CSE 410: Assignment 2 (Jan 2023) 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 six stages. You will implement roughly 4 stages of the pipeline.

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

Your program will output five files: `stage1.txt`, `stage2.txt`, `stage3.txt`, `z-buffer.txt`, and `out.bmp`. The first three files will contain the output of the first three stages, respectively. The fourth file will contain z-buffer values (only those which are less than the max value). And the fifth file will be a bmp image generated by the pipeline.

Three will be two input files: `scene.txt` and `config.txt`

The first file will contain the scene description and second file will contain the necessary information for Z-buffer algorithm.

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

Lines 1-3 of scene.txt state the parameters of the `gluLookAt` function and Line 4 provides the `gluPerspective` parameters.

The display code contains 7 commands as follows:

1. `triangle` command – this command is followed by three lines specifying 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();
```

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.

In this assignment, you will generate the output of the first three stages of the raster-based graphics pipeline according to the scene description provided in `scene.txt` file. The output of the stages should be put in `stage1.txt`, `stage2.txt`, and `stage3.txt` file.

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. We also have to maintain a matrix *M* which stores the current state of transformation. The transform closest to the object gets multiplied first. The pseudo-code for the modeling transformation phase is as follows:

```
initialize empty stack S
initialize M = Identity matrix
while true
        read command
        if command = “triangle”
                read three points
                for each three-point P
                        P’ <- transformPoint(M, P)
                        output P’
        else if command = “translate”
                read translation amounts
                generate the corresponding translation matrix T
                M = product(M, T)
        else if command = “scale”
                read scaling factors
                generate the corresponding scaling matrix T
                M = product(M, T)
        else if command = “rotate”
                read rotation angle and axis
                generate the corresponding rotation matrix T
                M = product(M, T)
        else if command = “push”
                S.push(M)
        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:

```
1   0   0   tx
0   1   0   ty
0   0   1   tz
0   0   0   1
```

Transformation matrix for Scaling

```
scale
sx sy sz
```

The transformation matrix for the above scaling is as follows:

```
sx   0   0   0
 0  sy   0   0
 0   0  sz   0
 0   0   0   1
```

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:

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

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 (S) State after Lines											
Code	0	1	2	3	4	5	6	7	8	9	10	11
1.Push 2.Translate1 3.Push 4.Rotate1 5.Pop 6.Scale1												
7.Push												
8.Rotate2												
9.Pop												
10.Scale2				T1	T1			T_1S_1	T_1S_1			
11.Pop		I	I	I	I	I	I	I	I	I	I	

	Transformation Matrix (M) State after Lines											
	I	I	T1	T1	T_1R_1	T1	T_1S_1	T_1S_1	$T_1S_1R_2$	T_1S_1	$T_1S_1S_2$	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.

```
1    0    0    -eyeX
0    1    0    -eyeY
0    0    1    -eyeZ
0    0    0      1
```

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.

```
 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
```

Thus the view transformation matrix *V=RT*.

Stage 3: 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 (`fovX`) axis 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:

```
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
```

Refer to [1] for understanding the 3rd row of the above projection matrix. Image source: [2].

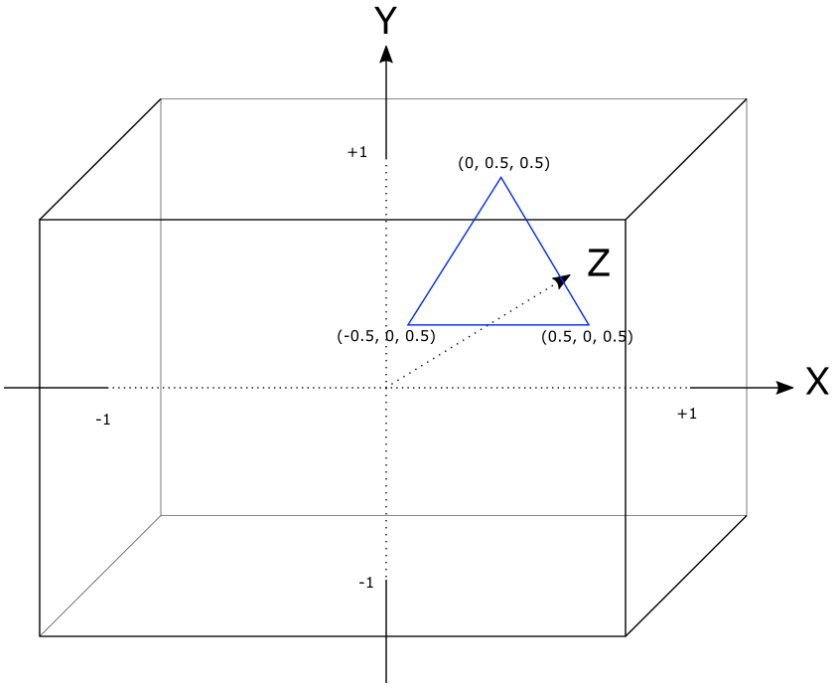
Stage 4: Clipping & scan conversion using Z-buffer algorithm

