Computer Graphics - Assignment 1

Christopher Braathen & Michael Brusegard

August 2024

1 Drawing your first triangle

c) Here is a screenshot showing 5 distinct triangles within a VAO.

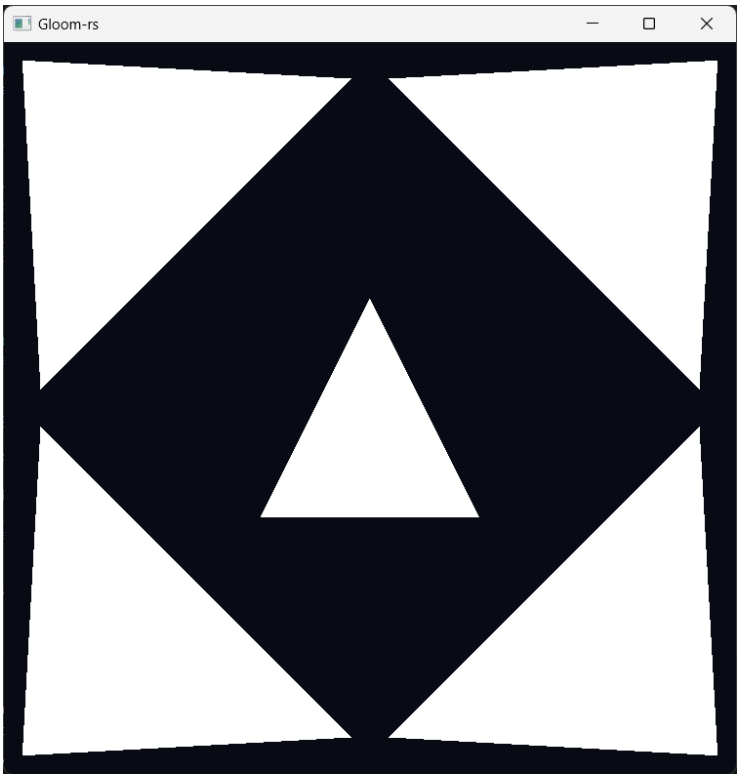


Figure 1: Five distinct triangles.

2 Geometry and Theory

a) Here is a triangle outside the clip space boundary.

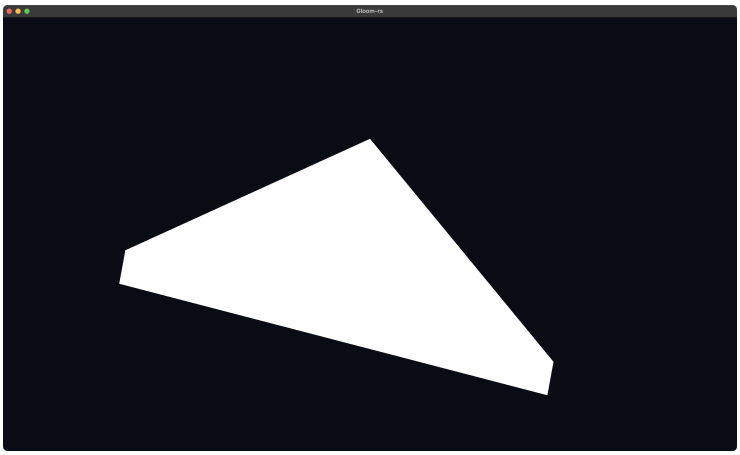
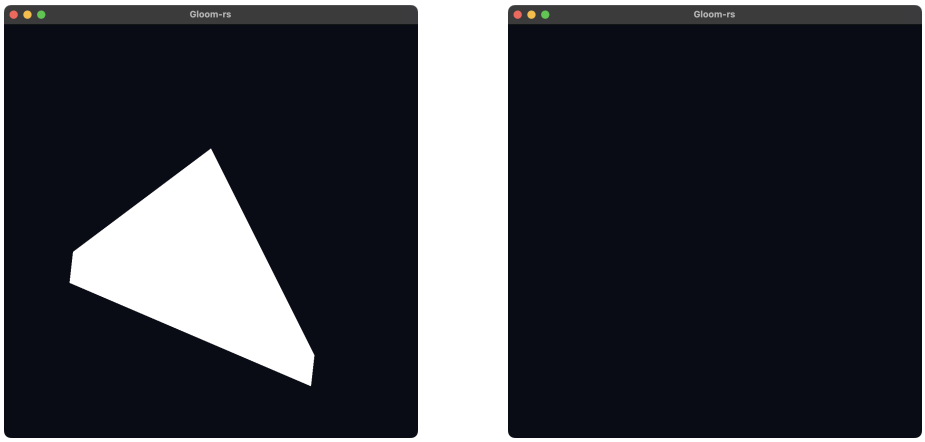


Figure 2: A single triangle outside the clip space boundary.

