CSE 410 – July 2018

Assignment 3

Raster Based Graphics Pipeline

Overview:

In this assignment, you will develop the raster based graphics pipeline used in OpenGL. The pipeline can be thought of as a series of four stages:

1. Stage 1: modeling transformation
2. Stage 2: view transformation
3. Stage 3: clipping and projection transformation
4. Stage 4: scan conversion using z-buffer algorithm

Your program will output four files: `stage1.txt`, `stage2.txt`, `stage3.txt` and `out.bmp`. The first three files will contain the output of the first three stages, respectively. The fourth file will be a bmp image generated by the pipeline.

Scene description:

You will be given a text file named "`scene.txt`". This will contain the following lines:

Line 1: eyeX eyeY eyeZ
Line 2: lookX lookY lookZ
Line 3: upX upY upZ
Line 4: fovY aspectRatio near far
Line 5: screen_width screen_height
Line 6: r g b

eyeX, eyeY and eyeX are eye position. lookX, lookY, lookZ denote 3D point where eye is looking. upX, upY and upZ are the up vector. These nine numbers are the nine parameters of the `gluLookAt` function.

Line 4 gives the field of view along Y axis, followed by the aspect ratio. Following two numbers indicate the near and far plane.

Line 5 contains two integers that indicate the size of the image. Then, line 6 contains **r g b** which denote the components of background color and can have any integer value between 0 and 255.

The rest of `scene.txt` contains the *display code* to generate/draw the model. The display code contains 7 commands as follows:

1. triangle command – this command is followed by four lines. The first three lines specify the coordinates of the three points of the triangle to be drawn. The points being p1, p2, and p3, 9 double values, i.e., p1.x, p1.y, p1.z, p2.x, p2.y, p2.z, p3.x, p3.y, and p3.z indicate the coordinates. This is equivalent to the following in OpenGL code.

   ```
   glBegin(GL_TRIANGLE);{
         glVertex3f(p1.x, p1.y, p1.z);
         glVertex3f(p2.x, p2.y, p2.z);
         glVertex3f(p3.x, p3.y, p3.z);
   }glEnd();
   ```

   The fourth line contains the r, g, b values of the triangle respectively, and can have any integer value with in range 0 and 255.

2. translate command – this command is followed by 3 double values (tx, ty, and tz) in the next line indicating translation amounts along X, Y, and Z axes. This is equivalent to `glTranslatef(tx, ty, tz)` in OpenGL.

3. scale command – this command is followed by 3 double values (sx, sy, and sz) in the next line indicating scaling factors along X, Y, and Z axes. This is equivalent to `glScalef(sx, sy, sz)` in OpenGL.

4. rotate command – this command is followed by 4 double values in the next line indicating the rotation angle in degree (angle) and the components of the vector defining the axis of rotation(ax, ay, and az). This is equivalent to `glRotatef(angle, ax, ay, az)` in OpenGL.

5. push command – This is equivalent to `glPushMatrix` in OpenGL.

6. pop command – This is equivalent to `glPopMatrix` in OpenGL.

7. end command – This indicates the end of the display code.

Please check the scene.txt carefully to have a more comfortable understanding of the input.

Stage 1: Modeling Transformation

In the Modeling transformation phase, the display code in `scene.txt` is parsed, the transformed positions of the points that follow each triangle command are determined, and the transformed coordinates of the points are written in `stage1.txt` file. We maintain a stack S of transformation matrices which is manipulated according to the commands given in the display code. The pseudo-code for the modeling transformation phase is as follows:

```
initialize empty stack S
S.push(identity matrix)
while true
      input command
      if command = "triangle"
            input three points and the color of the triangle
            for each of these three point P
                  P' <- transformPoint(S.top,P)
                  output P'
      else if command = "translate"
            input translation amounts
            generate the corresponding translation matrix T
            S.push(product(S.top,T))
      else if command = "scale"
            input scaling factors
            generate the corresponding scaling matrix T
            S.push(product(S.top,T))
      else if command = "rotate"
            input rotation angle and axis
            generate the corresponding rotation matrix T
            S.push(product(S.top,T))
      else if command = "push"
            //do it yourself
      else if command = "pop"
            //do it yourself
      else if command = "end"
            break
```