In this stage, you have to implement a very simple Hidden Surface Removal algorithm for the objects within a bounding box. You have to work with `triangles` only, as specified in `scene.txt`. The output generated by your program after the third stage, `stage3.txt` and the given input file `config.txt` will work as the input for this stage. However, during implementation it is recommended to test with smaller cases so that you can debug easily.

After the third stage, the viewing frustrum turns into a viewing volume, which is a 2x2x2 cube. X, Y and Z coordinates of every point inside the viewing frustrum get normalized within $[-1,1]$. The `stage3.txt` file generated by your program will contain each triangle information as three lines specifying the coordinates of the three points of the triangle. Suppose, the there is only one triangle and the file contains the following:

```
0.50 0.00 0.50
-0.50 0.00 0.50
0.00 0.50 0.50
```

The triangle position can be shown by the following figure within the viewing volume.



***config.txt* description:** The first Line of file contains two integers, representing Screen_Width and Screen_Height respectively. The final rendered image will have the width and height equal to these values respectively.

Example:

```
500 500
```

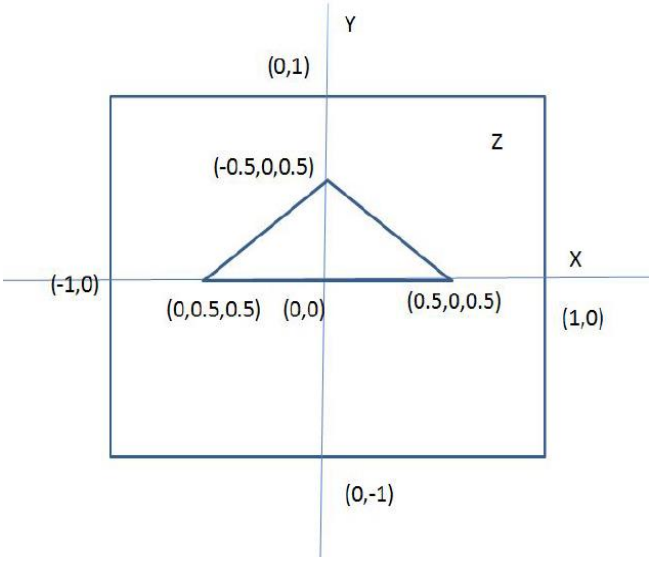
Tasks:

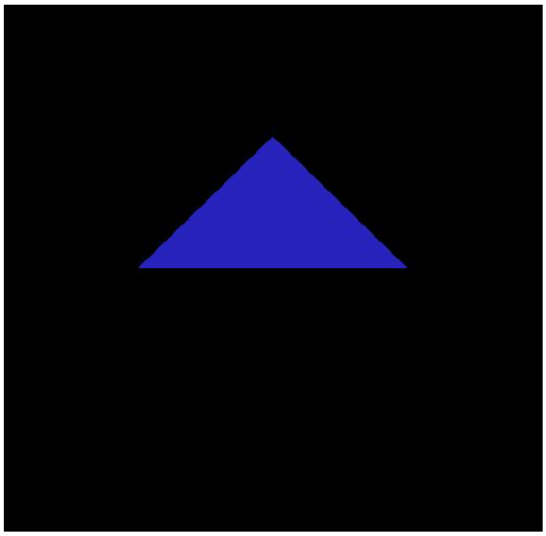
1. Clip everything outside the viewing volume.
2. Considering yourself as a parallel viewer from the XY-plane, generate the image (dimension: Screen_Width X Screen_Height) that can be seen on the XY plane after computing the necessary clipping and depth information of objects (triangles) within this viewing volume.
3. Print z-buffer values into a file named z-buffer.txt . (only those values where z-buffer[row][col] < z_max).

*Check “Procedures” for more details and further instructions.

Output:

The output from the viewing plane for the aforementioned configuration and triangle is shown by the following figures (the first one is for your understanding and the second one is the actual output).





You should save the output image in a file named “out.bmp” and the z-buffer values in a text file named “z-buffer.txt”.

