

Com S 336

Fall 2020

Homework 1

Here are a few exercises to try out some of the fundamentals we've looked at so far:

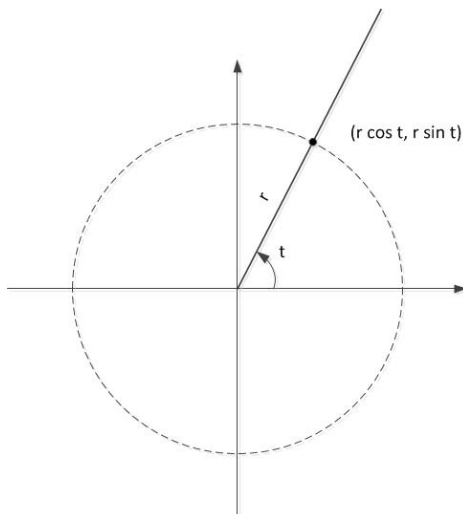
- get comfortable with JavaScript
- carefully read the sample code and know what each line of code is doing
- understand interpolation

None of them is very long in terms of lines of code (e.g. a few dozen lines at most), but you are certain to get stuck somewhere, so start early and talk to me or Jie as needed!

Please submit ONE zip archive on Canvas, containing the files indicated at the beginning of each problem. When working from an existing example, do not unnecessarily reformat or otherwise modify the code that does not need to be changed (so that we can diff your submission against the original).

1. (Please turn in two new files named *problem1.html* and *problem1.js*.)

Modify `GL_example1a_with_animation` so that instead of shifting the figure from side to side, it moves in a circle of radius 0.75 at a rate of one degree per frame. This is straightforward, you'll just need another uniform variable for the y shift. To calculate the position, use the sine and cosine functions as pictured below. Trig functions are available in JavaScript as `Math.cos()`, `Math.sin()`, and of course you might also need `Math.PI`. (Remember the JS trig functions expect radian measure! If you wish, you can use the handy function `toRadians` in `util/cs336util.js`.)



Provide an html text box to adjust the size of the square with a scale factor for the square (another uniform variable in the vertex shader). *Tip*: scale first, then shift the position.

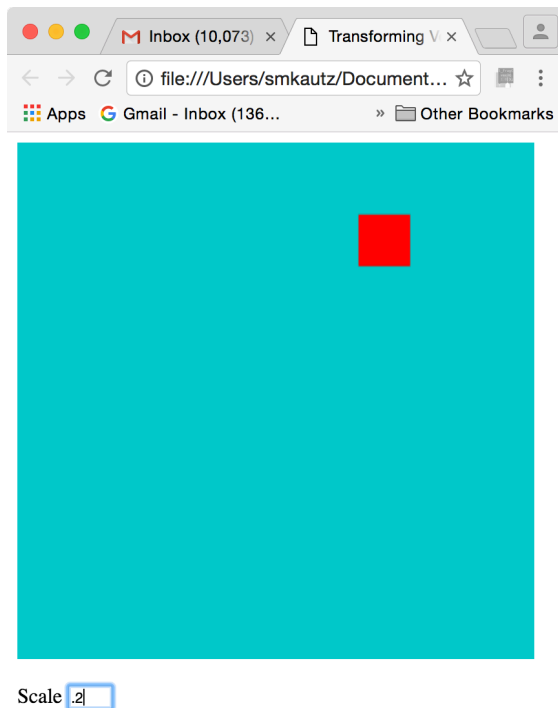
The html code for a text box looks like:

```
Scale: <input id="scaleBox" type="text" value="1.0" size=4/>
```

where `value` is the default text and `size` is the width. To get the value, use the built-in JS function `parseFloat`, e.g.,

```
var scale = parseFloat(document.getElementById("scaleBox").value);
```

See the sample code `examples/intro/text_box.html` for an example using a text box and attaching a handler.

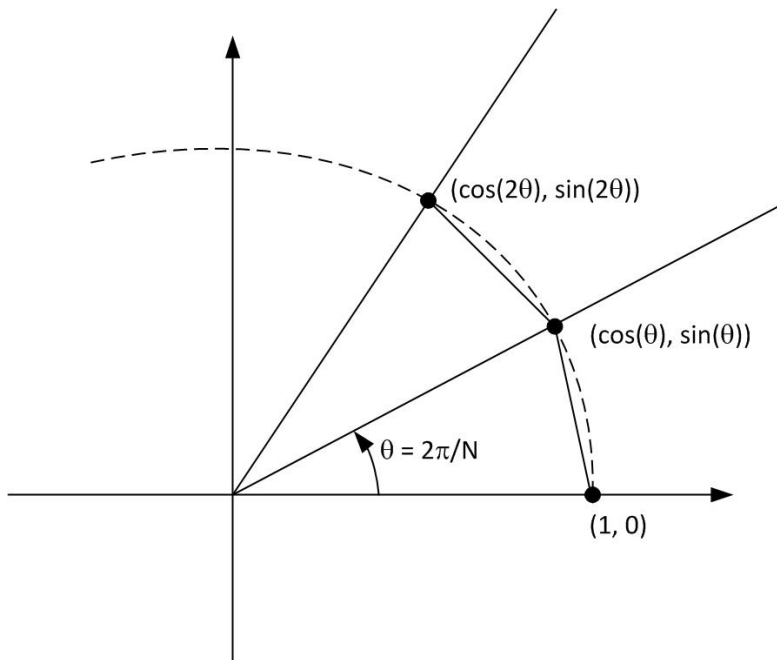


2. (Please turn in two new files named `problem2.html` and `problem2.js`.)

