TDT4195: Visual Computing Fundamentals

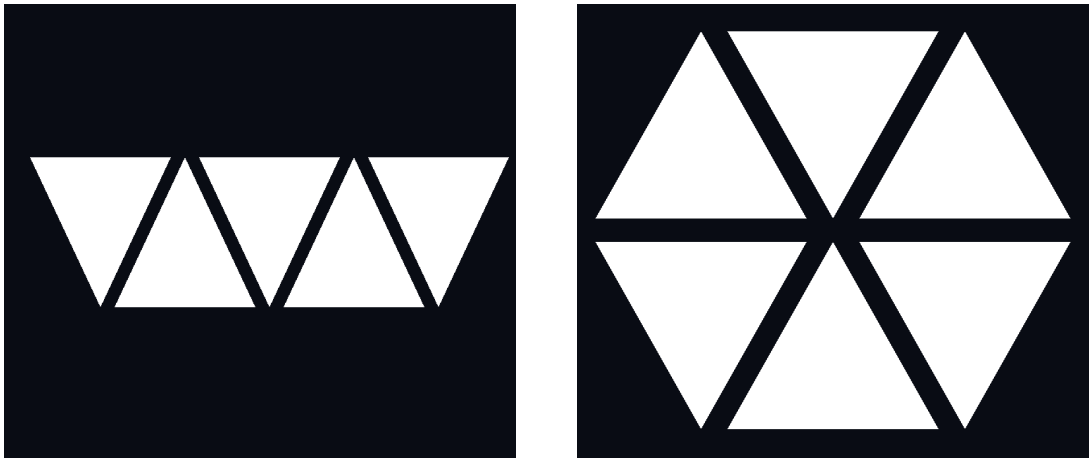
Computer Graphics - Assignment 1 Report

9th September 2023

Oluwatobi OJEKANMI

1 Drawing your first triangle

To implement this task, I completed the Vertex Specification block (i.e., the Vertex Array Object, Vertex Buffer Object, and Index Buffer Object) of the OpenGL pipeline, linked the given vertex and fragment shaders to the pipeline, specified separate vertices and VAOs for 5 and 6 distinct triangles, and called the DrawElements function to draw these triangles as shown in Figure 1 below.



(a) 5 distinct triangles (b) 6 distinct triangles

Figure 1: 5 and 6 Distinct Triangles

2 Geometry and Theory

2.1 Draw a single triangle passing through the following vertices:

$$v_0 = \begin{bmatrix} 0.6 \\ -0.8 \\ -1.2 \end{bmatrix} \quad v_0 = \begin{bmatrix} 0 \\ 0.4 \\ 0 \end{bmatrix} \quad v_0 = \begin{bmatrix} -0.8 \\ -0.2 \\ 1.2 \end{bmatrix}$$

The triangle of the given vertices is shown in Figure 2 below

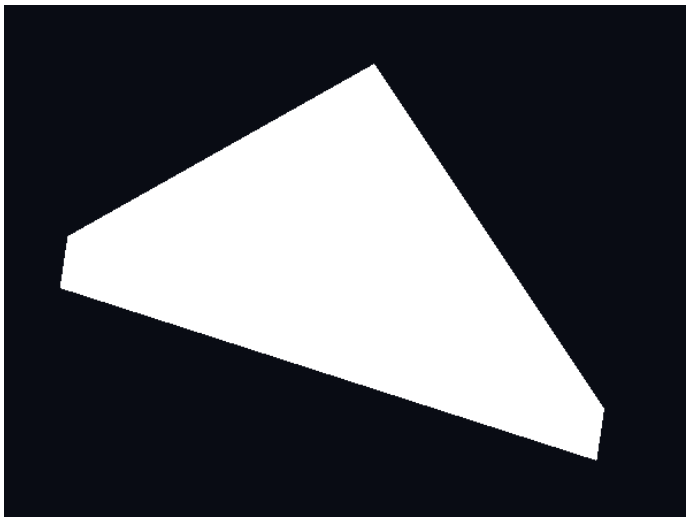


Figure 2: Triangle of the given vertices

1. What is the name of this phenomenon?

   This phenomenon is known as **CLIPPING**. It is the process of removing or discarding parts of a graphical object or scene geometry that fall outside the viewable area or a defined boundary, ensuring only the visible portions are further processed and displayed.

