

**Brock Barlow**

**ID #1113**

**Assessment ADGP 201 - Graphics**

### Graphics Assessment Documentation for “Rendering Geometry”

#### Purpose:

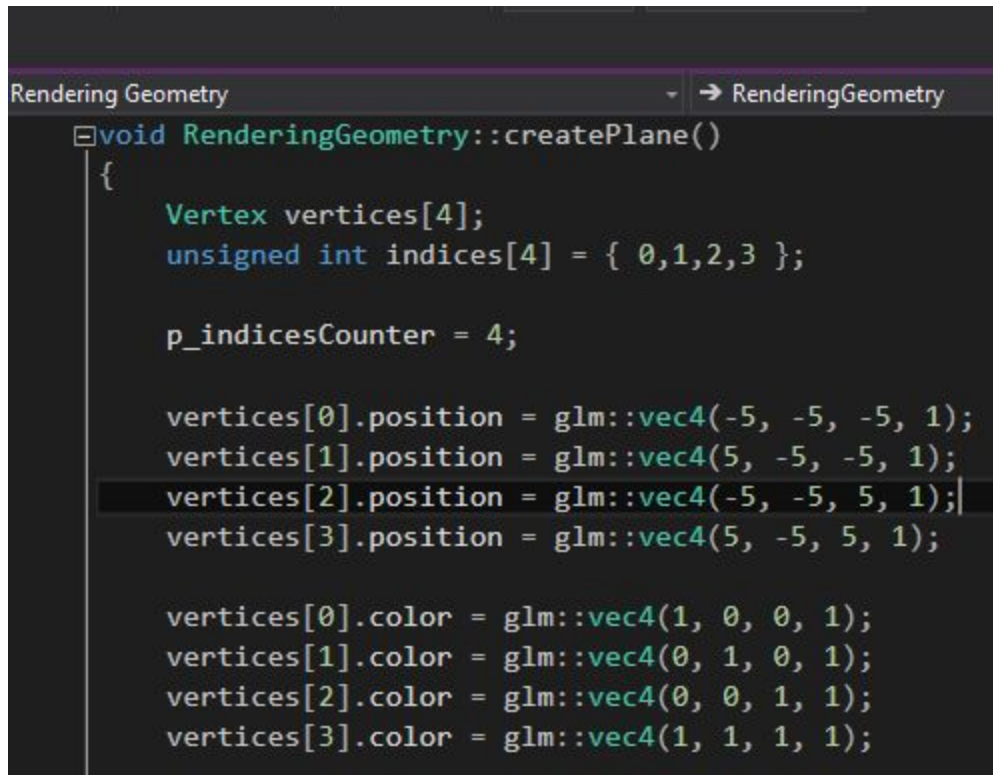
Introduce the steps needed to render a static plane, a static cube, a procedural sphere and take in separate shader files.

#### Learning Outcomes:

- 1) Render Plane (Static).
- 2) Render Cube (Static).
- 3) Render Sphere (Procedural) && (Method described in class).
- 4) Shaders separate file.

#### Evidence:

This project has the ability to render a static plane using the createPlane function. The plane has a vertices and indices array of four where the values have been set manually.



```
Rendering Geometry -> RenderingGeometry
void RenderingGeometry::createPlane()
{
    Vertex vertices[4];
    unsigned int indices[4] = { 0,1,2,3 };

    p_indicesCounter = 4;

    vertices[0].position = glm::vec4(-5, -5, -5, 1);
    vertices[1].position = glm::vec4(5, -5, -5, 1);
    vertices[2].position = glm::vec4(-5, -5, 5, 1);
    vertices[3].position = glm::vec4(5, -5, 5, 1);

    vertices[0].color = glm::vec4(1, 0, 0, 1);
    vertices[1].color = glm::vec4(0, 1, 0, 1);
    vertices[2].color = glm::vec4(0, 0, 1, 1);
    vertices[3].color = glm::vec4(1, 1, 1, 1);
}
```

