CS380 PA4 Report

20170690 최종윤

1. TriangulatePolygon

다각형의 꼭짓점이 n개라고 할 때 만들어지는 삼각형의 수는 n-2개이다. 따라서 삼각형의 세 꼭짓점을 *triangleVerts* 에 넣는 작업을 n-2번 해준다. 꼭짓점을 넣어줄 때 한 점은 고정하고 나머지 두 꼭짓점을 index를 늘려주면 모든 삼각형을 만들 수 있다.

```
if (polygonVerts.size() >= 3) {
    int triNum = polygonVerts.size() - 2;
    for (int i = 1; i < triNum + 1; i++) {
        triangleVerts.push_back(polygonVerts[0]);
        triangleVerts.push_back(polygonVerts[i]);
        triangleVerts.push_back(polygonVerts[i + 1]);
    }
    return true;
}
else {
    return true;
}
```

2.1. RasterizeTriangle

첫번째로 삼각형의 모든 edge equation이 필요하다. 모든 edge equation을 구해준 후에는 꼭짓점의 순서를 판단한다. CCW인 경우 $E(x,y) > 0$ 일 때 삼각형 내부이고, CW인 경우는 $E(x,y) < 0$ 일 때 삼각형 내부이다. 이는 삼각형의 중점으로 판단하고, 이를 edge 변수에 저장한다.

```
// get the edge equations
glm::vec3 e01 = glm::vec3(v0.position[1] - v1.position[1], v1.position[0] - v0.position[0],
                          v0.position[0] * v1.position[1] - v1.position[0] * v0.position[1]);
glm::vec3 e12 = glm::vec3(v1.position[1] - v2.position[1], v2.position[0] - v1.position[0],
                          v1.position[0] * v2.position[1] - v2.position[0] * v1.position[1]);
glm::vec3 e20 = glm::vec3(v2.position[1] - v0.position[1], v0.position[0] - v2.position[0],
                          v2.position[0] * v0.position[1] - v0.position[0] * v2.position[1]);

// check the result of edge equation of the center of triangle
glm::vec3 center = glm::vec3((v0.position[0] + v1.position[0] + v2.position[0]) / 3,
                             (v0.position[1] + v1.position[1] + v2.position[1]) / 3, 1);
int val0 = dot(e01, center);
int val1 = dot(e12, center);
int val2 = dot(e20, center);
bool edge = true;
if ((val0 < 0) && (val1 < 0) && (val2 < 0)) edge = false;
```

그 후 각 꼭짓점의 색을 *texture.GetPixel()* 을 통해서 받아온 후 각각 *color0, color1, color2* 에 저장한다. 다음으로는 linear interpolation을 하기 위해서 interpolate equation을 구한다. Color의 속성인 R, G, B, A, 그리고 z-buffering을 위한 Z에 대해서도 구해주었다. 식은 교재를 참고했다.

```
 // get interpolation equation
float A = (e01[2] + e12[2] + e20[2]);
glm::mat3 Es;
Es[0] = e12; Es[1] = e20; Es[2] = e01;
Es = transpose(Es);

glm::vec3 linInterpR = (1 / A) * glm::vec3(color0[0], color1[0], color2[0]) * Es;
glm::vec3 linInterpG = (1 / A) * glm::vec3(color0[1], color1[1], color2[1]) * Es;
glm::vec3 linInterpB = (1 / A) * glm::vec3(color0[2], color1[2], color2[2]) * Es;
glm::vec3 linInterpA = (1 / A) * glm::vec3(color0[3], color1[3], color2[3]) * Es;
glm::vec3 linInterpZ = (1 / A) * glm::vec3(v0.position[2], v1.position[2], v2.position[2]) * Es;
```

삼각형의 bounding box 안에 있는 각 픽셀을 traverse 하기 위해서 x, y의 max, min 값을 찾아주었다. 이때 최댓값의 index를 찾아주는 FindXMax(), FindYMax() 함수를 만들어서 사용했다. x와 y좌표는 position의 index 차이만 있으므로 x만 표시했다.

```
int MyGL::FindXMax(GLVertex verts[3]) {
    for (int i = 0; i < 3; i++) {
        if ((verts[i].position[0] >= verts[(i + 1) % 3].position[0]) &&
             (verts[i].position[0] >= verts[(i + 2) % 3].position[0]))
            return i;
    }
}

 // find xmax and xmin
 int xindex = FindXMax(verts);
 int xmax = verts[xindex].position[0];
 int xmin;
 if (verts[(xindex + 1) % 3].position[0] > verts[(xindex + 2) % 3].position[0])
    xmin = verts[(xindex + 2) % 3].position[0];
 else
    xmin = verts[(xindex + 1) % 3].position[0];
```