Modify `GL_Example1a` to draw a regular N-gon instead of a square. It is not hard to compute the coordinates in a loop using the sine and cosine functions as pictured below. If you have not used JS arrays before, note you can append elements using the `push()`

method (like a Java ArrayList). Trig functions are available in JS as `Math.cos()`, `Math.sin()`, and of course you might also need `Math.PI`. Scale the radius down to 0.8 so it's all visible. N should be selectable via an html text box.

You should load the new vertex data into the buffer only when a new value of N is selected, not every frame. See the sample code `examples/intro/text_box.html` for an example using a text box and attaching a handler.



Note: Remember that the argument to the constructor `Float32Array` or `Uint16Array` is a JavaScript array, not a sequence of numbers. That is, we can write

```
var myVertices = new Float32Array([1.0, 2.0, 3.0, 4.0]);
```

but NOT

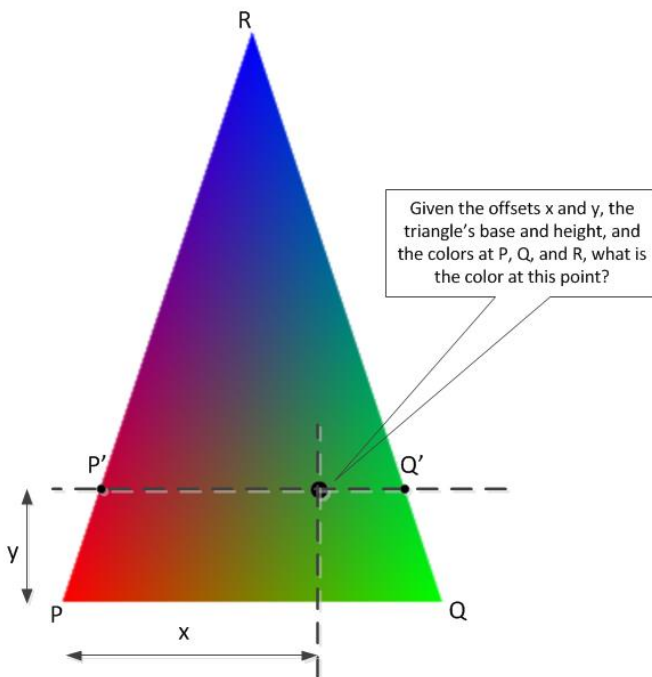
```
var myVertices = new Float32Array(1.0, 2.0, 3.0, 4.0);
```

3. (Please turn in your modified version of `color_interpolator.js`.)

Write a JS function that performs interpolation of colors associated with the corners of an isosceles triangle (this simulates what is done by the fragment shader in `GL_example2`).

See the figure below. That is, given base and height of the triangle, colors for the three corners, and an integer x and y offset within the triangle, find the interpolated values for red, green, and blue at the given location. (Don't worry about alpha here, assume it's always 1.0.) The exact signature for the function is in the file `color_interpolator.js` along with a definition for a struct representing an RGBA color. You can find this file in the `examples/intro` directory. For simplicity, you can assume that the base of the triangle is always aligned with the bottom of the canvas. Turn in your completed version of `color_interpolator.js`.

There are several ways you could do this. One is to use a Fahrenheit-to-Celsius conversion to find interpolated values at points P' and Q' along the sides, and then do it again to interpolate between P' and Q' . A slicker and more general solution would be to use *barycentric coordinates*, but this is optional. If you are interested, check the text by Angel; or, a quick Google search will turn up lots of references too, e.g. https://www.rose-hulman.edu/class/cs/esse351/m10/triangle_fill.pdf looks good.



4. (Please turn in the two new files `problem4.html` and `problem4.js`.)

We have seen in `GL_example1` how to use a uniform variable in a shader to shift a figure to the left or right, and we have seen in `GL_example2` how to use varying variables to

have colors associated with vertices interpolated across a triangle. Based on these examples, create files `problem4.js` and `problem4.html` as follows:

a) Draw a colored triangle, similar to the figure above, shifted to the left side of the canvas and draw a solid colored square shifted to the right side. You'll need a different shader for each figure. The one on the left can use the shader from `GL_example2_varying_variables` and the one on the right can use the shader from `GL_example1a_with_uniform_color` (you'll need to be able to set the color from your JS code). (Obviously the shaders will need different ids in the html file.)

b) Add a mouse handler that, when the mouse is clicked on the left triangle, the handler will *use your function from problem 4* to calculate what color will be at that pixel, and the application then draws the right-hand square that color. (A rudimentary color-picker!) You can assume that the triangle coordinates and shift amount are fixed, but you must allow for the fact that the canvas can be resized. That is, if the canvas is 400 x 400 pixels and the triangle's lower left corner is at $(-0.5, -0.75)$, then (linear interpolation!) that location has window (i.e. canvas) coordinates $(100, 75)$. But if the canvas width is 600 with height 400 then the same point would have window coordinates $(150, 75)$. In your JS code you can always get the current size of the canvas from its attributes `canvas.width` and `canvas.height` (where `canvas` is your JS variable referring to the html canvas element).

See `examples/intro/mouse.html` for how to get the mouse coordinates. After you get the mouse coordinates don't forget that the y-value is upside-down so you have to subtract from the canvas height to flip it. (For the mouse, $(0, 0)$ is the upper left corner of the canvas, but for the function from problem 3, $(0, 0)$ is the lower left corner, as it would be in the framebuffer.)