COSC6372 HW2 – Rasterization

Jiechang Guo UHID: 2084258

February 26, 2023

1 Problem

The problem for this assignment is to implement the Scan Line Algorithm for the Gz library. And then using the implemented rasterization function to render some triangles from the text file.

2 Method

In this assignment, two methods of rasterization were implemented. The first method is calculating a bounding box for a triangle, for each pixel testing if it is inside the triangle. The second method is the scan line algorithm. The first method is simpler but slower than the scan line method.

2.1 Barycentric Algorithm

Bounding Box Calculation

The first step in the barycentric algorithm is calculating a bounding box of the triangle to avoid checking every pixel on the screen. The example of a 2D bounding box of a triangle is shown in figure 1.

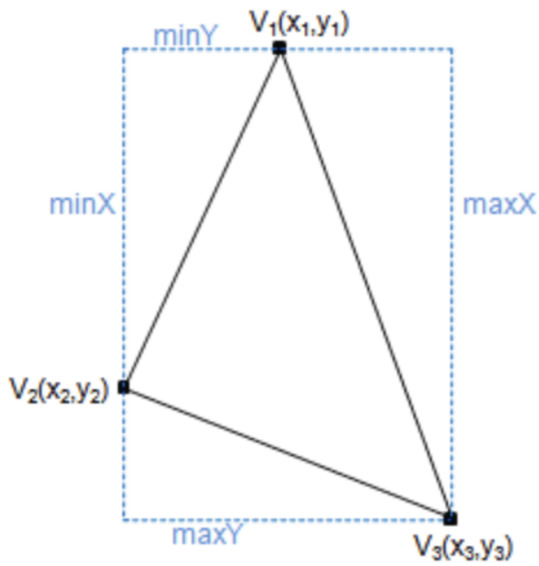

Figure 1: 2D bounding box of a triangle[2]

Pixel Inside Triangle

The second step is for every pixel inside the bounding box to check if it is inside the triangle. A simple way to do this is by calculating the dot product of the vector from the pixel point to each triangle vertex with the vector of each side. The sign of the dot product indicates the position of the pixel relative to the side of the triangle. Let the pixel point be $P = (P_x, P_y)$, we will calculate the dot product in a counter-clockwise fashion. If the signs of the dot product are the same, then the pixel is inside the triangle.

$$dot1 = dot(V1\vec{V}2, V\vec{1}P)$$

$$dot2 = dot(V2\vec{V}3, V\vec{2}P)$$

$$dot3 = dot(V3\vec{V}1, V\vec{3}P)$$

Barycentric Interpolation Last, except for the X,Y position on the screen, other rendering attributes like color and z-value in this assignment are needed to be determined. Here we use the barycentric coordinate to interpolate these attributes at the three vertices of the triangle[1]. The barycentric coordinates equation is as followed. The weights for $v1, v2, v3$ is the weights we use to interpolate color and z values.

$$P_x = W_{v1}X_{v1} + W_{v2}X_{v2} + W_{v3}X_{v3}$$

$$P_y = W_{v1}Y_{v1} + W_{v2}Y_{v2} + W_{v3}Y_{v3}$$

$$W_{v1} + W_{v2} + W_{v3} = 1$$

With given P_x, P_y, the equations for the weights are:

$$W_{v1} = \frac{(Y_{v2} - Y_{v3})(P_x - X_{v3}) + (X_{v3} - X_{v2})(P_y - Y_{v3})}{(Y_{v2} - Y_{v3})(X_{v1} - X_{v3}) + (X_{v3} - X_{v2})(Y_{v1} - Y_{v3})}$$

$$W_{v2} = \frac{(Y_{v3} - Y_{v1})(P_x - X_{v3}) + (X_{v1} - X_{v3})(P_y - Y_{v3})}{(Y_{v2} - Y_{v3})(X_{v1} - X_{v3}) + (X_{v3} - X_{v2})(Y_{v1} - Y_{v3})}$$

$$W_{v3} = 1 - W_{v1} - W_{v2}$$

By using the above equation to calculate $W1, W2, W3$, we can get smooth interpolation.

