TDT4195 assignment report 1

Nils Reed

September 2021

1c

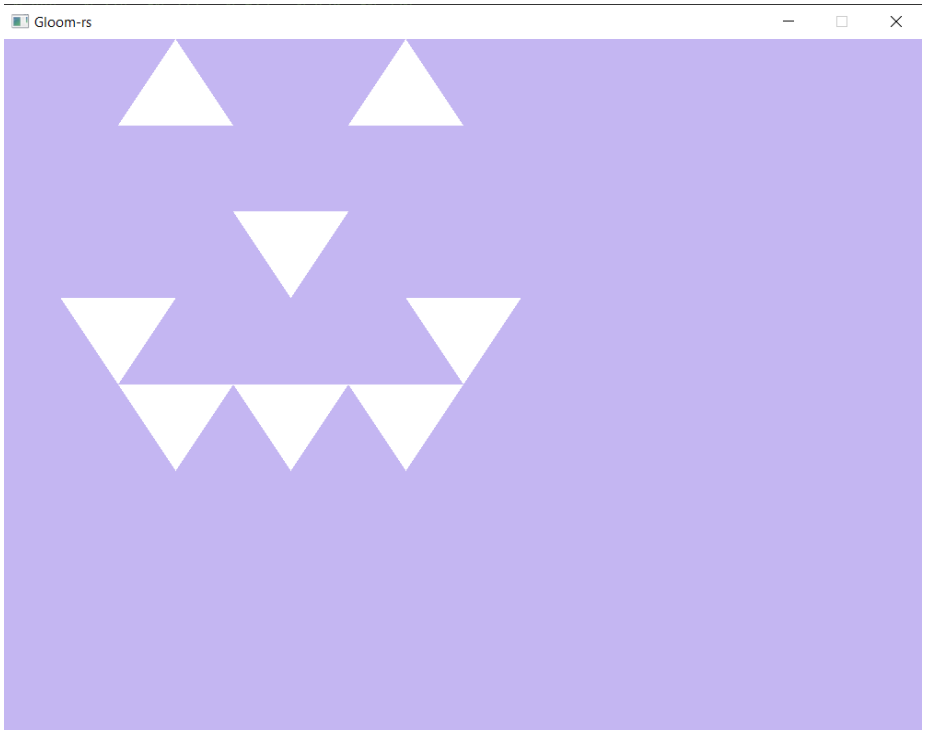

Figure 1: 8 triangles making up a cat smiling :)

2a

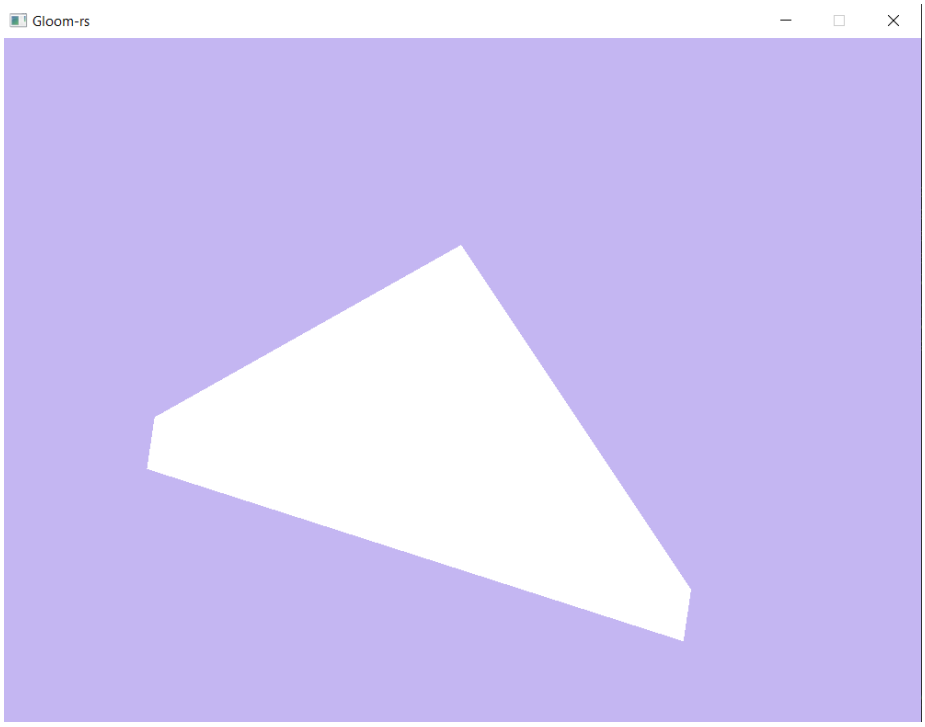

Figure 2: Triangle generated from vertices given in task

i) The name of the observed phenomenon is *clipping*.

ii) Clipping occurs when one or more of the components of a vertex has absolute value larger than 1. It is more apparent when the x and y components are larger than 1, because this stretches the figure alle the way to the border of the window. When the z components are larger than 1 (like in this case) it'll not be as apparent straight away, but it is the same phenomenon.

iii) The purpose of clipping is to make sure that elements that are not "In the view of the program" at the moment in the scene, are not rendered to the screen. In other words: it ensures that no out-of-bounds (wrt screen and window) rendering happens on the screen.

