ECS762P

Computer Graphics

Sean Sanii Nejad

Contents

Exercise A

A1

Output

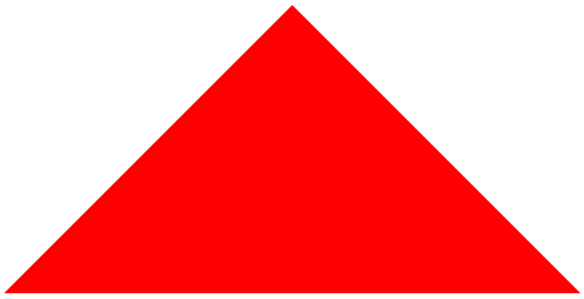
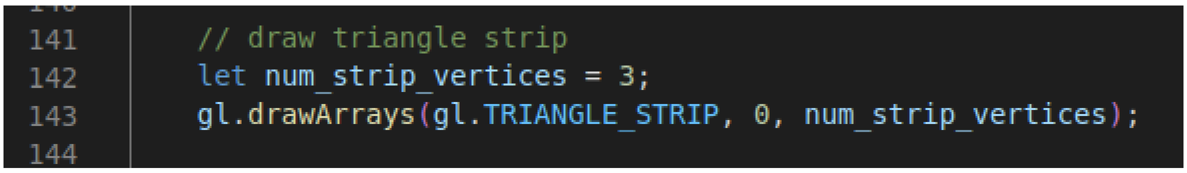


Figure 1: A single geometric primitive

JavaScript Code Snippet



```
    // draw triangle strip
    let num_strip_vertices = 3;
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, num_strip_vertices);
```

Figure 2: Code Snippet from 'rotating-square.js' file

Triangle Strip Example

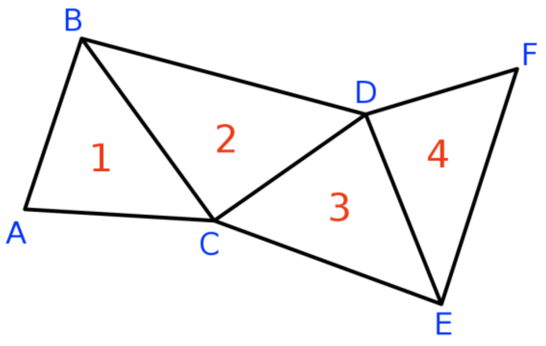


Figure 3: Triangles constructed with 'Triangle Strip' method

Explanation

Changing the 3rd parameter of the function drawArrays() in the 'rotational-square' JavaScript file changes the square into a triangle. This is because count specifies the number of vectors to be rendered, in this case 3. The WebGL API renders the primitives from the array data which holds the attributes for the shape.

gl.drawArrays(mode, first, count)

Also, because the mode is set to TRIANGLE_STRIP, new triangles are created with every additional vertex, as vertices are shared between them. This is shown in the Triangle Strip Example above.

References

https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/drawElements
https://en.wikipedia.org/wiki/Triangle_strip

A2

Output

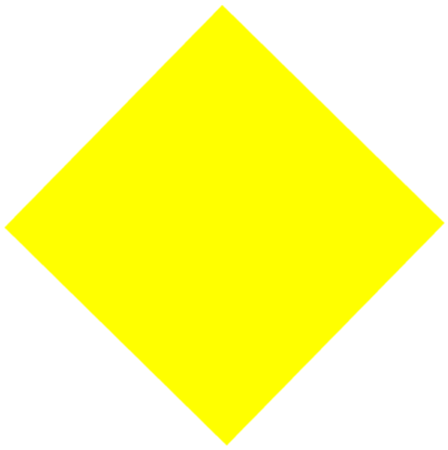
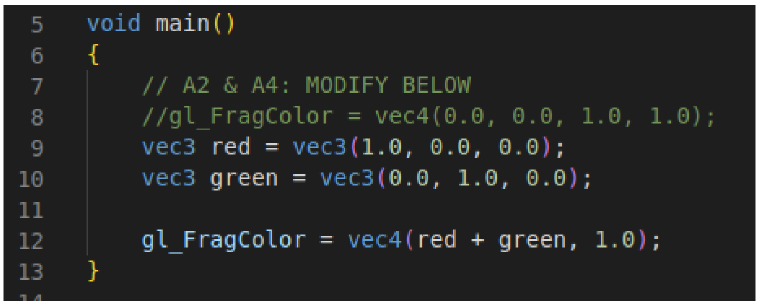


Figure 4: Primitive RGBA attribute set to yellow

Fragment Shader Code Snippet

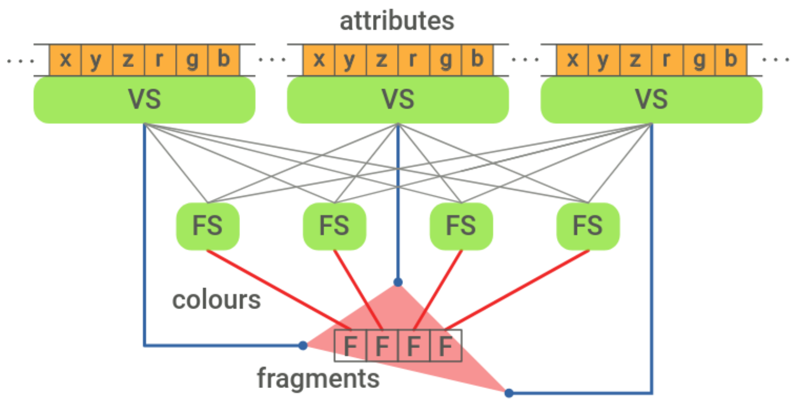


```
void main()
{
    // A2 & A4: MODIFY BELOW
    //gl_FragColor = vec4(0.0, 0.0, 1.0, 1.0);
    vec3 red = vec3(1.0, 0.0, 0.0);
    vec3 green = vec3(0.0, 1.0, 0.0);

    gl_FragColor = vec4(red + green, 1.0);
}
```

Figure 5: Code snippet from 'rotating-shape-frag.glsl' file

Shader Rendering Outline



Explanation

This image shows us that we modified the individual fragments within the square to be rendered in yellow instead of red. This was achieved by altering the 'rotating-shape-frag.glsl class. I created two vec3 objects in main called 'red' and 'green'. Vec3 objects consist of three parameters. These are 3 floats between 0 and 1 reflecting RGB format. For instance, (1.0, 0.0, 0.0) will be red. I then used arithmetic addition on these two vec3 objects

vec3 red = vec3(1.0, 0.0, 0.0);
vec3 green = vec3(0.0, 1.0, 0.0);

vec3 yellow = red + blue; // => [1.0, 1.0, 0.0]

The new vec3 object was directly used in the parameters of the vec4 objected called 'gl_Fragcolor'. The vec4 object as an additional parameter for opacity.

gl_FragColor = vec4(1.0, 1.0, 0.0, 1.0);

As a result, the fragment shader filled in the two triangles fragments coloured in yellow.

References
https://thebookofshaders.com/06/

A3
Output

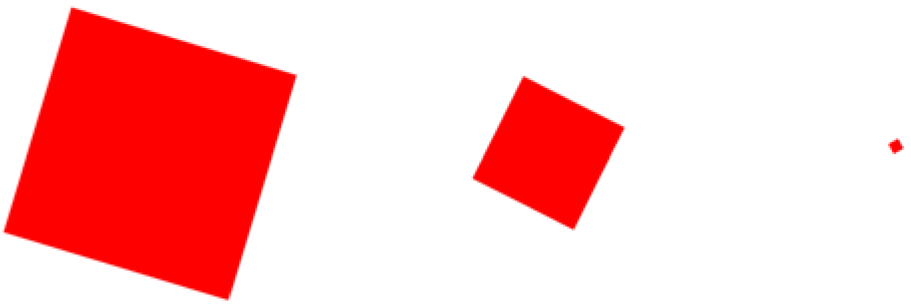
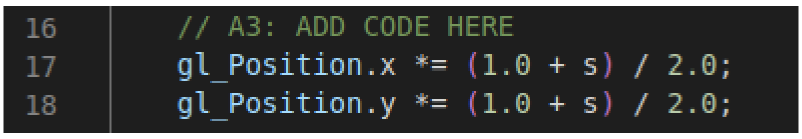


Figure 6: Primitive Shrinking and Growing repeatedly

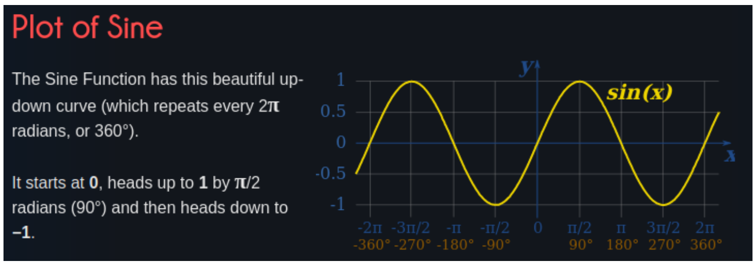
Vertex Shader Code Snippet



```
// A3: ADD CODE HERE
gl_Position.x *= (1.0 + s) / 2.0;
gl_Position.y *= (1.0 + s) / 2.0;
```

Figure 7: Code Snippet from 'rotating-shape-vert.glsl' file

Sine Graph



Plot of Sine

The Sine Function has this beautiful up-down curve (which repeats every 2π radians, or 360°).