x, y의 man, min을 구한 후에는 bounding box의 픽셀들을 찍어야 한다. *frameBuffer.GetSize()* 로 w, h를 가져온다. 이때 픽셀의 좌표가 0보다 작거나 *framebuffer* 의 w, h보다 크면 그리지 않는다. 각 픽셀이 삼각형의 내부인지 아닌지를 *EdgeCheck()* 로 확인한다. 이때 *edge* 변수를 확인하여 꼭짓점의 ccw, cw를 확인하고 그에 따른 알맞은 답을 리턴한다. *EdgeCheck()* 결과 삼각형의 내부일 경우 색을 찍어줘야 한다. 이 작업은 *InterpolateColor()* 함수에 미리 구해놓은 Interpolation equation 값과 좌표를 넣고 linear interpolate 된 색을 구해서 *frameBuffer.SetPixel()* 함수를 이용해서 색을 찍는다.

```
bool MyGL::EdgeCheck(float x, float y, glm::vec3 e01, glm::vec3 e12, glm::vec3 e20,
                     bool edge) {
    glm::vec3 pos = glm::vec3(x, y, 1.0);
    int val0 = dot(e01, pos);
    int val1 = dot(e12, pos);
    int val2 = dot(e20, pos);

    if (edge) {
        if ((val0 >= 0) && (val1 >= 0) && (val2 >= 0)) return true;
        else return false;
    }
    else {
        if ((val0 < 0) && (val1 < 0) && (val2 < 0)) return true;
        else return false;
    }
}
```

```
void MyGL::InterpolateColor(int x, int y, glm::vec3 linInterpR, glm::vec3 linInterpG,
                                          glm::vec3 linInterpB, glm::vec3 linInterpA) {
   float rv = dot(linInterpR, glm::vec3(x, y, 1.0));
   float gv = dot(linInterpG, glm::vec3(x, y, 1.0));
   float bv = dot(linInterpB, glm::vec3(x, y, 1.0));
   float av = dot(linInterpA, glm::vec3(x, y, 1.0));
   frameBuffer.SetPixel(x, y, glm::vec4(rv, gv, bv, av));
}
```

```
 int w, h;
frameBuffer.GetSize(w, h);
for (int y = ymax; y >= ymin; y--) {
    for (int x = xmin; x <= xmax; x++) {
        if (EdgeCheck(x, y, e01, e12, e20, edge)) {
            if (y < 0 || x < 0 || y >= h || x >= w)
                continue;
            InterpolateColor(x, y, linInterpR, linInterpG, linInterpB, linInterpA);
            }
        }
    }
}
```

2.2. Z-Buffering

Z-Buffering은 depth가 0~1 사이인 물체들 중에서 depth가 작을수록 앞에 보이게 하는 기능을 구현한다. 일단 해당 픽셀의 depth를 구하기 위해서 linear interpolation을 이용해서 *depth* 를 구했다. 구현 중 문제점은 배경의 depth가 0이어서 모든 물체보다 앞에 보인다는 점이었다. 따라서 배경을 구분하기 위해서 배경의 color 속성을 이용해서 배경을 판단하고, depth를 0.99로 설정하여 맨 뒤에 그려지도록 했다. 따라서 depth가 1 이상인 경우에는 배경 뒤에 가려져서 보이지 않는다. 또한 depth가 0 이하인 경우도 너무 가까워서 보이지 않는 경우이므로 색을 칠하지 않았다. 현재 픽셀에 저장되어있는 depth 값을 *frameBuffer.GetDepth()* 를 통해서 받아오고, 해당 픽셀의 *depth* 값과 비교한다. 해당 픽셀의 *depth* 가 현재 depth 값보다 작을 경우 해당 픽셀의 *depth* 값으로 *framebuffer* 의 depth 값을 바꿔준다. 그리고 그 후에 *InterpolateColor()* 함수를 이용해서 색을 칠해준다. 이렇게 색을 칠하면 depth 값이 가장 작은 경우에만 색을 찍으므로, depth가 작은 object가 앞에 오게 된다.

```
if (EdgeCheck(x, y, e01, e12, e20, edge)) {
    if (y < 0 || x < 0 || y >= h || x >= w)
        continue;
    float depth = dot(linInterpZ, glm::vec3(x, y, 1.0));
    float rv = dot(linInterpR, glm::vec3(x, y, 1.0));
    float gv = dot(linInterpG, glm::vec3(x, y, 1.0));
    float bv = dot(linInterpB, glm::vec3(x, y, 1.0));
    if ((rv == gv) && (bv == 2*rv)) depth = 0.99;           // it is background
    if (depth < 0) continue;
    if (depth < frameBuffer.GetDepth(x, y)) {
        frameBuffer.SetDepth(x, y, depth);
        InterpolateColor(x, y, linInterpR, linInterpG, linInterpB, linInterpA);
    }
}
```

