TDT4195 – Assignment 1

Task 1

Task 2

a)

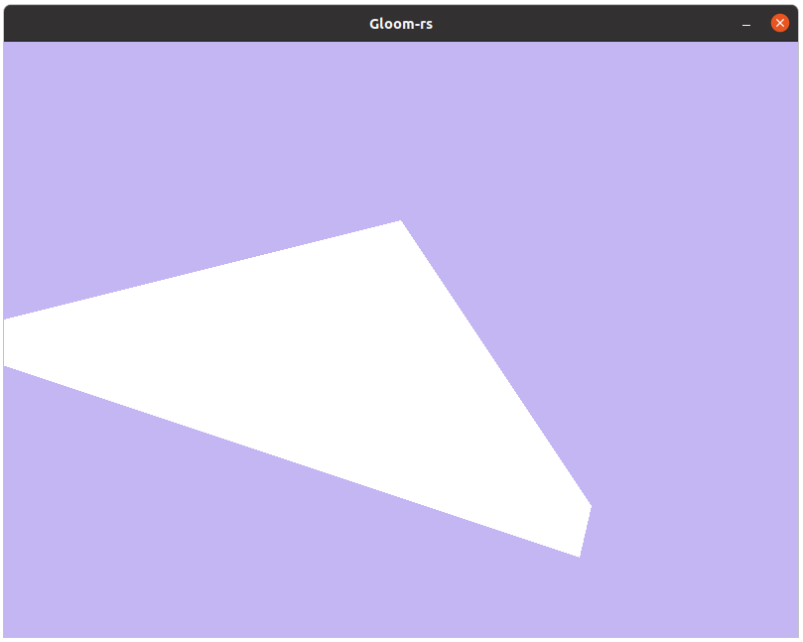

Figure 1: Task 2a)

i) The name of the phenomenon in figure 1 is called clipping.

ii) The box we place our triangle in is in 3D, the same as the coordinates for vertices. The box(also called clipping object) is a 2x2x2, giving us a limit for the length each side in the triangle to be between 2 (along the axes) and 2sqrt(3) (along the diagonal). The clipping on the left looks natural in our figure, because it comes from the left wall of our box doing the clipping. The right clipping does on the other hand not look natural, since it is “in the middle” of the frame. Since we only can see the box in 2D, we expect the clipping to also happen in 2D.This is however not the case. The right clipping is because the triangle is rotated, and goes through the back wall, making the clipping appear out of nowhere. If we had moved the back vertex towards us by 0.2, then it would have been inside the clipping object, and no unnatural clipping would have occurred.

iii) The purpose of clipping is to avoid giving the the display unit “out of bound/range” values and therefore removing them from the viewing pane. We don’t want the screen to get values that it cant show.

