TDT 4195 - Lab Assignment 1

Kari Lovise Ness (MTING)
Håvard Kjellmo Arnestad (MTFYMA)

September 2019

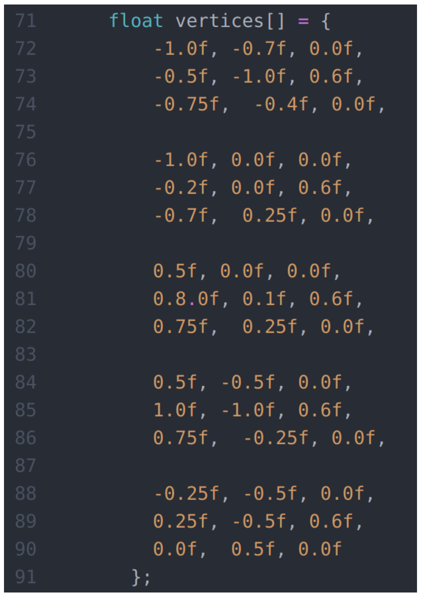
1 Drawing your first triangle

a) and b)

Not requested.

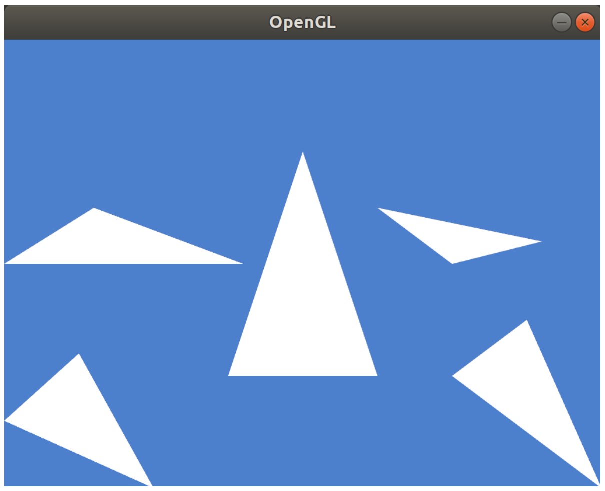
c)

The defined vertices and a picture of the 5 triangles can be seen in Fig. 1 and Fig. 2



```
    float vertices[] = {
        -1.0f, -0.7f, 0.0f,
        -0.5f, -1.0f, 0.6f,
        -0.75f,  -0.4f, 0.0f,

        -1.0f, 0.0f, 0.0f,
        -0.2f, 0.0f, 0.6f,
        -0.7f,  0.25f, 0.0f,

        0.5f, 0.0f, 0.0f,
        0.8.0f, 0.1f, 0.6f,
        0.75f,  0.25f, 0.0f,

        0.5f, -0.5f, 0.0f,
        1.0f, -1.0f, 0.6f,
        0.75f,  -0.25f, 0.0f,

        -0.25f, -0.5f, 0.0f,
        0.25f, -0.5f, 0.6f,
        0.0f,  0.5f, 0.0f
      };
```

Figur 1: The coordinates of the vertices making up 5 triangles. Each line represents the x,y,z coordinate of a vertex.



Figur 2: 5 triangles drawn from the vertices in Fig. 1.

2 Geometry and theory

a)

i)

As seen in figure 3, the triangle is missing two of its corners. This phenomenon is called clipping, as parts of the triangle outside the clip box is not displayed. The slight non-vertical clipping lines comes from the way the triangle intersects the clip box. It is not so easy to see how the triangle is oriented, but it covers the whole depth field, with the left corner clipped behind the viewer.

ii)

Clipping is a process in which one ensures that the geometry is within the bounds of the screen. The volume within the bounds is called the clipbox. In OpenGL this volume is defined as a cube around the origin, ranging between -1 and 1 along the x-, y- and z-axis.

