0][1] = s[1][0];
  viewMatrix[0][2] = s[2][0];

  viewMatrix[1][0] = u[0][0];
  viewMatrix[1][1] = u[1][0];
  viewMatrix[1][2] = u[2][0];

  viewMatrix[2][0] = -F[0][0];
  viewMatrix[2][1] = -F[1][0];
  viewMatrix[2][2] = -F[2][0];

  GzMatrix translate = Identity(4);
  translate[0][3] = -eyeX;
  translate[1][3] = -eyeY;
  translate[2][3] = -eyeZ;

  viewMatrix = viewMatrix * translate;
}
```

Putting MVP all together

```
    void Gz::end() {
  //This function need to be updated since we have introduced the viewport,
  //projection, and transformations.
  //In our implementation, all rendering is done when Gz::end() is called.
  //Depends on selected primitive, different number of vetices, colors, ect.
  //are pop out of the queue.
  switch (currentPrimitive) {
    case GZ_POINTS: {
      while ( (vertexQueue.size()>=1) && (colorQueue.size()>=1) ) {
      }
    } break;
    case GZ_TRIANGLES: {

      //Put your triangle drawing implementation here:
      //   - Extract 3 vertices in the vertexQueue
      //   - Extract 3 colors in the colorQueue
      //   - Call the draw triangle function
      //      (you may put this function in GzFrameBuffer)
      while ( (vertexQueue.size()>=3) && (colorQueue.size()>=3) ) {
        GzVertex v1=vertexQueue.front(); vertexQueue.pop();
        GzVertex v2=vertexQueue.front(); vertexQueue.pop();
        GzVertex v3=vertexQueue.front(); vertexQueue.pop();
        vector<GzVertex> vertices;
        vertices.push_back(v1);
        vertices.push_back(v2);
```

```
        vertices.push_back(v3);

        //apply model_view, projection matrix
        for(GzInt i = 0; i < vertices.size(); i++)
        {
          GzMatrix v_mat;
          //create a vector from vertex in object space
          v_mat.fromVertex(vertices[i]);
          //transform the object to world space then to view space
          //v_mat = v_mat;
          //transform to clip space
          v_mat = prjMatrix*viewMatrix*transMatrix*v_mat;
          vertices[i] = v_mat.toVertex();
          //viewport transform to screen space
          vertices[i][0] = (vertices[i][0] + 1.0)*(wViewport/2.0) + xViewport;
          vertices[i][1] = (vertices[i][1] + 1.0)*(hViewport/2.0) + yViewport;
          vertices[i][2] = (vertices[i][2] + 1.0)/2.0;
        }

        GzColor c1=colorQueue.front(); colorQueue.pop();
        GzColor c2=colorQueue.front(); colorQueue.pop();
        GzColor c3=colorQueue.front(); colorQueue.pop();

        vector<GzColor> colors;
        colors.push_back(c1);
        colors.push_back(c2);
        colors.push_back(c3);