It starts at **0**, heads up to **1** by π/2 radians (90°) and then heads down to **−1**.

Figure 8: Sine Function Graph

Explanation
By including the two lines as shown above. The immediate effect on the square is it is continuously alternating between growing and shrinking at certain depths.

We can see in the code that 1.0 is being added to sin(theta). Sin(theta) range is between -1 and 1 (As shown in the image above). Giving as a value between 0-2. It is then divided by 2, finally giving as a value between 0-1.

This value is then being multiplied by the vertex shading attributes x and y coordinates each frame.

gl_Position.x *=
gl_Position.y *=

As the value decreases towards 0, the square will shrink into nothing, and as the value increases, the shape will grow back to its original size.

References

https://www.mathsisfun.com/algebra/trig-sin-cos-tan-graphs.html

A4

Output

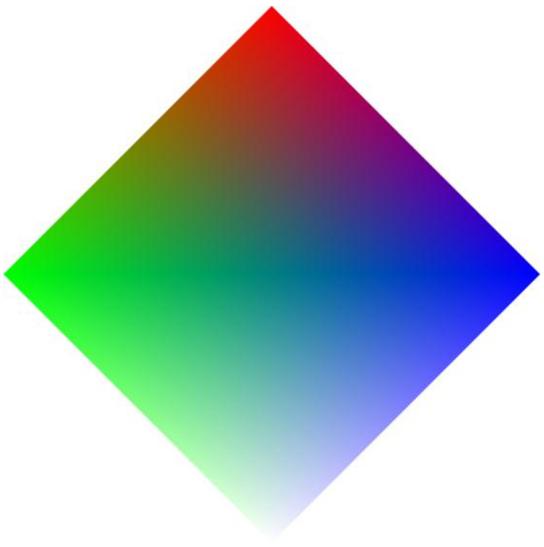
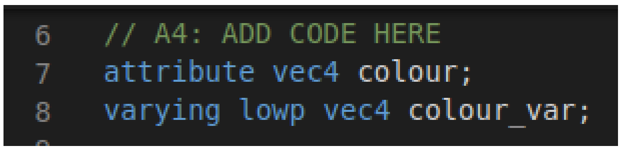


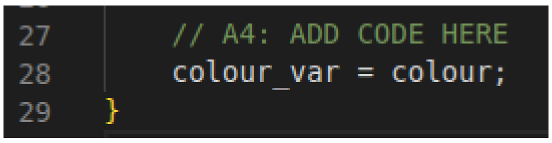
Figure 9: Primitive with interpolated fragment colours

Vertex Shader Code Snippet



```glsl
// A4: ADD CODE HERE
attribute vec4 colour;
varying lowp vec4 colour_var;
```

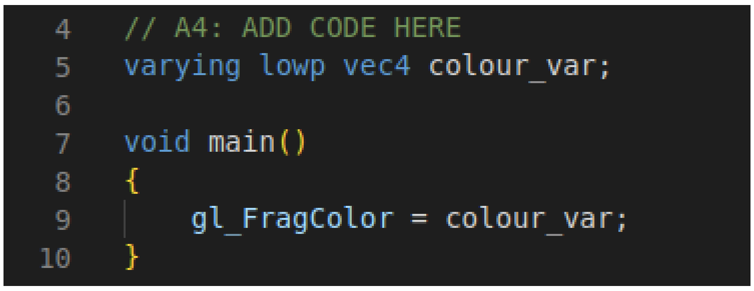
Figure 10: Code Snippet from 'rotating-shape-vert.glsl' file



```glsl
    // A4: ADD CODE HERE
    colour_var = colour;
}
```

Figure 11: Code Snippet from 'rotating-shape-vert.glsl' file

Fragment Shader Code Snippet

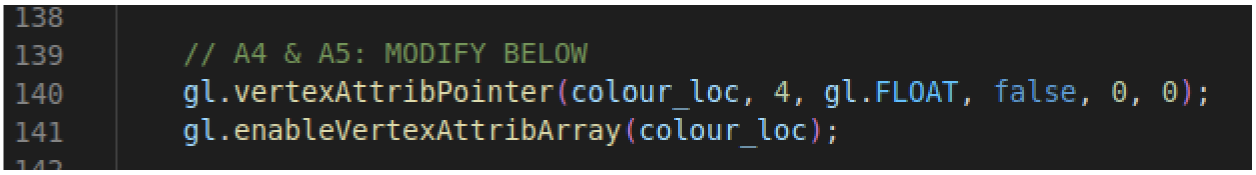


```glsl
// A4: ADD CODE HERE
varying lowp vec4 colour_var;

void main()
{
    gl_FragColor = colour_var;
}
```

Figure 12: Code Snippet from 'rotating-shape-frag.glsl' file

JavaScript Code Snippet



```
    // A4 & A5: MODIFY BELOW
    gl.vertexAttribPointer(colour_loc, 4, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(colour_loc);
```

Figure 13: Code Snippet from 'rotating-square.js' file

Shader Rendering Outline

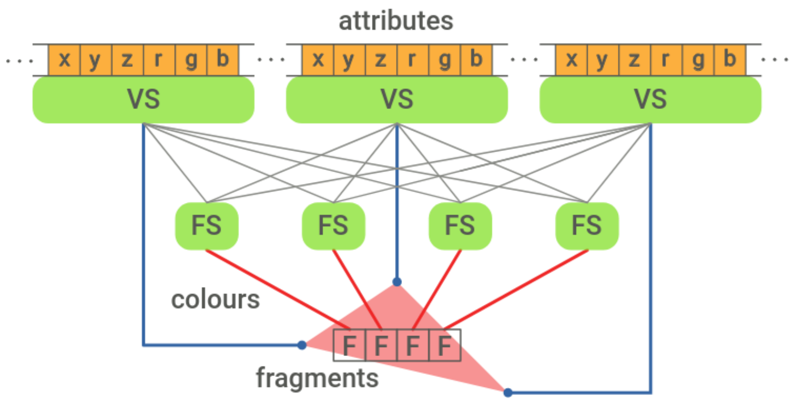


Figure 14: Diagram demonstrating connection from Vertex shader, to Fragment Shader, to fragments and vertices

Explanation

First, I created an attribute vec4 objected called ‘colour‘ and a varying lowp vec4 objected called ‘colour_var’.

An attribute is a GLSL variable that is only available to the Vertex shader and the JavaScript code and is shared between them. Varying objects are used to pass down data from the vertex shader to the fragment shader and will be interpolated. Meaning that each fragment’s colour will be updated between the 4 vertices accordingly.

This is shown in the fragment shader snippet as the varying colour_var object is assigned to gl_FragColor.

I then included the vertexAttribPointer function. This function has 6 parameters. The first being index, this specifies the index of the vertex attribute. The second is size, which is the number of components per vertex attribute. In this case, we used 4 for the Vec4 values RGBA. I then set the array to gl.FLOAT as this corresponds to our vec4 values. Finally normalised, stride and offset are set to false or 0.

Lastly, we call the enableVertexAttribArray() function which the WebGL API turns on the attribute array into the list of attribute arrays as shown at the top of the Shader Rendering Outline image.