2.2 Scan Line Algorithm

The scan line algorithm used in this assignment is the standard algorithm mentioned in the post[2]. This algorithm takes the advantage of two basic triangles which are a flat bottom and a flat top. The idea of drawing these two basic triangles is to sample both legs of the triangle and draw a horizontal line between the sampled endpoints on the two legs. The figures 2 show how the scan line works in basic cases.

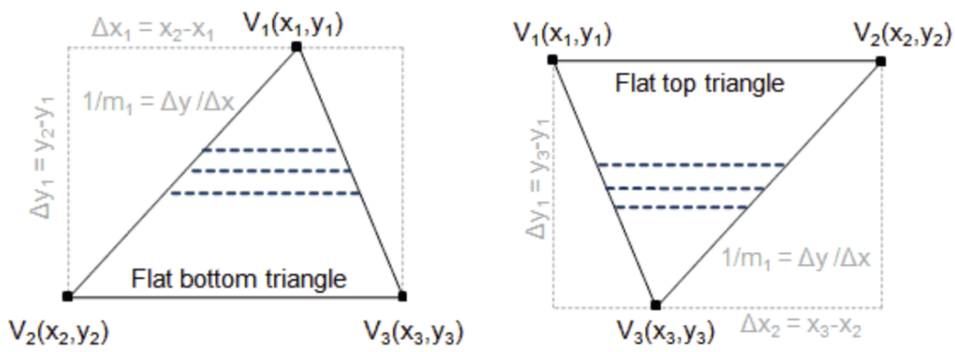

Figure 2: Basic Cases[2]

For the general case, we can simply split the triangle into a bottom flat triangle and a top flat triangle as shown in the figure 3, and then utilize the basic algorithm for the new triangles which are the basic cases.

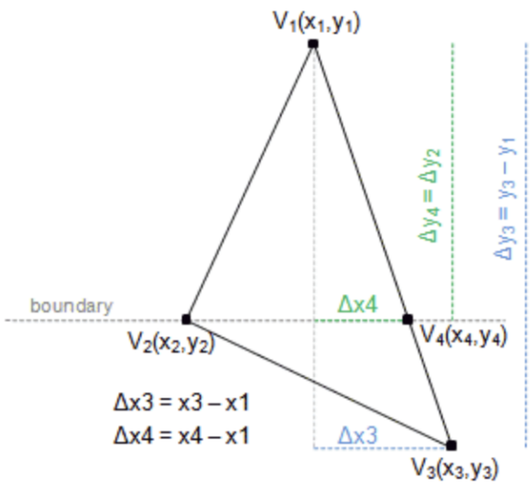

Figure 3: General Case[2]

However, this method may cause lots of numerical errors when the triangle is sharp and thin which in the end leads to the artifacts of the rendering.

3 Implement

The code is implemented using C++ on MacOS. The following section will go through the implementation in detail.

Updates in GzCommon.h

In order to process triangles conveniently, a struct for a triangle is defined as GzTriangle, a struct for a 2D bounding box is defined as Gz2DBoundingBox, and also the color attribute is added to the GzVertex struct.

```
struct GzVertex:public vector<GzReal> {
        GzVertex():vector<GzReal>(3, 0) {}
        GzVertex(GzReal x, GzReal y, GzReal z):vector<GzReal>(3, 0) {
                at(X)=x; at(Y)=y; at(Z)=z;
        }
        GzColor color; //color attribute of the vertex
};
//struct for triangle
struct GzTriangle{
        GzTriangle(){};
        GzTriangle(GzVertex a, GzVertex b, GzVertex c)
        {
                vertices.clear();
                vertices.push_back(a);
                vertices.push_back(b);
                vertices.push_back(c);
        }
        vector<GzVertex> vertices;
};

//struct for 2D bounding box
struct Gz2DBoundingBox{
        Gz2DBoundingBox(){}
        GzVertex min;
        GzVertex max;
};
```