i) The name of this phenomenon is called clipping (or culling). Clipping is a process in the graphics pipeline which improves the performance due to avoiding unnecessary calculations to determine a shape. In this specific case this is called frustum culling.

ii) Clipping occurs when shapes are outside of the clip space boundary. The clip-space boundary (atleast in OpenGL) are between -1.0 and 1.0 and thus the triangle in Figure 2 will be clipped since some of its vertices are outside this boundary. This means if we want to transform any shape from view-space to clip-space we have to perform a projection matrix transformation.

iii) As mentioned it improves performance because it removes unnecessary calculations. The GPU recieves all the necessary data it needs to draw one frame. Therefore it is important to construct this data such that it does not contain elements that are irrelevant in said frame.



a) [0, 1, 2] b) [2, 1, 0]

Figure 3: A triangle (a) disappearing due to back-face culling (b).

b) *i*) The triangle disappears, which suggests it is not being considered in the rendering of the frame.

ii) This happens due to face-culling (or back-face culling). In 3D graphics we can determine a "back-face" with either a normal vector given to the triangle or a by using the index buffer and determining said triangles winding order. OpenGL uses a winding order of counter-clockwise (CC), but this is just due to convention and standards.

iii) A rule we determined was that; the index buffer has nothing to do with the culling of the triangle. A triangle must be formed such that a point which leads to another point is a straight line, the third point of the triangle must be a point such that it will form a counter-clockwise loop back to the starting point.

c) *i*) The depth buffer is a tool that keeps track of how far objects are from the camera to ensure proper rendering. The depth buffer needs to be reset each frame to avoid leftover depth data from previous frames messing up the rendering of the new frame. In a scene with a sphere moving leftwards, if you do not reset it, you will observe a ghost trail of the sphere as it moves leftward, with parts of the old positions still appearing in the scene.

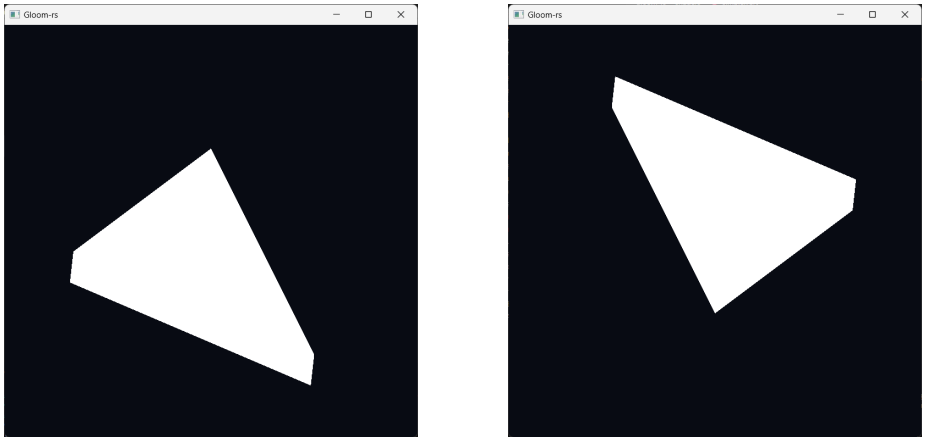
ii) The Fragment Shader determines the final color of each pixel on the screen, and running it multiple times for the same pixel can be beneficial for creating complex effects like blending, shadows, or transparency. The Fragment Shader can be run multiple times for the same pixel when different objects overlap on the screen, when blending is used for transparency, when using a stencil buffer, or during multi-pass rendering. (Assuming we do not use multisampling.)

iii) The two most commonly used shaders are the Fragment (Pixel) Shader and the Vertex Shader. The Fragment Shader's main responsibility is to determine the final color of each pixel (fragment) on the screen. It applies textures, lighting calculations and color effects, as well as shading (shadows) and blending (transparency). It outputs the final color and possibly other values like depth or stencil data for each pixel.
The Vertex Shader's main responsibility is to handle vertex attributes like position, color and texture coordinates. It processes each vertex transforming its position from model coordinates (3D space) to screen coordinates (2D space). Additionally, it applies affine transformations and outputs the vertex position.

iv) Using an index buffer to determine which vertices should be connected into triangles as opposed to relying on order is common because it reduces memory usage by allowing vertices to be used for multiple triangles which improves performance. It also simplifies geometry manipulation by allowing you to modify the connections between vertices without altering the vertex data itself, since you can modify the index buffer directly which is less expensive.

v) You pass a non-zero value to glVertexAttribPointer() when your vertex data is interleaved in a single buffer, meaning multiple attributes (like position and color) are stored together. The non-zero value specifies the offset where each attribute starts within the buffer. This is necessary to correctly point to the start of each attribute, such as when color data follows position data.

d) *i*) To achieve a mirrored/flipped scene, both horizontally and vertically, we multiplied the vertex shader positions with -1 at both the x-position and the y-position.



a b

Figure 4:
The same triangle without multiplying -1 (a), and with multiplying -1 (b).

ii) To change the color of the triangles we edited the fragment-shader such that the `color` variable has different RGB values. We chose a burgundy-red color and converted the RGB values to normalized versions.