The function then generates buffers and vertex arrays. It then binds the buffer and sets the buffer data. It then enables the vertex attrib array and sets the vertex attrib pointers.

```

Rendering Geometry
RenderingGeometry
createPlane()

vertices[3].color = glm::vec4(1, 1, 1, 1);

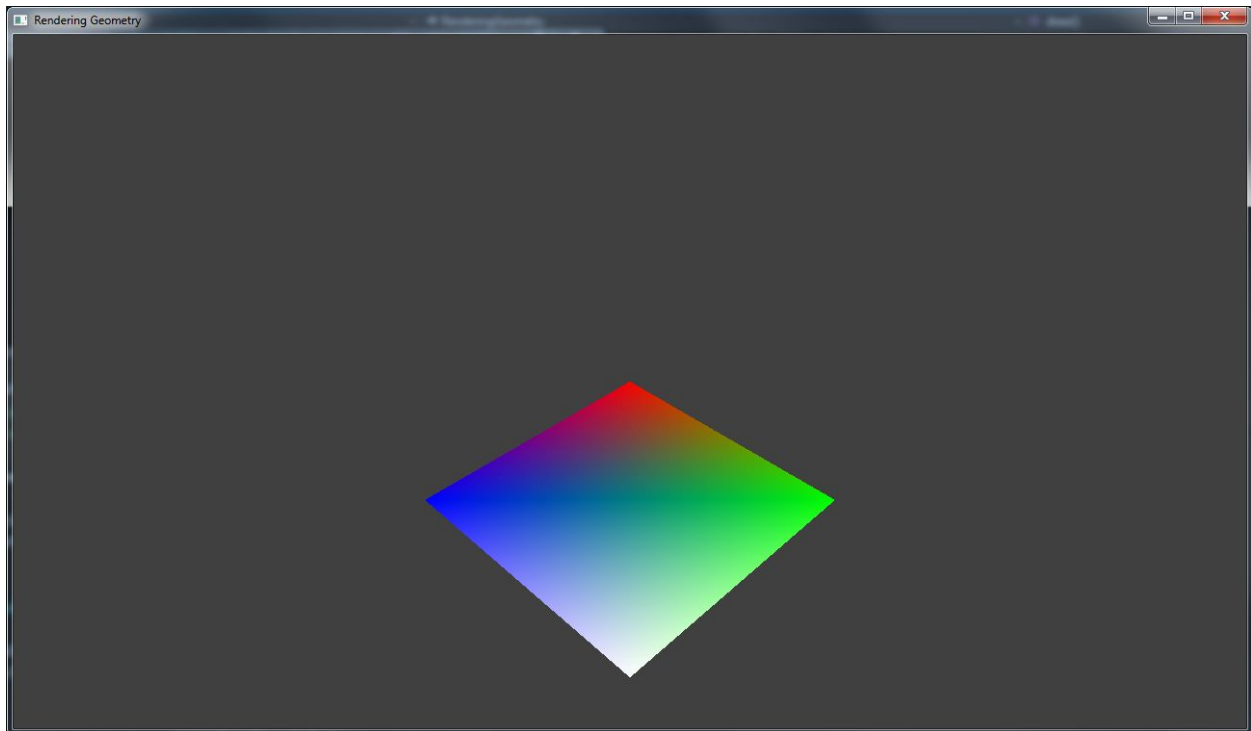
glGenBuffers(1, &p_VBO);
glGenBuffers(1, &p_IBO);
glGenVertexArrays(1, &p_VAO);
glBindVertexArray(p_VAO);

glBindBuffer(GL_ARRAY_BUFFER, p_VBO);
glBufferData(GL_ARRAY_BUFFER, 4 * sizeof(Vertex), vertices, GL_STATIC_DRAW);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, p_IBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, 4 * sizeof(unsigned int), indices, GL_STATIC_DRAW);

glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), 0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)(sizeof(glm::vec4)));

```

When the createPlane function is called, this is the result:



