Assignment 2

Visual Computing Fundamentals

Pablo Díaz Viñambres : *pablodi@ntnu.no*

September 22, 2022

Contents

1 Task 1: Pre-Vertex Colors

On this first task, we modified the VAO to add colors in the scene via a new color buffer that asigns RGBA values to the vertices. We also added some utility functions `add_x()` that append geometric data to the vertex and index buffers to simplify figure drawing.

To complete the exercise, we drew 3 triangles with different colors for each of their vertices. We can observe that the their surface becomes a color gradient, caused by an interpolation of the vertices colors. By default, OpenGL applies barycentric interpolation to get those colors, but that behaviour can also be changed to achieve other effects. We also included the Spanish flag on the composition for fun!

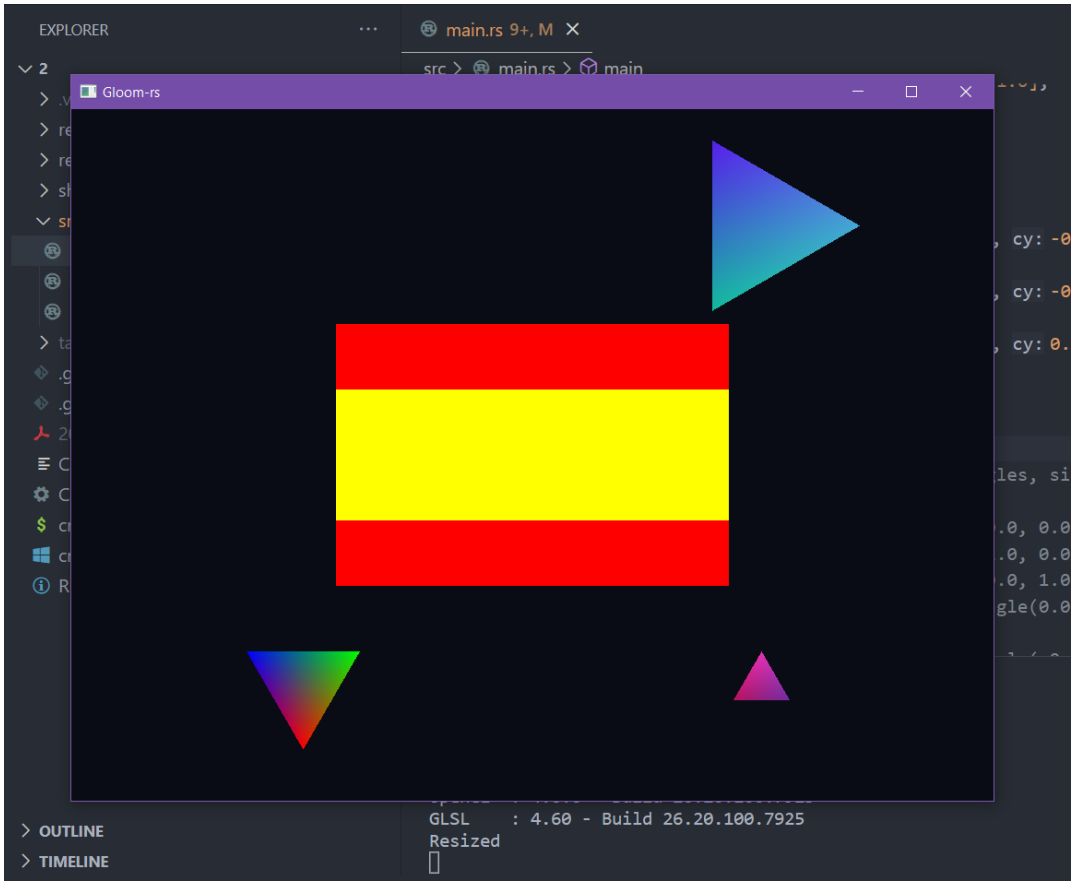

Figure 1: Task 1 composition.

2 Task 1: Alpha blending and Depth

For this task, we changed the composition and drew 3 triangles, one red, one green and one blue. We positioned them around the center with slight X and Y offsets and different Z coordinates, rotations and sizes. We also lowered their alpha-values to 0.5, making them transparent.

