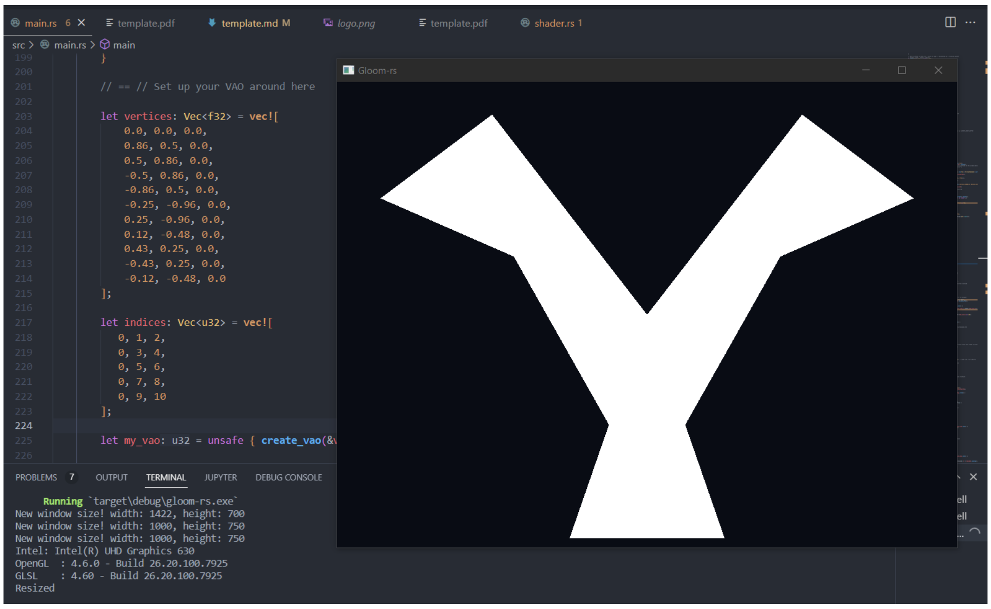
TDT4195 Exercise 1

Pablo Díaz Viñambres

September 8, 2022

1 Task 1: Basic triangle composition

For task 1, exercise c), we arranged a composition of 5 triangles roughly making an ‘Y’ shape. In order to reduce memory consumption, We reused the central vertex at (0, 0, 0) for all the triangles using the Vertex Index Buffer. The coordinates and indexing of the vertices are also included in the screenshot.



```rust
    }

    // == // Set up your VAO around here

    let vertices: Vec<f32> = vec![
        0.0, 0.0, 0.0,
        0.86, 0.5, 0.0,
        0.5, 0.86, 0.0,
        -0.5, 0.86, 0.0,
        -0.86, 0.5, 0.0,
        -0.25, -0.96, 0.0,
        0.25, -0.96, 0.0,
        0.12, -0.48, 0.0,
        0.43, 0.25, 0.0,
        -0.43, 0.25, 0.0,
        -0.12, -0.48, 0.0
    ];

    let indices: Vec<u32> = vec![
       0, 1, 2,
       0, 3, 4,
       0, 5, 6,
       0, 7, 8,
       0, 9, 10
    ];

    let my_vao: u32 = unsafe { create_vao(&v
```