References

https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/vertexAttribPointer
https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/enableVertexAttribArray

A5

Output

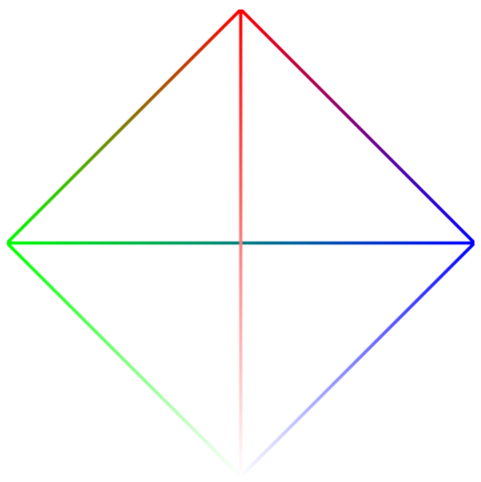
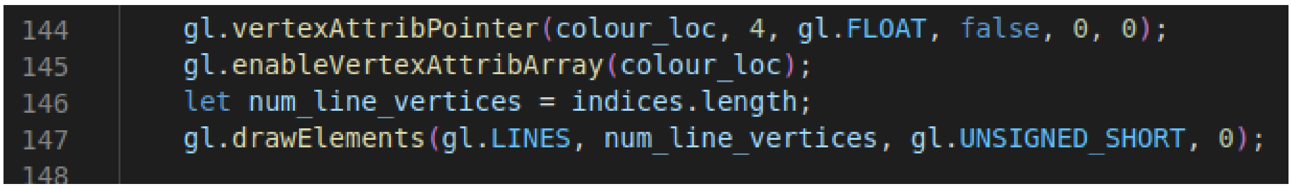


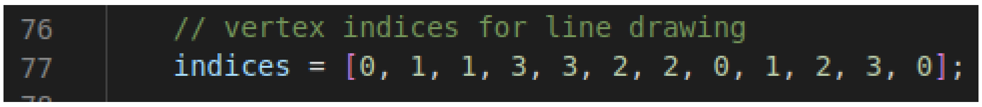
Figure 15: Lines drawn between vertices with interpolated colours

JavaScript Code Snippet



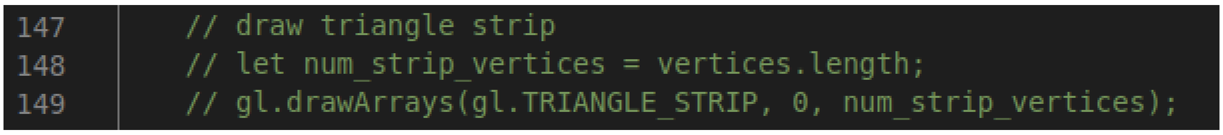
```
    gl.vertexAttribPointer(colour_loc, 4, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(colour_loc);
    let num_line_vertices = indices.length;
    gl.drawElements(gl.LINES, num_line_vertices, gl.UNSIGNED_SHORT, 0);
```

Figure 16: Code Snippet from 'rotating-square.js' file



```
    // vertex indices for line drawing
    indices = [0, 1, 1, 3, 3, 2, 2, 0, 1, 2, 3, 0];
```

Figure 17: Code Snippet from 'rotating-square.js' file



```
    // draw triangle strip
    // let num_strip_vertices = vertices.length;
    // gl.drawArrays(gl.TRIANGLE_STRIP, 0, num_strip_vertices);
```

Figure 18: Code Snippet from 'rotating-square.js' file

Explanation

In this exercise I called the vertexAttriPointer() and enableVertexAttribArray() functions as previously explained in A4. I then created a local variable called 'num_line_vertices' and assigned it to the length of the indices array. In this case, it is 12.

From here, I called the drawElements() function. This function is a part of WebGL API which renders primitives from an array data type.

drawElements(mode, count, type, offset)

The first parameter we set as gl.LINES. This mode draws lines between a pair of vertices. We then assign the count to num_line_vertices which is 12 as previous shown. We specify that we're using UNSIGNED_SHORT and finally set the offset to 0.

Lastly, we create the array of indices that will have lines drawn between them. Indices[0] will have a line drawn to Indices[1] and indices[2] will I have a line drawn to indices[3] and so on.

We removed the code in JavaScript code snippet #3 as that follows a different method to render primitives.

References

https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/drawElements

A6

Output

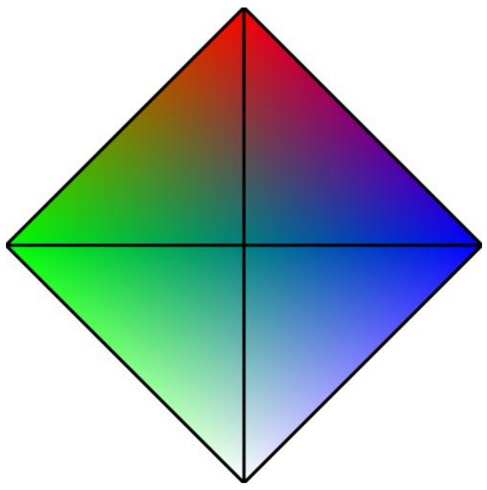
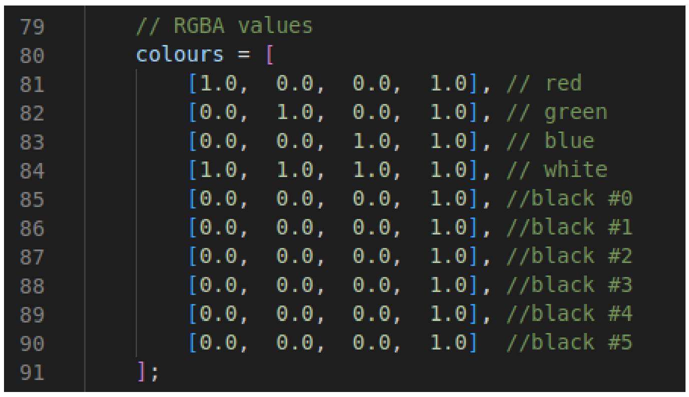


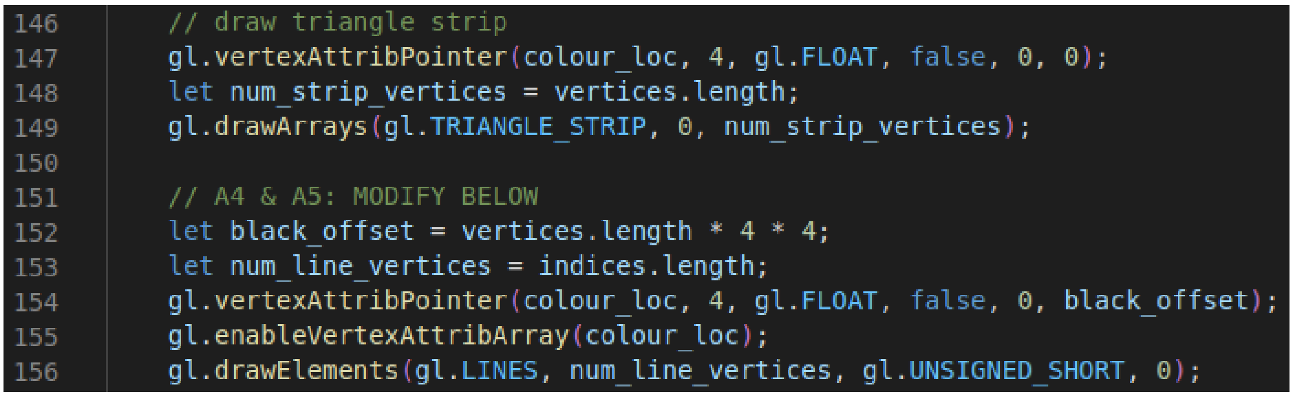
Figure 19: Two Triangle Primitives drawn with interpolated colours and 6 lines drawn over the top

JavaScript Code Snippet



```javascript
// RGBA values
colours = [
    [1.0,  0.0,  0.0,  1.0], // red
    [0.0,  1.0,  0.0,  1.0], // green
    [0.0,  0.0,  1.0,  1.0], // blue
    [1.0,  1.0,  1.0,  1.0], // white
    [0.0,  0.0,  0.0,  1.0], //black #0
    [0.0,  0.0,  0.0,  1.0], //black #1
    [0.0,  0.0,  0.0,  1.0], //black #2
    [0.0,  0.0,  0.0,  1.0], //black #3
    [0.0,  0.0,  0.0,  1.0], //black #4
    [0.0,  0.0,  0.0,  1.0]  //black #5
];
```

Figure 20: Code Snippet from 'rotating-square.js' file



```javascript
// draw triangle strip
gl.vertexAttribPointer(colour_loc, 4, gl.FLOAT, false, 0, 0);
let num_strip_vertices = vertices.length;
gl.drawArrays(gl.TRIANGLE_STRIP, 0, num_strip_vertices);

// A4 & A5: MODIFY BELOW
let black_offset = vertices.length * 4 * 4;
let num_line_vertices = indices.length;
gl.vertexAttribPointer(colour_loc, 4, gl.FLOAT, false, 0, black_offset);
gl.enableVertexAttribArray(colour_loc);
gl.drawElements(gl.LINES, num_line_vertices, gl.UNSIGNED_SHORT, 0);
```

Figure 21: Code Snippet from 'rotating-square.js' file

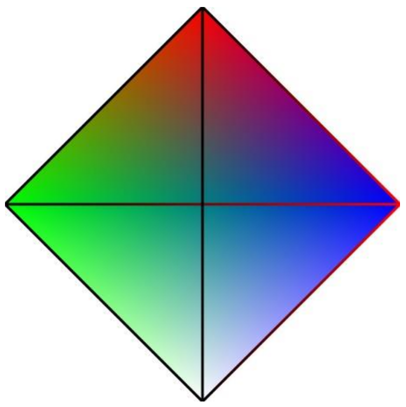
Explanation

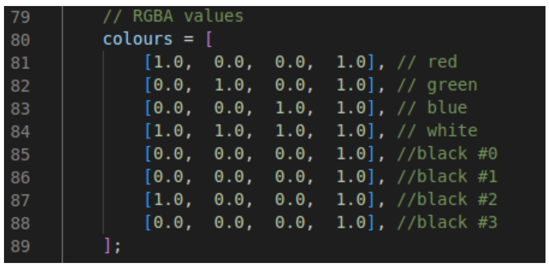
First, I added the code from A4. This is to ensure that the two triangles are drawn first with interpolated colours, allowing the 6 black lines to be drawn over the top. I then created 6 additional arrays which vec4 values will set the colour the black. These arrays were created within an array (2D array). I then created a black_offset variable at 64, to move the vertex pointer to black#0 as shown in JavaScript code snippet #1. This is done by putting the variable in the offset parameter vertexAttribPointer(). From here, the lines are drawn the same as in A5, however, the colours attribute for the 4 vertices have been changed to black. As all 4 vertices are black, are the fragments within the lines are drawn as if they were constant.

