Lab 2 - Hardware Rasterization

You may work in pairs on this assignment. To receive credit, demonstrate your completed lab to me by the end of my office hours (2-3pm in EPS358) on Feb 6th.

In this assignment, you'll be using OpenGL and GLSL to render a single triangle just like in lab 1. Copy the user input code you wrote in lab 1 into the provided program and modify it to behave similarly to lab 1. Instead of writing out an image file, you'll be rendering straight to the screen using OpenGL calls and GLSL shaders. I recommend breaking the assignment into the following parts.

Part 1 - Normalized Device Coordinates vs Window Coordinates

In lab 1, all the points were input as window coordinates with the origin in the top left with positive x going right and positive y going down. Window coordinates refer to the actual column and row for each pixel in the image or window. OpenGL uses normalized device coordinates which is always -1 to 1 from left to right in x and -1 to 1 from bottom to top in y regardless of the dimensions of the window. Convert between the two using these equations:

$$x_{nd} = -1 + x_w(\frac{2}{width}) \qquad y_{nd} = 1 - y_w(\frac{2}{height})$$

For part 1, implement a conversion function, w2nd, that converts from window coordinates to normalized device coordinates. Use the function to convert the coordinates of your triangle.

Part 2 - Vertex Attributes: Position & Color

In lab 1, you used barycentric coordinates to determine which pixels were inside a triangle and to blend between the three different vertex colors, all in software. In this assignment, you'll let OpenGL do that work on your graphics card (the GPU). The provided code defines a vertex shader with a position vertex attribute. You'll need to add another vertex attribute for color.

Part 2a - Vertex Shader Start by adding another attribute for color in the vertex shader. Copy the definition for position and change its name. Since you'll also need color in the fragment shader (unlike position), you'll need to add a color output from the vertex shader.

Part 2b - Fragment Shader Add color as an input to your fragment shader. Make sure the name matches the output from the vertex shader.

Part 2c - Putting it together Look through the code and find where the position data is being loaded onto your graphics card (GPU). Modify the code so it uses the transformed input coordinates. Make it so the color data is also uploaded and bound correctly to your shaders.

Things To Notice

Uneven scaling

Try resizing the window and see what happens. Most likely your triangle will scale unevenly in x and y as the relationship between the width and height of your window changes. The coordinates haven't changed, but the shape of the triangle is changing. What's happening?

Because we converted our input coordinates to normalized device coordinates, which are independent of window size, the triangle is scaled to fit the window. Is this a desired behavior?

The answer is it depends on the purpose of your program, but generally not. Usually, you'll want your geometry to scale uniformly in x and y regardless of how the width and height of your window are set. Imagine scaling the window of a game you're playing and having everything stretched - not ideal. Think about how you might overcome this problem. We'll go over ways to fix it in future labs.

Window coordinates

The equations we used above convert from window coordinates with the origin in the top left corner. We want it this way because we're mimicking the behavior of lab 1. Many image libraries use (0,0) as the top left pixel and positive x goes to the right and positive y goes *down*, just like in lab 1. Mouse or touch coordinates are also usually specified this way. OpenGL's definition of window coordinates is a little different with the origin in the bottom left with positive x going to the right and positive y going *up*. The function, glViewport, tells OpenGL how to map from normalized device coordinates to window coordinates (the ones with the origin in the bottom left). The documentation for glViewport provides equations similar to, but different than the ones above for this reason. The fragment shader variable gl_FragCoord returns the window coordinates of the current fragment and by default also uses the bottom left as the origin. See the documentation for gl_FragCoord for how to change that behavior.

Recommended Reading

OpenGL Programming Guide, 8th Edition - Chapter 1, pages 1-32