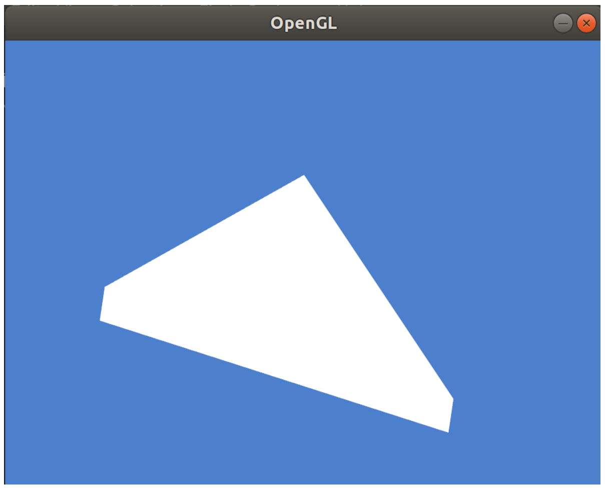
iii)

The purpose of clipping is to ensure that the geometry fits within the screen.

b)

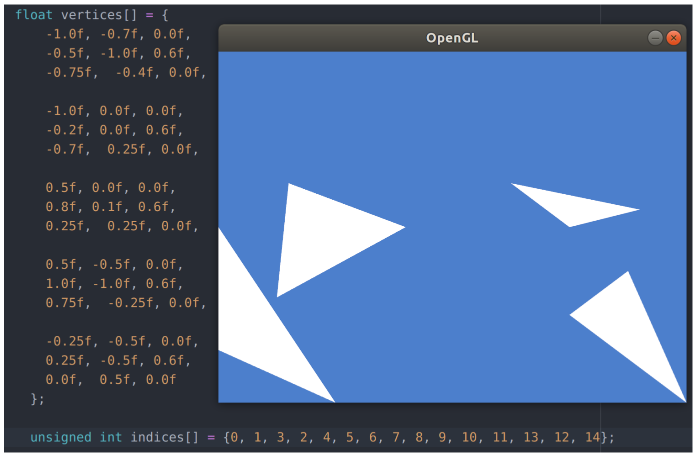
i) What happens?

Figure 4 shows that the indices 2 and 3 is swapped, essentially swapping the corners of two of the triangles on the left side of Fig. 2. We can also see that



Figur 3: Clipped triangle. Only the top vertex fits within the clip box.

index 12 and 13 is swapped, but these two are concerning the same triangle, and as a result it is not drawn.



```
float vertices[] = {
    -1.0f, -0.7f, 0.0f,
    -0.5f, -1.0f, 0.6f,
    -0.75f,  -0.4f, 0.0f,

    -1.0f, 0.0f, 0.0f,
    -0.2f, 0.0f, 0.6f,
    -0.7f,  0.25f, 0.0f,

    0.5f, 0.0f, 0.0f,
    0.8f, 0.1f, 0.6f,
    0.25f,  0.25f, 0.0f,

    0.5f, -0.5f, 0.0f,
    1.0f, -1.0f, 0.6f,
    0.75f,  -0.25f, 0.0f,

    -0.25f, -0.5f, 0.0f,
    0.25f, -0.5f, 0.6f,
    0.0f,  0.5f, 0.0f
  };

  unsigned int indices[] = {0, 1, 3, 2, 4, 5, 6, 7, 8, 9, 10, 11, 13, 12, 14};
```

Figur 4: Swapped buffer indices.

ii) Why does it happen?

This happens because the first triangle is now drawn with the vertices 0, 1 and 3 so that it looks different from a triangle drawn with vertices 0, 1 and 2.

For the middle triangle it is not rendered because it is facing away from us. It uses the same vertices, but in a different order essentially going in a different direction which redefines which way the surface faces. This is called back-face culling, and borrowing a citation from Wikipedia: *It is a step in the graphical pipeline that tests whether the points in the polygon appear in clockwise or counter-clockwise order when projected onto the screen. If the user has specified that front-facing polygons have a clockwise winding, but the polygon projected on the screen has a counter-clockwise winding then it has been rotated to face away from the camera and will not be drawn.* It improves performance by only drawing polygons visible to the viewer.

iii) What is the condition under which this effect occurs? Determine a rule.

It depends on what way the vertices are traversed, but essentially a cyclic change of indices will yield the same result. So, if the triangle with indices (0,1,2) is shown, so will triangles with (2,0,1) and (1,2,0). The orientation will be changed with for instance (0,2,1), so that will not be visible.

c)

i) Why does the depth buffer need to be reset each frame?

The depth buffer is related to the vertex shader, which defines the perspective of the frame. Each new frame depends on some kind of change of vertices, and the vertex shader is run every time a vertex is drawn.

ii) In which situation can the Fragment Shader be executed multiple times for the same pixel?