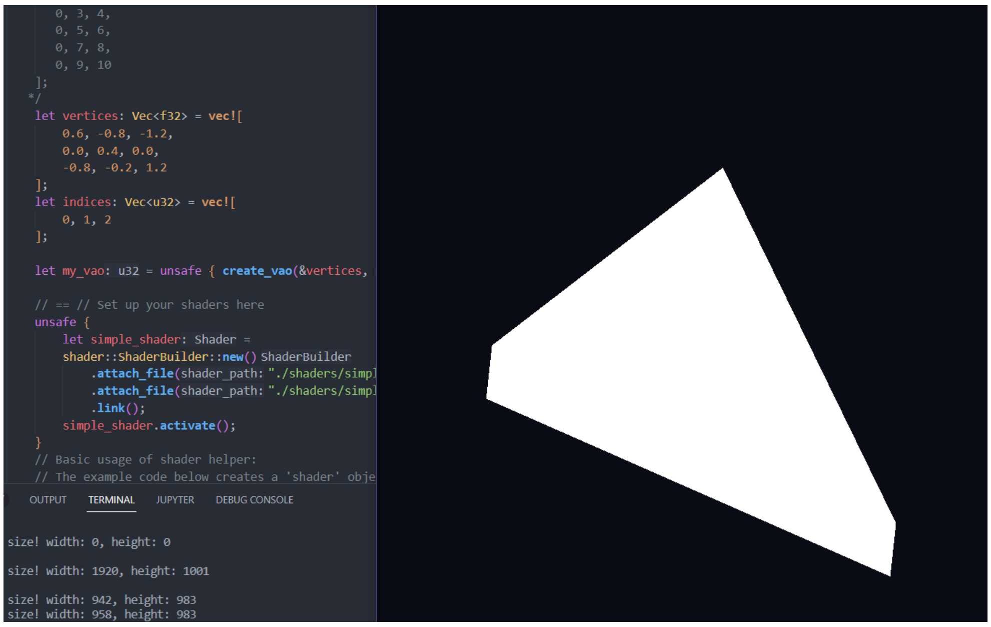
```
     Running `target\debug\gloom-rs.exe`
New window size! width: 1422, height: 700
New window size! width: 1000, height: 750
New window size! width: 1000, height: 750
Intel: Intel(R) UHD Graphics 630
OpenGL  : 4.6.0 - Build 26.20.100.7925
GLSL    : 4.60 - Build 26.20.100.7925
Resized
```

2 Task 2: Geometry and theory

2.1 Exercise a)

When we draw the triangle we get a pentagon shape instead of a triangle. This happens because of camera clipping. We can recall that in OpenGL, the camera faces the negative Z axis by default. In this case, one of the vertices of the triangle has a positive Z value (+1.2), making it being placed behind the camera. Because of this, OpenGL clips the part of the triangle that ends outside of the camera field of view.

The purpose of this mechanism is to reduce the amount of pixels that need to be drawn. Since pixels behind the camera won't have valid device coordinates and won't be visible, we save the effort of drawing them by clipping them beforehand.



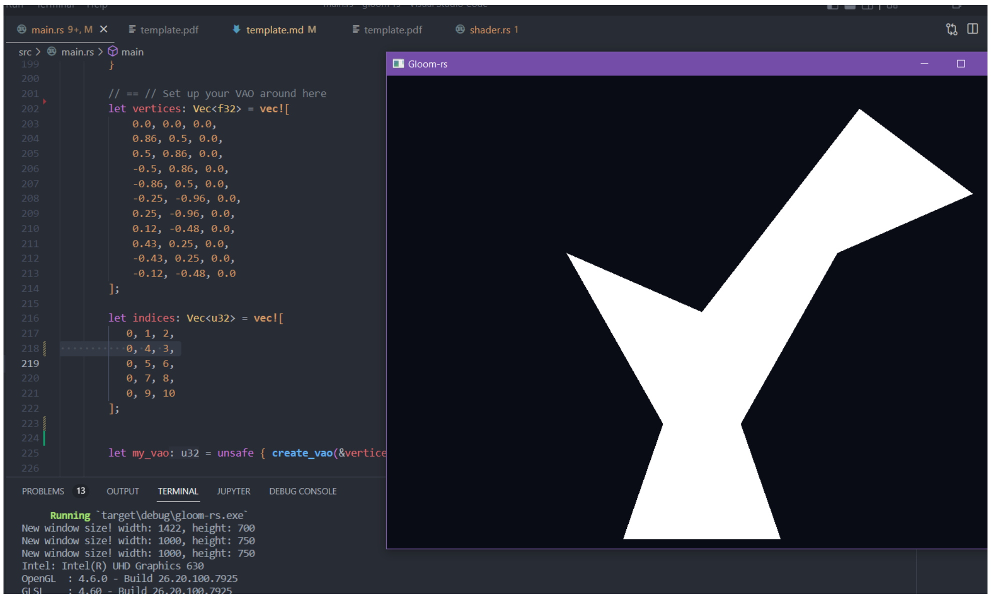
```
        0, 3, 4,
        0, 5, 6,
        0, 7, 8,
        0, 9, 10
    ];
   */
    let vertices: Vec<f32> = vec![
        0.6, -0.8, -1.2,
        0.0, 0.4, 0.0,
        -0.8, -0.2, 1.2
    ];
    let indices: Vec<u32> = vec![
        0, 1, 2
    ];

    let my_vao: u32 = unsafe { create_vao(&vertices,

    // == // Set up your shaders here
    unsafe {
        let simple_shader: Shader =
        shader::ShaderBuilder::new() ShaderBuilder
            .attach_file(shader_path: "./shaders/simpl
            .attach_file(shader_path: "./shaders/simpl
            .link();
        simple_shader.activate();
    }
    // Basic usage of shader helper:
    // The example code below creates a 'shader' obje
```

