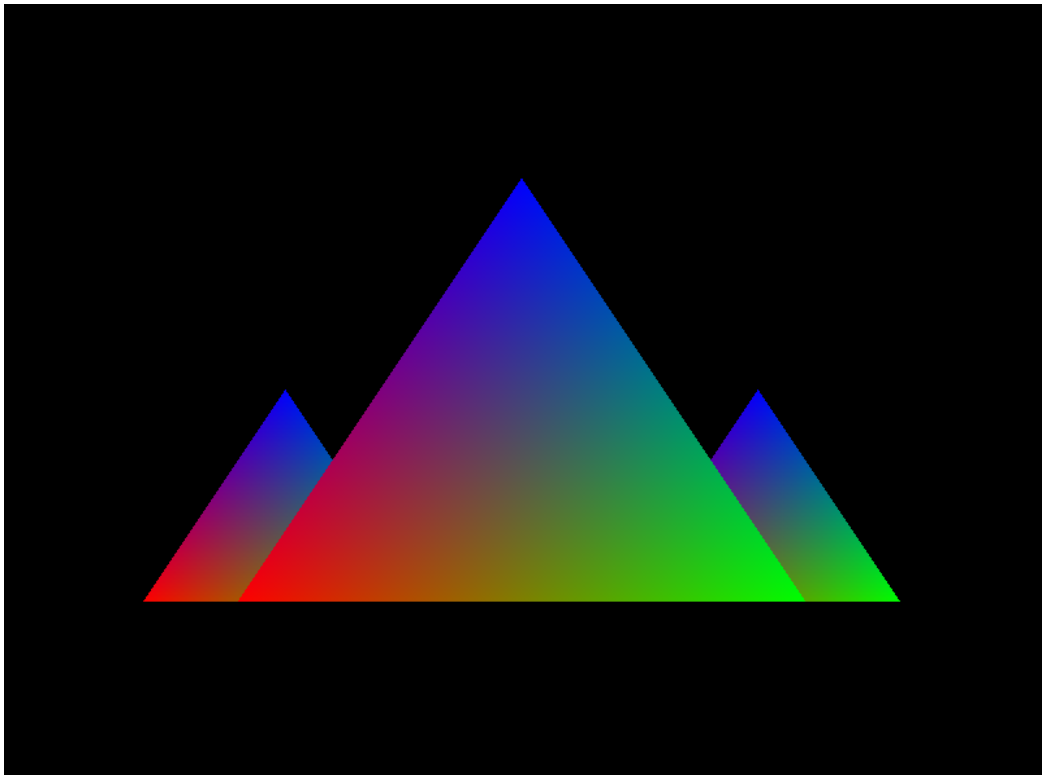
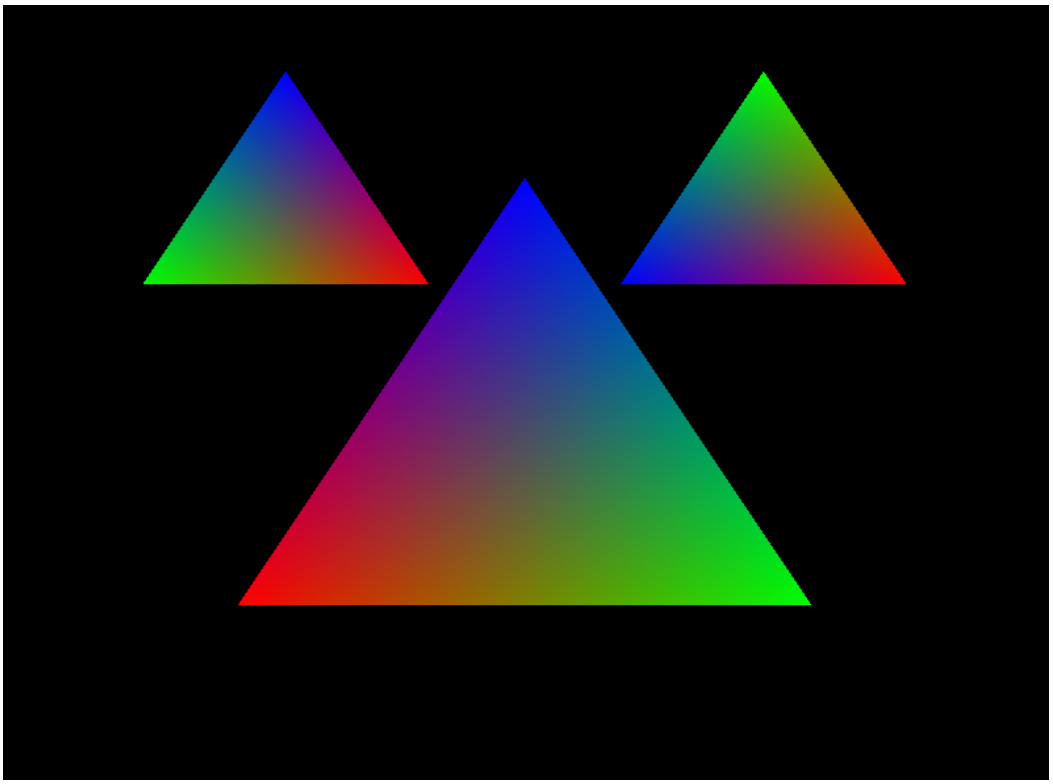
Computer Graphics - Assignment 2 - Erik Galler