Fragments are pixels OpenGL attempts to draw on the geometries in the clip-box. These pixels might end up behind other geometries, and these pixels are therefore not necessarily the same as the pixels on screen. Rendering complicated scenes might therefore need to render a significantly higher number of fragments than pixels visible on the final frame.

iii) What are the two most commonly used types of Shaders? What are the responsibilities of each of them?

A shader is a small programme running in the GPU and is processing input. The two shaders commonly used are the vertex shader and the fragment shader.

The vertex shader is in charge of transforming each vertex in the scene. This means that the vertex shader is responsible for things like translating, rotating and scaling the vertex. In addition, the vertex shader is responsible for the projection of items on to the camera. This means that the vertex shader will take into account for example perspective.

The fragment shader is in charge of determining the colour of each fragment after the vertex shader has processed all of the vertices. The programme is running once per every fragment.

iv) Why is it common to use an index buffer to specify which vertices should be connected into triangles, as opposed to relying on the order in which vertices are specified in the vertex buffer(s)?

Using an index buffer enables the programmer to reuse the vertices. For example, if one wants to draw a square out of two triangles, two of the vertices will be common. With an index buffer, one can reuse the vertices, and in that way only store 4 vertices. Without an index buffer, one need to write the vertices in both triangles, and in that way store 6 vertices. In that way, the data stored when using an index buffer will be less and one avoid duplicate information.

v) While the last input of glVertexAttribPointer() is a pointer, usually a simple integer value is passed in. Describe a situation in which you would pass a non-zerovalue into this function.

The pointer defines the number of bytes until the first value in the buffer. The x, y and z coordinates start at index 0, while for instance the texture coordinates may start at byte $3 \cdot 4 = 12$. With only a single entry type in your buffer, this parameter is usually 0. With more entries, like the normal, colour etc. this pointer specifies the offset. One will also then have to include a stride length.

d)

i)

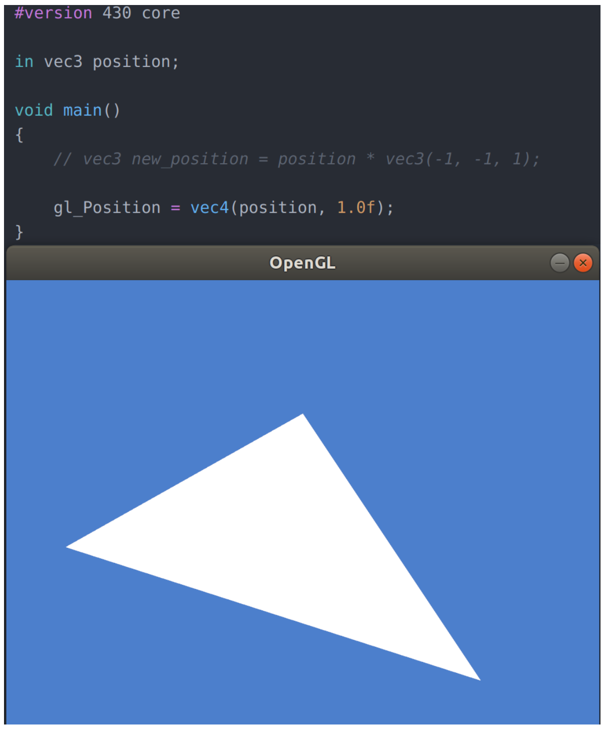
In figure 5 one can see the triangle before changing the vertex shader. In figure 6 one can see the mirrored triangle after changing the vertex shader.

ii)

As seen in 7, the colour of the triangle has been changed by changing the fragment shader.

