NTNU

Assignment 2

Grunnleggende Visuell Databehandling

Aditi Ravi Deshpande

October 3, 2025

Contents

1 Task 1: Per-Vertex Colors

1.1 a)

I had already done the necessary changes for the shader files in the first assignment, but I added another VBO for the colour vector in the create_vao() function.

1.2 b)

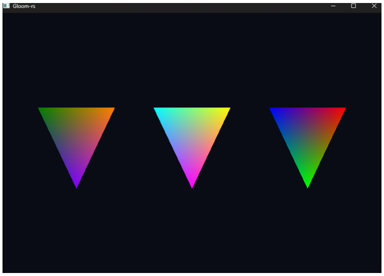


Figure 1: Three triangles with different colours on each vertex

The rasterizer fills each triangle and interpolates the colour values for every fragment using interpolation. The fragment shader receives and writes that interpolated colour so that we can see a smooth gradient across each triangle. This interpolation means that even though we only specify one colour at each corner, every pixel inside the triangle is shaded with a colour that is a weighted mix of its three vertices based on how close it is to the vertex. As a result, the triangles don't have a flat red/green/blue colour per vertex, but instead smoothly blend from one corner's colour to the others across the surface. See Figure 1.

2 Task 2

2.1 a)

I added a new float variable in simple.frag called alpha to introduce transparency. Each triangles got a constant value for z, and the same colour for each vertex. The alpha value is added for transparency, and I implemented 3D depth using different Z-coordinates (0.3,0.0,-0.3). I drew the triangles in back-to-front order for correct alpha blending and used three separate draw calls for each triangle to control rendering order. See Figure 2 for the results.

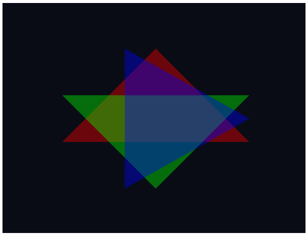


Figure 2: Depth and transparency

2.2 b)

i. The order was red, green, blue, and I swapped it to blue,red and green the results are seen in Figure 3. We can instantly see that the colour intensity of all the triangles have changed. The blue triangle that was at the front in Figure 2 is more transparent than the one in Figure 3.

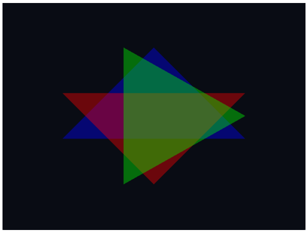


Figure 3: Swapped colours

The colour of the overlap region shifts toward the colour of the front-most triangle, and the apparent intensity of each colour changes. Swapping the per-vertex colours reassigns which geometric layer carries each RGB without altering the depth(alpha), so the change in the blended colour is purely due to order-dependent drawing of the triangles, caused by the standard alpha blending (`SRC_ALPHA`, `ONE_MINUS_ALPHA`).

ii. I changed the depth size z of the red and blue triangles. The result is seen in Figure 4. The red triangle does not have transparency. The transparency with the green and blue triangle is there. The depth buffer causes this error as the red triangle is drawn first. This is because with depth testing (`GL_LESS`) and *depth writes enabled*, the first triangle that is nearer at a pixel stores its depth in the Z-buffer. Since the red triangle is drawn first and is closer, it writes a smaller z.

Later blue fragments at the same pixels have $z_{\text{blue}} > z_{\text{red}}$ and fail the depth test, so the blend equation is never executed there so red looks opaque. Only regions not covered by red can blend normally. To get the intended transparency, we need to render back-to-front with `glDepthMask(GL_FALSE)` or re-sort draw calls.