Transformation matrix for Translation

```
translate
tx ty tz
```

The transformation matrix for the above translation is as follows:

$$\begin{vmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Transformation matrix for Scaling

```
scale
sx sy sz
```

The transformation matrix for the above scaling is as follows:

$$\begin{vmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Transformation matrix for Rotation

Remember that, the columns of the rotation matrix indicate where the unit vectors along the principal axes (namely, `i`, `j`, and `k`) are transformed. We will use the vector form of Rodrigues formula to determine where `i`, `j`, and `k` are transformed and use those to generate the rotation matrix. The vector form of Rodrigues formula is as follows:

$$R(\vec{x},\vec{a},\theta) = \cos\theta\,\vec{x} + (1-\cos\theta)(\vec{a}.\vec{x})\,\vec{a} + \sin\theta(\vec{a} X \vec{x})$$

In the above formula, $\vec{a}$ is a unit vector defining the axis of rotation, θ is the angle of rotation, and $\vec{x}$ is the vector to be rotated.

Now we outline the process of generating transformation matrix for the following rotation:

```
rotate
angle ax ay az
```

We denote the vector `(ax, ay, az)` by `a`. The steps to generate the rotation matrix are as follows:

```
a.normalize()
c1=R(i,a,angle)
c2=R(j,a,angle)
c3=R(k,a,angle)
```

The corresponding rotation matrix is given below:

$$\begin{vmatrix} c1.x & c2.x & c3.x & 0 \\ c1.y & c2.y & c3.y & 0 \\ c1.z & c2.z & c3.z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Managing Push and Pop

The following table demonstrates how `push` and `pop` works. The state of the transformation matrix stack after execution of each line of the code in the left is shown in the right. Design a data structure that manages `push` and `pop` operations on the transformation matrix stack accordingly.

	Stack State after Lines											
Code	0	1	2	3	4	5	6	7	8	9	10	11
1.Push												
2.Translate$_1$												
3.Push												
4.Rotate$_1$												
5.Pop												
6.Scale$_1$												
7.Push												
8.Rotate$_2$									$T_1S_1R_2$		$T_1S_1S_2$	
9.Pop					T_1R_1		T_1S_1	T_1S_1	T_1S_1	T_1S_1	T_1S_1	
10.Scale$_2$			T_1	T_1	T_1	T_1	T_1	T_1	T_1	T_1	T_1	
11.Pop	I	I	I	I	I	I	I	I	I	I	I	I

Stage 2: View Transformation

In the view transformation phase, the `gluLookAt` parameters in `scene.txt` is used to generate the view transformation matrix *V*, and the points in `stage1.txt` are transformed by *V* and written in `stage2.txt`. The process of generating *V* is given below.

First determine mutually perpendicular unit vectors `l`, `r`, and `u` from the `gluLookAt` parameters.

```
l = look - eye
l.normalize()
r = l X up
r.normalize()
u = r X l
```

Apply the following translation *T* to move the eye/camera to origin.

$$\begin{bmatrix} 1 & 0 & 0 & -eyeX \\ 0 & 1 & 0 & -eyeY \\ 0 & 0 & 1 & -eyeZ \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Apply the following rotation *R* such that the `l` aligns with the -Z axis, `r` with X axis, and `u` with Y axis. Remember that, the rows of the rotation matrix contain the unit vectors that align with the unit vectors along the principal axes after transformation.

$$\begin{bmatrix} r.x & r.y & r.z & 0 \\ u.x & u.y & u.z & 0 \\ -l.x & -l.y & -l.z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Thus the view transformation matrix *V*=*RT*.

Stage 3: Clipping and Projection Transformation

Projection Transformation:

In the projection transformation phase, the `gluPerspective` parameters in scene.txt are used to generate the projection transformation matrix *P*, and the points in `stage2.txt` are transformed by *P* and written in `stage3.txt`. The process of generating *P* is as follows:

First compute the field of view along X axis (`fovX`) and determine `r` and `t`.

```
fovX = fovY * aspectRatio
t = near * tan(fovY/2)
r = near * tan(fovX/2)
```

The projection matrix *P* is given below:

$$\begin{bmatrix} near/r & 0 & 0 & 0 \\ 0 & near/t & 0 & 0 \\ 0 & 0 & -(far+near)/(far-near) & -(2*far*near)/(far-near) \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