Task 1: Repetition [0.5 points]

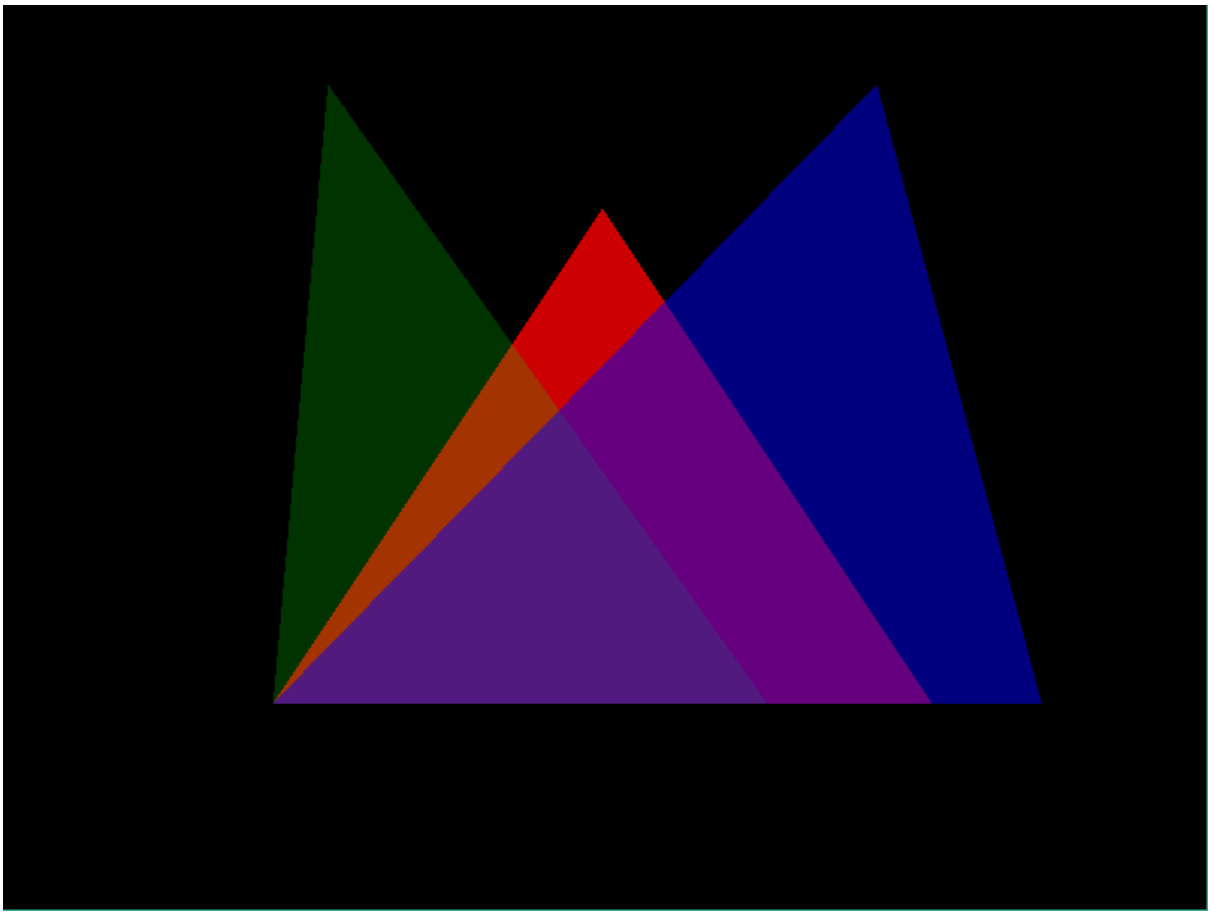
Render a scene containing at least 3 different triangles, where each vertex of each triangle has a different color. Put a screenshot of the result in your report. All triangles should be visible on the screenshot.





