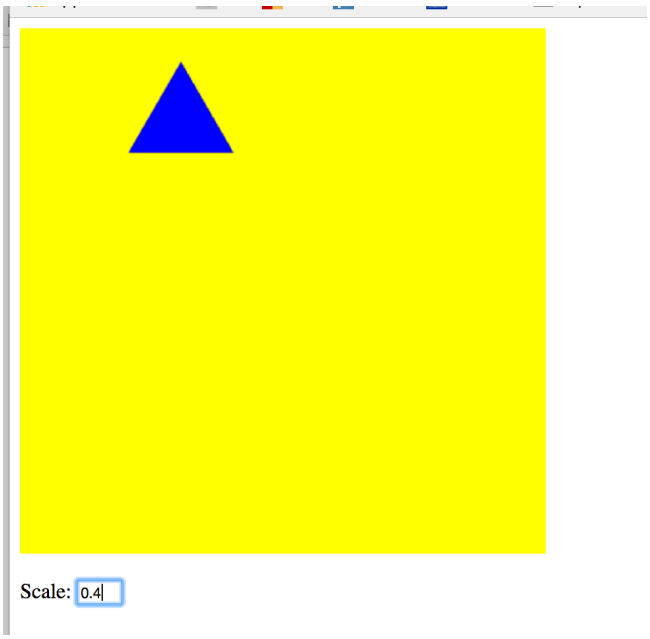
Com S 336
Fall 2017
Homework 1

Here are a few exercises to try out some of the fundamentals we've looked at so far. None of them is very long in terms of lines of code (e.g. a few of dozen lines at most), but you are bound to get stuck somewhere, so start early and talk to me as needed!

Please submit an archive on Blackboard containing the files indicated at the end of each problem.

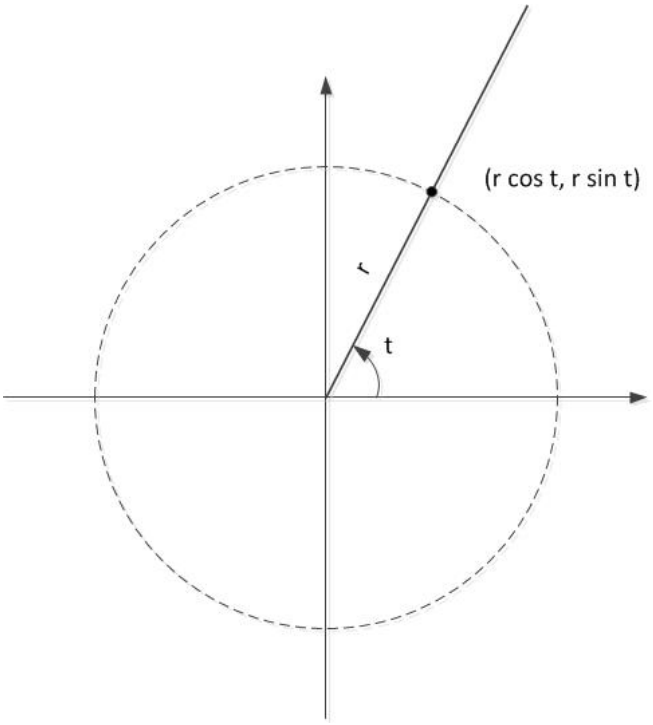
1. (*Please turn in two new files named* **`homework1a.html`** *and* **`homework1a.js`**.)
Modify **`GL_Example1_animated`** so that

- instead of a red unit square on a cyan background, it draws a blue equilateral triangle with sides of length 1 on a yellow background
- instead of moving back and forth, it moves so that its *center* travels in a circle of radius 0.8 at a rate of one degree per frame
- the page has a text box for entering a scale factor (this should scale the triangle, *not* the radius)



Tips: Moving in a circle 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!)



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.,

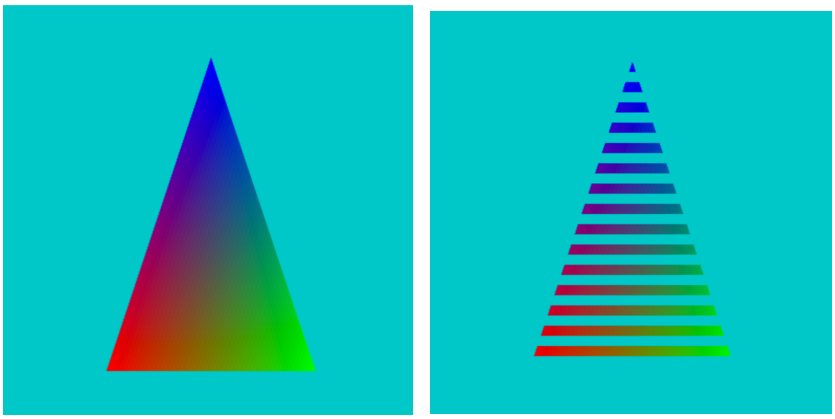
```
var scale = parseFloat(document.getElementById("scaleBox").value);
```

2. *(Please turn in one new file homework1b.html containing the modified code.)*
In a fragment shader, you have access to the current fragment's screen coordinates using the built in variable values **`gl_FragCoord.x`** and **`gl_FragCoord.y`**. These values are in a range from zero up to the canvas width and height, respectively (it's an integer value, but has type **`float`**). There is also a keyword **`discard`** that essentially says "do nothing at this fragment location." Modify the fragment shader from GL_example2 so that it discards fragments in a striped pattern 10 pixels wide (illustrated below for a triangle drawn with that shader, but your code should work regardless of what is being drawn).

