Assignment 2: Rasterization

Mai Trinh
COSC 6372 Computer Graphics*, University of Houston

March 10, 2021

1 Problem

The goal of this assignment is to implement rasterization process using the Scan Line Algorithm for 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. Several classes, functions, and data types are already given in the *Gz* library. We were to complete the implementation of the *GzFrameBuffer* class to render the output image.

2 Methods

The main challenge of this assignment is the triangle rasterization. The general idea of a scanline rasterization approach is to scan the triangle from top to bottom, and determine, for each scanline, where the triangle starts and ends along the x-axis. Then we can fill in all the pixels at the scanline (all the pixels in between the endpoints). The scanline algorithm works on row-by-row basis, considering all rows that can possibly overlap with triangle. All of the polygons to be rendered are first sorted using y-coordinate in increasing order. The general algorithm for triangle rasterization using Scan Line algorithm method is given below:

```
Algorithm:
    1. Sort vertices based on y-coordinate in ascending order
    2. For each row until the last y-coor of triangle, do the following:
        a. Scanline algorithm starting at the top of the triangle.
        b. Determine the intersections at scanline:
            Sort intersection by x-coord in increasing order /*left to right*/
            left point = point with smaller x
            right point = point with larger x
            Perform linear interpolation  for endpoints /* z-value and color*/
            Fill scanline /*draw all pixels in between left point and right point*/
```

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, pass it to the frame buffer in order to render the output image.

To implement the frame buffer for our graphic library, the *GzFrameBuffer* class has the following structures:

```
class GzFrameBuffer {
public:
        /* Assuming that methods & variables from Assignment 1 are included.
        * Here, we only consider additional implementation needed for Assignment 2.*/
        void drawTriangle(GzVertex* vqueue, GzColor* cqueue, GzFunctional status);
private:
        void sortY(GzVertex* vqueue, GzColor* cqueue);
        GzColor colorInterpolate(double startX, double endX,
                                 GzColor start, GzColor end, double x);
        double Interpolate(double x0, double x1, double y0, double y1, double x);
        void fillScanLine(GzVertex& left, GzVertex& right,
                          GzColor& leftColor, const GzColor& rightColor,
                          double scanline, GzFunctional status);};

/* define the structure of new data type Edge */
typedef struct Edge {
        GzVertex start, end;
        GzColor cstart, cend;
        Edge(const GzVertex& v1, const GzVertex& v2, const GzColor& c1, const GzColor& c2) {
                start = v1; end = v2;
                cstart = c1; cend = c2;}
```

*Course assignments

3.1 Rasterization

We will be using scanline algorithm to render triangle. First, we will sort the vertex list by its y-coordinate and form 3 edges of the triangle.

3.1.1 Sorting vertex based on y-coordinate

To perform the sorting, we used the Insertion sort algorithm. Iterate over the entire vertex list, and color list. Compare the y-value of the current element to the y-value of the previous element. If y-value is smaller than that of the previous element, compare to the element before. Move the vertex one position up to make space for swapping. The implementation of sort function is shown below:

```
for (int i = 1; i < 3; i++) {
        vertexTemp = vTriangle[i];
        colorTemp = cTriangle[i];
        int j = i - 1;
        while (j >= 0 && vTriangle[j].at(Y) > vertexTemp.at(Y)) {
                vTriangle[j + 1] = vTriangle[j];
                cTriangle[j + 1] = cTriangle[j];
                j = j - 1;}
        vTriangle[j + 1] = vertexTemp;
        cTriangle[j + 1] = colorTemp;}
```

Once we have the ordered list of vertices. Then, we will form 3 edges of triangle based on the vertex coordinate.

```
Edge edge01(vTriangle[0], vTriangle[1], cTriangle[0], cTriangle[1]);
Edge edge12(vTriangle[1], vTriangle[2], cTriangle[1], cTriangle[2]);
Edge edge02(vTriangle[0], vTriangle[2], cTriangle[0], cTriangle[2]);
```

In the next step, we will perform the scanline check row-by-row.

3.1.2 Scan Line algorithm

The main idea of scanline algorithm is to fill up the polygons using horizontal lines. Starting from the top of the triangle, we traverse down to the bottom of the triangle (determine by its Y-coordinate). At each scanline, we determine the 2 endpoints where the scanline intersection with the triangle. Note that the 2 points are on the same line as the scanline (i.e. same y-coordinate). We then identify the left point and the right point based on its X-coordinate. Note that there are several cases we must consider:

- Case 1: First vertex is the only vertex on the top of the triangle
  - If second vertex and third vertex are on the same line, then we have a flat-bottom triangle
  - If second vertex and third vertex are not on the same line (i.e. they have different Y-coordinate), then the left side and right side will change once the scanline hits the second Y-coordinate because we now encounter a new edge.
- Case 2: First vertex and second vertex are both at the top of the triangle (i.e. they are on the same line), thus we have a flat-top triangle like an upside down triangle.