Task 2: Alpha Blending and Depth [0.5 points]

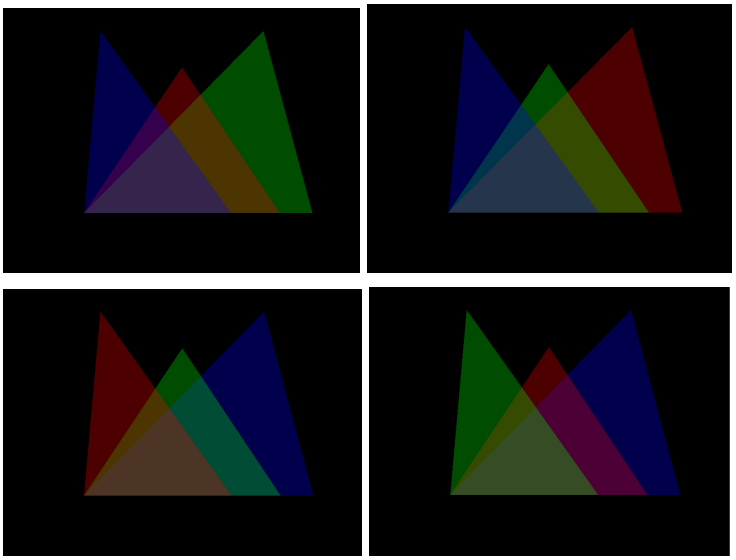
a) **[0.2 points] [report] For this task, we want to show you what happens to a blended color when multiple triangles overlap from the camera's perspective (where the triangles are positioned at different distances from the camera). To this end, draw at least 3 triangles which satisfy the following conditions:**
• There exists a section on the xy-plane on which all triangles overlap.
• For any single triangle, the three vertices of that triangle have the same z-coordinate.
• No two triangles share the same z-coordinate.
• All triangles have a transparent color (alpha < 1).
• All triangles have a different color.
• Each triangle's vertices have the same color.
I do not recommend drawing the exact same triangle 3 times at different depths; it will make the next question more difficult to solve. Remember to turn on alpha blending, as described in the previous task. Put a screenshot in your report.



b) **[0.3 points] [report] First make sure your triangles are being drawn back to front. That is, the triangle furthest away from the screen is drawn first, the second-furthest one next, and so on. You can change the draw order of triangles by modifying the index buffer. Remember the coordinate space of the Clipbox here.**

i) **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?**

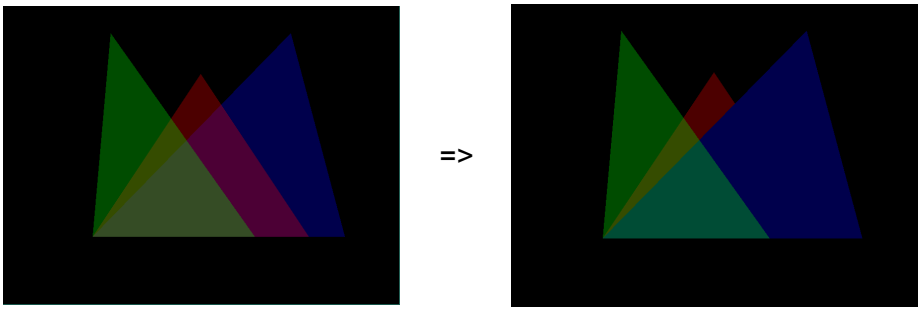
Left triangle (x, y, z = -1) => Rendered last in the front
Middle triangle (x, y, z = 0) => Rendered in middle
Right triangle (x, y, z = 1) => Rendered first in the back