References

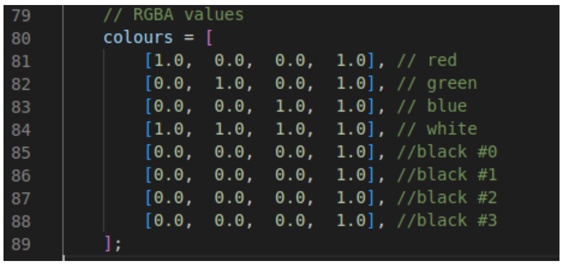
https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/vertexAttribPointer
https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/drawElements

Correction





```
    // RGBA values
    colours = [
        [1.0,  0.0,  0.0,  1.0], // red
        [0.0,  1.0,  0.0,  1.0], // green
        [0.0,  0.0,  1.0,  1.0], // blue
        [1.0,  1.0,  1.0,  1.0], // white
        [0.0,  0.0,  0.0,  1.0], //black #0
        [0.0,  0.0,  0.0,  1.0], //black #1
        [1.0,  0.0,  0.0,  1.0], //black #2
        [0.0,  0.0,  0.0,  1.0], //black #3
    ];
```



```
    // RGBA values
    colours = [
        [1.0,  0.0,  0.0,  1.0], // red
        [0.0,  1.0,  0.0,  1.0], // green
        [0.0,  0.0,  1.0,  1.0], // blue
        [1.0,  1.0,  1.0,  1.0], // white
        [0.0,  0.0,  0.0,  1.0], //black #0
        [0.0,  0.0,  0.0,  1.0], //black #1
        [0.0,  0.0,  0.0,  1.0], //black #2
        [0.0,  0.0,  0.0,  1.0], //black #3
    ];
```

Figure 47: Correction code snippet and output

From testing, I can see that only 4 black vec4 arrays are needed to get the desired output. As shown above, when I change black #2 to red, it changes the colours of three lines. This means that the lines fragment colours are influenced by the 4 vertices and are not needed for each line. This is because we are doing index drawing instead of direct drawing. The last code image shows what I used in the end.

Exercise B

B1

Output

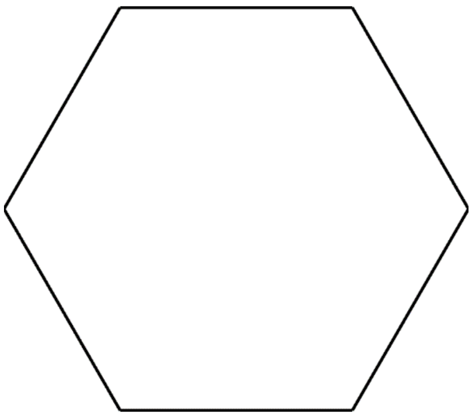
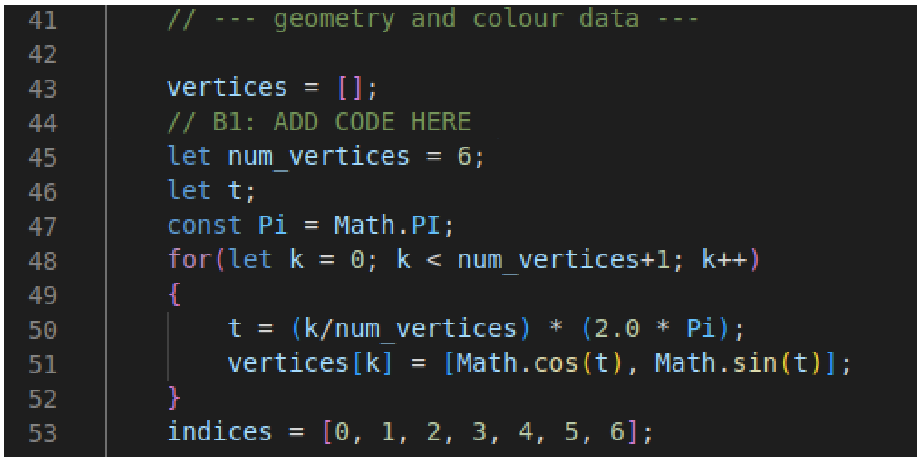


Figure 22: Hexagon created with cos / sin math equation on vertices x and y location

JavaScript Code Snippet



```
// --- geometry and colour data ---

vertices = [];
// B1: ADD CODE HERE
let num_vertices = 6;
let t;
const Pi = Math.PI;
for(let k = 0; k < num_vertices+1; k++)
{
    t = (k/num_vertices) * (2.0 * Pi);
    vertices[k] = [Math.cos(t), Math.sin(t)];
}
indices = [0, 1, 2, 3, 4, 5, 6];
```

Figure 23: Code Snippet from 'colour-hexagon.js' file

Explanation

For this exercise, I first created 3 variables. num_vertices, t and Pi. I then created a for loop, which loops through the number of vertices + 1. With each iteration the following equation is used and assigned to t.

(k / num_vertices) * (2.0 * Pi)

t is then applied to Math.cos and Math.sin parameters and formatted in another array, which is finally assigned to the vertices array.

The vertices and indices arrays are then used with WebGL API to initialize and create buffer objects data stores with gl.bufferData() function.

References

https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Constants#rendering_primitives
https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/bufferData