3.1 Rendering triangles

The vertices of the triangles were loaded to the memory, and each time three vertices were popped from the vertex queue and rendered. The code in the "Gz.cpp" is as followed,

```
void Gz::end() {
        //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) ) {
                                GzVertex v=vertexQueue.front(); vertexQueue.pop();
                                GzColor c=colorQueue.front(); colorQueue.pop();
                                frameBuffer.drawPoint(v, c, status);
                        }
                } break;
                case GZ_TRIANGLES: {
                        //Put your triangle drawing implementation here:
                        //   - Pop 3 vertices in the vertexQueue
                        //   - Pop 3 colors in the colorQueue
                        //   - Call the draw triangle function
                        //     (you may put this function in GzFrameBuffer)
                        //int i = 0;
                        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();

                                GzColor c1=colorQueue.front(); colorQueue.pop();
                                GzColor c2=colorQueue.front(); colorQueue.pop();
                                GzColor c3=colorQueue.front(); colorQueue.pop();
                                v1.color = c1;
                                v2.color = c2;
```

```
                                    v3.color = c3;
                                    //cout << i << endl;
                                    GzTriangle tri(v1,v2,v3);
                                    frameBuffer.drawTriangle(tri,status);
                                    //i++;
                                }
                            }
                        }
}
```

3.2 Rasterization

The rasterization of the triangle was implemented in the GzFrameBuffer class. As mentioned in section 2, two methods were implemented and can be switched using the define command. The implementation of the two methods followed the methods mentioned above, please check the source code for the detailed implementation. Before rasterization, we need to transform the triangle from object coordinate to the screen coordinate.

```
void GzFrameBuffer::drawTriangle(const GzTriangle& tri, GzFunctional status)
{
    //transform from object coordinate to image coordinate
    GzTriangle tri_transed = tri;
    for(auto& vert : tri_transed.vertices)
    {
        vert.at(Y) = height_ - vert.at(Y) - 1;
    }
    status_ = status;
#ifdef BBOX
    drawTriangleUseBbox(tri_transed);
#else
    drawTriangleUseScanline(tri_transed);
#endif
}
```

4 Result

In this section, the final result of the rasterization of the triangles and the intermediate results of the experiment will be illustrated.

The results for the two methods are shown in figures 4 and 5. The result of the barycentric algorithm is very smooth. The result of the scan line algorithm, which is smooth in most of the area, however with some artifacts in the edge of the teapot. The reason for the artifacts is because of the numerical error caused by the splitting of the thin and sharp triangle. A possible solution is instead of splitting the triangle, we can directly sample the endpoints on the three sides of the triangle.