OUTPUT TERMINAL JUPYTER DEBUG CONSOLE

```
size! width: 0, height: 0

size! width: 1920, height: 1001

size! width: 942, height: 983
size! width: 958, height: 983
```

2.2 Exercise b)

Let's start with the 5 triangle figure shown in Task 1. If we, for instance, change the order vertex of the second triangle from 0 -> 3 -> 4 to 0 -> 4 -> 3 the triangle disappears. What happened is that we changed the "winding direction" of the triangle. By default, triangles with vertices specified in counter-clockwise order are interpreted by OpenGL as front-facing. By changing the vertex order to clockwise, the triangle will be back-facing. Therefore, because of the mechanism known as backface calling, the triangle will not be drawn. This mechanism is used to reduce the number of polygons that need to be drawn, since backfacing polygons are usually behind the front part of a solid model and hence not visible.



```
        }

        // == // Set up your VAO around here
        let vertices: Vec<f32> = vec![
            0.0, 0.0, 0.0,
            0.86, 0.5, 0.0,
            0.5, 0.86, 0.0,
            -0.5, 0.86, 0.0,
            -0.86, 0.5, 0.0,
            -0.25, -0.96, 0.0,
            0.25, -0.96, 0.0,
            0.12, -0.48, 0.0,
            0.43, 0.25, 0.0,
            -0.43, 0.25, 0.0,
            -0.12, -0.48, 0.0
        ];

        let indices: Vec<u32> = vec![
           0, 1, 2,
           0, 4, 3,
           0, 5, 6,
           0, 7, 8,
           0, 9, 10
        ];


        let my_vao: u32 = unsafe { create_vao(&vertice
```

```
     Running `target\debug\gloom-rs.exe`
New window size! width: 1422, height: 700
New window size! width: 1000, height: 750
New window size! width: 1000, height: 750
Intel: Intel(R) UHD Graphics 630
OpenGL  : 4.6.0 - Build 26.20.100.7925
GLSL    : 4.60 - Build 26.20.100.7925
```

2.3 Exercise c)

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

The depth buffer is used to find which objects are in front of others. If an object A on Frame X is in front of object B on pixel P, the Z-buffer algorithm will only draw A on that pixel (assuming A is not transparent). Now let's say that object A disappears on frame X+1. If we didn't reset the depth buffer beforehand, object B still won't be drawn on pixel P, since its z-value on that pixel is still less than the old z-value from object A in frame X.

In which situation can the Fragment Shader be executed multiple times for the same pixel?

Since the fragment shader is executed for every fragment, when multiple different fragments share a pixel, it will various times on that pixel. This implies that, in complex scenes, we may render more fragments than pixels on the screen.

What are the two most commonly used types of Shaders? What are the responsibilities of each of them?

They are the vertex shader and the fragment shader. The vertex shader runs once for every drawn vertex and is responsible for transforming them (translating, rotating, scaling, flipping. . .) and projecting them into the camera. The fragment shader runs once for every fragment and determines its color and depth value.

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)?

