Assignment 1

TDT4195 – Visual Computing Fundamentals

Sigurd Totland — MTTK

September 3, 2019

Task 1

c)

We start by defining a simple triangle with coordinates

$$v_0 = \begin{bmatrix} -0.0866 \\ -1 \\ 0 \end{bmatrix}, v_1 = \begin{bmatrix} 0.0866 \\ -1 \\ 0 \end{bmatrix}, v_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}. \quad (1)$$

Using simple rotation and translation matrices we can create a nice triangle pattern of 5 triangles. (The rotations are all done with pen and paper for now.) The resulting triangles are shown below in figure 1.

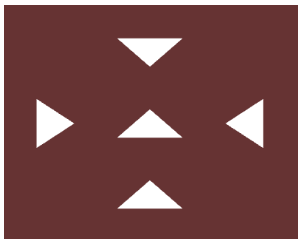

Figure 1: 5 triangles in one VAO

Task 2

a)

We now insert the single triangle

$$v_0 = \begin{bmatrix} 0.6 \\ -0.8 \\ -1.2 \end{bmatrix}, v_1 = \begin{bmatrix} 0 \\ 0.4 \\ 0 \end{bmatrix}, v_2 = \begin{bmatrix} -0.8 \\ -0.2 \\ 1.2 \end{bmatrix}. \quad (2)$$

This results in the clipped shape shown in figure 2 below.

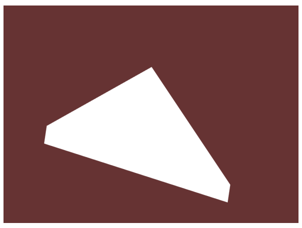

Figure 2: Clipped triangle

- The phenomenon we are seeing is clipping with the OpenGL unit-size world cube.
- It happens when the z-value of our verticies goes outside the 1-by-1 box that OpenGL renders in. This causes parts of the triangle – in this case the two lower corners – to fall outside this box and in turn not be rendered.
- The purpose of this is to not render parts of the objects that are outside the scene. This speeds up rendering.

b)

For this problem, we draw two triangles with equal proportions spaced along the x-axis:

$$v_0 = \begin{bmatrix} -0.1 \\ 0 \\ 0 \end{bmatrix}, v_1 = \begin{bmatrix} -0.1 \\ 0.5 \\ 0 \end{bmatrix}, v_2 = \begin{bmatrix} -0.7 \\ 0 \\ 0 \end{bmatrix} \quad (3)$$

$$v_3 = \begin{bmatrix} 0.7 \\ 0 \\ 0 \end{bmatrix}, v_4 = \begin{bmatrix} 0.7 \\ 0.5 \\ 0 \end{bmatrix}, v_5 = \begin{bmatrix} 0.1 \\ 0 \\ 0 \end{bmatrix} \quad (4)$$

Passing the indices $\{0, 1, 2, 3, 4, 5\}$ to the index buffer causes the two triangles to be drawn as expected, shown in figure 4 below.

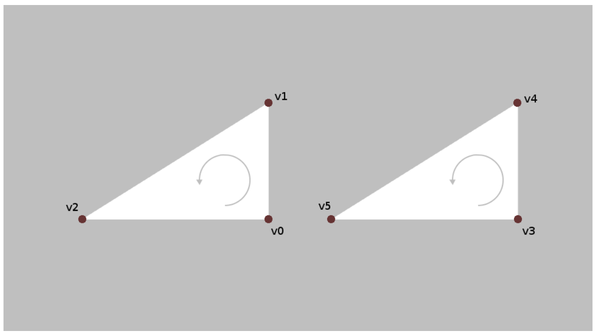

Figure 3: Non-culled triangles.

However, if we flip the *orientation* for the left triangle by using the indices $\{0, 2, 1, 3, 4, 5\}$ instead, the triangle dissapears entirely!

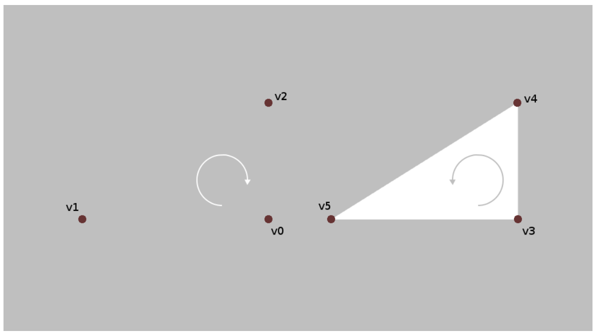

Figure 4: Left triangle culled, right triangle not.

The result we see in figure 4 might seem strange, but it is in fact a very important part of the OpenGL rendering pipeline called *face culling*. The rule OpenGL goes by in the default case is to only render faces (triangles) that are *counter clockwise winding*, like we see here. This rule can be toggled to render the opposite winding by use of the function `void glFrontFace(GLenum mode);` which takes in either `GL_CW` or `GL_CCW` for clockwise and counter clockwise winding, respectively.

c)

1. The depth buffer is used to determine which parts of a scene should be rendered and which should not due to other objects being in front of them. This test, called a *depth test* is enabled with the function `glEnable(GL_DEPTH_TEST)`. Since this option is enabled, we must clear the depth buffer after each render, or the current frame will use the depth values from the last frame.

