NTNU

Assignment 1

Grunnleggende Visuell Databehandling

Aditi Ravi Deshpande

September 7, 2025

Contents

1 Task 1: Drawing your first triangle

1.1 c)

I defined five distinct triangles placed symmetrically around the center of the screen (left, right, top, bottom, and center) as you can see in figure 1. Each triangle is represented by three unique

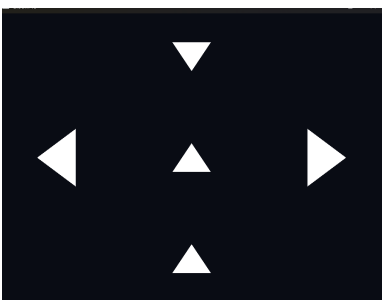

Figure 1: Five triangle

vertices (x, y, z), with z = 0 since I am working in 2D. The triangles are placed so that they are within the clip range and do not overlap. For example, the left triangle has vertices clustered around x -0.7, the right triangle around x +0.7.

2 Task 2: Geometry and Theory

2.1 a)

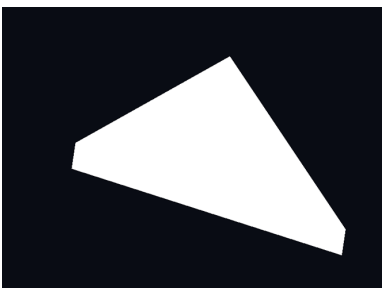

Figure 2: Clipped triangle

i. This phenomenon is called clipping.

ii. It happens if we insert out of range values to the display on which OpenGL works on. In this case, the lower parts of the triangle have not been rendered as it has gone out of the box.

iii. The purpose of clipping is to ignore/discard the parts of the input that are outside the specified range as the rendering process becomes more effective.

2.2 b)

$$v_0 = \begin{bmatrix} -0.8 \\ -0.2 \\ 0.0 \end{bmatrix}, \quad v_1 = \begin{bmatrix} -0.4 \\ -0.2 \\ 0.0 \end{bmatrix}, \quad v_2 = \begin{bmatrix} -0.6 \\ 0.2 \\ 0.0 \end{bmatrix},$$

$$v_3 = \begin{bmatrix} 0.4 \\ -0.2 \\ 0.0 \end{bmatrix}, \quad v_4 = \begin{bmatrix} 0.8 \\ -0.2 \\ 0.0 \end{bmatrix}, \quad v_5 = \begin{bmatrix} 0.6 \\ 0.2 \\ 0.0 \end{bmatrix}.$$

Indicies : Triangle 1: $(0, 1, 2)$ (counter clock-wise)

Triangle 2: $(3, 4, 5)$ (clock-wise, swapped order)

These vertices and indices form two identical triangles as shown in Figure 3:

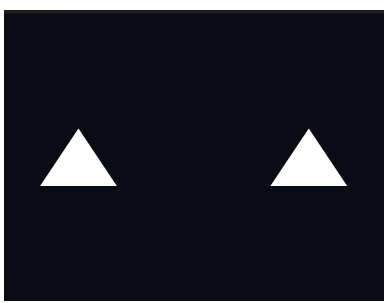

Figure 3: Two triangles

But if I swap the indices on the second triangle:

Triangle 1: $(0, 1, 2)$

Triangle 2: $(3, 5, 4)$

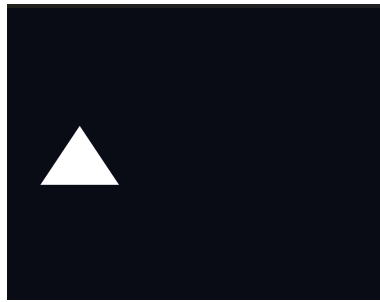

Figure 4: Swapped index

i. The second triangle disappears.

ii. Swapping the indices reverses the drawing order of the triangle, which makes OpenGL think that it is facing the other way. With back-face culling on, the back face is discarded.

iii. The rule is that triangles drawn counter-clockwise are treated as the front face. Triangles drawn clockwise are treated as the back and are culled (not displayed).

2.3 c)

i. The depth buffer stores depth information for each pixel to ensure only the closest objects to the camera are drawn. This ensures that OpenGL knows what to draw in front and what to hide behind. If the buffer is not cleared every frame, the old depth information from the previous frame stays there and blocks new objects incorrectly.

ii. The fragment shader can be executed multiple times for the same pixel when more than one triangle covers the same pixel. Each triangle produces a fragment for that pixel, so the fragment shader runs once per fragment. Afterwards, depth testing decides which one is actually visible. This is seen in :

```
gl::Enable(gl::DEPTH_TEST);
```

iii. The two types of shaders are **fragment** and **vertex** shaders. **Vertex shaders** run once for each vertex. It decides the position of the vertex on the screen by applying transformation techniques. The vertex shaders also have the task of projecting the scene, as objects need to be placed further away or closer to the camera, which imitates the way our eyes percept the world around us.
The **fragment shader** runs for each fragment (also called pixel). It decides the colour of the fragment.

iv. When drawing objects, it becomes clear that the vertices of an object are shared across several triangles that make up the object. Therefore, an effective way to store data and save memory is by indexing. We can first define the vertices, and then combine them by using their index.

v. It is a null pointer as the position data starts at the beginning of the buffer. When for example colour values are also stored, then VertexAttribPointer() will not be null, it will be the offset where the colour data starts.

2.4 d)

i. The vertex shader runs once for each vertex, and decides the final position. We can just tell the shader to change the sign of the x and y value by multiplying by (-1,-1). This is done in the simple.vert file. Figure 5 shows the result, the triangles are from b).

ii. Since I added colour data for this question, I added gradient colour shading to make it interesting. Instead of each vertex having the same colour, they all got different colours. When OpenGL rasterizes the triangle, it interpolates, so each fragment gets a mix based on how close it is to the vertex. The result can be seen in Figure 6.

Figure 5: Flipped triangles

Figure 6: Gradient colours

3 Bonus

3.1 a) Checkerboard

First, we tile the screen saying 24 x 24 is the size of the checkerboard square. We can then divide the pixel's x and y coordinates by the tile size, and then cast them to integers. The squares need to alternate between black and white. If the sum of the indices is even, then the colour is white; otherwise it is black.

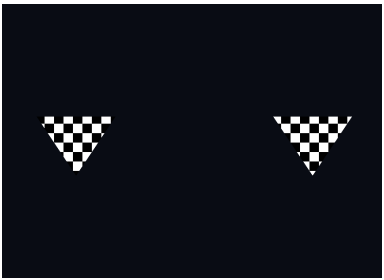

Figure 7: Checkerboard

3.2 e) Square

A square cannot be drawn directly in OpenGL. So we can combine two triangles to make a square. The vertices have a colour data

attached. The indices join the two triangles together, as seen in figure 8.

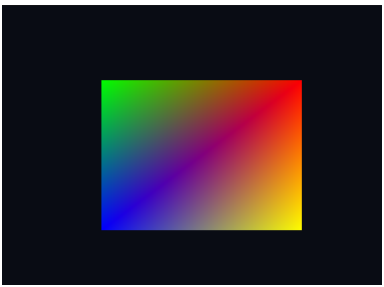

Figure 8: Square