Note: you can use conditional statements, just remember that GLSL is very fussy about number types and won't allow a comparison or assignment between **`float`**

and `int` values. There is a built-in `mod(a, b)` function that takes two floating-point values that might be useful. Also remember that the first statement in your fragment shader has to be the precision declaration, e.g.

```
precision mediump float;
```



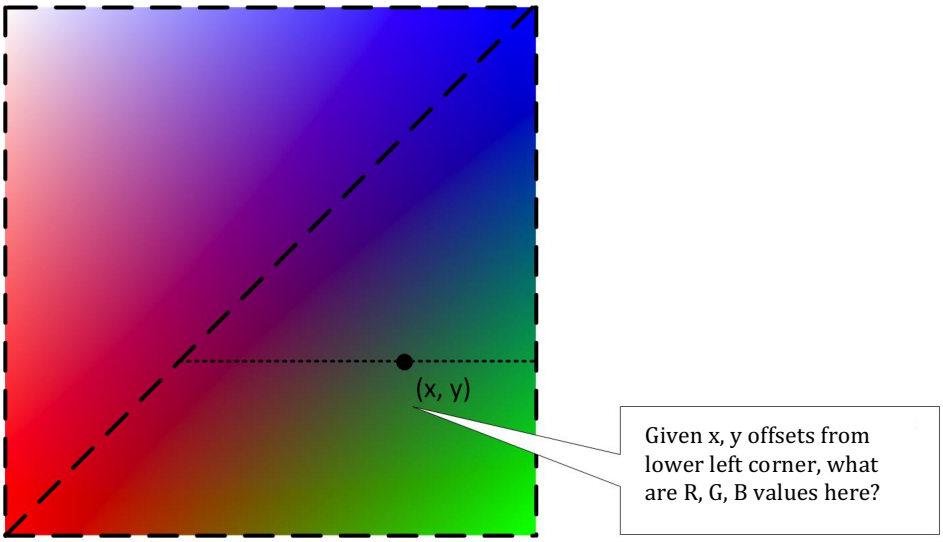
3. *(Please turn in one new file homework1c.html containing the modified code.)* Modify the fragment shader from GL_example1_with_uniform_variable so that instead of a fixed color, the fragment gets a value interpolated between two colors. You will to define two uniform variables for the two colors, say `left_color` and `right_color`, and two uniform variables `min_x` and `max_x` representing the left and right boundaries, such that any fragment drawn to the left of `min_x` is `left_color`, any fragment to the right of `max_x` is `right_color`, and those in between have interpolated values. (The min and max variables may, or may not, correspond to the left and right boundaries of the figure being drawn.) Here is our example with the `min_x` at 150 and `max_x` at 250 using yellow and red:

Tip: remember that any color component values that are outside of the range [0, 1] are automatically clamped, so you don't have to worry about going out of range. Also

remember that the + and * operators are overloaded in GLSL for vector operations. So if v1 and v2 are two values and you want to be p percent of the way from v1 to v2, you can write v1 + p(v2 - v1), just like converting Fahrenheit to Celsius (where v1 would be 0 and v2 would be 100). However, this works perfectly well in GLSL if v1 and v2 are vectors. For another way of thinking about it with nice pictures, see: https://classes.soe.ucsc.edu/cmps160/Fall10/resources/barycentricInterpolation.pdf

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

Write a JavaScript function that performs interpolation of colors associated with the corners of a rectangle (simulating what is done by the fragment shader in GL_example2). That is, given the *size* of the rectangle, *colors* for the four corners, and an integer x and y



offset within the rectangle, find the interpolated values for red, green, and blue at (x, y). (Don't worry about alpha here, assume it's always 1.0.) Assume the rectangle consists of two triangles as shown by the dashed line in the example above. (This is important, because interpolation is done **only** within triangles.) The exact signature for the function is in the file **`color_interpolator.js`** along with a definition for a simple type representing an RGBA color.

> There are several ways you could do this. One is to use a Fahrenheit-to-Celsius conversion to find interpolated values along the vertical legs of the triangle that contains (x, y), and then do it again to interpolate horizontally between those two values (i.e., along the smaller dashed line in the figure). A slicker and more general solution would be to use *barycentric coordinates*, this is optional. If you are interested, see https://classes.soe.ucsc.edu/cmps160/Fall10/resources/barycentricInterpolation.pdf to get started.

5. (*Please turn in the two new files problem5.html and problem5.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 problem5.js and problem5.html as follows:

a) Draw a colored square, similar to the figure from problem 4, on the left side of the canvas and draw a solid colored square on the right side. You'll need a different shader for each figure. The one on the left can use the shader from GL_example2 and the one on the right can use the shader from GL_example1_with_uniform_variable (you'll need to be able to set the color from the JS code).

b) Add a mouse handler that, when the mouse is clicked on the left square, the handler will *use your function from problem 4* to calculate what color will be at that pixel, and it draws the right-hand square that color. (Kind of a rudimentary color-picker!) You can assume for simplicity that the canvas size is fixed at 400 x 400 and that the square position is hard coded.

> See MouseExample.js for how to get the mouse coordinates. After you get the mouse coordinates don't forget that the y-value is upside-down. (For the mouse, (0, 0) is the upper left corner of the canvas, but for the function from problem 4, (0, 0) is the lower left corner, as it would be in the framebuffer.)