2. The fragment shader is executed once *for each fragment*, not each pixel. These concepts, although similar, are not entirely the same. This means that the shader can sometimes be executed more than once for the same pixel. Examples include if we want to make some kind of transparent materials, or try to implement antialiasing in the fragment shader. Say OpenGL is trying to color a pixel that happens to be at the very corner of a yellow triangle that is in turn in front of a red background. The pixel will be colored somewhere in-between yellow and red, instead of one or the other.

3. The two kinds of shaders we use a lot are the *vertex shader* and the *fragment shader*. The vertex shader is one of the first steps in the OpenGL rendering pipeline, and it handles taking the vertices given to it as vertex attribute data and actually generates vertices (transforms and projects them onto the camera) for further processing by the vertex post-processing stage. The fragment shader takes in *rasterized* fragments and colors them. In addition, it sets the depth value of the fragment.

4. By specifying the order the vertices should be drawn in through indices, OpenGL makes it possible to render the same vertex multiple times whithout having to specify it more than once. This is hugely beneficial when rendering more complex objects than a simple triangle, e.g. a cube, where the corner vertices will be part of multiple triangles. Many executions of the vertex shader can be saved for each object like this.

5. The pointer value tells OpenGL where it should start reading the buffer. By passing a non-zero value, we tell it to start reading further into the buffer than from the beginning.

d)

We begin with the following scene (figure 5).

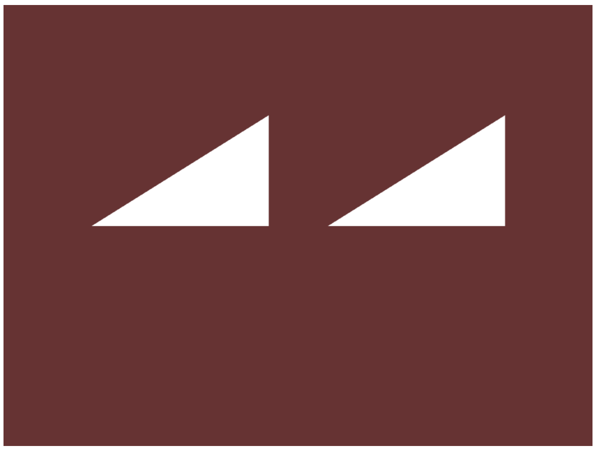

Figure 5: Two triangles on a red background.

The first task is to mirror this scene both horisontally and vertically. This is easily accomplished by flipping the signs on the x and y coordinates of each vertex in the vertex shader. This results in the following mirrored scene (figure 6).

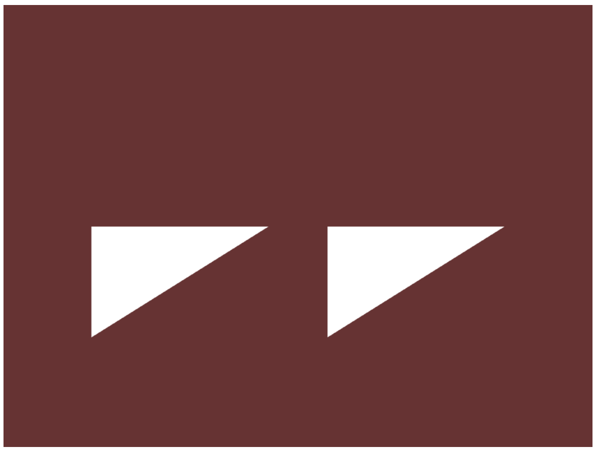

Figure 6: Two flipped triangles mirrored both horisontally and vertically.

Finally we will change the color of the triangles, and this time we want to try our hands at a gradient. We can obtain the position of each fragment from `vec4 gl_FragCoord`, so an easy way to do this would be to make the r, g and b values depend on for instance the x value of the current fragment, which would make a nice grayscale gradient going across the screen. RGB values are however quite unintuitive to work with, so it would be much better to have a hue-based color mapping. A quick google search yields us a GLSL snippet for converting from the HSV color space into RGB. Feeding in the x coordinate into the hue while leaving the saturation and value both at 1 produces the slick triangles in figure 7.

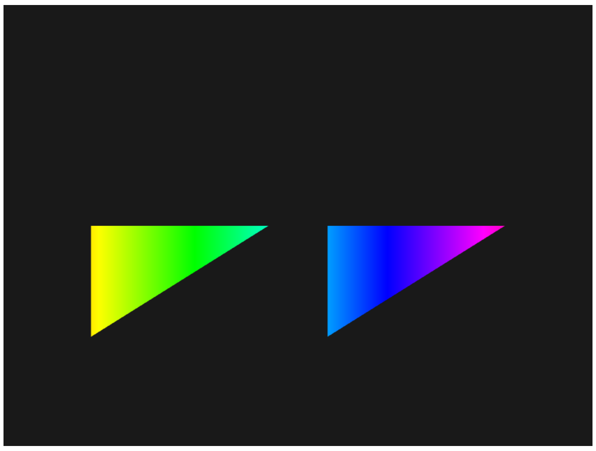

Figure 7: Two gradient-colored triangles.