After enabling alpha blending and drawing them from furthest to nearest, we got the following image.

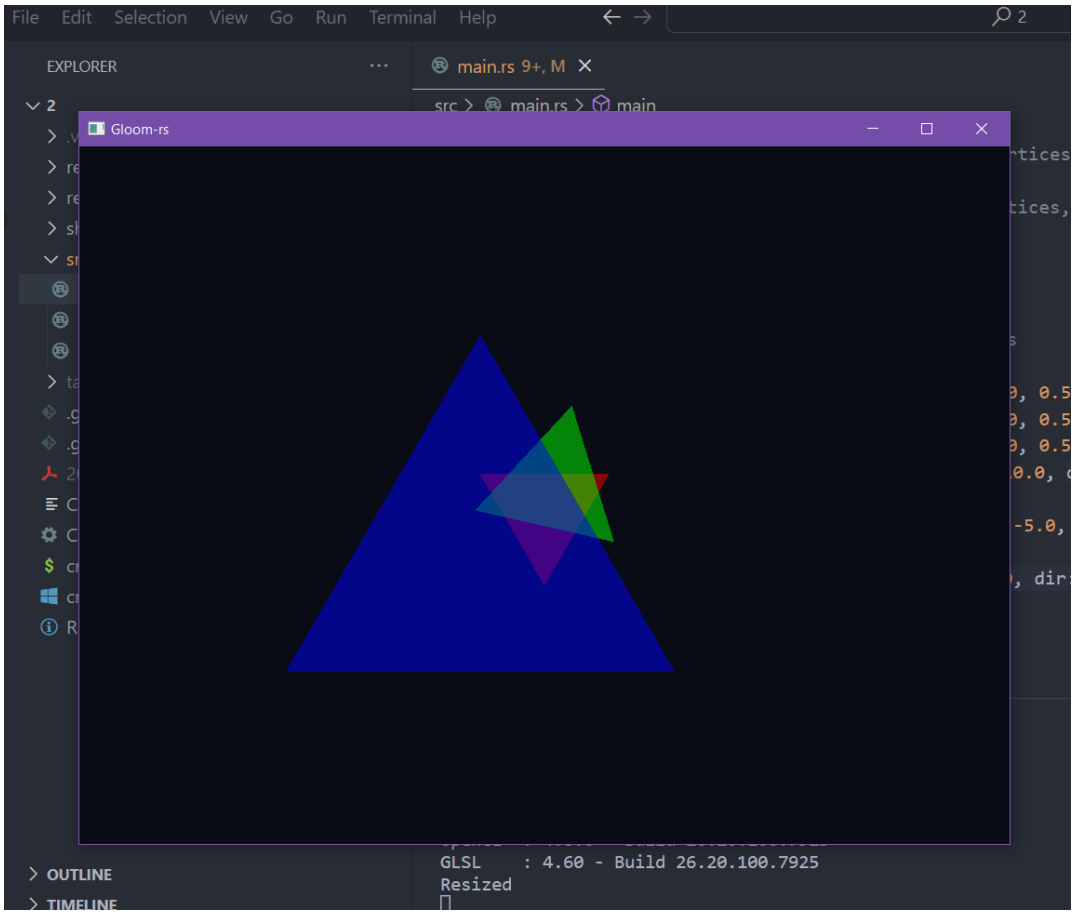

Figure 2: Task 2 composition.

1. Swap the colors of different triangles by modifying the VBO containing the color Vertex Attribute. Observe the effect of these changes on the blended color of the area in which all triangles overlap. What effects on the blended color did you observe, and how did exchanging triangle colors cause these changes to occur?