We can observe that the color of the area where all triangles overlap changes based on the order for colors rendered. For example the overlapping area is more blue dominated when blue is the last color being rendered (i.e the left triangle), and vice versa for when it is red. This is because of the way new colors are defined by the OpenGL function gl::BlendFunc(gl::SRC_ALPHA, gl::ONE_MINUS_SRC_ALPHA) where $Color_{New} = Color_{Source} \cdot Alpha_{Source} + Color_{Destination} \cdot (1 - Alpha_{Source})$. This equation tells us that the new color is $(Alpha_{Source} \cdot 100)\%$ consisting of the overlapping color and the rest is from the already existing color(s). Therefore making the left triangle more dominant, because the other colors are reduced multiple times.

ii) **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?**

Swapping the first triangle's z-coordinate (blue) with the middle triangle's:



Even though blue is transparent, red seems to be totally blocked by it. This is because of the depth test. First the blue triangle gets rendered (because of the index buffer), and the depth coordinates get saved in the depth buffer (which is initially empty). Next the red triangle gets rendered, and the depth test checks if it is "behind" anything, which it partially is (red: z = 1.0, blue: z = 0.0), and only the pixels which are not behind the blue triangle gets saved in the depth buffer. Pixels that are not saved in the depth buffer are overlooked. So because the blue triangle is calculated first, and the red triangle is actually behind it, the

$$Color_{Destination} \cdot (1 - Alpha_{Source})$$

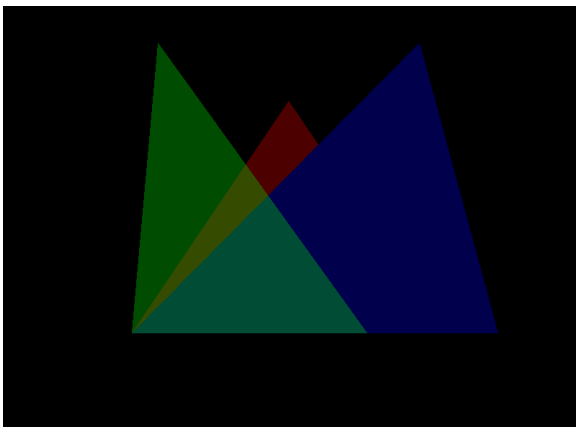
part of the equation does not reference the red triangle, and blue is not blended with red.

Task 3: The Affine Transformation Matrix [0.7 points]

a) **[0.2 points] Modify your Vertex Shader so that each vertex is multiplied by a 4x4 matrix. The elements of this matrix should also be defined in the Vertex Shader. Change the individual elements of the matrix so that it becomes an identity matrix. Render the scene from Task 1b to ensure it has not changed. Note that matrices in GLSL are column major. As such, individual elements in a matrix can be addressed using:**

matrixVariable[column][row] = value;
It is also possible to assign a vec4 to a column, to set all 4 values at once:
matrixVariable[column] = vec4(a, b, c, d);
Refer to the guide for more information.

```
#version 430 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec4 color;

out VS_OUTPUT {
    vec4 color;
} OUT;


uniform mat4 mirror;

void main()
{
    float a = 1.0;
    float b = 0.0;
    float c = 0.0;
    float d = 0.0;
    float e = 1.0;
    float f = 0.0;

    mat4 identity_matrix = mat4(
        a, b, 0.0, c,
        d, e, 0.0, f,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0
    );
    gl_Position =  identity_matrix * vec4(position, 1.1f);
    OUT.color = color;
}
```



b) **[0.4 points] [report] 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.**