B2

Output

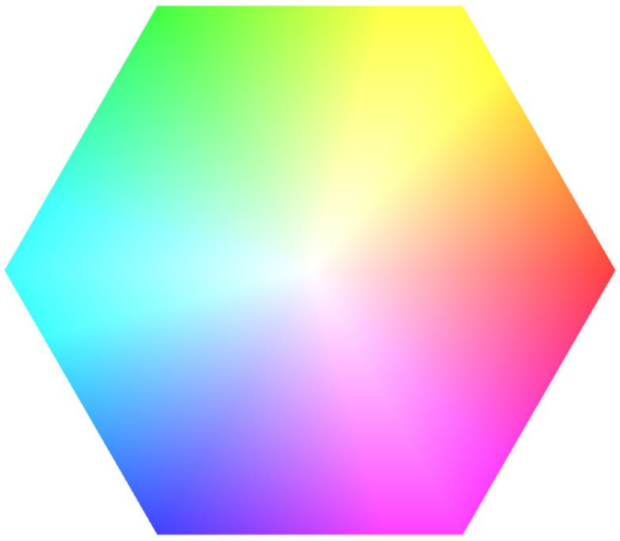
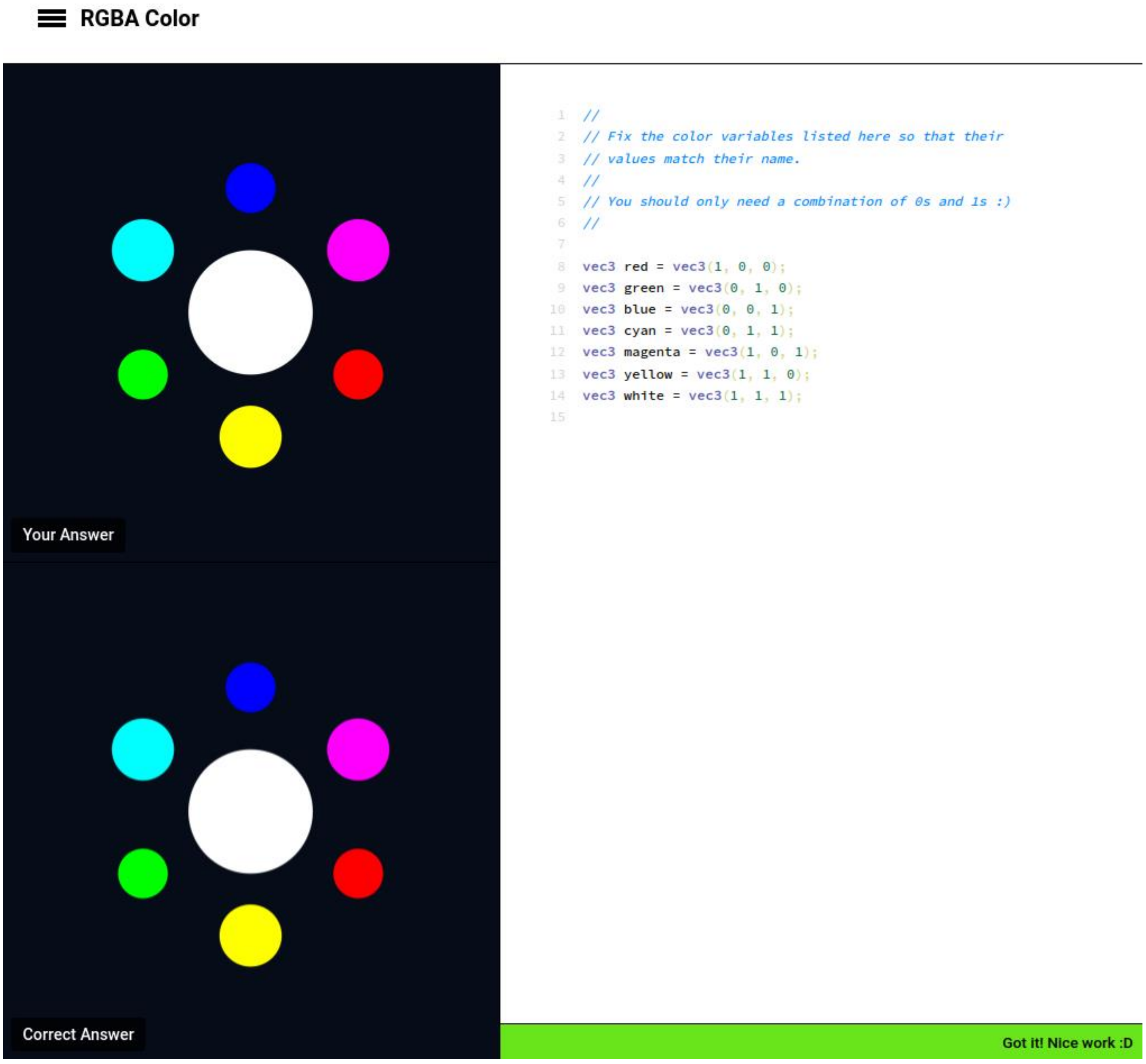


Figure 24: Hexagon created with Triangle Fan and interpolated colours

RGBA Colour Example



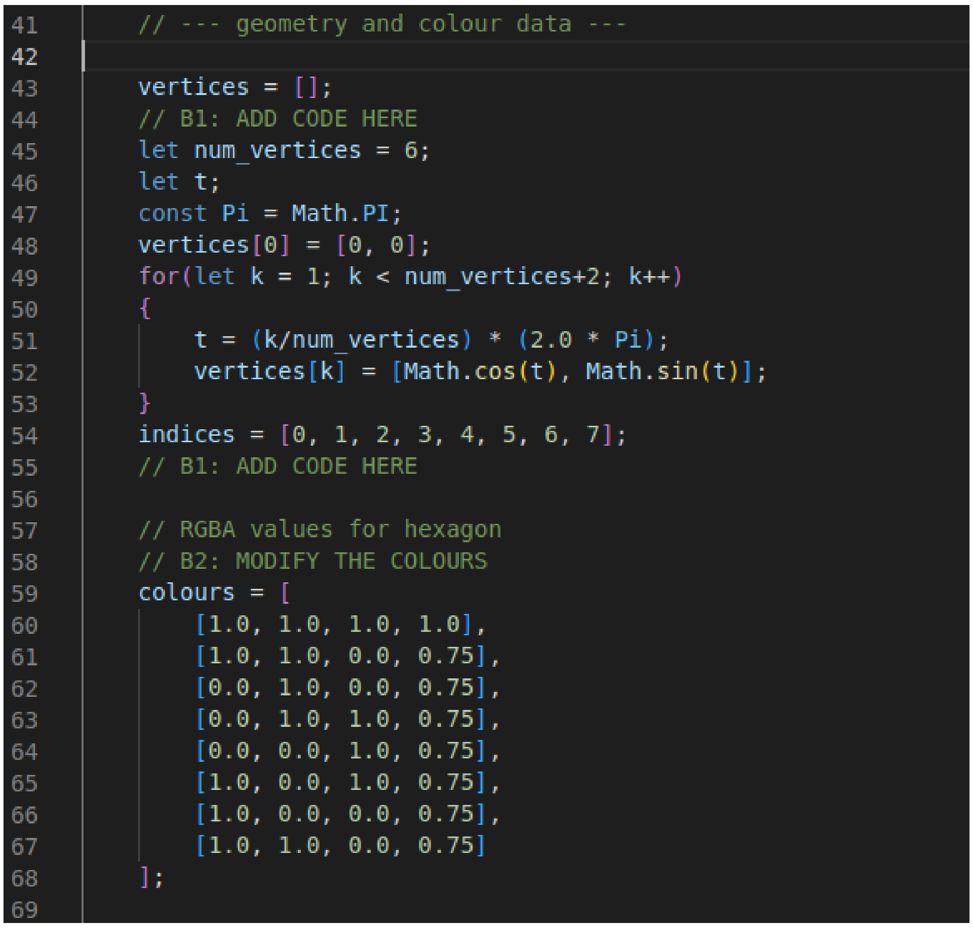
```
//
// Fix the color variables listed here so that their
// values match their name.
//
// You should only need a combination of 0s and 1s :)
//

vec3 red = vec3(1, 0, 0);
vec3 green = vec3(0, 1, 0);
vec3 blue = vec3(0, 0, 1);
vec3 cyan = vec3(0, 1, 1);
vec3 magenta = vec3(1, 0, 1);
vec3 yellow = vec3(1, 1, 0);
vec3 white = vec3(1, 1, 1);

```

Figure 25: Website testing the correct parameters for colours

JavaScript Code Snippet



```
    // --- geometry and colour data ---

    vertices = [];
    // B1: ADD CODE HERE
    let num_vertices = 6;
    let t;
    const Pi = Math.PI;
    vertices[0] = [0, 0];
    for(let k = 1; k < num_vertices+2; k++)
    {
        t = (k/num_vertices) * (2.0 * Pi);
        vertices[k] = [Math.cos(t), Math.sin(t)];
    }
    indices = [0, 1, 2, 3, 4, 5, 6, 7];
    // B1: ADD CODE HERE

    // RGBA values for hexagon
    // B2: MODIFY THE COLOURS
    colours = [
        [1.0, 1.0, 1.0, 1.0],
        [1.0, 1.0, 0.0, 0.75],
        [0.0, 1.0, 0.0, 0.75],
        [0.0, 1.0, 1.0, 0.75],
        [0.0, 0.0, 1.0, 0.75],
        [1.0, 0.0, 1.0, 0.75],
        [1.0, 0.0, 0.0, 0.75],
        [1.0, 1.0, 0.0, 0.75]
    ];

```

Figure 26: Code Snippet from 'colour-hexagon.js' file

Triangle Fan Example

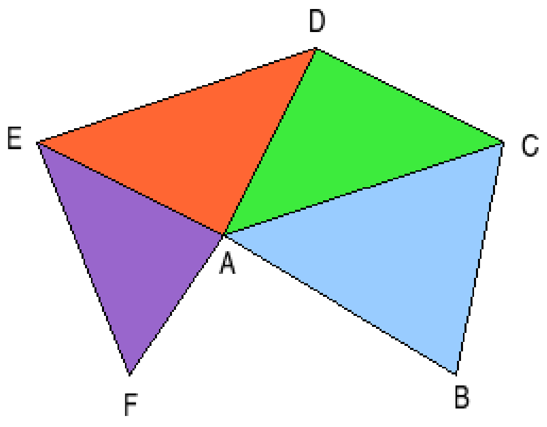


Figure 27: Triangles constructed with Triangle Fan method

Explanation

For this exercise, I changed the drawElements() function mode parameter to TRIANGLE_FAN. Triangle fan is a primitive that connects triangles together by sharing the first one in the centre as shown in (Triangle Fan Example). From here, I created the first vertex position 0,0. I set the colour to this vertex to white so I can get the desired effect as required. From here, I looped through the array, creating the appropriate vertices as done in B1.

I used a separate JavaScript code to learn how to create the necessary colours as shown in RGBA Colour Learning image. From here, I created the colours array.