Task 3 Bonus!

a) Checkerboard

Drawing a checkerboard can be done by dividing the x coordinate with some length (that will be the size of our checker squares) and then taking the modulus with 2. This first result in a zebra pattern. If we do the same calculation with the y coordinate as well and compute the exclusive OR with the x-one, we get the checker pattern we are after. See figure 8.

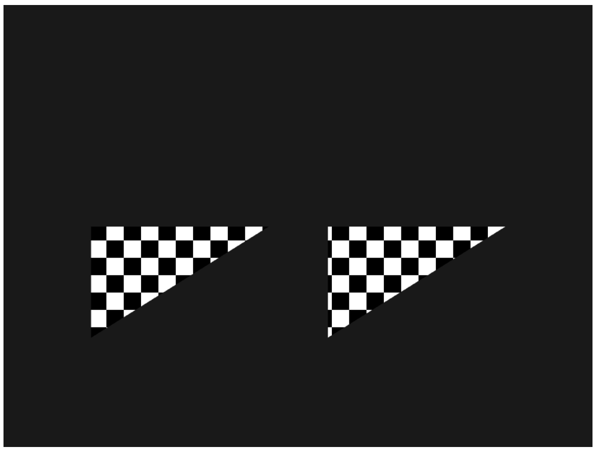

Figure 8: Checkerboard triangles.

b) Circle and sine function

We define a function to take in a vector of x coordinates and a function, and output a vector corresponding of y coordinates. This can in turn be used to create a circle, by using the functions

$$f_{\mathrm{U}}(x) = \sqrt{r^2 - x^2}, \quad f_{\mathrm{L}}(x) = -\sqrt{r^2 - x^2} \tag{5}$$

as input, or a sine by using a `std::sin` as input. For the circle, we render it using `GL_TRIANGLE_FAN`, making sure to orient the upper and lower half-circle according to the face-culling rule we have chosen. This results in the render in figure 9 below.

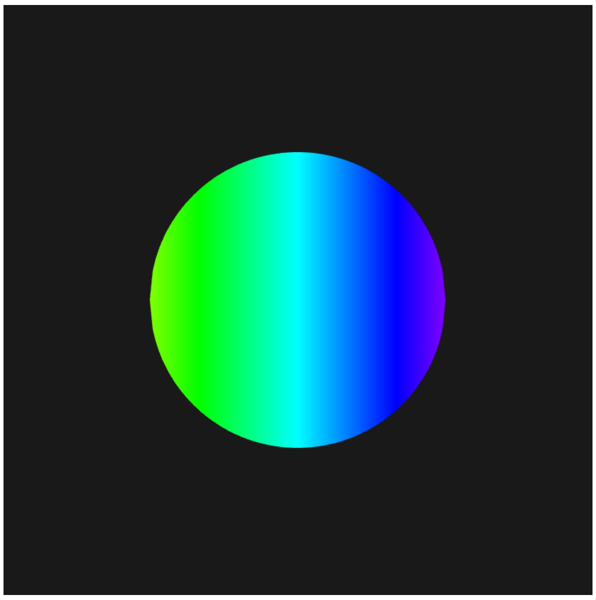

Figure 9: Circle

For the sine (this is really task **e**), we use `GL_LINE_STRIP` to produce the render in figure 10 below.

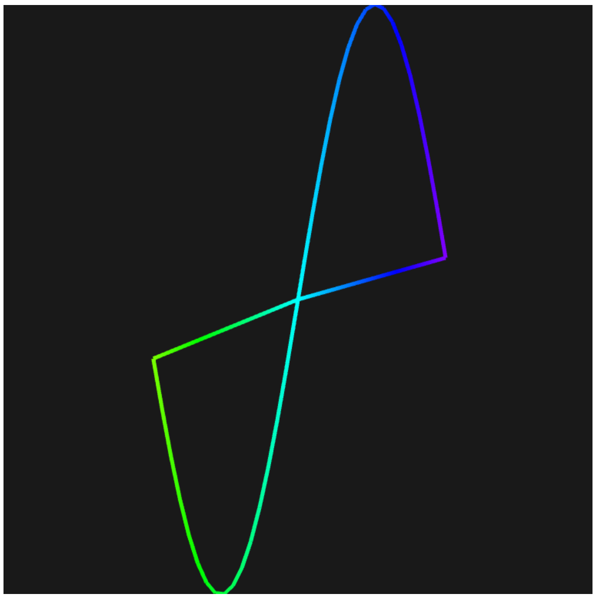

Figure 10: Sine function.

Notice how OpenGL draws a rogue line that is not really part of the sine function.