Lecture/Lab 20 - WebGL and GLSL - Lab

Overview

By the end of this lab, you will have prodded at some WebGL code a little bit, and written a tiny bit of your own. Hopefully along the way, you'll get a feel for how **attributes**, **uniforms**, **varyings**, and **buffers** interact with **shaders**.

Grab the skeleton code from the course website, linked from today's lecture in the schedule table.

Open up `glDemo.js` in an editor of your choice. In `glDemo.js`, there are four sections:

1. WebGL Mechanics Helper Functions
2. Source Code for Shaders
3. Triangle Code
4. GLDemo - main code

I'd like you to scroll past the first section as quickly as possible. Don't look! Okay you can look, but the stuff there is details that I'd rather you not focus on first. You will be writing code in **only** the second and third sections, which hides details by using the helpers from the first section.

Open `hellogl.html` in a browser of your choice. I've tested this in Firefox, but it should work in any modern browser. You should see a white canvas on a white background. Actually, you're looking at a white triangle on a white canvas on a white background! Your first task involving code changes will be to change the triangle color so it's visible on the white background.

Tasks

-1. I suggest opening the developer console of your browser of choice right now, so you will see any errors that come up in the console. In Firefox, you can access it via the menu at the top-right > Web Developer > Web Console. The A3 handout links to instructions for other browsers.

0. Starting at the `glDemo` function, read through the code in the `GLDEMO - main code` and `TRIANGLE CODE` sections in the following order:
    1. `glDemo` is called first to set things up; it in turn calls the `Triangle`'s constructor
    2. Read the `Triangle` constructor; don't dive into the helper functions it calls
    3. Once things are set up, the `glDemo.prototype.render` function is called to draw things to the screen. The main render function calls the `Triangle`'s `render` function.
    4. Read the `Triangle.prototype.render` function, which draws the geometry to the screen. Again, don't dive into the helpers yet.

1. In the fragment shader source code, set the fragment color to black instead of white so the triangle actually shows up on the canvas. Remember that the fourth color channel represents transparency, with 0 being transparent and 1 being opaque.
2. Now we'll add a `uniform` that allows us to transform the triangle's geometry.
    1. Uncomment the two lines indicated in `Triangle.prototype.render` to set the `matrix` javasript variable to the shaders as a `uniform` called `Matrix`.
    2. Add this line at the beginning of the vertex shader source code to declare the uniform:

    ```
    uniform mat4 Matrix;
    ```

    3. Change the main function so it sets the vertex's position (`gl_Position`) to the product of `Matrix` with the `Position` attribute. At this point you should be able to refresh the page and see the triangle enlarged by a factor of 1.5, with the right-most corner of the triangle cut off a bit.
    4. Adjust the matrix's entries in `glDemo.prototype.render` to shift the triangle to the left a bit so it fits in the canvas again.
3. Make the triangle multi-colored. We'll do this by (1) specifying a `Color` attribute at each vertex; (2) passing this into a `varying` parameter so it gets interpolated and passed into the fragment shader; and (3) setting the color in the fragment shader. Here are the steps you should take:
    1. Create a VBO for color data in the `Triangle` constructor function. Use the code above that creates the `positionVbo` above as a template.
    2. Modify the `Triangle`'s `render` function to bind the VBO you just created, then link the `Color` attribute to the VBO. Use the code directly above, which does this same thing for the `Position` attribute, as a template. Note that you need to bind the color VBO, but you don't need to re-bind the IBO.
    3. In the vertex shader source code, add a delaration for the `Color` attribute underneath the delcaration for the `Position` attribute.
    4. Also in the vertex shader source code (still outside the `main` method), add a declaration for a `varying` parameter called `vColor`:

    ```
    varying vec3 vColor;
    ```

    5. In the vertex shader's main function, set the `vColor` varying parameter to the value of the `Color` attribute.
    6. Declare `vColor` in the fragment shader, just as you did in the vertex shader. Although the name is the same, remember that the value given to the fragment shader has been *interpolated* from the nearby vertices so it smoothly varies across the triangle.
    7. Finally, assign the varying's value to `gl_FragColor`. Refresh the page and you should see a much more colorful triangle!

4. *If you have extra time:* The `glDemo` `render` function is actually called repeatedly. This means that if you change the values in `Matrix` over time, you can animate the triangle. Use javascript's Datetime.now() to make change the `matrix` over time to make the triangle change shape or pose over time. Can you also change the data in the buffers (e.g., the color VBO) over time?