References
https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/drawElements
https://en.wikipedia.org/wiki/Triangle_fan

Exercise C
C1
Output

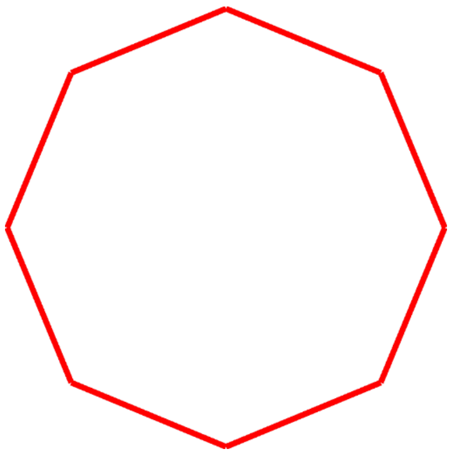
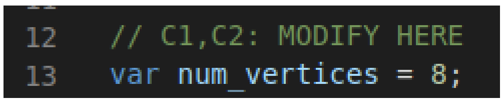


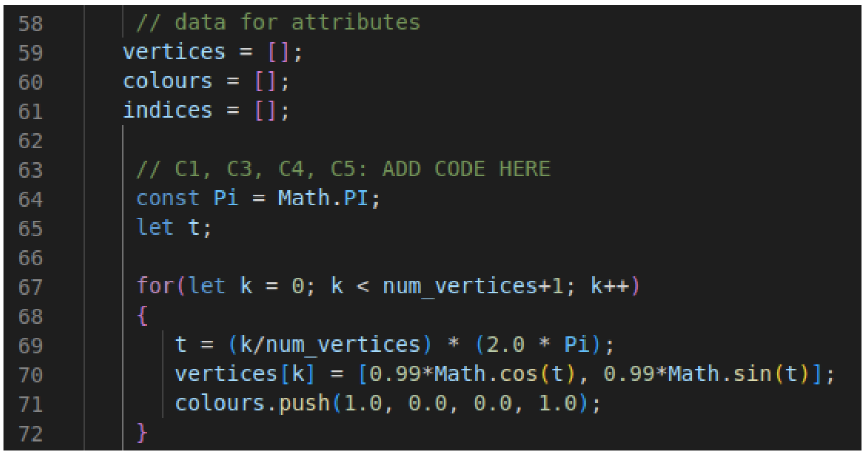
Figure 28: Octagon constructed with Line_Strip and colour red attribute

JavaScript Code Snippet



```
// C1,C2: MODIFY HERE
var num_vertices = 8;
```

Figure 29: Snippet Code from 'colour-spiral.js' file



```
 // data for attributes
vertices = [];
colours = [];
indices = [];

 // C1, C3, C4, C5: ADD CODE HERE
 const Pi = Math.PI;
 let t;

 for(let k = 0; k < num_vertices+1; k++)
 {
    t = (k/num_vertices) * (2.0 * Pi);
    vertices[k] = [0.99*Math.cos(t), 0.99*Math.sin(t)];
    colours.push(1.0, 0.0, 0.0, 1.0);
 }
```

Figure 30: Snippet Code from 'colour-spiral.js' file

Explanation
I first populated the vertices array with a for loop. As shown in B1, however I pushed '(1.0, 0.0, 0.0, 1.0)' array for each vertex to populate colours array as-well (JavaScript Code Snippet).

I also changed num_vertices = 8 so we render 8 vertices that can be used to draw lines between them using the drawArrays() function with LINE_STRIP primitive. Both x and y were multiplied by 0.99 to fit the octagon nicely in frame.

References
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push
https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/bufferData

C2

Output

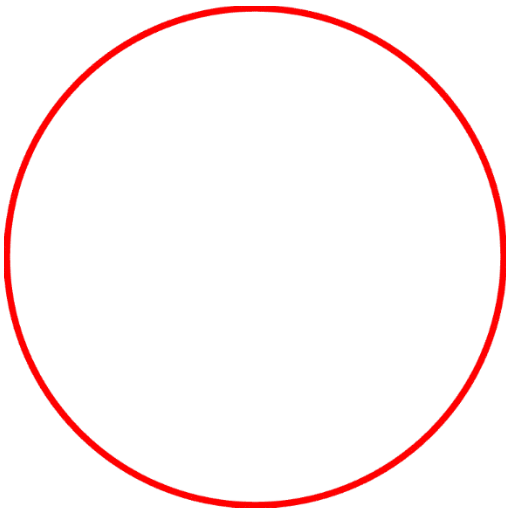
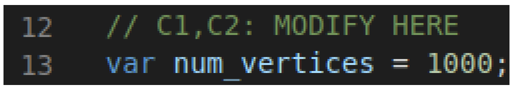


Figure 31: Circle constructed with Line_Strip and 1000 vertices

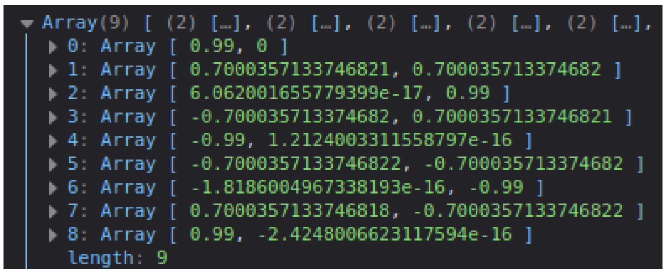
JavaScript Code Snippet



```javascript
// C1,C2: MODIFY HERE
var num_vertices = 1000;
```

Figure 32: Code Snippet from 'colour-spiral.js' file

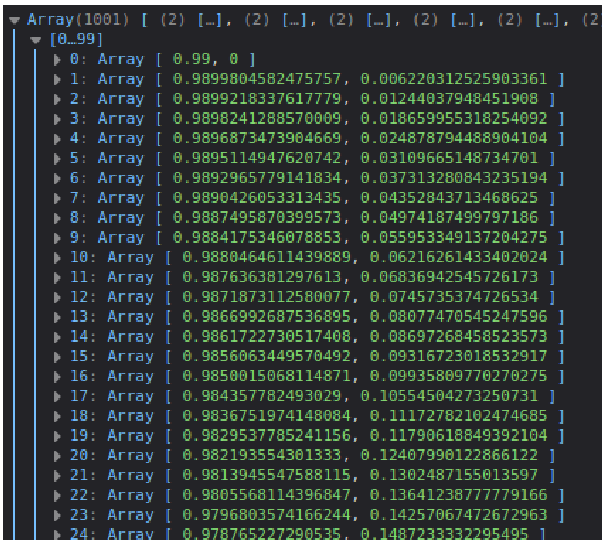
Vertices Array Size: Octagon



```
▼ Array(9) [ (2) […], (2) […], (2) […], (2) […], (2) […],
  ▶ 0: Array [ 0.99, 0 ]
  ▶ 1: Array [ 0.7000357133746821, 0.700035713374682 ]
  ▶ 2: Array [ 6.062001655779399e-17, 0.99 ]
  ▶ 3: Array [ -0.700035713374682, 0.7000357133746821 ]
  ▶ 4: Array [ -0.99, 1.2124003311558797e-16 ]
  ▶ 5: Array [ -0.7000357133746822, -0.700035713374682 ]
  ▶ 6: Array [ -1.8186004967338193e-16, -0.99 ]
  ▶ 7: Array [ 0.7000357133746818, -0.7000357133746822 ]
  ▶ 8: Array [ 0.99, -2.4248006623117594e-16 ]
    length: 9
```

Figure 33: Console log of vertex array for octagon

Vertices Array Size: Circle



```
▼ Array(1001) [ (2) […], (2) […], (2) […], (2) […], (2) […], (2
  ▼ [0…99]
    ▶ 0: Array [ 0.99, 0 ]
    ▶ 1: Array [ 0.9899804582475757, 0.006220312525903361 ]
    ▶ 2: Array [ 0.9899218337617779, 0.01244037948451908 ]
    ▶ 3: Array [ 0.9898241288570009, 0.018659955318254092 ]
    ▶ 4: Array [ 0.9896873473904669, 0.024878794488904104 ]
    ▶ 5: Array [ 0.9895114947620742, 0.03109665148734701 ]
    ▶ 6: Array [ 0.9892965779141834, 0.037313280843235194 ]
    ▶ 7: Array [ 0.9890426053313435, 0.04352843713468625 ]
    ▶ 8: Array [ 0.9887495870399573, 0.04974187499797186 ]
    ▶ 9: Array [ 0.9884175346078853, 0.055953349137204275 ]
    ▶ 10: Array [ 0.9880464611439889, 0.06216261433402024 ]
    ▶ 11: Array [ 0.987636381297613, 0.06836942545726173 ]
    ▶ 12: Array [ 0.9871873112580077, 0.0745735374726534 ]
    ▶ 13: Array [ 0.9866992687536895, 0.08077470545247596 ]
    ▶ 14: Array [ 0.9861722730517408, 0.08697268458523573 ]
    ▶ 15: Array [ 0.9856063449570492, 0.09316723018532917 ]
    ▶ 16: Array [ 0.9850015068114871, 0.09935809770270275 ]
    ▶ 17: Array [ 0.984357782493029, 0.10554504273250731 ]
    ▶ 18: Array [ 0.9836751974148084, 0.11172782102474685 ]
    ▶ 19: Array [ 0.9829537785241156, 0.11790618849392104 ]
    ▶ 20: Array [ 0.982193554301333, 0.12407990122866122 ]
    ▶ 21: Array [ 0.9813945547588115, 0.1302487155013597 ]
    ▶ 22: Array [ 0.9805568114396847, 0.13641238777779166 ]
    ▶ 23: Array [ 0.9796803574166244, 0.14257067472672963 ]
    ▶ 24: Array [ 0.978765227290535, 0.1487233332295495 ]
```

Figure 34: Console log of vertex array for circle

Explanation

Changing the num_vertices variable increased the number of vertices being rendered from 8 to 1000. This is shown in the console with console.log(vertices). We can see that the vertices size array for the circle is a lot larger in comparison to the Octagon. Each holding the x and y position for the individual vertices.