   Swapping the green and blue triangles, we get a different result:

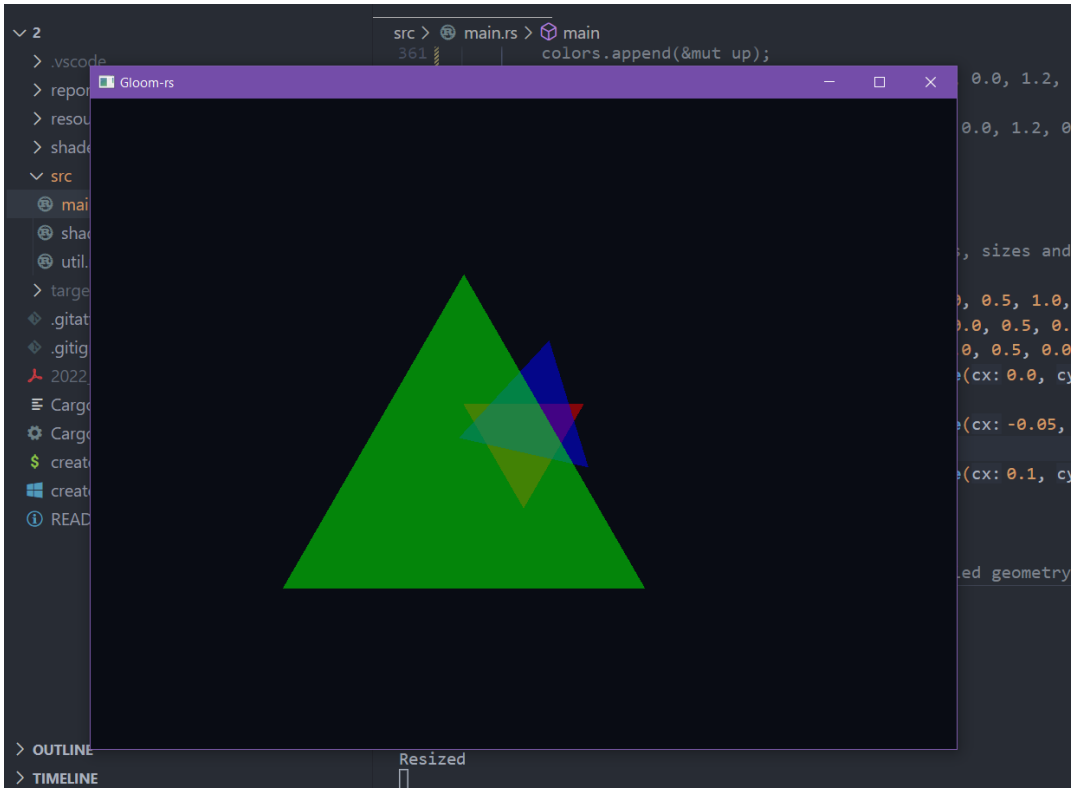

Figure 3: Task 2 composition with swapped colors.

As we can see, the blended color between the blue and green triangles goes from being mostly blue to being mostly green. This happens because of how the alpha blending formula we are using works.

Since $\alpha = 0.5$ and we are using the formula $\alpha_{\text{image}} = \alpha_a + \alpha_b(1 - \alpha_a)$ for a the primitive at front and b the primitive at back, then $\alpha_a = 0.5$ and $\alpha_b = 0.5^2 = 0.25$, making the blue triangle much less noticable, since the color of the blue triangle is multiplied by its alpha value.

Additionally, we get $\alpha_r = 0.5^3 = 0.125$, making it really hard to notice the difference between the green-blue area and the area with all the three colors. Because of this, the color we perceive at the intersection of the tree triangles is $(0.125, 0.5, 0.25)$.

2. Swap the depth (z-coordinate) at which different triangles are drawn by modifying the VBO containing the triangle vertices. Again observe the effect of these changes on the blended color of the overlapping area. Which changes in the blended color did you observe, and how did the exchanging of z-coordinates cause these changes to occur? Why was the depth buffer the cause this effect?