2. When does it occur?

   Clipping takes place when portions of a graphical object or scene geometry extend beyond the viewable area or a set boundary, resulting in the removal or clipping of these out-of-bounds segments. In OpenGL, the defined clipping space uses normalized coordinates, with x, y, and z values limited to the range of -1.0 to 1.0. Consequently, any primitives or components not fitting within these normalized device coordinates after vertex processing in the vertex shader are clipped and will not be visible following the viewport transformation.

   In this exercise, we have triangle vertices whose z-coordinates have values of -1.2. and 1.2. Consequently, the geometry undergoes clipping, removing any portions of the geometry with z-coordinate values less than -1.0 or greater than 1.0.

   To prove this, I ensured all the coordinates of the given vertices have values in the range of -1.0 and 1.0 by changing the z-coordinate values of these affected vertices from -1.2 and 1.2 to -1.0 and 1.0 respectively. As a result, we have a complete triangle shown in Figure 3 below

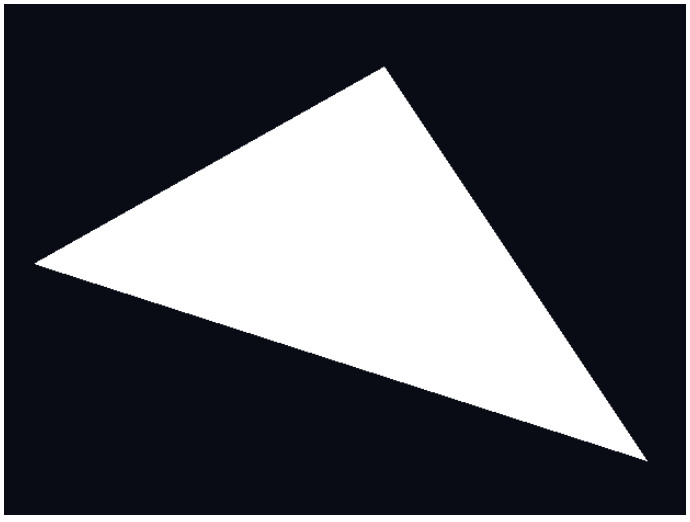


Figure 3: New Triangle using the modified vertices

3. What is its purpose?

   Clipping helps to determine which objects or parts of a graphical object/scene that are visible from a viewport and should be rendered. This helps to eliminate unnecessary calculations, optimize performance, and improve pipeline efficiency. Besides, it helps to guarantee the display of only the visible portions of objects, resulting in faster rendering times and a more realistic representation of a scene.

2.2 While drawing one or more triangles, try to swap the order in which two of the vertices of a single triangle are drawn by modifying the index buffer.

(a) Original triangles (b) After swapping the order of the 1st triangle

Figure 4: Triangles before and after reordering vertex indices

1. What happens?

   The triangle whose vertices were reordered disappeared (i.e., was not rendered).

2. Why does it happen?
   This is because the triangle has been removed by back-face culling as it cannot be viewed from the given point of view (viewing direction or viewport).

   OpenGL uses the order of the vertex indices (winding order) to determine if a triangle is front-facing or back-facing. By default, triangles defined with counter-clockwise vertices are processed as front-facing triangles while triangles defined with clockwise vertices are processed as back-facing triangles. Back-facing triangles are by default not visible from the viewpoint, thus, they are discarded in the back-face culling process.

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

   The condition under which culling occurs is typically when a primitive is invisible from a specific viewpoint. OpenGL uses the winding order (order to plot a triangle's vertices) to determine if a triangle is front-facing/visible (counter-clockwise winding order) or back-facing/not visible (clockwise winding order). Front-facing triangles are kept for further processing while back-facing triangles are culled/discarded and not processed any further.