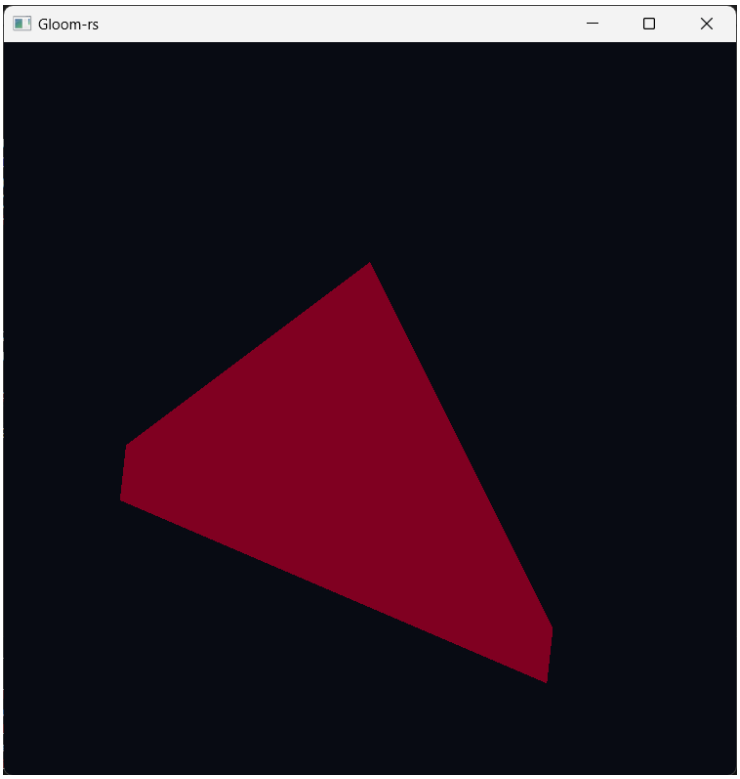


Figure 5: A triangle with burgundy-red color

3 Optional Bonus Challenges

a) Here is the checkerboard pattern. We used the combined value of the `x` and `y` fragment coordinates to determine if they are either even or odd with the formula: $(\lfloor \frac{x}{\text{boxSize}} \rfloor + \lfloor \frac{y}{\text{boxSize}} \rfloor)\%2$.

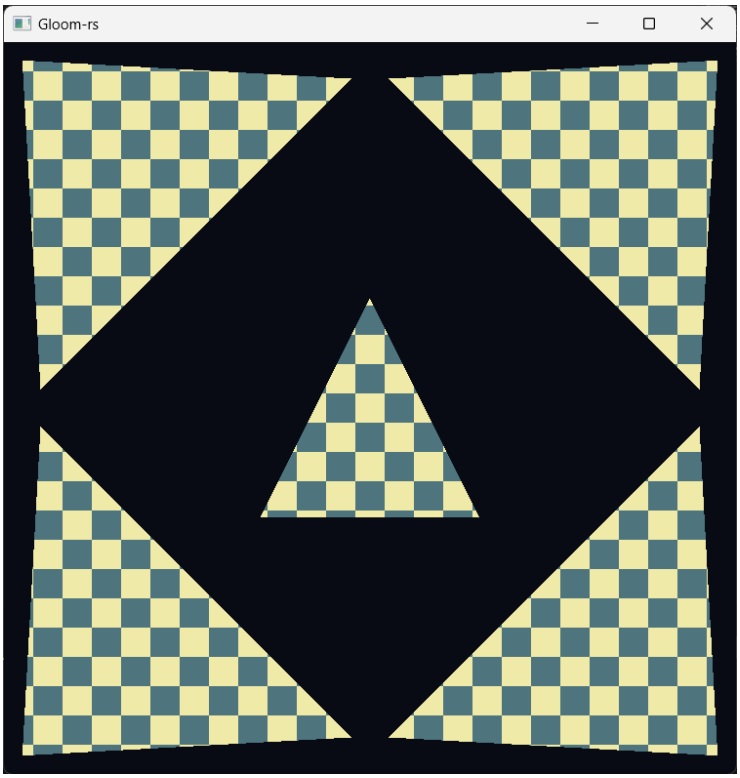


Figure 6: 5 distinct triangles now with checkerboard pattern.