This project has the ability to render a static cube using the createCube function. The cube has a vertices array of eight and a indices array of seventeen where the values have been set manually.

```

Vertex vertices[8];
unsigned int indices[17] = { 0,1,2,3,6,7,4,5,0,1,5,3,7,6,4,2,0 };

c_indicesCounter = 17;

vertices[0].position = glm::vec4(-2, 4, -2, 1);
vertices[1].position = glm::vec4(2, 4, -2, 1);
vertices[2].position = glm::vec4(-2, 4, 2, 1);
vertices[3].position = glm::vec4(2, 4, 2, 1);
vertices[4].position = glm::vec4(-2, 6, -2, 1);
vertices[5].position = glm::vec4(2, 6, -2, 1);
vertices[6].position = glm::vec4(-2, 6, 2, 1);
vertices[7].position = glm::vec4(2, 6, 2, 1);

vertices[0].color = glm::vec4(1, 0, 0, 1);
vertices[1].color = glm::vec4(0, 1, 0, 1);
vertices[2].color = glm::vec4(0, 0, 1, 1);
vertices[3].color = glm::vec4(1, 1, 1, 1);
vertices[4].color = glm::vec4(1, 0, 0, 1);
vertices[5].color = glm::vec4(0, 1, 0, 1);
vertices[6].color = glm::vec4(0, 0, 1, 1);
vertices[7].color = glm::vec4(1, 1, 1, 1);

```

The function then generates buffers and vertex arrays. It then binds the buffer and sets the buffer data. It then enables the vertex attrib array and sets the vertex attrib pointers.

```

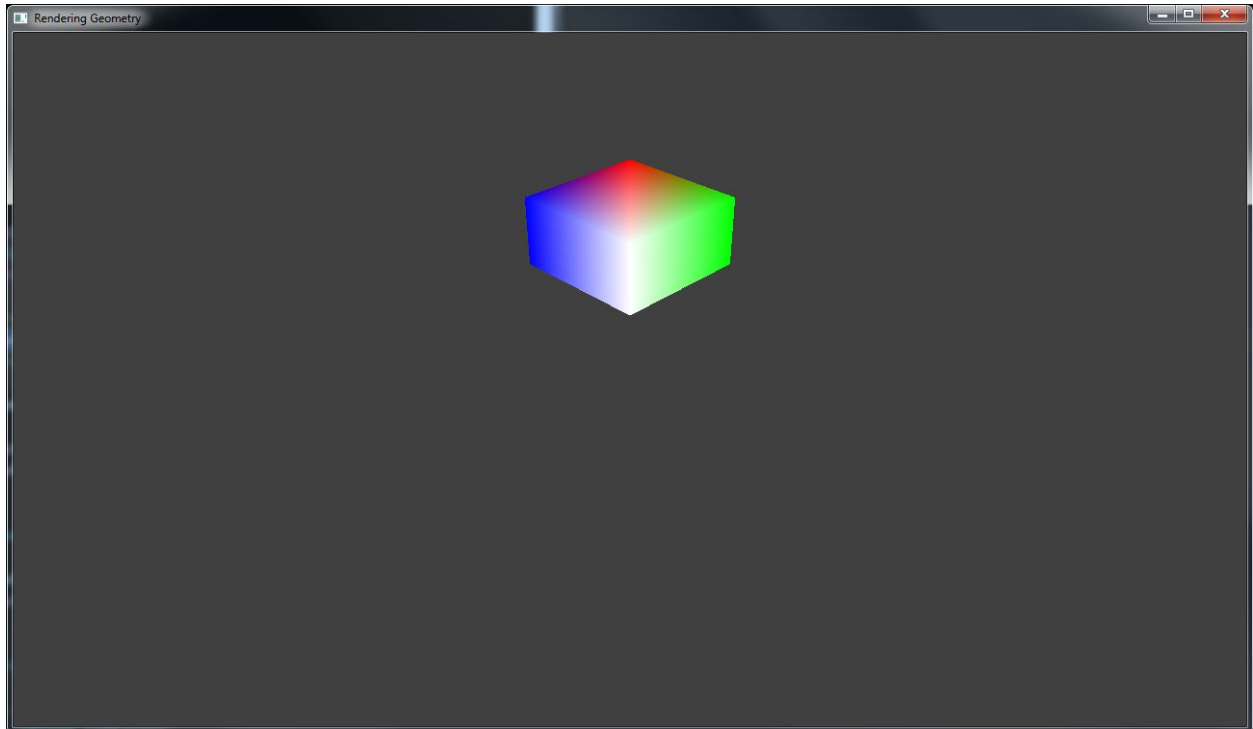
glGenBuffers(1, &c_VBO);
glGenBuffers(1, &c_IBO);
glGenVertexArrays(1, &c_VAO);
glBindVertexArray(c_VAO);

glBindBuffer(GL_ARRAY_BUFFER, c_VBO);
glBufferData(GL_ARRAY_BUFFER, 8 * sizeof(Vertex), vertices, GL_STATIC_DRAW);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, c_IBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, 17 * sizeof(unsigned int), indices, GL_STATIC_DRAW);

glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), 0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)(sizeof(glm::vec4)));