2b

Figure 3: Result of switching around the values in the index buffer.

i) For certain permutations of the index buffer, the triangle disappears. For others, it's still drawn as normal.

ii) It happens because OpenGL draws points in counter-clockwise order, and when the vertices corresponding to the indices (set in order) can't be drawn in counter-clockwise order (when they're drawn in order from lowest to highest index), it simply won't draw them.

iii) As mentioned above, it happens when the vertices corresponding to the indices can't be drawn in counter-clockwise order. In other words, when the indices are not in order from smallest to largest in the vector (this includes the vector wrapping around itself). E.g. for a VAO containing indices 0, 1 and 2, this includes the vectors:

   - $\begin{bmatrix}0 & 1 & 2\end{bmatrix}$
   - $\begin{bmatrix}1 & 2 & 0\end{bmatrix}$
   - $\begin{bmatrix}2 & 0 & 1\end{bmatrix}$

2c

i) The depth buffer is used to compare the depths of primitives in the scene, to know what primitive is closest to the viewer an thus being the one to be rendered in that pixel. If one does not clear the depth buffer each frame, one would be comparing depths of primitives in the current frame with primitives already rendered to the screen, which does not make sense.

ii) When more than one object is placed in the same point by the vertex shader (this means one object in practice lies in front of the other(s)), the fragment shader will be executed for all of them, even though only one of them will show as a pixel on the screen.

iii) The two most commonly used shaders are vertex shaders and fragment shaders. The vertex shader is responsible for projecting the scene onto the camera, that is scaling and placing the primitives so that they give a sense of perspective as if we were viewing them as real objects in the real world even though they're just pixels lighting up on a screen. The responsibility of the fragment shader is to give each fragment (the things that OpenGL tries to draw) colour.

iv) It's common to use an index buffer to specify the which vertices should be connected into triangles because for large and complex models you can save quite a bit of space doing this, since many of the vertices probably overlap. This way, you don't have to input the same vertex many times, but rather specify in the index buffer each time a vertex should be used.

v) The last input defines how many bytes into the buffer the interesting data starts. Setting this to a non-zero value is relevant when the vector we sent to the buffers as input contain more than one entry type (like vertices, colour etc.), and the entry type we're interested in at the moment is not the first in the array.

