­CIS367 - GFX

Winter 2023

**Lab 2.2 : Interaction and Animation**

**Due February 17th by 11:59pm**

**Synopsis:**

Time to move things around the screen! Make a copy of your triangle.js / triangle.html files from before, call them triangle-anim.js / triangle-anim.html. Update the reference in the HTML file appropriately.

**1 - Move a triangle**

Let's first get this triangle bouncing around the screen, bounded by clipping plane. Previously, we had defined basic pass-through shaders and left them at that. The fragment shader will still be pass-through, but we're going to add a little bit to the vertex shader.

First, add two uniform variables, x and y. (This should be outside the main function, underneath the vPosition declaration in your HTML file). (Naturally, you should make them floats).

Then, rewrite the position code to consume these variables. Since we're in 2D, we're going to set the Z coordinate to be static.

gl\_Position.x += x + vPosition.x;

gl\_Position.y += y + vPosition.y;

gl\_Position.z = 0.0;

gl\_Position.w = 1.0;

**Do a little Googling and/or reading and describe what the W attribute of gl\_Position is in Q2! (We also talked about it in class, but I'm looking for a bit more detail).**

Here, we're updating the x and y position each loop. The x and y variables are defined as uniform, one of those special variable types we need. We'll consume them in JavaScript momentarily.

We've defined our shader variables externally, now it is time to update them.

First, define some new globals (we're in the JavaScript file now). We'll need those shader variables, local versions, and a directional indicator.

var x = 0.0;

var y = 0.0;

var xLoc, yLoc;

var xDir = 1.0;

var yDir = 1.0;

Let's also add that random function as well (we should probably add it to the Common folder at some point):

function getRandomArbitrary(min, max) {

return Math.random() \* (max - min) + min;

}

Next, let's make our triangle *smaller*. This means changing its vertices. In the vertex definition list, change all values of 1 to 0.25. Feel free to play with different sizes here as you go.

After the call to gl.useProgram(program), add the following:

xLoc = gl.getUniformLocation(program, "x");

yLoc = gl.getUniformLocation(program, "y");

This will link up the shader to our application. Main application is done, time to update render! Previously, this was two lines of code. Add the following prior to the call to gl.clear.

x += 0.05 \* xDir;

y += 0.1 \* yDir;

gl.uniform1f(xLoc, x);

gl.uniform1f(yLoc, y);

**Don't forget to start animating!**

Drop one of these at the end of the render() function:

window.requestAnimFrame(render);

**Refresh and see what happens. Describe in Q3 what needs to be done!**

Clearly, there is a problem here, as we don't want our triangle disappearing off the screen. Add the following (after y += …):

if (y > 0.9) { // top hit -- reverse y but keep x

y = 0.9;

yDir \*= -1.0;

}

if (x > 0.9) { // right hit -- reverse x but keep y

x = 0.9;

xDir \*= -1.0;

}

if (y < -0.9) { // bottom hit -- reverse y but keep x

y = -0.9;

yDir \*= -1.0;

}

if (x < -0.9) { // left hit -- reverse x but keep y

x = -0.9;

xDir \*= -1.0;

}

Refresh your browser and now you should see your triangle bouncing around. Congrats, you've achieved (bounded-ish) movement!

**2 - Rotate a triangle**

Go back to the original triangle files and make another copy. Call them triangle-rotate.html/js. Now, check out the class GitHub page and take a look at Rotating Square 1. Translate the code from that file to make your triangle rotate in a similar fashion. Hint: this is fairly similar to the last step of this lab, except a *lot* easier. **Take three screenshots and paste them into Q1 (to show a few frames in different orientations).**

**3 - Make a point follow your mouse**

Now let's do some picking, and by picking I mean that we're going to dynamically reposition our Sierpinski vertices. This could be accomplished in several different ways, however we're going to leverage the power of HTML here to select which active vertex we're working on.

Make a copy of https://efredericks.github.io/CIS367-ComputerGraphics/Chap3/gasket5.html/js and call them sierpinski-click.html/js. You may notice that this file has a slider to select the number of recursive subdivisions to allow in the fractal.

Let's first handle a mouse click event. For us, we're going to handle the mouseup event. Why mouseup instead of mousedown? We want it to happen when somebody releases their button, not presses it.

First, let's define some new variables. In the global space (i.e., at the top with the other globals), define variables for x1..x3 and y1..y3. For each of these, set their initial value to an existing triangle vertex. For instance,

var x1 = -1;

var y1 = -1;

…

Then, replace the values in var vertices with these variables, i.e.,

var vertices = [

vec2(x1, y1),

vec2(x2, y2),

vec2(x3, y3)

];

When you refresh the page, it shouldn't look any different!