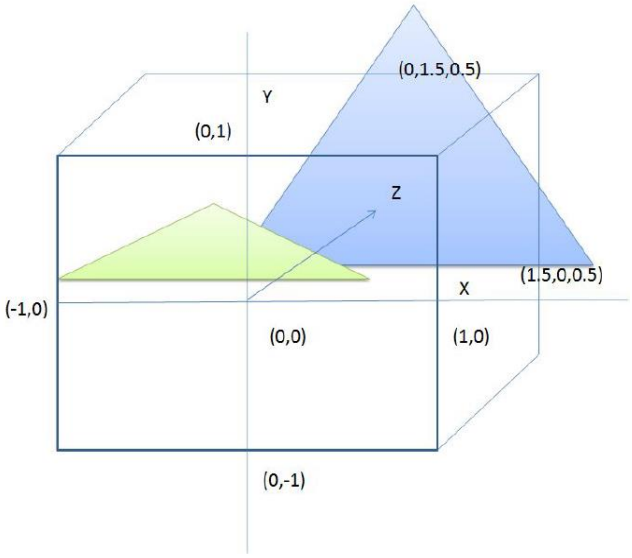
Another Sample:

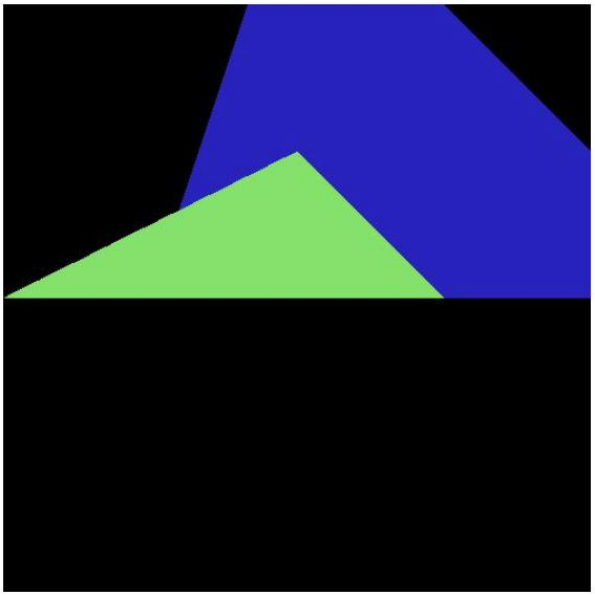
Suppose. `stage3.txt` contains the following while the `config.txt` is the same as before.

```
1.5 0 0.5
-0.5 0 0.5
0 1.5 0.5

0.5 0 0.25
-1.0 0 0.25
0 0.5 0.25
```

In this case, the output will be as follows (the first one is for your understanding and the second one is the actual output).





Color:

Assign the colors (RGB values) of a triangle using this random function.

```
static unsigned long int g_seed = 1;
inline int random()
{
    g_seed = (214013 * g_seed + 2531011);
    return (g_seed >> 16) & 0x7FFF;
}
```

For example, for the 1st triangle, first draw an integer from random() function. Assign this value as it's RED component. Similarly, draw two more random integer. Assign them as GREEN and BLUE component respectively. Do the same for the rest of the triangles.

Save Image as a .bmp File:

To render an bmp image file, use the `bitmap_image.hpp` from the following Github repository: https://github.com/ArashPartow/bitmap
You will need the following functions:

- Constructor: bitmap_image()
- Set color of a pixel: set_pixel()
- Save the image as a file: save_image()

Procedure:

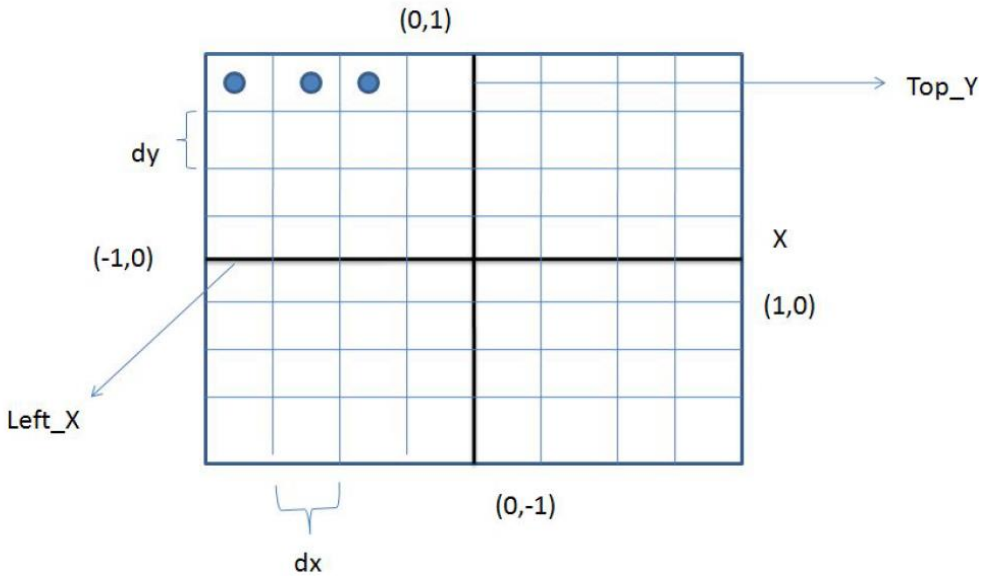
The aforementioned tasks can be divided into the following sub-tasks inside your main function.