Because most models are formed of primitives that share vertices. If there wasn't an index buffer pointing this out, all shared vertices would be repeated in the vertex data buffer, making it memory inefficient.

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 last input is an offset value, changing it to a non-zero value o would indicate that the attribute we are looking for in the vertex buffer starts at byte o. Per example, if the attribute we are loading is RGB color values and the format for the data of a vertex in the buffer is:

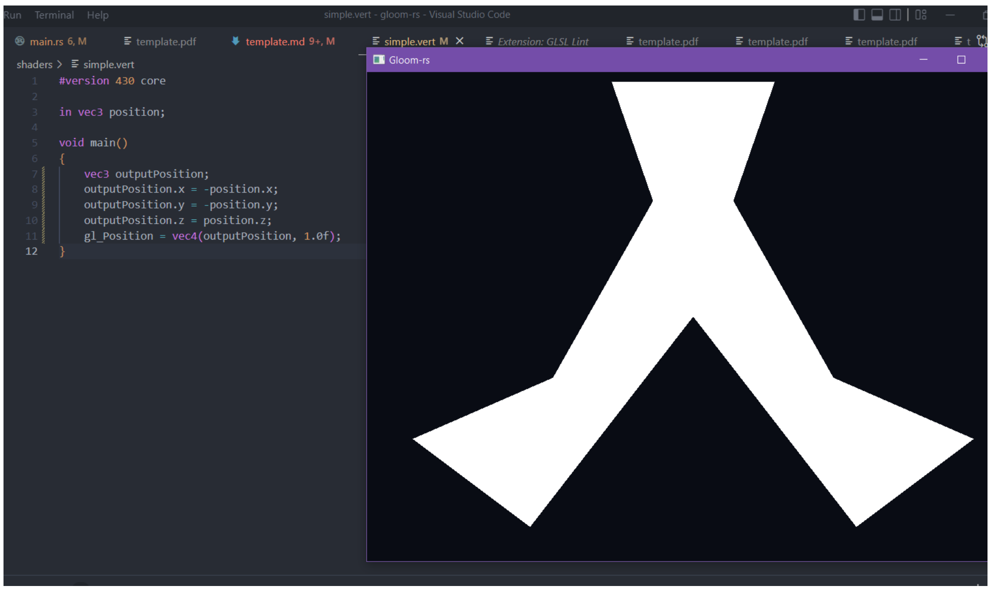
x	y	z	r	g	b
4 bytes float	4 bytes float	4 bytes float	1 byte uint	1 byte uint	1 byte uint

Then the offset should be 12, since we need to skip the first 12 floats that contain position information, additionally, the stride would need to be 15, in order to get to the next vertex RGB values.

2.4 Exercise d)

2.4.1 Mirroring

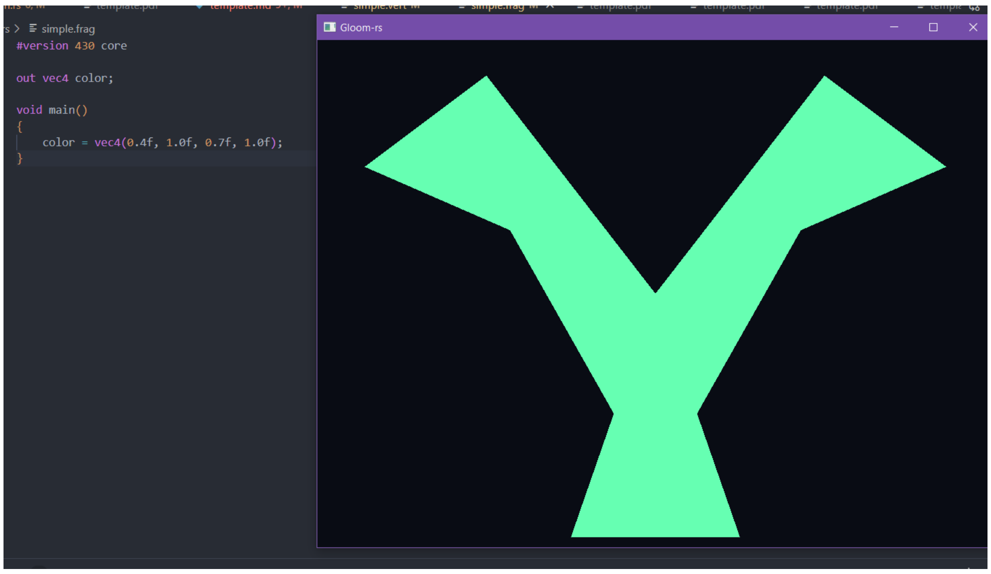
To achieve this effect, we changed the vertex shader output position. To do this, we defined a new variable `vec3 outputPosition` equal to the input position, but with flipped x and y coordinates. We then passed it to the `gl_Position` output.



```
#version 430 core

in vec3 position;

void main()
{
    vec3 outputPosition;
    outputPosition.x = -position.x;
    outputPosition.y = -position.y;
    outputPosition.z = position.z;
    gl_Position = vec4(outputPosition, 1.0f);
}
```