```

When the createCube function is called, this is the result:



This project has the ability to render a procedural sphere using multiple functions. Before creating the whole sphere, the vertices for a half sphere need to be generated first. In the `generateHalfSphereVertices` function, a new vertex array is created. A for loop is then used to calculate the half spheres position. The position is calculated by multiplying the radius with the `cos(angle)` and with the `sin(angle)`.

```

Vertex* RenderingGeometry::generateHalfSphereVertices(unsigned int np, const int &rad)
{
    Vertex* vertices = new Vertex[np];
    for (int i = 0; i < np; i++)
    {
        float angle = (pi * i) / (np - 1);
        vertices[i].position = glm::vec4(rad * std::cos(angle), rad * std::sin(angle), 0, 1);
    }
    return vertices;
}

```

The next step is to generate the sphere vertices. In the `generateSphereVertices` function, a new vertex array is created. We then use two for loops: one for the meridians and one for the sides. In the sides for loop, we calculate the y and z positions while leaving x alone (we are rotating around the x axis). To calculate y, we take the y position and multiply it by `cos(phi)` minus the z position multiplied by `sin(phi)`. To calculate z, we take the z position and multiply it by `cos(phi)` plus the y position multiplied by `sin(phi)`.

```

Vertex* RenderingGeometry::generateSphereVertices(const unsigned int &sides, const unsigned int &mirid, Vertex* &halfSphere)
{
    int count = 0;
    Vertex* vertices = new Vertex[sides * mirid];

    for (int i = 0; i < mirid; i++)
    {
        float phi = (2.0f * pi) * ((float)i / (float)mirid);
        for (int j = 0; j < sides; j++, count++)
        {
            float x = halfSphere[j].position.x;
            float y = halfSphere[j].position.y * std::cos(phi) - halfSphere[j].position.z * std::sin(phi);
            float z = halfSphere[j].position.z * std::cos(phi) + halfSphere[j].position.y * std::sin(phi);

            vertices[count].position = glm::vec4(x, y, z, 1);
            vertices[count].color = glm::vec4(1, 0, 0, 1);
        }
    }
    return vertices;
}

```

The next step is to generate the indices for the sphere. In the generateSphereIndices function, a new unsigned int indices variable is created. We then calculate variables botL and botR. botR is calculated by taking the beginning variable (i times vertices) plus vertices plus j modded by vertices times mirid. botL is calculated by taking the beginning variable plus j modded by vertices times mirid. We then push\_back both botL and botR to the indicesHolder variable. We then use a for loop to place all of the info stored in the indicesHolder to the indices variable.

```

unsigned int* RenderingGeometry::generateSphereIndices(const unsigned int &vertices, const unsigned int &mirid)
{
    unsigned int* indices = new unsigned int[2 * (vertices * (mirid + 1))];
    s_indicesCounter = 2 * (vertices * (mirid + 1));

    for (unsigned int i = 0; i < mirid; i++)
    {
        unsigned int beginning = i * vertices;
        for (int j = 0; j < vertices; j++)
        {
            unsigned int botR = ((beginning + vertices + j) % (vertices * mirid));
            unsigned int botL = ((beginning + j) % (vertices * mirid));
            indicesHolder.push_back(botL);
            indicesHolder.push_back(botR);
        }
        indicesHolder.push_back(0xFFFF);
    }

    for (int i = 0; i < indicesHolder.size(); i++) {
        indices[i] = indicesHolder[i];
    }
    return indices;
}