   In addition to back-face culling, the other popular conditions for determining primitives/polygons to cull/discard are:

   (a) Primitive is occluded by another object/primitive. Occlusion culling techniques are used to remove such primitive

   (b) Object/Primitive is outside the field-of-view (viewing frustum). Frustum culling techniques are used to remove such primitive

2.3 Explain the following in your own words:

1. Why does the depth buffer need to be reset for each frame?

   To ensure accurate rendering in each frame, it's crucial to clear the depth buffer at the beginning of every frame. This action erases the depth data from the previous frame, allowing us to utilize the current frame's depth values (z-indexes) for rendering. Otherwise, all the pixels of the current image will be determined against the depth values from the previous frame which could result in certain pixels not being rendered as they might appear obscured by residual depth information from the previous frame.

2. In which situation can the Fragment Shader be executed multiple times for the same pixel? *(Assume we do not use multisampling.)*

   The Fragment Shader can be executed multiple times when there is an overdraw. This occurs when multiple fragments pass the depth test and cover the same pixel, the Fragment Shader will be executed for each of them. This can result in multiple shader executions for the same pixel.

3. What are the two most commonly used types of Shaders?

   The **Vertex Shader** and the **Fragment Shader**.

   (a) Vertex Shader - This is a programmable step of the graphics pipeline used for pre-processing the data from the input buffer(s). It is used to transform (translate, scale, shear, rotate, etc.) each vertex. It is run once for each drawn vertex.

   (b) Fragment Shader - This is also a programmable stage of the graphics pipeline used for determining the final color of each fragment. It is also run once for each fragment.

4. 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 the vertices are specified in the vertex buffer(s)?

   In many 3D models, vertices are shared among multiple triangles. Therefore, using an index buffer to specify how the vertices should be connected into triangles helps to avoid needless repetition of vertices, optimize memory usage, optimize data transfer to the GPU, and enhance rendering efficiency.

5. While the last input of gl::VertexAttribPointer() is a pointer, we usually pass in a null pointer. Describe a situation in which you would pass a non-zero value into this function.

   The pointer describes the offset of where the position data for an attribute begins in the buffer. Therefore, if the buffer contains data of multiple vertex attributes, the pointer is used to determine the position of the first data for a specific vertex attribute being linked to the VAO.
   For instance, let us consider a Vertex Buffer Object (VBO) that holds data for both vertex coordinates and color attributes in the following order: [x, y, z, R, G, B]. Assuming each value occupies 4 bytes of memory, we would need to specify a pointer value of 0 or a null pointer when linking the coordinate data to the VAO using gl::VertexAttribPointer(). However, when linking the color data to the VAO using the same function, we would need to provide a pointer value of 12 bytes to ensure that it starts reading the color data at the correct location within the VBO. In this case, the non-zero pointer value is necessary to correctly bind the color attribute to the VAO.

Since the position data is at the start of the data array this value is just 0. We will explore this parameter in more detail later on

2.4 Modify the source code of the shader pair to:

1. Mirror/flip the whole scene both horizontally and vertically at the same time.

To flip the whole scene both horizontally and vertically at the same time, I multiplied each vertex's coordinates with

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

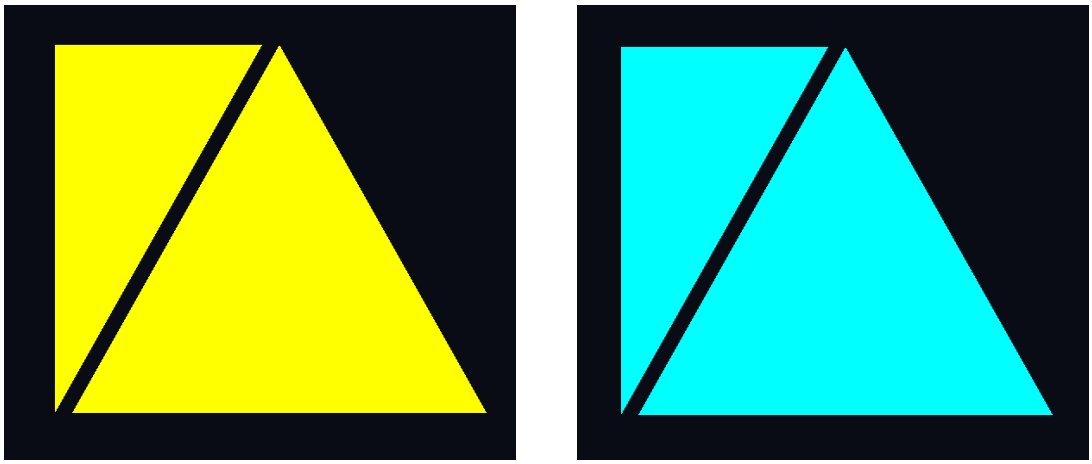
The initial and corresponding flipped scenes are therefore shown in Figure 5 below