2.4.2 Color change

To achieve this effect, we changed the fragment shader output color to the greenish tone from RGB values `(0.4, 1.0, 0.7)`.



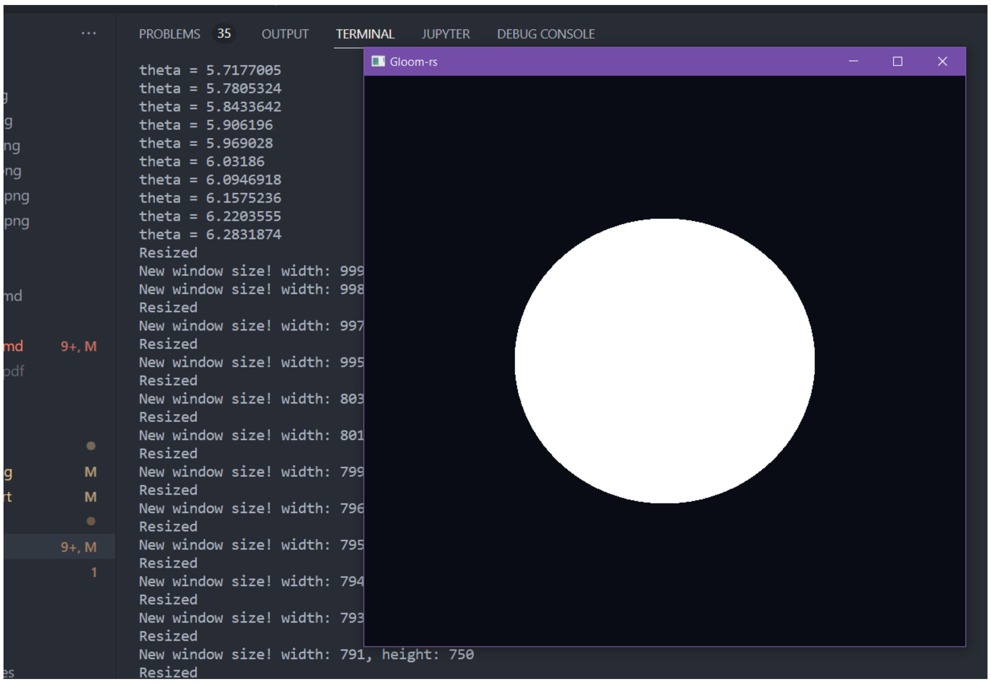
```
#version 430 core

out vec4 color;

void main()
{
    color = vec4(0.4f, 1.0f, 0.7f, 1.0f);
}
```

3 Bonus challenges

We implemented two of the given bonus challenges, drawing a circle shape and making its color change over time.