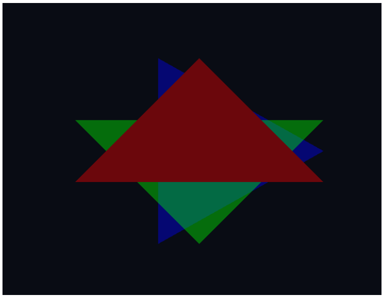
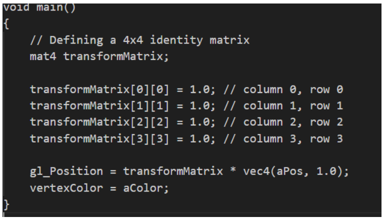


Figure 4: Changing depth size to triangles

3 Task 3: The Affine Transformation Matrix

3.1 a)



```
void main()
{
    // Defining a 4x4 identity matrix
    mat4 transformMatrix;

    transformMatrix[0][0] = 1.0; // column 0, row 0
    transformMatrix[1][1] = 1.0; // column 1, row 1
    transformMatrix[2][2] = 1.0; // column 2, row 2
    transformMatrix[3][3] = 1.0; // column 3, row 3

    gl_Position = transformMatrix * vec4(aPos, 1.0);
    vertexColor = aColor;
}
```

Figure 5: Identity matrix

3.2 b)

A true 2D rotation has the form

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix},$$

In our experiment we changed *one* entry at a time in $\begin{bmatrix} a & b & 0 & c \\ d & e & 0 & f \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, which cannot satisfy the required coupling $a = e = \cos\theta$ and

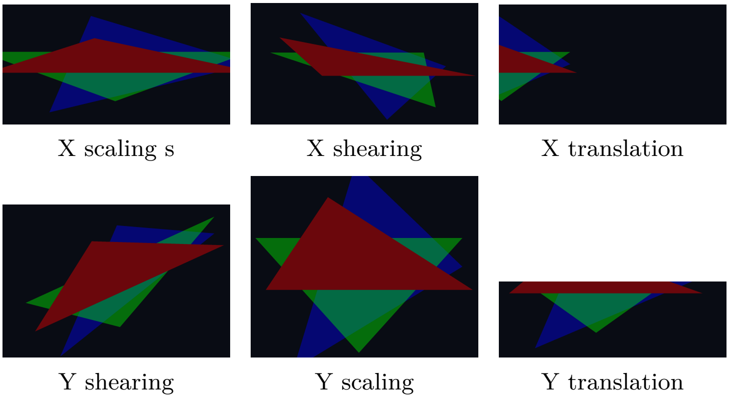
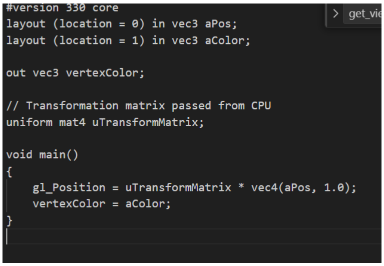


Figure 6: Overview of results arranged in a 2×3 grid.

$b = -d = \sin\theta$. Single-entry changes therefore produced translation (c, f), non-uniform scaling (a or e), or shear (b or d), not rotation. Rotations also preserve lengths and angles; our results showed significant size/shape changes.

4 Task 4 :Combinations of Transformations

4.1 a)



```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

out vec3 vertexColor;

// Transformation matrix passed from CPU
uniform mat4 uTransformMatrix;

void main()
{
    gl_Position = uTransformMatrix * vec4(aPos, 1.0);
    vertexColor = aColor;
}
```

Figure 7: Transformation matrix as a uniform variable

4.2 b)

We pre-compute a single 4×4 matrix per object,

$$M = P \cdot T$$

and send it as a uniform to the vertex shader. T translates the triangle along -z (into the screen). P = perspective(aspect, fovy, near, far) is applied last.

Push an object farther (more negative z) $\Rightarrow$ divide by a bigger number $\Rightarrow$ it looks smaller. That is why the red triangle looks huge, the green looks smaller, and the blue looks tiny, ignoring FOV/aspect constants.