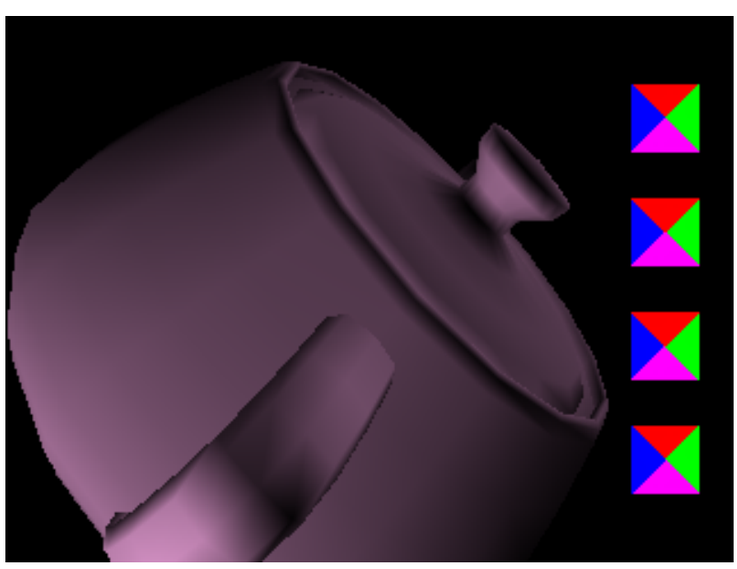

Figure 4: Final result of the Barycentric Algorithm

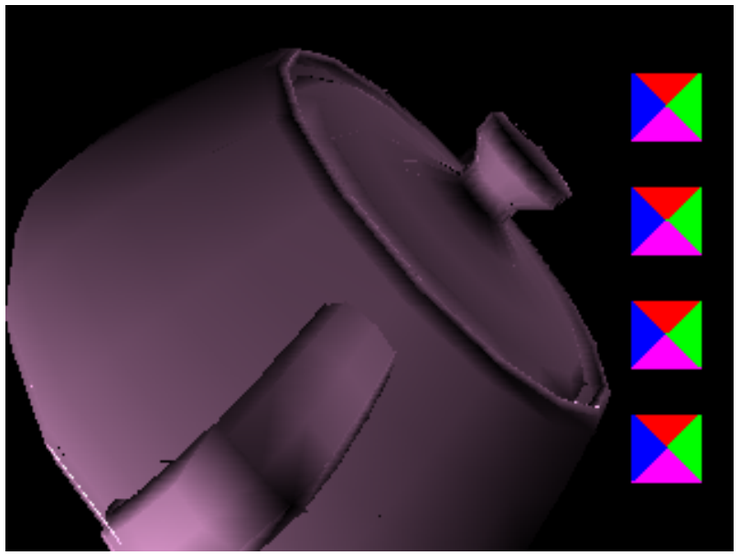

Figure 5: Final result of the Scan Line Algorithm.

The intermediate results are as followed. Note that the following result failed to transform the triangles from world space(with Y up)to screen space(with Y down).

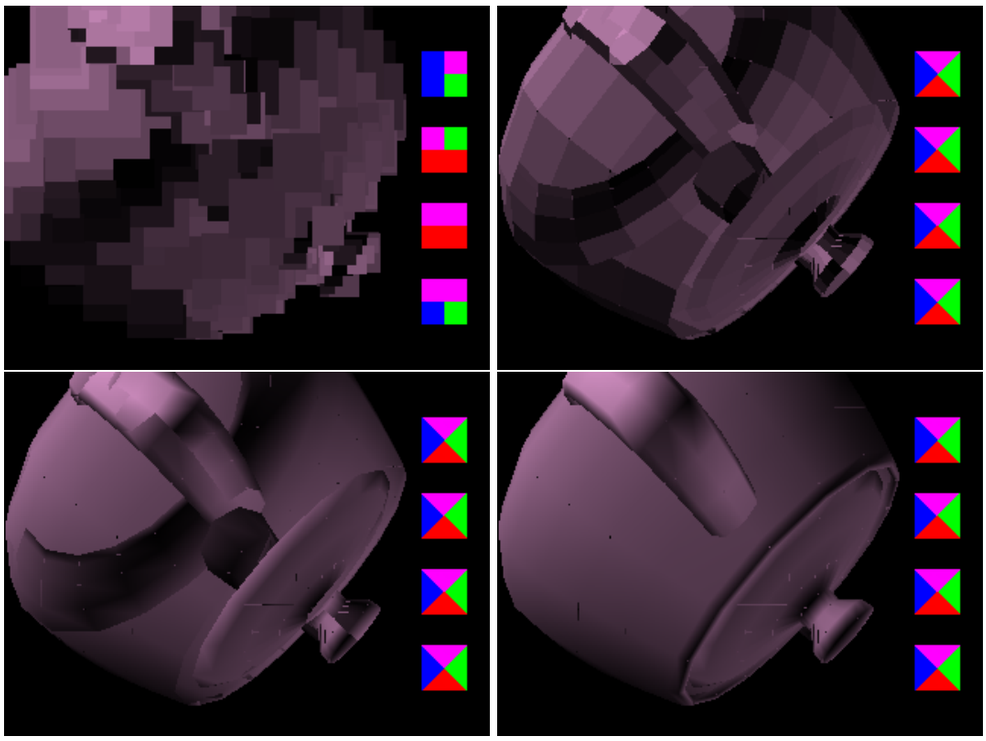

Figure 6: Intermediate results of the Barycentric algorithm. Left top: render calculated bounding box, Right top: coarse result without color interpolation, Left bottom: fine result with color interpolation, Right bottom: better result with depth test.

References

[1] CodePlea. *Interpolating in a Triangle*. `https://codeplea.com/triangular-interpolation`.

[2] Sunshine. *Software Rasterization Algorithms for Filling Triangles*. `http://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html`.