2d

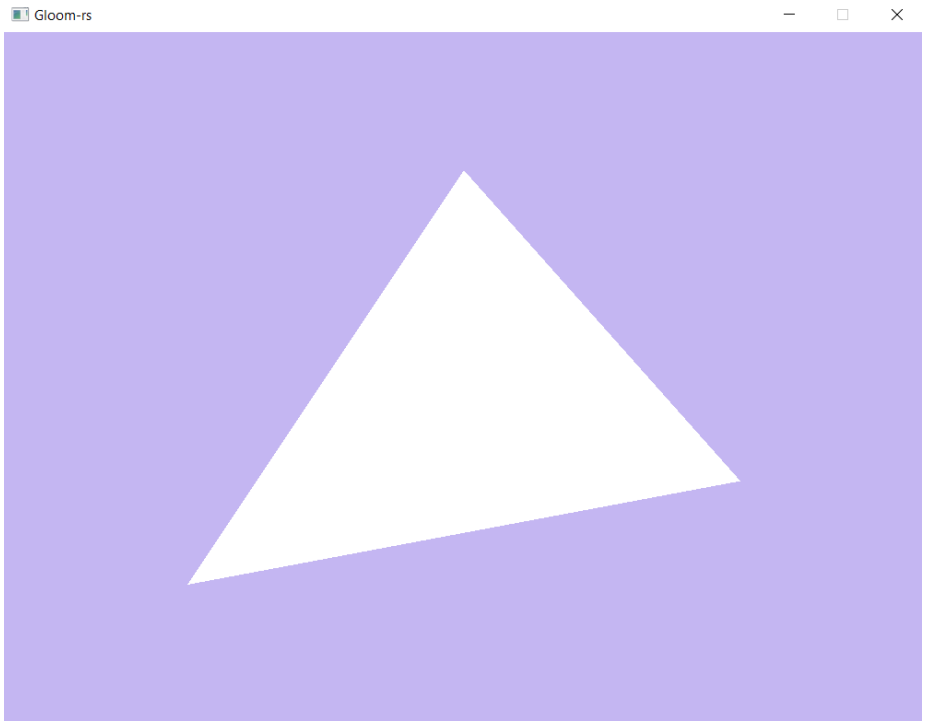

Figure 4: Before mirroring and colouring

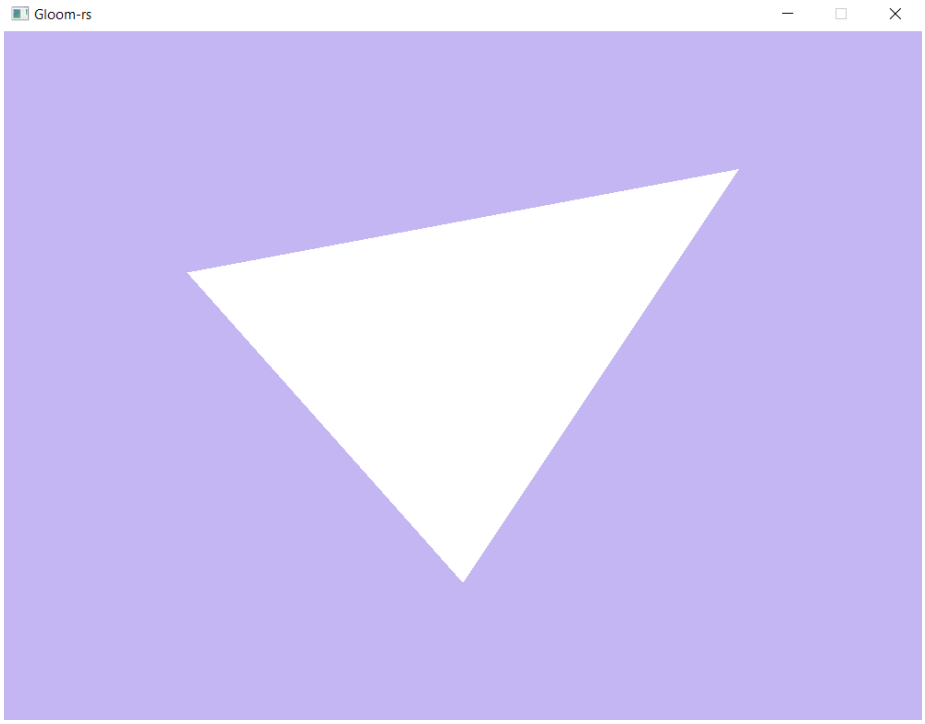

Figure 5: After mirroring

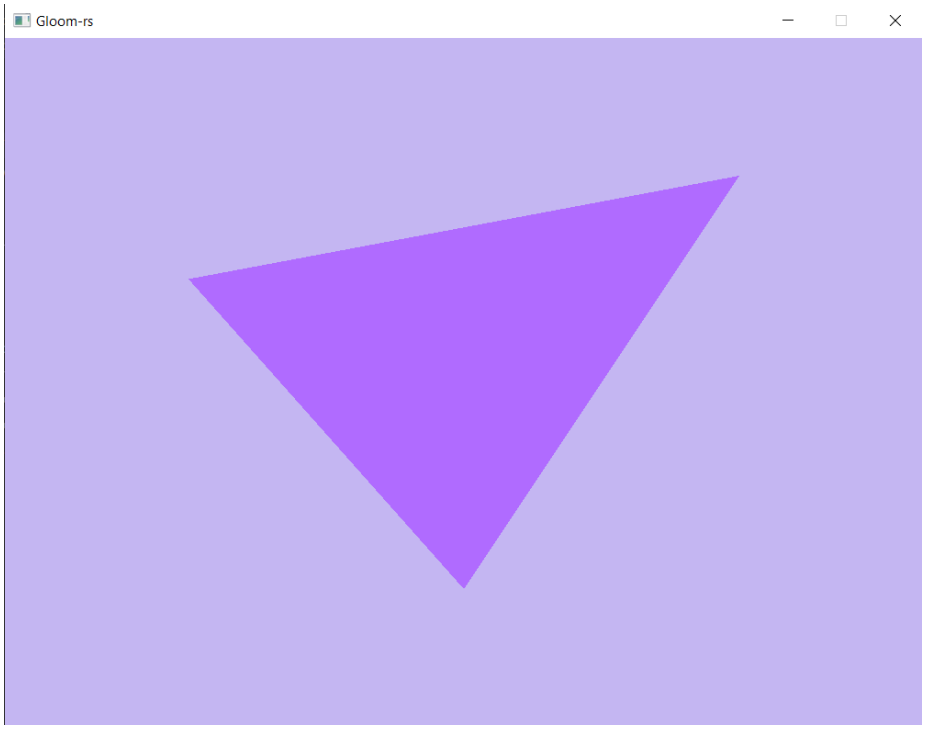

Figure 6: After colouring

i) Mirroring on both x and y axis was acheived by multiplying the input `position` to the vertex shader by a scaling matrix with $s_x = s_y = -1$ and $s_z = 1$ in `simple.vert`, and the feeding the result of this into the new `gl_Position` instead of `position`

ii) Changing the colour of the rendered triangle was acheived by changing the values of the three first components in the `color` output variable in `simple.frag`