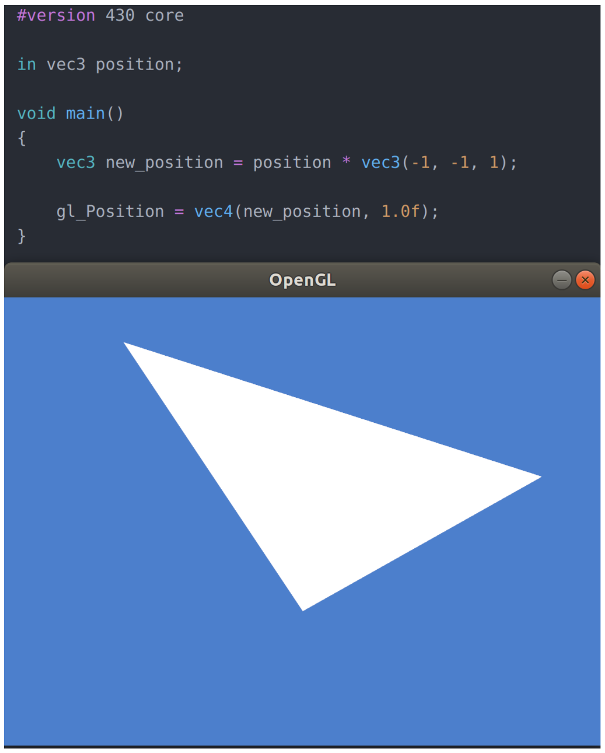
3 Bonus challenge: drawing a circle

As seen in figure 8, a circle is drawn with the triangle fan method.



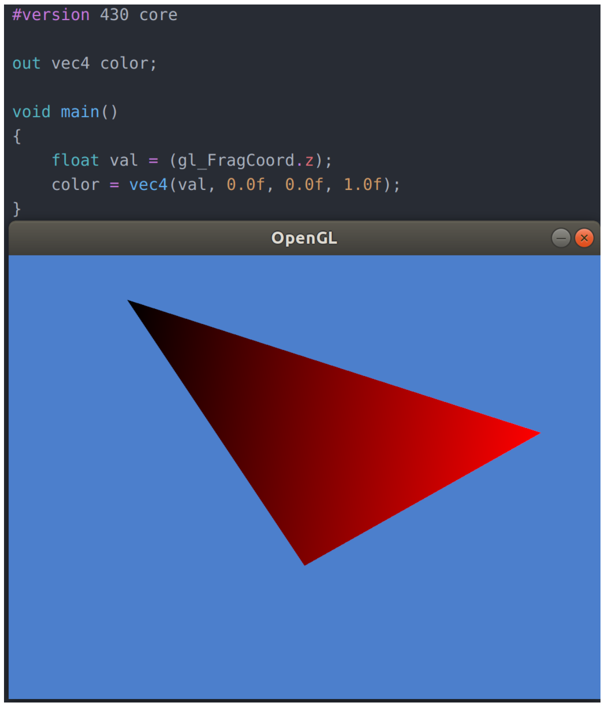
```
#version 430 core

in vec3 position;

void main()
{
    // vec3 new_position = position * vec3(-1, -1, 1);

    gl_Position = vec4(position, 1.0f);
}
```

Figur 5: Before changing the vertex shader.



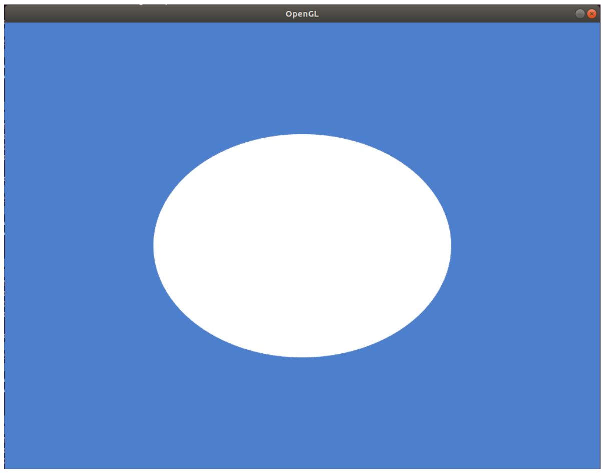
```
#version 430 core

in vec3 position;

void main()
{
    vec3 new_position = position * vec3(-1, -1, 1);

    gl_Position = vec4(new_position, 1.0f);
}
```

Figur 6: After changing the vertex shader to mirror the scene. As seen the x and y coordinates are mirrored by a multiplication with -1 while z is kept.



```
#version 430 core

out vec4 color;

void main()
{
    float val = (gl_FragCoord.z);
    color = vec4(val, 0.0f, 0.0f, 1.0f);
}
```

Figur 7: After changing the fragment shader to colour the triangle more red with increasing depth in the z direction. The value of the z coordinate is obtained by using *fl_ FragCoord.z*. The value is passed on to the red color value, so the deeper into the frame a fragment is, the more red it becomes.



Figur 8: Circle drawn with the triangle fan method