2.3. Back-face Culling

Back-face culling이 되는 경우를 살펴보자. 해당 삼각형의 면적이 0일 경우에는 보이지 않는 삼각형이고, 음수일 경우에는 뒤를 향해있는 삼각형이다. 따라서 삼각형의 면적을 구하고 이 면적이 0 이하일 때 *return false* 하고, 아래에서 삼각형의 픽셀을 그리는 과정을 스킵하여 back-face culling을 진행한다.

```
float A = (e01[2] + e12[2] + e20[2]);

// cull if triangle is back-face
if (cullFaceEnabled) {
    if (A <= 0) return false;
}
```

Back-face culling을 확인하기 위해 실제 rasterize 하는 삼각형의 개수를 count해야 한다. GLRenderer.cpp 파일에 삼각형을 세는 *cnt* 변수를 global로 선언한다. 그리고 *processPolygon()* 함수에서 raterization 하는 부분에 *RasterizeTriangle()* 함수가 제대로 실행되었을 경우 1씩 더한다. 그리고 이 *cnt* 변수는 스크린이 clear 되었을 때 초기화 해주어야 한다. 따라서 Clear() 함수에서 0으로 초기화해준다.

```
int cnt = 0;

void GLRenderer::processPolygon( vector<GLVertex> &verts ) {
    // rasterization
    for( i = 0; i < ( int )triVerts.size(); i += 3 ) {
        if( !RasterizeTriangle( &triVerts[i] ) ) {
            passVerticesToGL( triVerts, coords, true );
            return;
        }
        cnt += 1;
    }
}

void GLRenderer::Clear( GLbitfield mask ) {
    cnt = 0;
}
```

이 *cnt* 변수를 출력하기 위해서는 화면이 나타날 때 한번 출력하면 된다. 따라서 *Display()* 함수에 출력하는 부분을 추가한다. 이때 'B' 키로 인해서 culling 상태인지를 *cullFaceEnabled* 변수를 통해 확인하고 상태를 함께 출력한다.

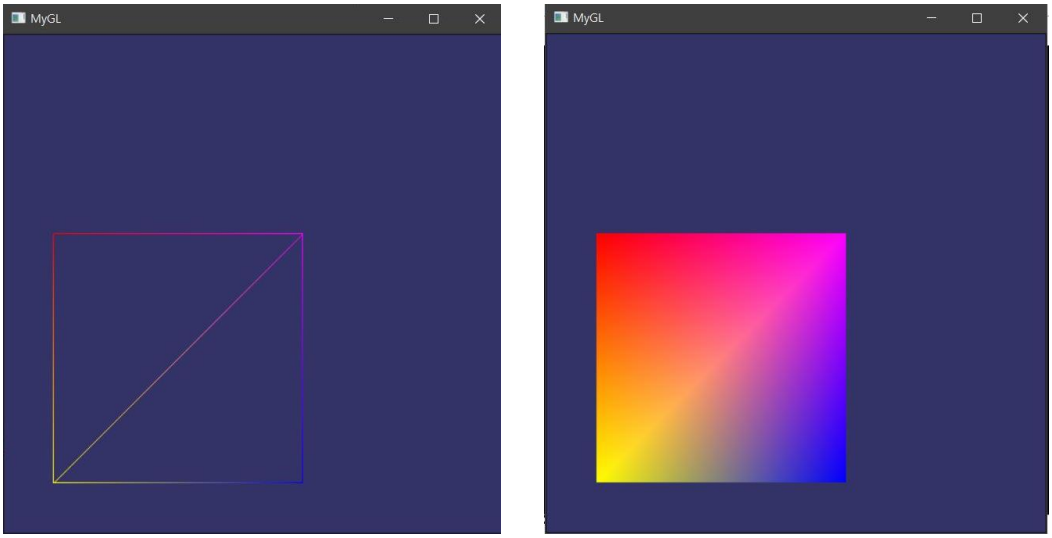
```
void GLRenderer::Display( bool setDepth ) {
    if (cullFaceEnabled) {
        printf("Culling... -> ");
    }
    else {
        printf("No cull... -> ");

    }
    printf("triangle: %d\n", cnt);
}
```