References

https://www.w3schools.com/java/java_arrays.asp

C3

Output

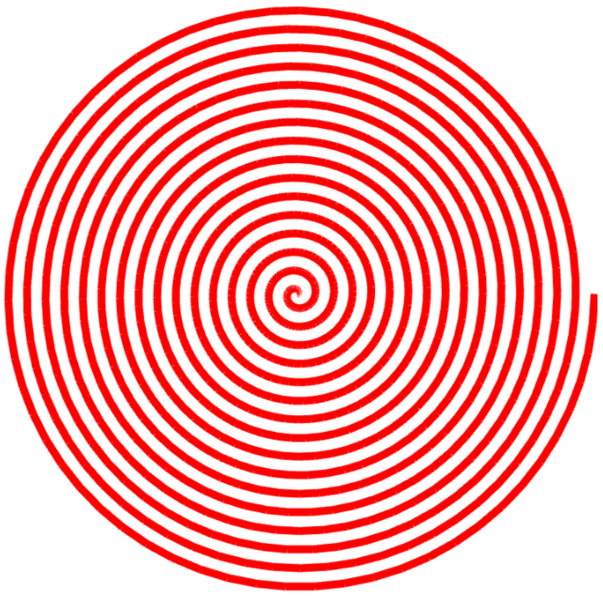
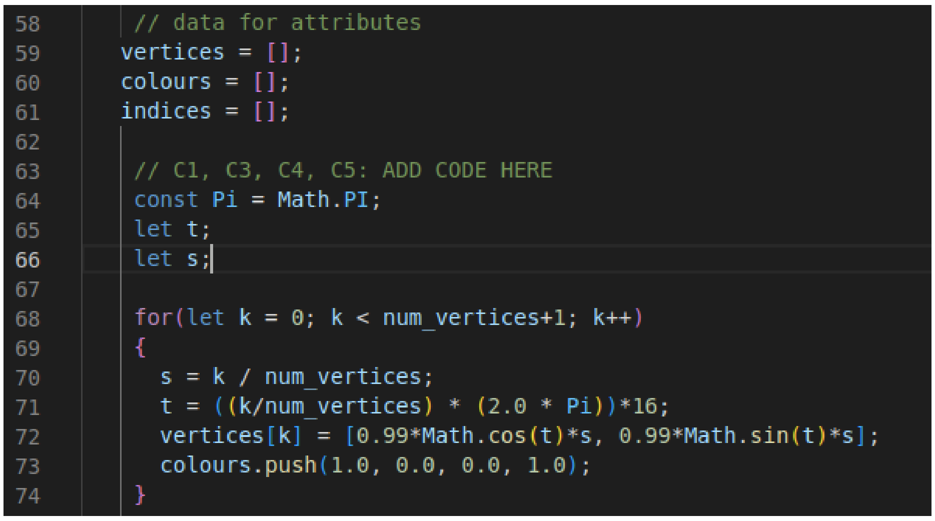


Figure 35: Spiral constructed with 0-1 scalar on x and y coordinates

JavaScript Code Snippet



```
    // data for attributes
    vertices = [];
    colours = [];
    indices = [];

    // C1, C3, C4, C5: ADD CODE HERE
    const Pi = Math.PI;
    let t;
    let s;

    for(let k = 0; k < num_vertices+1; k++)
    {
      s = k / num_vertices;
      t = ((k/num_vertices) * (2.0 * Pi))*16;
      vertices[k] = [0.99*Math.cos(t)*s, 0.99*Math.sin(t)*s];
      colours.push(1.0, 0.0, 0.0, 1.0);
    }
```

Figure 36: Snippet Code from 'colour-spiral.js' file

Explanation

For this exercise, I created a scalar which I multiply with the x and y coordinates. The scalar range is between 0-1 and increments the same number of vertices that exist. This gradually renders larger circles, creating a spiral effect as the x and y positions of each vertex is multiplied by this scalar.

I also multiplied the angle by 16. This is to create a sharper angle, so more spirals are rendered.

References
n/a

C4
Output

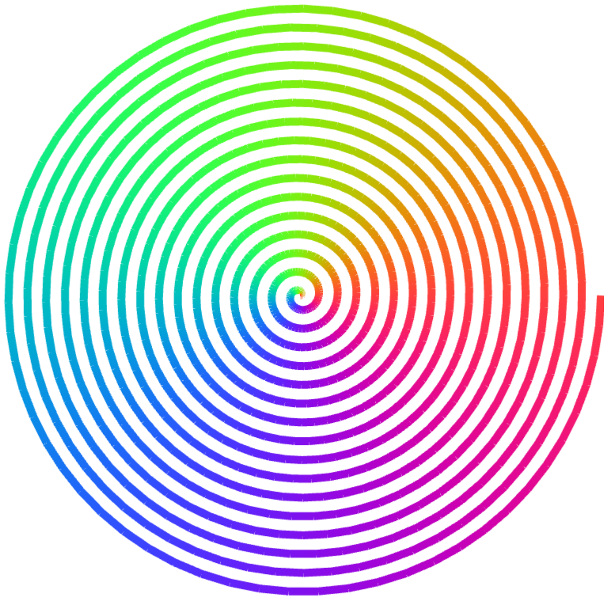
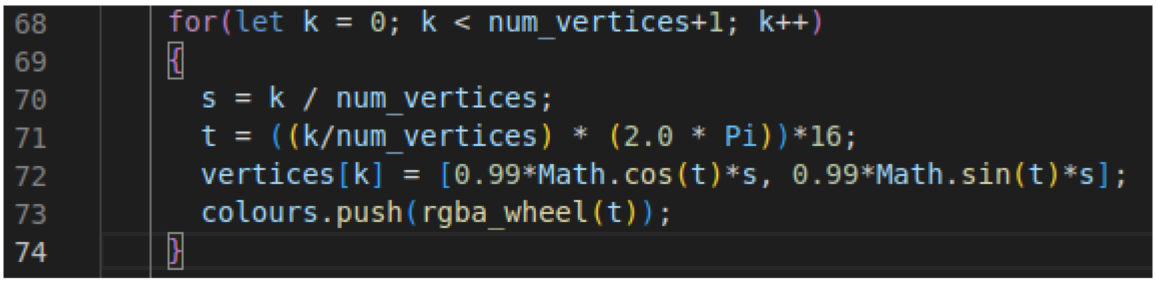



Figure 37: Uses RGBA_Wheel Function to generate rainbow colour

JavaScript Code Snippet

```
for(let k = 0; k < num_vertices+1; k++)
{
  s = k / num_vertices;
  t = ((k/num_vertices) * (2.0 * Pi))*16;
  vertices[k] = [0.99*Math.cos(t)*s, 0.99*Math.sin(t)*s];
  colours.push(rgba_wheel(t));
}
```
Figure 38: Code Snippet in 'colour-spiral.js' file

Explanation
For this exercise, I called the rgba_wheel(t) function inside the colours.push() function as shown in JavaScript Code Snippet . This returned specific cosine wave with offset by 120 degrees to render the colour scheme you see in output. This was then pushed to the colours array and then used in the bufferData() function.