a: This variable scales the figure in the x-axis where $X_{new} = scale * X_{old}$.
b: This variable shears the figure in the y-axis where $X_{new} = X_{old} + scale * Y_{old}$.
c: This variable translates the figure in the x-axis where $X_{new} = X_{old} + scale$.
d: This variable shears the figure in the x-axis where $Y_{new} = Y_{old} + scale * X_{old}$
e: This variable scales the figure in the y-axis where $Y_{new} = scale * Y_{old}$.
f: This variable translates the figure in the y-axis where $Y_{new} = Y_{old} + scale$.
Setup:

```
#version 430 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec4 color;

out VS_OUTPUT {
    vec4 color;
} OUT;


uniform mat4 mirror;
uniform float scale;

void main()
{
    float a = 1.0;
    float b = 0.0;
    float c = 0.0;
    float d = 0.0;
    float e = 1.0;
    float f = 0.0;

    mat4 identity_matrix = mat4(
        a, b, 0.0, c,
        d, e, 0.0, f,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0
    );
    gl_Position =  identity_matrix * vec4(position, 1.1f);
    OUT.color = color;
}
```

```
unsafe {
    let name = String::from("scale");
    let scale_location: gl::types::GLint = shaders.get_uniform_location(&name);
    gl::Uniform1f(scale_location, _elapsed.sin());
    gl::ClearColor(0.0, 0.0, 0.0, 1.0);
    gl::Clear(gl::COLOR_BUFFER_BIT | gl::DEPTH_BUFFER_BIT);

    // Issue the necessary commands to draw your scene here
    gl::Enable(gl::BLEND);
    gl::Disable(gl::CULL_FACE);
    gl::BlendFunc(gl::SRC_ALPHA, gl::ONE_MINUS_SRC_ALPHA);

    gl::BindVertexArray(vao); // Not really necessary because of 1 VAO
    gl::DrawElements(gl::TRIANGLES, 3*3, gl::UNSIGNED_INT, ptr::null());

}
```

c) [0.1 points] [report] Why can you be certain that none of the transformations observed were rotations?

Because you need to at least alter 2 individual parameters to rotate the figure, for example Z-axis rotation: $X_{new} = X_{old} * \cos\theta - Y_{old} * \sin\theta$.

Task 4: Combinations of Transformations [3.3 points]

Controller:

```
// Handle keyboard input
// Movement                                  Rotation
//$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
//$ Direction $   X   $   Y   $   Z   $  $ Direction $   Yaw     $  Pitch  $
//$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
//$ Forwards  $   A   $   E   $   W   $  $ Forwards  $   Right   $   Up    $
//$ Backwards $   D   $   Q   $   S   $  $ Backwards $   Left    $   Down  $
//$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

Calculation for each frame:

```
// Uniform scale based on time elapsed
let name = String::from("scale");
let scale_location: gl::types::GLint = shaders.get_uniform_location(&name);
gl::Uniform1f(scale_location, _elapsed.sin());

// Model
let model: glm::Mat4 = glm::translation(&glm::vec3(0.0, 0.0, -5.0));

// View
let xyz_movement: glm::Mat4 = glm::translation(&movement_coordinates);
//let scale_movement: glm::Mat4 = glm::scale(xyz_movement, );
let xy_rotation: glm::Mat4 =  glm::rotation(-rotation_coordinates[1], &glm::vec3(1.0, 0.0, 0.0))
* glm::rotation(rotation_coordinates[0], &glm::vec3(0.0, 1.0, 0.0));
let view:  glm::Mat4 = xy_rotation * xyz_movement;

// Projection
let projection: glm::Mat4 = glm::perspective(SCREEN_H as f32 / SCREEN_W as f32, 0.52, 1.0, 100.0);  // NB flips the z-axis

// Putting it all together
let mvp: glm::Mat4 =  projection * view * model;

// Sending uniform to vertex shader
let name = String::from("transformation");
let transformation_location: gl::types::GLint = shaders.get_uniform_location(&name);
gl::UniformMatrix4fv(transformation_location, 1, gl::FALSE, mvp.as_ptr());
```