* Table of the number of rendered triangles

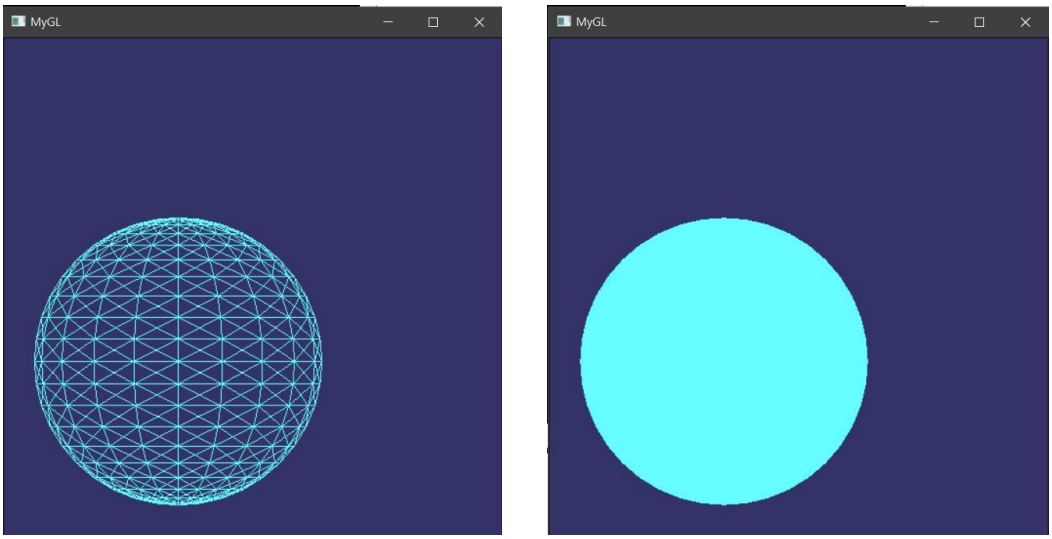
```
    Software rendering:on  - T(1):on  R(2):on
F1  No cull... -> triangle: 4     Culling... -> triangle: 4
F2  No cull... -> triangle: 7     Culling... -> triangle: 7
F3  No cull... -> triangle: 5     Culling... -> triangle: 5
F4  No cull... -> triangle: 762   Culling... -> triangle: 382
```

* Executed Images

a. Scene F1 and F4, with and without wireframe mode, `Software rendering:on - T(1):on R(2):off`

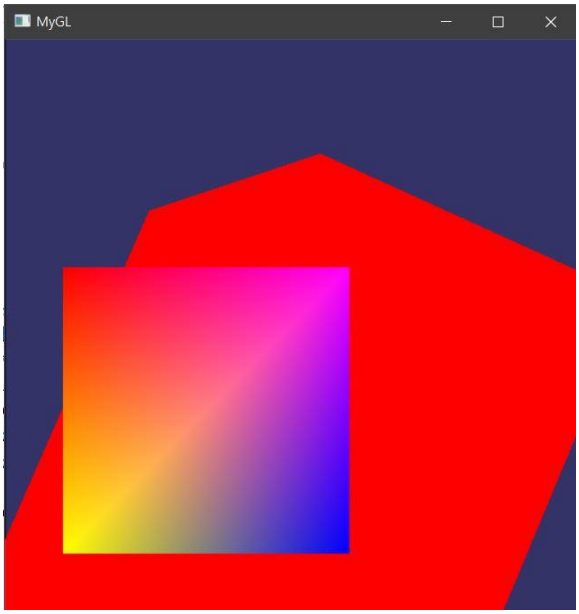


Scene F1, with / without wireframe mode



Scene F4, with / without wireframe mode

b. Scene F2, without wireframe mode, `Software rendering:on - T(1):on R(2):on`



c. Scene F2, without wireframe mode, the rectangle is occluded by the red polygon, `Software rendering:on - T(1):on R(2):on`