1. Read data
2. Initialize z-buffer and frame buffer
3. Apply procedure
4. Save
5. Free memory

The description of each of these sub-tasks is as follows.

1. Read data
    a. Read the `config.txt` file and store the values as Screen_Width, Screen_Height
    b. Work on the output of stage 3. You can either (i) use the variable that stored the coordinates of triangles after stage 3, or (ii) read the `stage3.txt` file.
    c. Use a suitable data structure to hold this information. Also associate a random color value(R, G, B) with each object as mentioned before. RGB values are bounded by 0-255.
    d. Print and check whether you have correctly read the information from the files (for debugging purposes).

2. Initialize z-buffer and frame buffer



    a. Create a pixel mapping between the x-y range values and the Screen_Width X Screen_height range.

$$dx = (\text{right limit} - \text{left limit along X-axis}) / \text{Screen\_Width}$$

$$dy = (\text{top limit} - \text{bottom limit along Y-axis}) / \text{Screen\_Height}$$

    Besides, specify Top_Y and Left_X values.

$$\text{Top\_Y} = \text{top limit along Y-axis} - dy/2$$

$$\text{Left\_X} = \text{left limit along X-axis} + dx/2$$

    b. During scanning from top to bottom and left to right, check for the middle values of each cell. e.g. Top_Y- row_no*dy, Left_X+col_no*dx

c. Create a z-buffer, a two dimensional array of Screen_Width X Screen_Height dimension. Initialize all the values in z-buffer with z_max. In the aforementioned examples, z_max = 2.0. The memory for z-buffer should be dynamically allocated (using STL is allowed).
d. Create a bitmap_image object with Screen_Width X Screen_Height resolution and initialize its background color with black.

3. Apply procedure
    a. Pseudocode:

```
for each object : Triangles
        Find top_scanline and bottom_scanline after necessary clipping

        for row_no from top_scanline to bottom_scanline
                Find left_intersecting_column and right_intersecting_column
                after necessary clipping

                for col_no from left_intersecting_column to right_intersecting_column
                        Calculate z values
                        Compare with z-buffer and z_front_limit and update if required
                        Update pixel information if required
                end
        end
end
```

    * Note that you should not update a z-buffer value if the point's z-coordinate < z_front_limit (invisible to the observer, but ignore its occlusion impact).

    b. Clipping:
        i. Compute the max_y for each triangle. If max_y > Top_Y, clip (i.e. ignore) the portion above Top_Y and start scanning from Top_Y. Otherwise find the suitable mapping of max_y to a top_scanline below Top_Y. Do a similar checking for min_y and Bottom_Y.
        ii. Compute min_x for each scan line for each triangle. If min_x < Left_X, clip (i.e. ignore) the portion to the left of Left_X. Otherwise find the suitable mapping of min_x to a column to the right of Left_X (left_intersecting_column). Do a similar checking for max_x and Right_X

4. Save
    a. Save the image as **"out.bmp"**
    b. Save the z_buffer values in **"z_buffer.txt".** Save only those values which are less than z_max i.e. for each row and col, z-buffer[row][col] < z_max.

    Check the sample output files for a better understanding.

5. Free memory
    a. Free image memory
    b. Free z-buffer memory

Do's and Don'ts

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. Do not use the matrix form of Rodrigues formula directly to generate the rotation matrix. Use the procedure shown above that uses the vector form of Rodrigues formula.
4. Do not specify `gluLookAt` parameters in `scene.txt` such that the looking direction, i.e., `look-eye`, becomes parallel to the up direction.
5. Make sure that the model is situated entirely in front of the near plane.

Reference

[1] http://www.songho.ca/opengl/gl_projectionmatrix.html
[2] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/