        frameBuffer.drawTriangle(vertices,colors,status);
      }
    }
  }
}
```

4 Result

In this section, the final result of the rendered teapots will be illustrated.
The final result is shown in figure 2

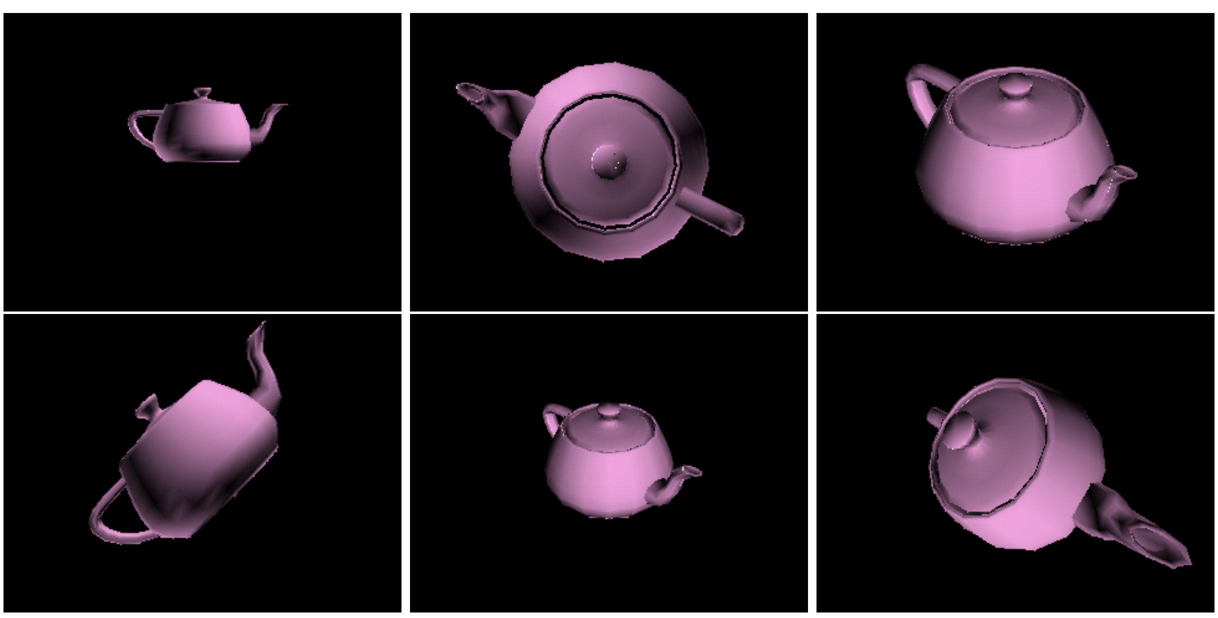

Figure 2: From left to right, from top to bottom. Teapot1, Teapot2, Teapot3, Teapot4, Teapot5, Teapot6.

Order matters
If the order of transformation does not follow the rule that scales first, then rotates, and translates in the end, the result won't be correct. In our case, the teapot 6 is supposed to be first rotated 45 degrees along the x-axis, then translated. In figure 3, the right teapot was not rendered correctly. Because it applied the translation matrix first.

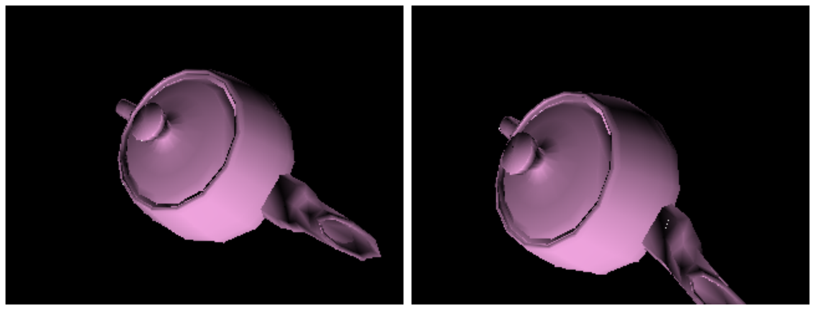

Figure 3: The result with wrong transformation order. Left: rotate first. Right translate first.

References

[1] Microsoft. *glViewport function*. `https://learn.microsoft.com/en-us/windows/win32/opengl/glviewport`.

[2] OpenGL. *glOrtho*. `https://registry.khronos.org/OpenGL-Refpages/gl2.1/xhtml/glOrtho.xml`.

[3] OpenGL. *gluPerspective*. `https://registry.khronos.org/OpenGL-Refpages/gl2.1/xhtml/gluPerspective.xml`.

[4] Joey de Vries. *Camera*. `https://learnopengl.com/Getting-started/Camera`.

[5] Joey de Vries. *Coordinate Systems*. `https://learnopengl.com/Getting-started/Coordinate-Systems`.

[6] Joey de Vries. *Transformations*. `https://learnopengl.com/Getting-started/Transformations`.