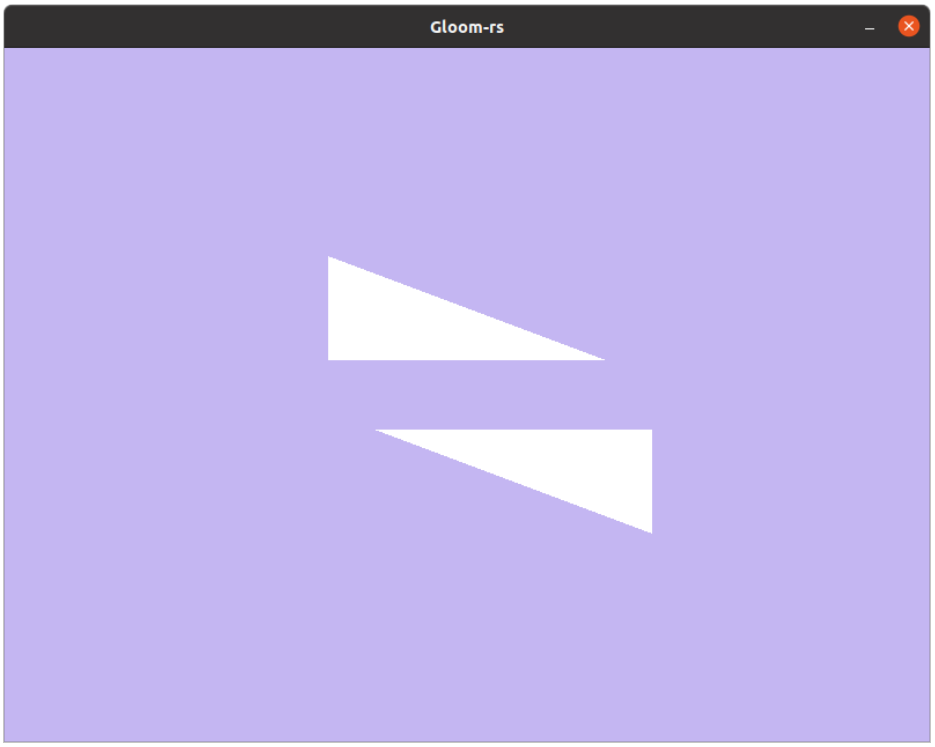

Figure 2: Task 2b) - Both triangles drawn counter-clockwise

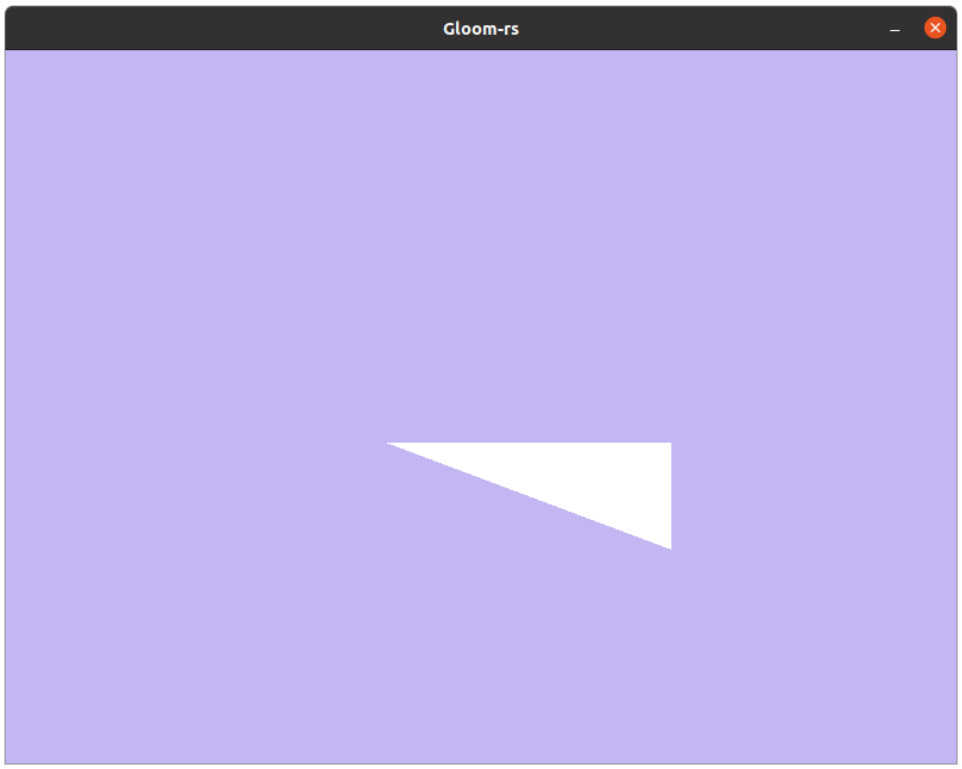

Figure 3: Task 2b) - Changed index order of upper triangle

b)
i) From the figure 2 to figure 3 the index order was changed from [0, 1, 2, 3, 5, 4] to [0, 2, 1, 3, 5, 4], making the upper triangle being drawn in a clockwise direction, instead of counter-clockwise.

This change of drawing order “removed” the upper triangle form the picture, while keeping the lower.(You may notice that I found out what it did after designing the vertex-cooridnates, since the lower triangle is drawn 3, 5, 4, and not 3, 4, 5 as I designed the picture.)

ii) iii)This happens because of back-face culling. This part in the pipe-line checks if a polygon is drawn clockwise or counter-clockwise, and then compares it to the what the user has set as front-facing. In my case, front-facing was counter-clockwise and therefore I had to set the indices to follow that rotation. More general, culling is a part in the pipeline that decides whether a polygon should be drawn or if it should be invisible; due to being outside the field of view(frustum culling), occluded by other objects(occlusion culling) or (in our case) they are occluded by front-facing primitives(back-face occlusion).

c)

i) The depth buffer needs to be reset each frame so the next frame doesn’t compare new depth values to old lowest value. Or in other words, OpenGL checks if the the depth of the object is lager or smaller than the previous closest depth depth value to determine if it should be drawn or is occlude. If it the new value is lager(further back) then it will not draw the new object. If it is lower(closer to the camera) then it will drawn the object. Now we can see that if the depth buffer is not cleared, then we might get compare new objects depth to totally wrong values.

ii) The Fragment Shader can be executed several times if the there exists multiple fragments that are at the same place in x-y-coordinates, but different value in z-coordinate. This is because the Fragment Shader does not check for depth, so it will always run the shader for a fragment that can can be a fragment that is shown in that pixel. Only after running the Fragment Shader will it discard fragments that are occluded, unless early depth-testing is enabled. So if there are 3 fragments at the same x-y-coordinates, but three different z-values, then then the Fragment Shader will run three times for that pixel.

iii) The two most common shaders are Vertex shader and Fragment(pixel) shader.
Vertex shader perform a number of operations the vertices themselves. These operations can include transforming the vertex to a screen position, generating texture coordinates and lighting the vertex to determine color. This is done before the Fragment shader.