```

We now create the sphere. In the createSphere function, we call the three previous functions. The function then generates buffers and vertex arrays. It then binds the buffer and sets the buffer data. It then enables the vertex attrib array and sets the vertex attrib pointers.



```

void RenderingGeometry::createSphere(const int radius, const unsigned int verts, const unsigned int halfSpheres)
{
    const unsigned int size = (verts) * (halfSpheres);

    Vertex* vertices = new Vertex[size];
    unsigned int* indices;

    Vertex* halfSpheresVerts = generateHalfSphereVertices(verts, radius);
    vertices = generateSphereVertices(verts, halfSpheres, halfSpheresVerts);
    indices = generateSphereIndices(verts, halfSpheres);

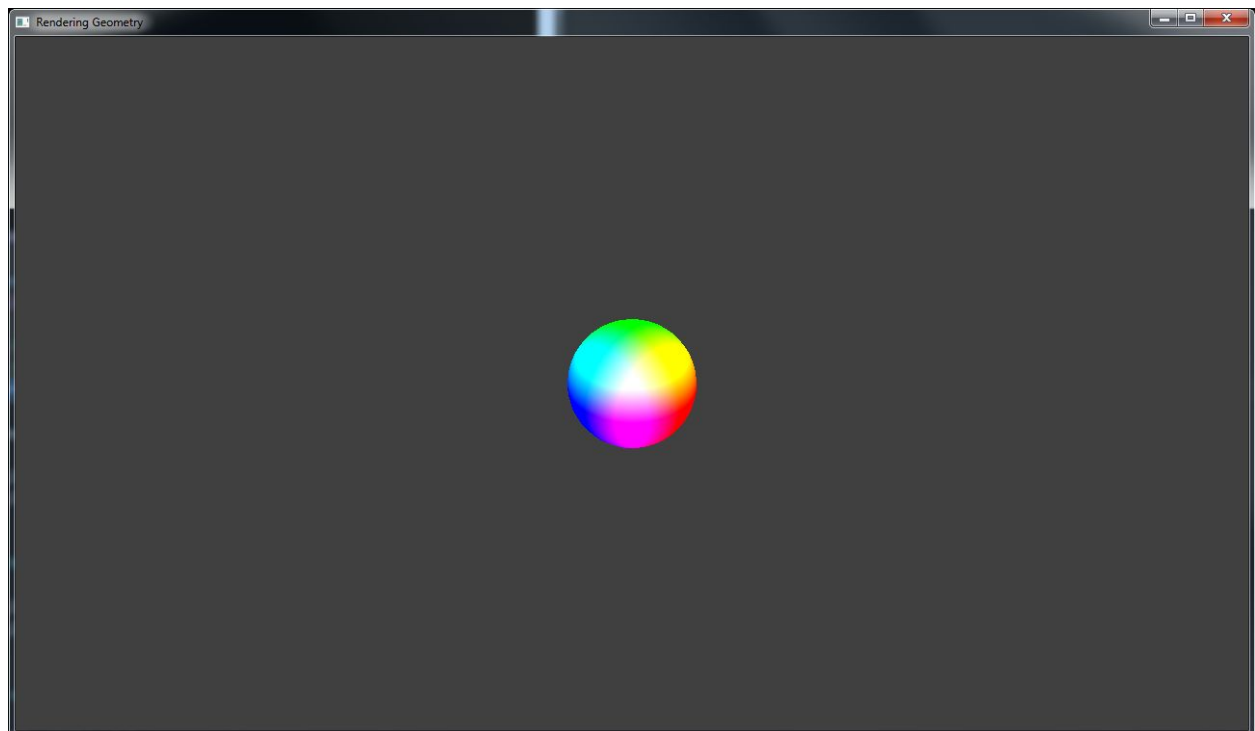
    glGenBuffers(1, &s_VBO);
    glGenBuffers(1, &s_IBO);
    glGenVertexArrays(1, &s_VAO);
    glBindVertexArray(s_VAO);

    glBindBuffer(GL_ARRAY_BUFFER, s_VBO);
    glBufferData(GL_ARRAY_BUFFER, size * sizeof(Vertex), vertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, s_IBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, (s_indicesCounter * sizeof(unsigned int)), indices, GL_STATIC_DRAW);

    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), 0);
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), 0);
}

```

When the createSphere function is called, this is the result:



The shader information has been placed into separate file: one for the vertex shader and one for the fragment shader. Both files are read from in the program.

```
const char* vsSource;  
std::string vs = ReadFromFile("vsInfo.txt");  
vsSource = vs.c_str();  
  
const char* fsSource;  
std::string fs = ReadFromFile("fsInfo.txt");  
fsSource = fs.c_str();
```