   For the next image, we changed the drawing order from furthest to nearest to nearest to furthest, we got the following image:

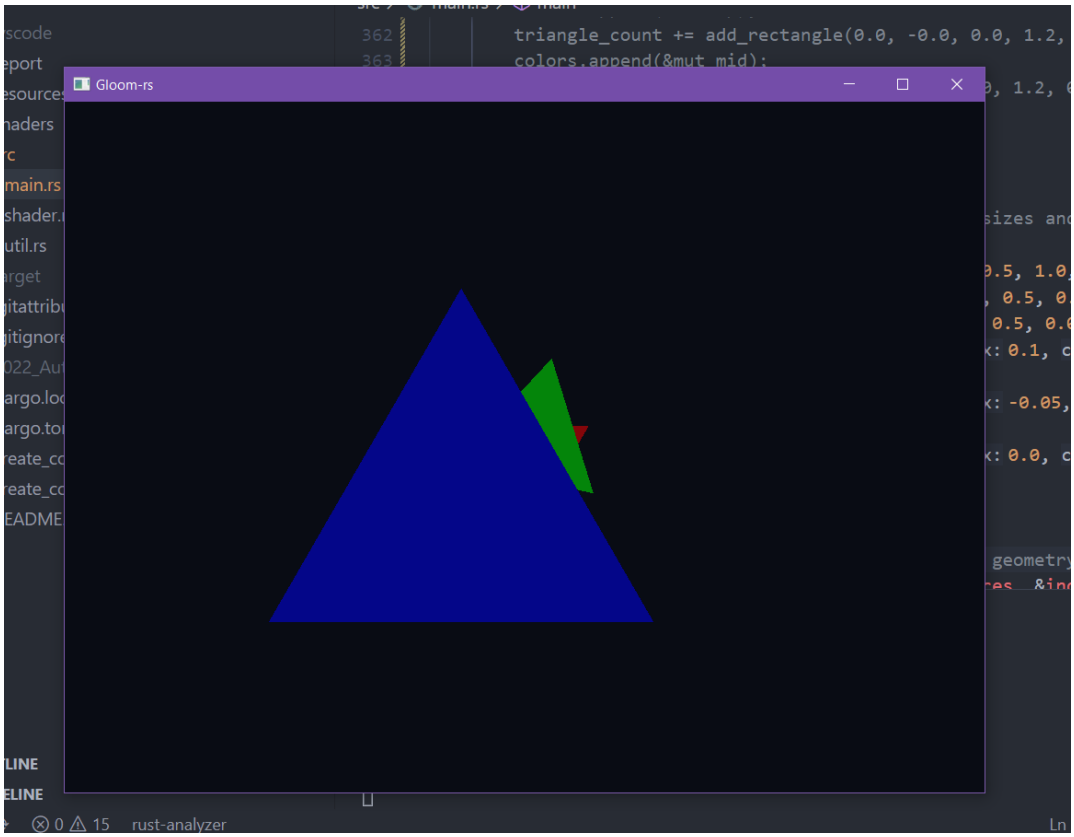

Figure 4: Task 2 composition with swapped drawing order.

As we can see, the transparency effect disappeared completely. This happens because the Z-Buffer algorithm receives the blue triangle first, and updates the minimum depth of all the pixels occupied by it. When the other triangles arrive, they can only be drawn at the positions not already occupied by another, nearer one, since their Z value is bigger than the one stored at the Z-Buffer.

This situation can be expanded to every scene that contains transparent objects. To solve this we need to sort all the transparent fragments based on their depth value, aiming at drawing them in the correct furthest to nearest order. Unfortunately, sorting has a complexity of $\mathcal{O}(n \log n)$, which could affect perfomance in complex scenes.

3 Task 3: The Affine Transformation Matrix

In this task, we first modified the vertex shader to take an input 4x4 matrix and multiply the output coordinates by it.

1. Individually modify each of the values marked with letters in the matrix in equation 2 below, one at a time. In each case use the identity matrix as a starting point. Observe the effect of modifying the value on the resulting rendered image. Deduce which of the four distinct transformation types discussed in the lectures and the book modifying the value corresponds to. Also write down the direction (axis) the transformation applies to.

   Passing the matrix

   $$\begin{bmatrix} a & b & 0 & c \\ d & e & 0 & f \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

   gives the following transformations with:

   - $a \neq 1$: Scaling in the X axis
   - $b \neq 0$: Shearing in the X axis
   - $c \neq 0$: Translation in the X direction
   - $d \neq 0$: Shearing in the Y axis
   - $e \neq 1$: Scaling in the Y axis
   - $f \neq 0$: Translation in the Y direction

2. Why can you be certain that none of the transformations observed were rotations?

   Rotations around a major axis have a matrix containing two cosines and two sines. From this follows that a rotation can not be achieved without changing multiple values at the same time.

