**Lab 3 Report**

*Name: Bingjing Xu Major:DMT Contact:* [*bingjing@kth.se*](mailto:bingjing@kth.se)

**1.0 Before we do the experiment…**

In this lab, we mainly focus on Rasterisation. In order to get the projection of 3D objects on 2D plane and move the origin of local coordinate system on the top left corner, we use following formulas:

**x = f\*X/Z+Screen\_width/2 (3)**

**y = f\*Y/Z+Screen\_height/2 (4)**

However, before we implement them into our lab, we should firstly transform points P from the world coordinate system to local coordinate system. To do this:

Let,

**P'(local coordinate system) = P(world coordinate system) – C(camera position)**

So, equation (3) and (4) will be changed as following:

**x = f\*(X-C.X)/(Z-C.Z)+Screen\_width/2 (3)**

**y = f\*(Y-C.Y)/(Z-C.Z)+Screen\_height/2 (4)**

* 1. **Draw Points**

If you follow the instructions below and write the VertexShader() function like this:

glm::ivec2 VertexShader(glm::vec3 vertice)

{

vec3 locVer=vertice-cameraPos;

ivec2 projPosition;

projPosition.x = focal\*locVer.x/locVer.z+SCREEN\_WIDTH/2;

projPosition.y = focal\*locVer.y/locVer.z+SCREEN\_HEIGHT/2;

return projPosition;

}

You will probably get the result like figure 1.

****

figure1 projected vertices

**1.2 Draw Edges**

Since we have already get the vertices of triangles on the screen, it would be easy for us to draw edges between vertices. The thought is that we do linear interpolation between vertices.

Therefore, we need a function Interpolation(ivec2 a, ivec2 b, vector<ivec2>& results) to handle drawing lines in two adjacent points.

Pay attention that we use pixels to describe the 2D points’ position in the canvas. So in (x, y), x and y should both be integers.



figure2 edges between vertices

**1.3 Filled Triangles**

Next, we need fill in triangles with pre-defined colors in TestModel.h. In this case, we should know the range of pixels for each triangle. So, we interpolate points along triangle’s edge and find the most left and right pixel for each row. Then we interpolate again along rows. After doing this, we store pixels within triangles.

The following is part of core code:

void CalculatingLeftRight(vector<PIXEL>& leftPixels, vector<PIXEL>& rightPixels, PIXEL a, PIXEL b, int j)

{

ivec2 delta = glm::abs( a.position2D - b.position2D );

int pixels = glm::max( delta.x, delta.y ) + 1;

vector<PIXEL> line (pixels);

Interpolate(a , b, line);

for (int i=0; i<line.size(); ++i)

{

if(line[i].position2D.y == leftPixels[j].position2D.y)

{

if(line[i].position2D.x<leftPixels[j].position2D.x)

{

leftPixels[j]=line[i];

}

if(line[i].position2D.x>rightPixels[j].position2D.x)

{

rightPixels[j]=line[i];

}

}

}

}

void ComputePolygonRows(vector<PIXEL>& leftPixels, vector<PIXEL>& rightPixels,vector<PIXEL>&projectedVertice, int ROWS)

{

int R;

for(R=0; R<ROWS; ++R)

{ for( int i=0; i<3; ++i )

{

int j = (i+1)%3; // The next vertex

CalculatingLeftRight(leftPixels, rightPixels,projectedVertice[i],projectedVertice[j],R);

}

}

}

****

figure3 filled triangles

**1.4 Depth Buffer**

Right now we have a rough sketch about the scene, consisted of different color block. One of main problems is that the blue cuboid should be behind the red cube from the perspective of camera position. However, the result we get actually is blue block covers part of red cube, which is not correspondent to the realistic point of view. Thus, we need depth buffer help us correct the result.

Principally, we compare z values of every 3D points in prospective projection. Bigger value in z means further distance to the camera position, which also means these points will be covered by points with smaller Z values if occlusion happens.



figure4 the effect with depth buffer algorithm

**1.5 Illumination**

Now, our task is how to make our scene more 3D. In order to realize it, we should implement certain illumination models to get light and shade relations among objects.

There are two methods for illumination in rasterization. The first one is Per Vertex Illumination and the other is Per Pixel Illumination.

According to the point light model, the following equation is used when we simulate illumination in our case.

**D=(P\*max(r̂ .n̂,0))/4πr2 (10)**

**R= ρ\*(D+N) (11)**

**1.5.1 Per Vertex Illumination**

We can always get some clues about how to illuminate our scene from the name. It indicates that the illumination will be calculated for each vertex. And then we use bio-linear interpolation to get approximate value of color for every pixel. The advantage of the method is it responds fast, but the illumination is not that precise.



figure 5 Per vertex illumination

**1.5.2 Per Pixel Illumination**

To get more accurate illumination, we should finally implement per pixel illumination method, which means longer responding time.

Firstly, we should interpolate 3D Points for every pixel, then we use these positions to get the pixels’ color.

In this process, we should be careful about the interpolation. Since these 3D points(Ps) are not linear in prospective projection, we have to interpolate Ps/z instead of Ps. After interpolation, we devide our result by the depth buffer 1/z, equivalent with multiplying with z, to get the 3D positions. When we did depth buffer, we have actually get proper depth buffer for every pixel. So what we should in this step is just to interpolate Ps/z and finally retrieve correct 3D positions.

****

figure6 Per Pixel Illumination