3.1 Circle shape

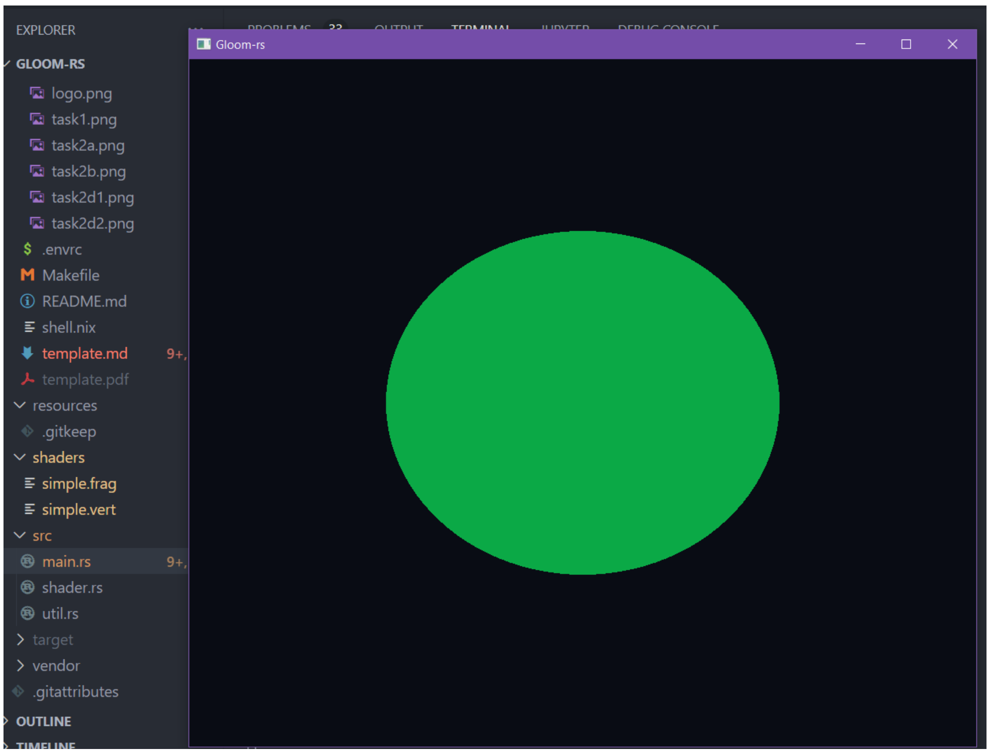
For this implementation, we created a new function `create_circle_vao()` that, given some 2D center coordinates, a radius and a number of triangles, creates a VAO that we can use to draw a circle. To accomplish this, we assemble the triangles in a regular polyhedron shape sharing the center vertex until the circle is complete. The result is a smooth circle when `triangle_count` is high enough.



```
theta = 5.7177005
theta = 5.7805324
theta = 5.8433642
theta = 5.906196
theta = 5.969028
theta = 6.03186
theta = 6.0946918
theta = 6.1575236
theta = 6.2203555
theta = 6.2831874
Resized
New window size! width: 999
New window size! width: 998
Resized
New window size! width: 997
Resized
New window size! width: 995
Resized
New window size! width: 803
Resized
New window size! width: 801
Resized
New window size! width: 799
Resized
New window size! width: 796
Resized
New window size! width: 795
Resized
New window size! width: 794
Resized
New window size! width: 793
Resized
New window size! width: 791, height: 750
Resized
```

3.2 Color change

To implement color change, we need to change the Fragment Shader color output dynamically for each frame drawn. To accomplish this, we passed a uniform value through the pipeline, by first passing it to the Vertex Shader and then to the Fragment Shader. For the color change itself, we coded a simple randomizer that modifies the RGB values each frame. The result is not the most beautiful, but it illustrates the concept well.


EXPLORER
GLOOM-RS
logo.png
task1.png
task2a.png
task2b.png
task2d1.png
task2d2.png
.envrc
Makefile
README.md
shell.nix
template.md 9+,
template.pdf
resources
.gitkeep
shaders
simple.frag
simple.vert
src
main.rs 9+,
shader.rs
util.rs
target
vendor
.gitattributes
OUTLINE
TIMELINE
PROBLEMS 33
OUTPUT
TERMINAL
JUPYTER
DEBUG CONSOLE
Gloom-rs