We must ensure that the left endpoint stays on the left and right endpoint stays on the right (i.e. left endpoint has smaller X-coordinate). Once we have the left and right endpoint, we fill the scanline by drawing all pixels in between using interpolation for Z-value and color value. The implementation of scanline algorithm is shown below:

```
for (double scanline = vTriangle[0].at(Y); scanline <= vTriangle[2].at(Y);scanline++){
    left[Y] = scanline;
    right[Y] = scanline;
    /*Determine the left endpoint and right endpoint that intersects the scanline*/
    if (scanline <= vTriangle[1][Y]){
        left[X] = Interpolate(edge01.start[Y], edge01.end[Y], edge01.start[X], edge01.end[X], scanline);
        left[Z] = Interpolate(edge01.start[Y], edge01.end[Y], edge01.start[Z], edge01.end[Z], scanline);
        lColor = colorInterpolate(edge01.start[Y], edge01.end[Y], edge01.cstart, edge01.cend, scanline);}
    else {
        left[X] = Interpolate(edge12.start[Y], edge12.end[Y], edge12.start[X], edge12.end[X], scanline);
        left[Z] = Interpolate(edge12.start[Y], edge12.end[Y], edge12.start[Z], edge12.end[Z], scanline);
        lColor = colorInterpolate(edge12.start[Y], edge12.end[Y], edge12.cstart, edge12.cend, scanline);}
    right[X] = Interpolate(edge02.start[Y], edge02.end[Y], edge02.start[X], edge02.end[X], scanline);
    right[Z] = Interpolate(edge02.start[Y], edge02.end[Y], edge02.start[Z], edge02.end[Z], scanline);
    rColor = colorInterpolate(edge02.start[Y], edge02.end[Y], edge02.cstart, edge02.cend, scanline);

    if (left[X] > right[X]){/* Swap left and right coordinate and color, if needed*/}

    /* Fill scanline (i.e.draw all pixels in between left point and right point) */
    fillScanLine(left, right, leftColor, rightColor, scanline, status);
}
```

3.2 Linear Interpolation

Linear interpolation is the process of constructing new data points within the range of a set of known data points. The linear interpolation between 2 known points is basically determined the points in between 2 points to form a straight line. Given 2 points with coordinates (x_0, x_1) and (y_0, y_1), and the known value at x, the formula to find y is:

$$y = y_0 + (x - x_0)\left(\frac{y_1 - y_0}{x_1 - x_0}\right) [1]$$

Using this equation, we can find the missing value of a vertex given its known value. For example, we have the Y-coordinates and Z-coordinates of 2 points (in this case, the 2 endpoints), and the y-coordinate of the target vertex, using the above formula, we can find the Z-value of the target point. We can also use this formula to find color value of a vertex. The implementation of linear interpolation for color vertex is shown below:

```
GzColor colorInterpolate(double x0, double x1, GzColor cstart, GzColor cend, double x){
    double dX = x1 - x0;
    GzColor color(0, 0, 0);
    for (int i = 0; i < 3; i++){ /*3 values for R,G,B*/
            if (dX != 0)
                    color[i] = cstart[i] + (x - x0) * ((cend[i] - cstart[i]) / dX);
            else
                    color[i] = cstart[i];
    }
    return color;
}
```

4 Results

The results of the implementation are shown below. While it renders a correct image, you can see that the image is not smooth. There are some crack in the teapot because the pixel at that point was not drawn. There are many reasons for such behavior such as error while interpolating color values, offset from edge to pixel center creating a gap. This indicates that scanline algorithm poses some limitations in the rasterizing process. Thus, there is a need for optimization method for smoothing the raster image.

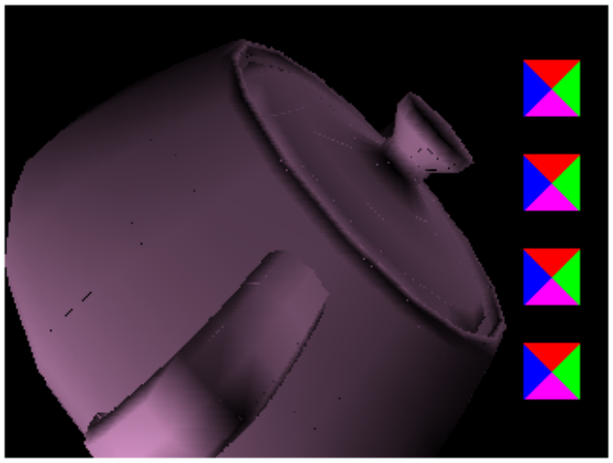

Figure 1: Rasterized image using scanline algorithm

.

References

[1] *Linear Interpolation*. URL: https://en.wikipedia.org/wiki/Linear_interpolation.