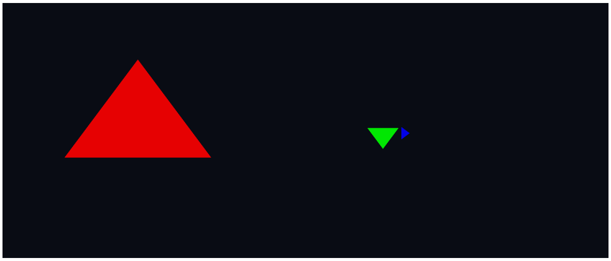
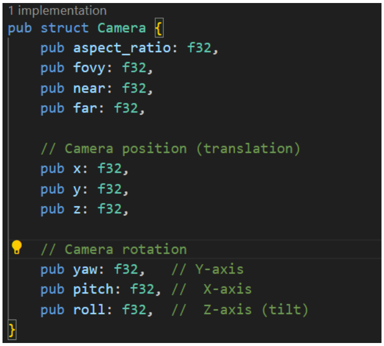


Figure 8: Perspective

4.3 c)

(a) A camera struct is created, and an instance is also created. Refer to source code and figure 9.


```
1 implementation
pub struct Camera {
    pub aspect_ratio: f32,
    pub fovy: f32,
    pub near: f32,
    pub far: f32,

    // Camera position (translation)
    pub x: f32,
    pub y: f32,
    pub z: f32,

    // Camera rotation
    pub yaw: f32,    // Y-axis
    pub pitch: f32, //  X-axis
    pub roll: f32,  //  Z-axis (tilt)
}
```

Figure 9: camera variable

(b) The purpose of the input handler is to manage all keyboard input for camera movement, implement a tripod-style camera system with 6 degrees of freedom and translate keyboard events into camera movements.

Control Scheme

Translation

- **W/S**: Forward/Backward movement
- **A/D**: Left/Right strafing
- **Space/LShift**: Up/Down movement

Rotation

- **Arrow Keys**: Yaw (left/right) and Pitch (up/down) rotation
- **Q/E**: Roll rotation (camera tilt)

Technical Details

- **Frame-rate independent**: All movements are scaled by `delta_time`.
- **Configurable speeds**: Easy to adjust movement and rotation speeds.
- **Continuous input**: Handles multiple keys pressed simultaneously.
- **Clean separation**: Input logic is separated from camera logic.

(c) Each frame I rebuild the camera transform from scratch:

  (a) Start with the identity matrix.

  (b) Apply the camera rotations (pitch, yaw, optional roll) as basic rotations.

  (c) Apply the camera position as a simple translation.

  (d) Build the final matrix as `MVP = Projection * View * Model` (projection last).

Each motion is a single basic affine transform (rotation or translation). Order matters: changing the order changes what you see.

(d)
- **One matrix per motion.** Each movement builds one basic matrix:
  - Move camera: a translation.
  - Turn camera (yaw/pitch[/roll]): a rotation.
  - Per object: a model transform (here, just a translation).
- **Build the view by doing the opposite to the world.**
  - Camera moves right $\Rightarrow$ world translates left.
  - Camera yaws right $\Rightarrow$ world rotates left around Y.
- **Order matters.** I multiply in this order (projection last):

$$\text{MVP} = \text{Projection} \times \text{View} \times \text{Model}.$$

What I do each frame

(a) Start from identity matrices.

(b) Make the **View** from camera state by applying opposite transforms: first translation by position, then rotations ($-$yaw about Y, $-$pitch about X)

(c) Make the **Model** per object (a simple translation in my scene).

(d) Make **Projection** from aspect, fov, near, far.

(e) Combine: `MVP = Projection * View * Model` and send it as the uniform to the vertex shader.

We apply one matrix per motion and multiply them so that translation happens before rotation (projection last):

$$(E \cdot D \cdot C \cdot B \cdot A)\, v,$$

where

- A: model transform (e.g. `translate(0,0,-2)`), optional,
- B: **translate** the world by $-$camera_position,
- C: **Rotation - yaw** by $-$camera_yaw (around Y) ,
- D: **Rotation - pitch** by $-$camera_pitch (around X),
- E: **perspective**(aspect, fovy, near, far).

Correction made in camera.rs, in get_view_matrix.