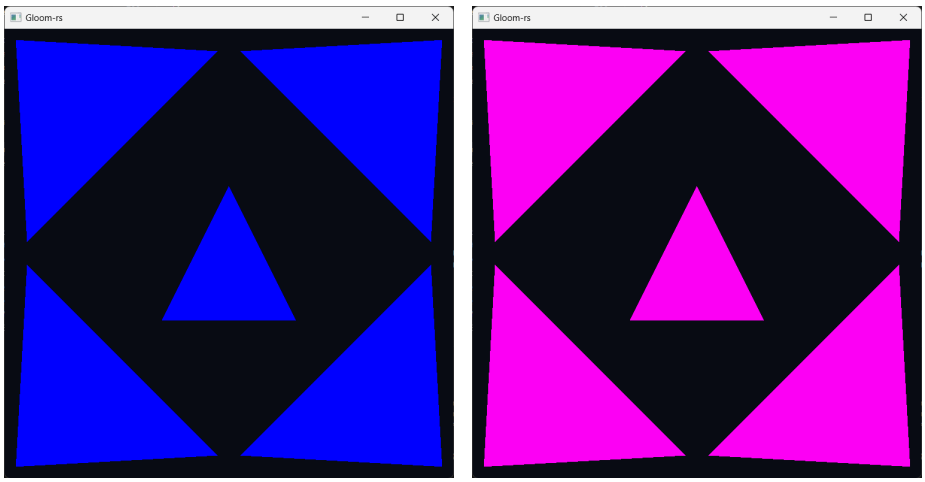
b) Here is a drawn circle with an attempt to represent the Japanese flag. We used a signed distance function (SDF) to calculate which colors to choose. The SDF was found through Inigo Quilez's website: https://iquilezles.org/.

Figure 7: Drawn circle.

c) Here is a drawn spiral. To achieve a spiral we used the fractal function in `GLSL` with an angle θ as an argument (and some other variables affecting the angle).

Figure 8: Spiral pattern.

d) Although we will not be able to show it animated here, the fragment shader code will include a commented section for a color changing slowly over time. To achieve this we passed on $sin(\textbf{time})$ as a value to one of the color arguments, and $cos(\textbf{time})$ as a value to the other arguments.



a b

Figure 9: Blue triangles (a) turning pink (b).

e) To create a triangle in the fragment shader we used Inigo Quilez's equation for a triangle given in his blog-article here. We iterated by finding linear equations to match where pixels should be drawn on the edge, but struggled to find out how to fill the inside. As a result we found Quilez's version and it worked like a charm.

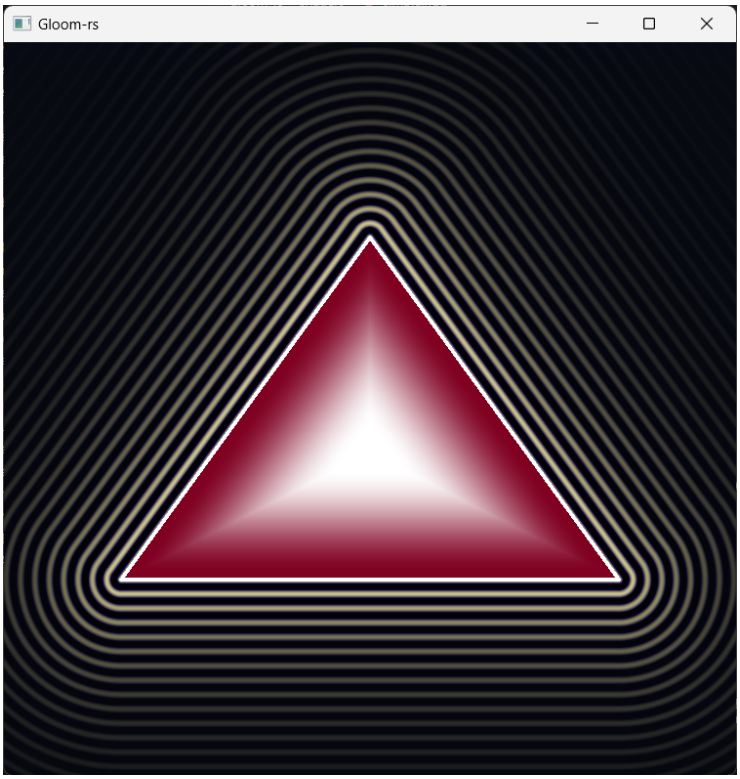


Figure 10: Cool triangle inspired by synth wave lines.

f) Here is our sine wave. It is drawn using an amplified sine wave where we match the given `glFragCoord.y` with the output given from the sine wave function. Such we have two different signs and therefore can draw multiple colors separating it. The crescent moon is represented by using some form distance-boolean mumbo jumbo. In short we define the space of two circles calculate its signed distance and only draw in the crescent moon space.

Figure 11: Sine waves with a crescent moon.