4 Tasks 4 and 5: Combinations of Transformations

The camera described on task 4 was implemented by multiplying different matrices to achieve the final transformation that projected the scene correctly and allowed for camera movement and rotation. To achieve it, we multiplied:

- `camera_transl`: The camera translation matrix, to account for different camera positions.
- `camera_rot`: The camera rotation matrix, to account for different camera yaw (Y axis rotation) and pitch (X axis rotation).
- `transl_persp`: A translation matrix to correct the coordinate shift after the projection inversion of Z coordinates.
- `persp`: The perspective matrix we configured with FOV = 100.0 and near and far clipping planes at 1.0 and 100.0, respectively.

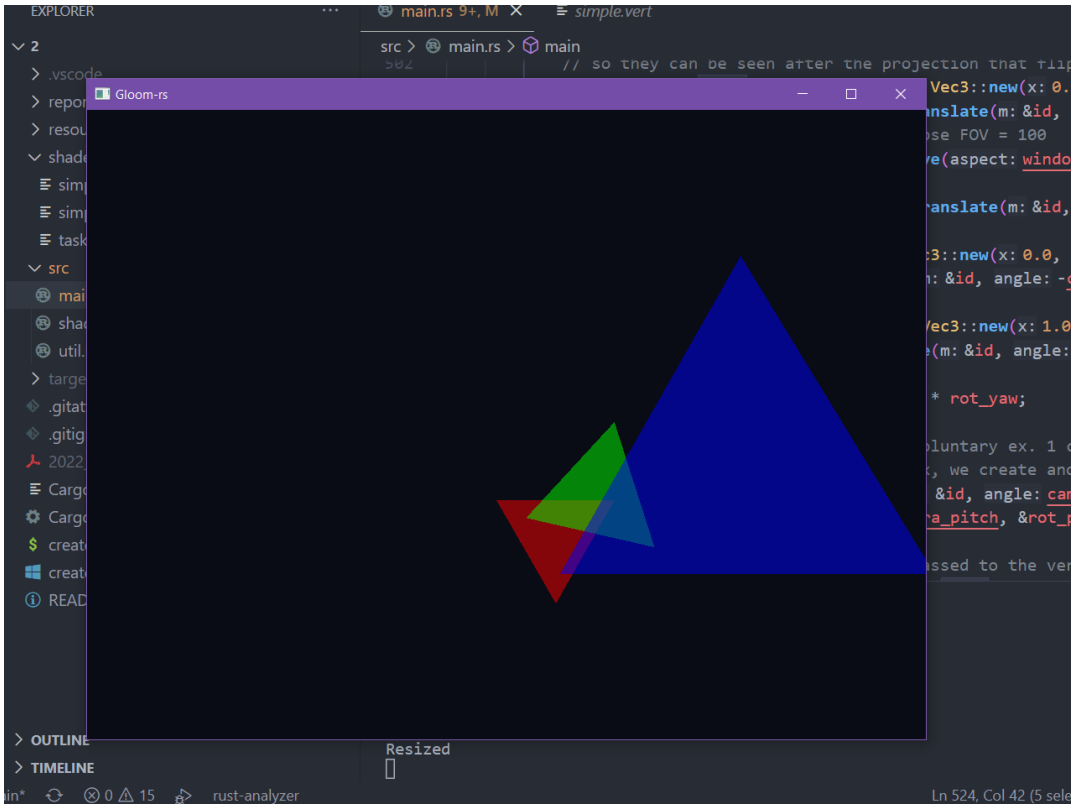

Figure 5: Task 2 composition from a different camera angle.

Additionally, the first exercise on task 5 was completed, making the camera more intuitive. To do this, we first saved the inverse of the rotation matrix (the rotation matrix with negative angles) for the yaw and pitch angles of the last frame. After moving the camera on one of the three axis, we multiplied the correspondent δ vector, (e.g. $\delta_{Y^+} = \delta_{SPACE} = (0, \delta, 0)$) by that inverse rotation matrix, to get the transformed movement in the correct direction.