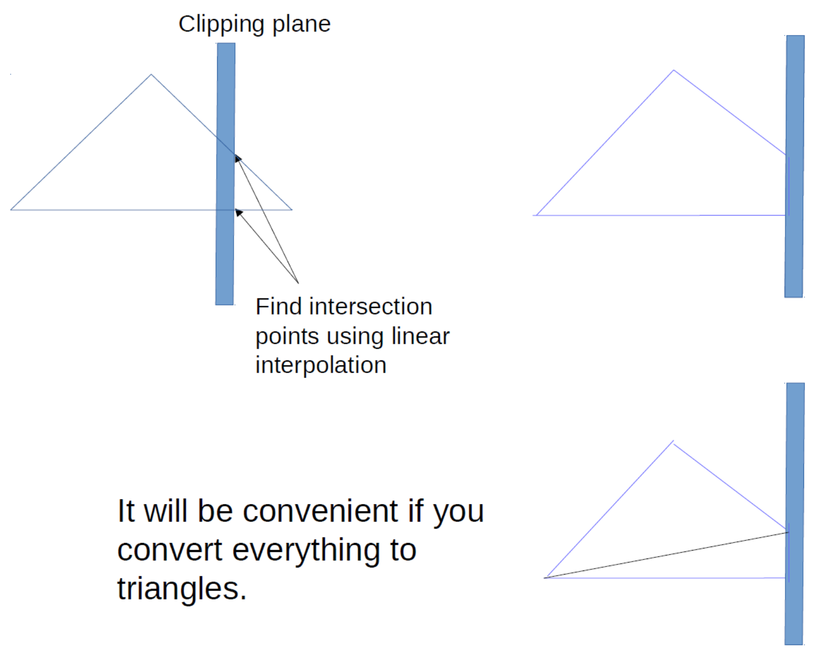
Clipping:

Having introduced the projection transformation, we can now introduce the fact that no point on xy plane can exist when we are trying to perform projection. (Refer to theory class for details). Therefore, ***before*** projection, you must clip everything with respect to near and far planes accordingly.

In this assignment, after view transformation, the camera is looking at –Z axis. Therefore, clipping with respect to `z=-near` is crucial before projection. However, we will perform clipping with respect to both `z=-near` and `z=-far` before projection. After clipping, the projected points will be outputted to `stage3.txt`.

PS: We also need to clip everything with respect to x and y axes as well (outside the field of view). These will be taken care of in the later stage.

Now, anything having z-value greater than (near) or less than (far) need to be clipped.



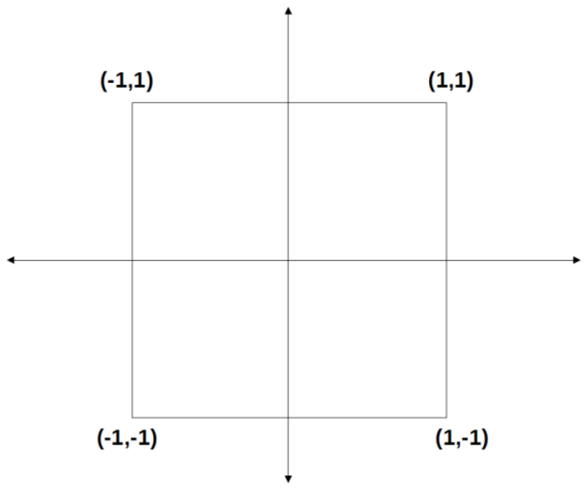
It will be convenient if you convert everything to triangles.

You may need to perform the clipping more than once on the same set of triangles, depending on how many points lie outside the clip region. Refer to class lecture for details.