Now, go down to where you see the onchange code to handle changes to the slider. Note that we are using HTML/JavaScript only, no WebGL to handle events! Again, appreciate how much easier this makes life for you!

But that's neither here nor there, you want to click and have things happen! Add a new listener, this time, for the mouse.

canvas.addEventListener("mouseup", function(event) {

console.log(event.clientX, event.clientY);

});

Open up the Developer Tools and go to the JavaScript console. Refresh your page. **What do you see there? Is there anything *wrong* with it for our (WebGL) purposes? Answer in Q5.**

Ok, comment out the console.log call. Add the following instead:

var rect = gl.canvas.getBoundingClientRect();

var newx = (event.clientX - rect.left) / canvas.width \* 2 - 1;

var newy = (event.clientY - rect.top) / canvas.height \* -2 + 1;

console.log(newx, newy);

**What was the purpose of this step? Describe in Q6. (Hint, we talked about it in class).**

Now, let's link it up with a vertex (this should still be within the mouseup listener).

Set:

x1 = newx;

y1 = newy;

Refresh again. Each time you click, you should be seeing the first vertex move around!

Last steps! Go back to the HTML file and add a new element. Add the following after the div element that defines the slider:

<div>

Vertex:

<input type="radio" name="vertex" value="0" checked> 0

<input type="radio" name="vertex" value="1"> 1

<input type="radio" name="vertex" value="2"> 2

</div>

Now back to the JavaScript file!

Replace your new code (the x1 = newx, y1 = newy) with the following:

var vertex\_id = document.querySelector('input[name="vertex"]:checked').value;

if (vertex\_id == 0) {

x1 = newx;

y1 = newy;

} else if (vertex\_id == 1) {

x2 = newx;

y2 = newy;