References
https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/bufferData

C5

Output

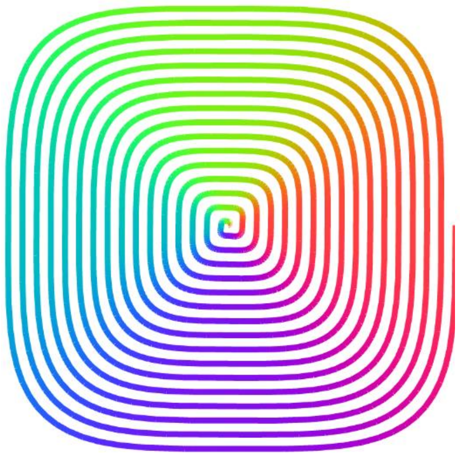


Figure 39: Spiral rendered with n = 4

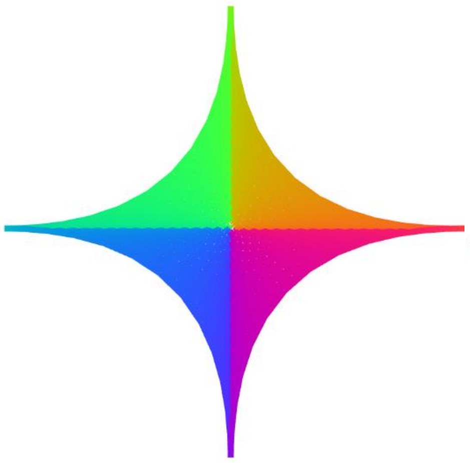


Figure 40: Spiral rendered with n = 0.5

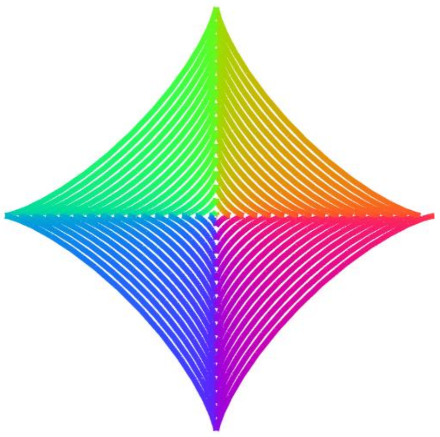


Figure 41: Spiral rendered with n = 0.75

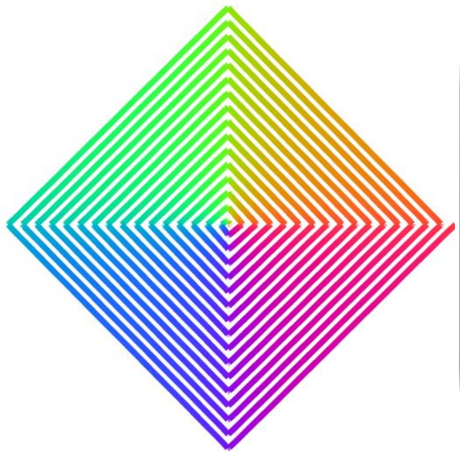


Figure 42: Spiral rendered with n = 4

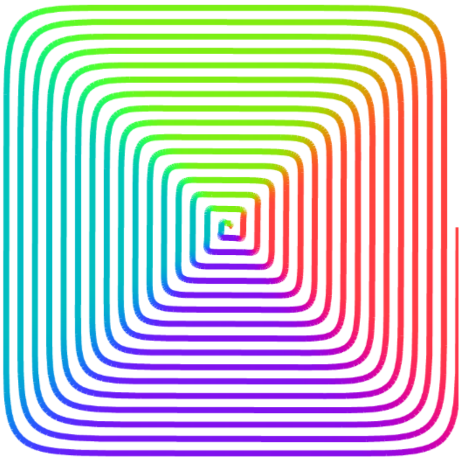


Figure 43: Spiral rendered with n = 10

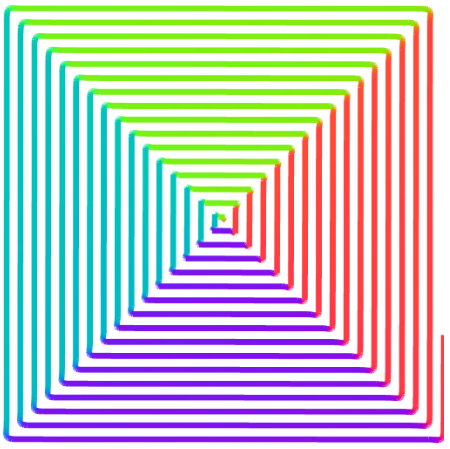
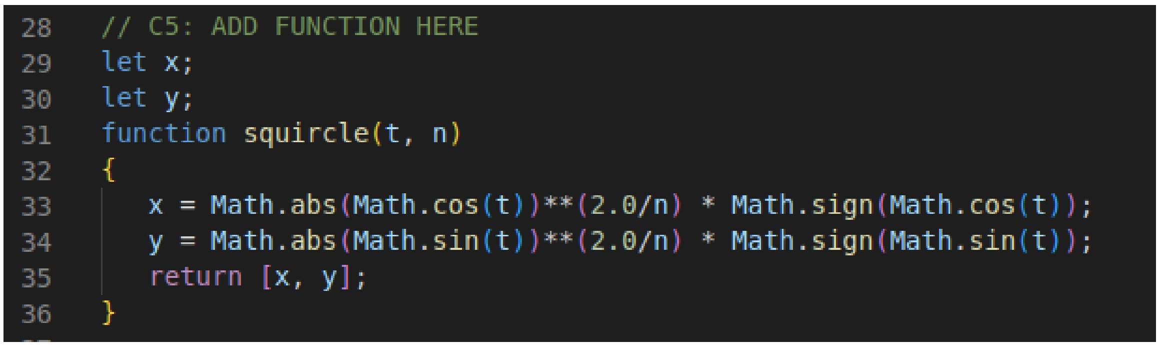


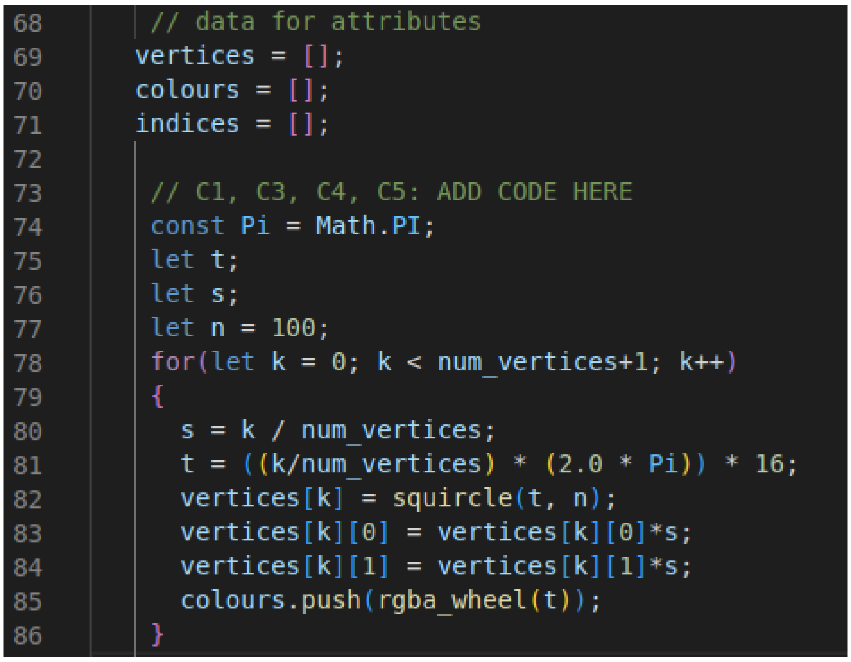
Figure 44: Spiral rendered with n = 100

JavaScript Code Snippet



```
// C5: ADD FUNCTION HERE
let x;
let y;
function squircle(t, n)
{
    x = Math.abs(Math.cos(t))**(2.0/n) * Math.sign(Math.cos(t));
    y = Math.abs(Math.sin(t))**(2.0/n) * Math.sign(Math.sin(t));
    return [x, y];
}
```

Figure 45: JavaScript Code Snippet from 'colour-spiral.js' file



```
68      // data for attributes
69     vertices = [];
70     colours = [];
71     indices = [];
72
73      // C1, C3, C4, C5: ADD CODE HERE
74      const Pi = Math.PI;
75      let t;
76      let s;
77      let n = 100;
78      for(let k = 0; k < num_vertices+1; k++)
79      {
80        s = k / num_vertices;
81        t = ((k/num_vertices) * (2.0 * Pi)) * 16;
82        vertices[k] = squircle(t, n);
83        vertices[k][0] = vertices[k][0]*s;
84        vertices[k][1] = vertices[k][1]*s;
85        colours.push(rgba_wheel(t));
86      }
```

Figure 46: JavaScript Code Snippet from 'colour-spiral.js' file

Explanation

For this exercise, I introduced the squircle(t, n) function as required. This takes two parameters t and n. t represents the angle, and n is a positive number. The squircle(t, n) function shapes the curve appropriately.

Then I scale both x and y by multiplying each coordinate by a s. As shown in lines 83 and 84.

Finally, I changed the n value, to get different patterns with n = 4 to get the pattern that is required.

References

https://en.wikipedia.org/wiki/Superellipse