Stage 4: Scan Conversion

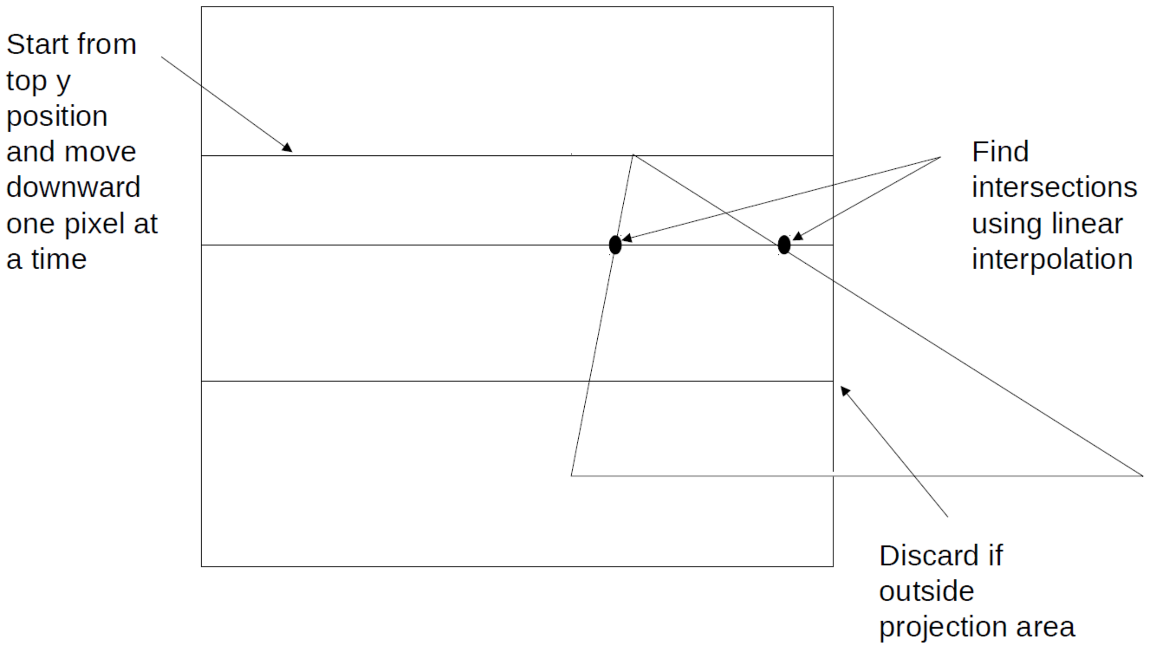
If you are done with step 1, your next step will start from the output of stage 3. After projection transformation, everything that will be visible will have all of x,y and z coordinates in [-1,1]. Since you have already performed clipping with respect to z-axis, z co-ordinates will be within this range by now.

Now, let us consider the plane as a square of length 2 on xy plane:



Now, divide the projection plane into a two dimensional grid of screen_width x screen_height. You have to calculate the center point of each small square to find out the color in the corresponding pixel. Construct a pixel buffer and a z-buffer of the same dimension and initialize them with appropriate values.

Now, you are ready for applying the z-buffer algorithm. Using the z-buffer algorithm, you will scan convert each triangle to fill up the pixel buffer.


Start from top y position and move downward one pixel at a time
Find intersections using linear interpolation
Discard if outside projection area

The steps are as follow:

- Each scan-line will intersect a triangle in at most two points. Find the co-ordinates from the coordinates of the vertices using linear interpolation.
- Find out the pixel position of the left intersection point and compare the z-value from the Z-buffer. If required conditions are satisfied, put the color of the triangle in the corresponding position of the pixel buffer and update the Z-buffer.
- Move one pixel right and calculate the z-value at that position from linear interpolation of the two intersection points.
- Repeat this step till you get to the right intersection point.
- At any point, if x or y value is outside [-1,1], discard it. This automatically takes care of the clipping.
- Handle special cases of a single intersection point and intersecting a horizontal line.
- Take some precautions to avoid precision related issues. Consider a small margin for floating point comparisons.
- Finally save the pixel buffer as an image. The code for saving image is already in the given template.

Do’s and Dont’s

1. Use homogeneous coordinates. The points should be represented by 4*1 matrices and transformations by 4*4 matrices.
2. While transforming a homogeneous point by multiplying it with a transformation matrix, don’t forget to scale the resultant point such that the w coordinate of the point becomes 1.
3. Use **the vector form** of Rodrigues formula to generate the rotation matrix in phase 1. Remember that, the columns of the rotation matrix indicate where the unit vectors along the principal axes are transformed. Use the vector form of Rodrigues formula to determine where `i`, `j`, and `k` are transformed and use them to generate the rotation matrix.
4. Notice that the rotation axis in the `rotate` command might not be normalized. Normalize it before using.
5. Do not specify `gluLookAt` parameters in `scene.txt` such that the looking direction, i.e., `look-eye`, becomes parallel to the up direction.
6. Make sure that the model is situated entirely in front of the near plane.