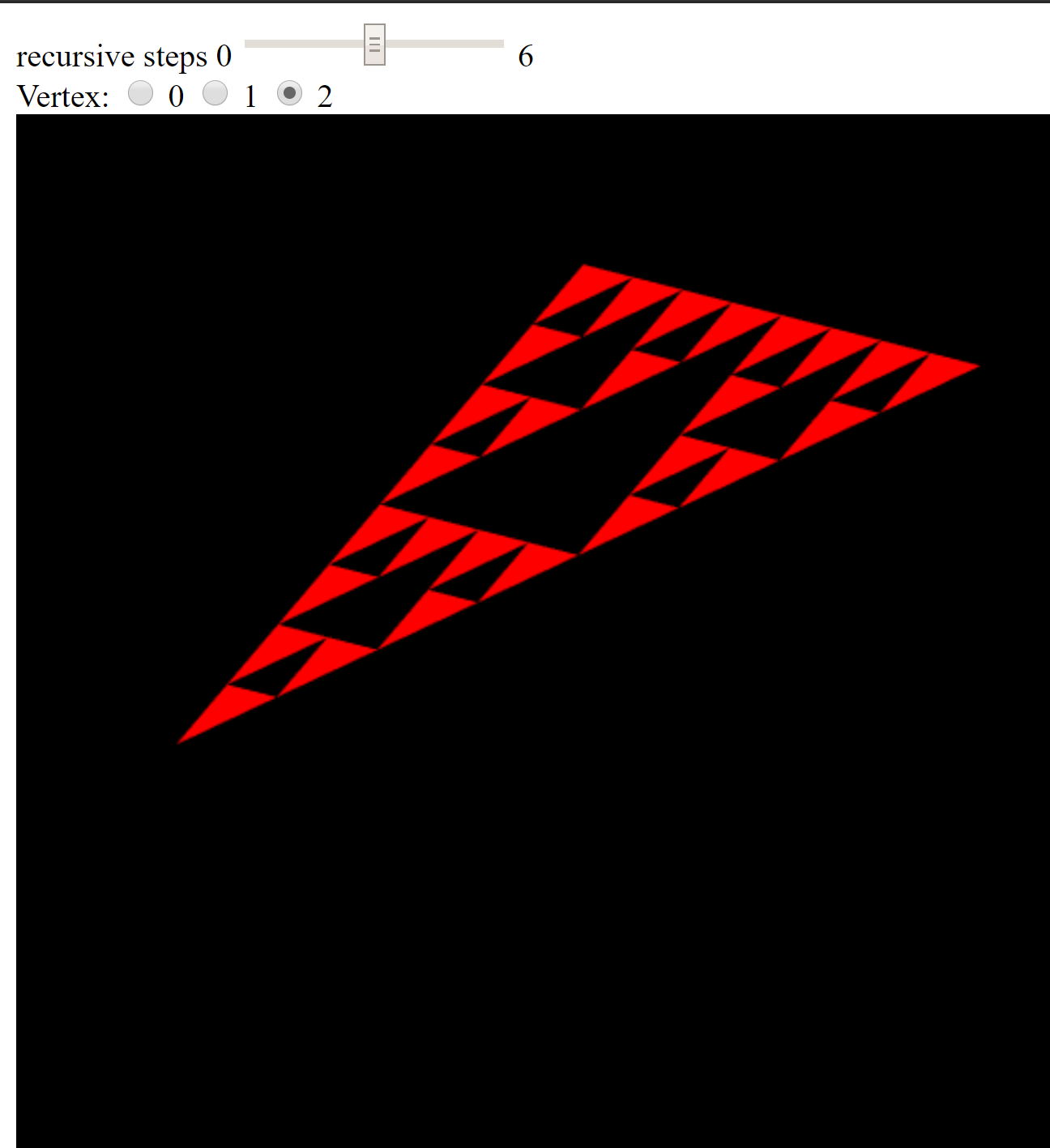
} else {

x3 = newx;

y3 = newy;

}

Now click around! What you should end up with is something like this (note, I changed the background color to make it easier to see the fractal):



**Take 3 screenshots of your gasket with various vertices from your clicks and paste into Q1**

**Upload all your files to your website and then update your index.html page appropriately. For reference, your homepage should now additionally list out triangle-anim.html, triangle-rotate.html, and sierpinski-click.html as list items. Points will be taken off if any are broken/missing.**

**Lab Report / Homework:**

1. **Paste all screenshots here**

**Shape, arrow

Description automatically generatedShape, arrow

Description automatically generatedShape, arrow

Description automatically generated**

**A picture containing text

Description automatically generatedText

Description automatically generatedShape

Description automatically generated with low confidence**

1. **What is the purpose of gl\_Position.w?**

gl\_Position.w is the homogeneous coordinate and tells us wether the element is a vector or a point.

1. **What happened after you added the initial code to triangle's render function and what do you have to do to fix it?**

The triangle flew off into the right corner and off the canvas. We need to prevent the triangle’s vertices from having values outside the canvas (must be less than 1). Basically we need to add collisions with the sides of the canvas.

1. **Assuming everything works fine with your animated moving triangle, add a delay so that it doesn't update as quickly as possible. Add a 100ms delay between each call to render. Copy and paste your code below. (Hint, see the slide titled 'Adding an Interval').**

function render() {  
 setTimeout(function() {  
 ***gl***.clear(***gl***.*COLOR\_BUFFER\_BIT*);  
 ***theta*** += 0.1;  
 ***gl***.uniform1f(***thetaLoc***, ***theta***);  
 ***gl***.drawArrays(***gl***.*TRIANGLES*, 0, 3);  
 ***window***.requestAnimFrame(render);  
 }, 100);  
}

1. **Answer the question regarding console.log from the Sierpinski Clicker section.**

The coordinates that are being displayed start in the top left and range from 0 to the canvas’ width/height, while WebGL uses a coordinate system that starts in the middle of the screen and range from 1 to -1.

1. **Answer the question regarding the math surrounding the mouse click event.**

We have to change coordinate spaces to use WebGL’s coordinate space. As described in the previous answer, the coordinates of the mouse’s position is relative to the top left of the canvas, but we need the coordinates relative to the center.

1. **Extend the Sierpinski in some fashion. Describe your extension and how you had to modify the original code to get it.**

I made the triangle bounce around the frame, similarly to how we animated the triangle in triangle-anim.html/js. I also added the clicking element from sierpinski-click.html/js. Both of these features do not overlap in their implementation so It was relatively easy to bring both together.

1. **Copy and paste the URL of your homepage (just in case if it has changed)**

<https://student.computing.gvsu.edu/johnsev/cis-367.html>

1. **Time to start thinking about your term project. For this, briefly describe *two* ideas of what you would like to do. *Note: you are not going to be held to these, but you will get feedback on them!***
   1. I’ve created a Ray Caster in pygame in the past and had a lot of fun doing it. I think it could be interesting creating one in WebGL and JavaScript as I have no experience working in JavaScript or HTML. I’m not sure If I really want to do this yet though, but It’s still on the board.
   2. I really liked the fluid simulation shown at the beginning of class. I’ve always been interested in different types of simulations so it might be interesting to look into creating something similar. I don’t have much experience in unity or blender so I’d love to spend more time in those programs.