(a) Initial Scene (b) Flipped Scene

Figure 5: Original and Flipped Scene

2. Change the colour of the drawn triangle(s) to a different colour.

   To change the color of the rendered triangles, I adjusted the color vector within the Fragment Shader. Specifically, I changed it from vec4(1.0f, 1.0f, 1.0f, 1.0f) to vec4(1.0f, 1.0f, 0.0f, 1.0f) for yellow and vec4(0.0f, 1.0f, 1.0f, 1.0f) for cyan. The corresponding triangles are shown in Figure 6 below



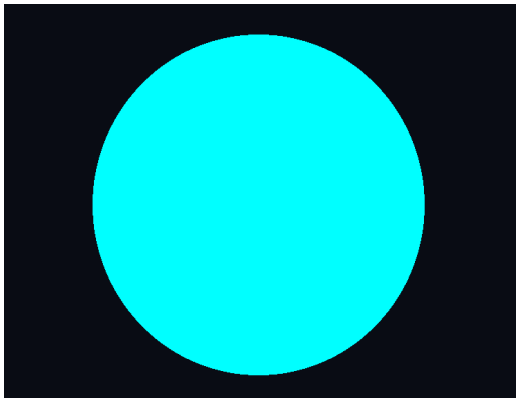
(a) Yellow Triangles (b) Blue Triangles

Figure 6: Triangles with different colors

PS:*I attempted 3 optional questions. Their plots are shown on the next page.*

3 Optional Tasks

3.1 Draw a circle.



(a) A circle approximated with triangles

(b) A circle approximated with Lines

Figure 7: Circle approximates

3.2 Draw a spiral.

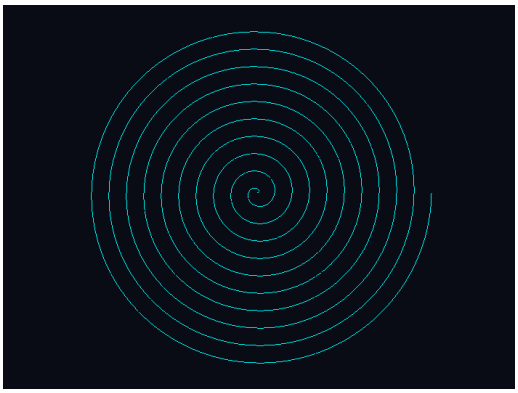
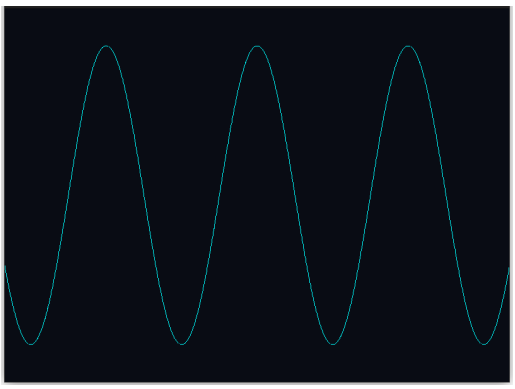
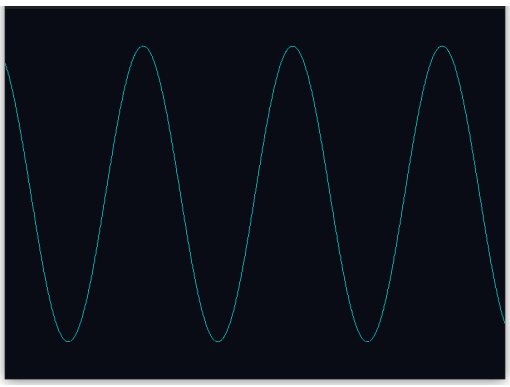


Figure 8: A spiral approximated with lines

3.3 Draw a sine curve.



(a) A cosine curve approximated with lines



(b) A sine curve approximated from lines

Figure 9: Cosine and Sine curves approximated with lines