Fragment shader comes after the rasterization, where a primitve broken into pixel-sized fragments for each pixel the primitve covers. The Fragment shader will process each fragment, and give it a depth value and color value(s). x-y-coordinates can not be changed by the Fragment shader. After that, it will pass the fragment on in the pipeline.

iv) With a index buffer you can reuse a vertex for different triangles, saving memory and making the process more efficient due to more use of cached vertices.

v) A non-zero pointer could be used when there is several types of data in the buffer. So if you want to get access to the second type of data in the buffer and that type of data start at the 3rd byte, then you could give a pointer to the 3rd byte, ignoring the first two bytes.

d)

Figure 4: Task 2 d) i) Pre mirror

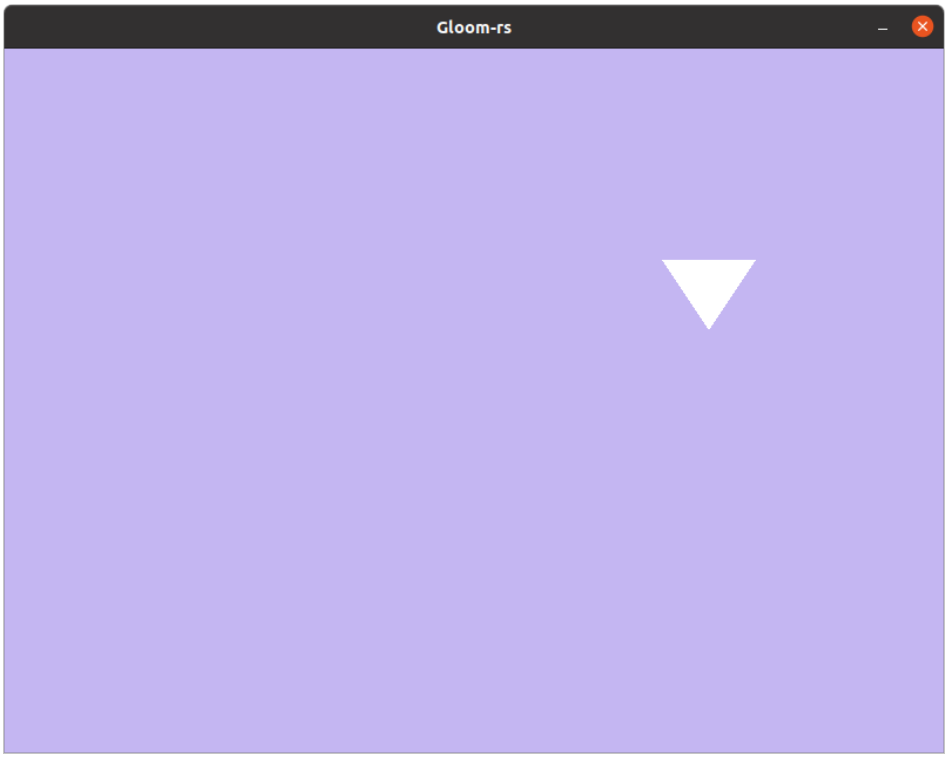

Figure 5: Task 2 d) i) Post mirror

i) To mirror the the whole image, I made inverted all x and y coordinates by adding a minus sign in front of position in the shader:

```
void main()
{
    gl_Position = vec4(-position, 1.0f);
}
```

ii)

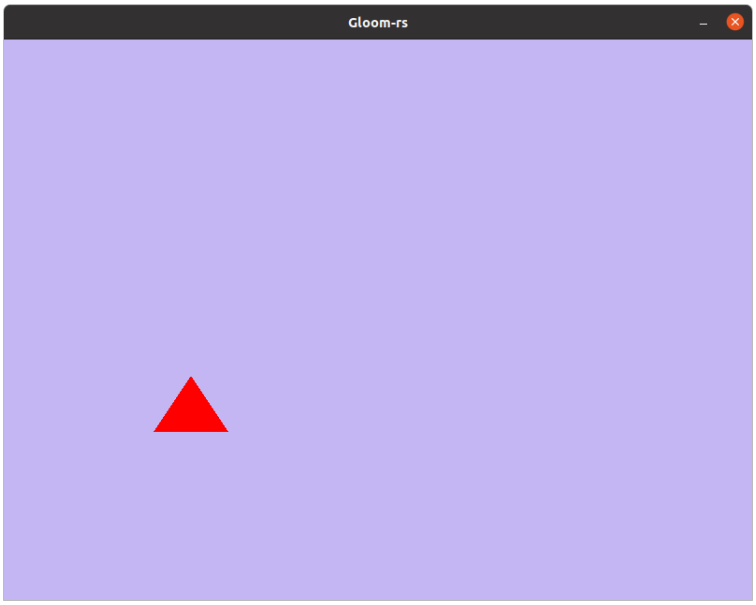

Figure 6: Task 2 d) ii) Changed color of the triangle

Pre change of color is in figure 4. In figure 6 the color of the triangle is changed to red. This was done by altering the RGBA values in the Fragment shader to

color = vec4(1.0f, 0.0f, 0.0f, 1.0f);

The RGBA values has to be between 0.0 and 1.0.

Task 3

My circle with 1000 line segments: