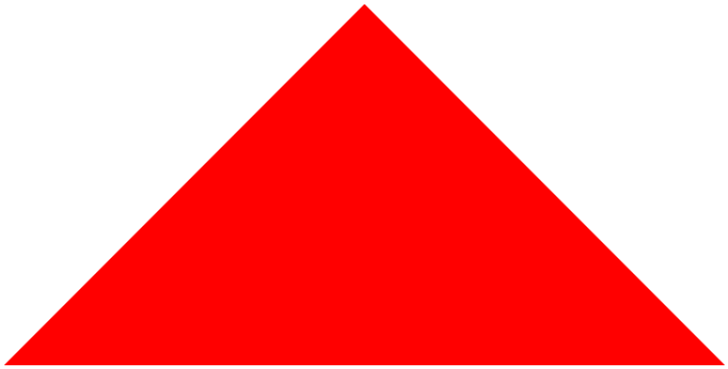
Computer Graphics Lab1

Full name (ID) : Erfan RafieiOskouei (240842587)

A1 — Modifying the drawing procedure



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

Changing num_strip_vertices to 3 in the gl.drawArrays(gl.TRIANGLE_STRIP, 0, num_strip_vertices) function causes only the first three vertices to be drawn, forming a triangle.

This demonstrates that the square is constructed from two triangles, and when only three vertices are rendered, only one of these triangles appears.

It highlights how the square is essentially a composition of triangular strips, a common approach in computer graphics for defining more complex shapes.

A2 — Modifying the fragment shader

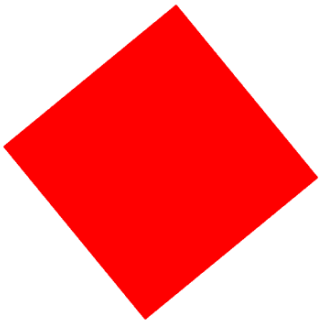

```
//gl_FragColor = vec4(0.0, 0.0, 1.0, 1.0);
vec3 red = vec3(v0: 1.0, v1: 0.0, v2: 0.0);
vec3 green = vec3(v0: 0.0, v1: 1.0, v2: 0.0);
vec4 add = vec4(v0: red + green, v1: 1.0);
gl_FragColor = add;
```

When you add the RGB vectors for red (1.0, 0.0, 0.0) and green (0.0, 1.0, 0.0) together, the result is the vector (1.0, 1.0, 0.0), which corresponds to the color yellow.

By converting this into an opaque RGBA vector, vec4(1.0, 1.0, 0.0, 1.0), and assigning it to gl_FragColor, the displayed shape will turn yellow.

This shows how combining basic colors (red and green) results in a new color, in this case, yellow, due to the additive nature of RGB color mixing.

A3 — Modifying the vertex shader



```
// A3: ADD CODE HERE

gl_Position.x *= (1.0 + s) / 2.0;
gl_Position.y *= (1.0 + s) / 2.0;
```

Introducing such lines in a vertex shader creates a pulsing or breathing effect on the rotating square appearing in that the square will seem to shine out and shrink as rotation occurs.

The change made is a scaling transformation, and the scale at any moment while it is changing is calculated using this formula: (1.0 + s)/2.0. The reason for this trigonometric manipulation is that "s" is the sine of the rotation angle so it varies from -1 to 1.

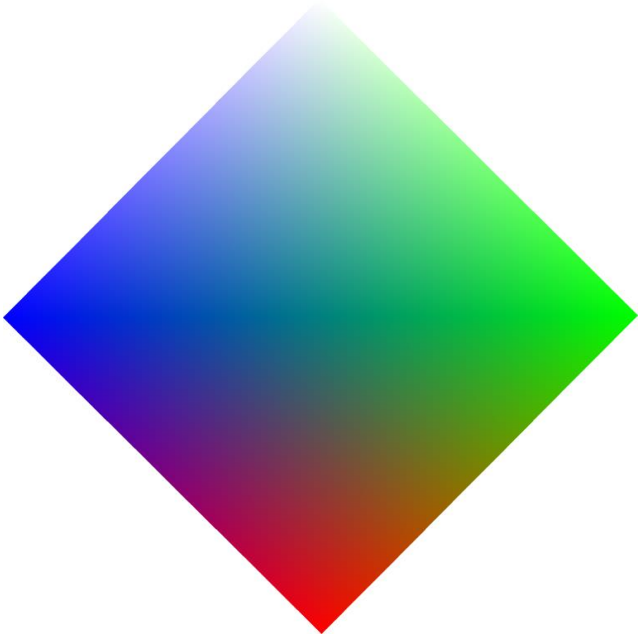
In other words,

When s=-1, the scaling factor, becomes 0 that is, in fact, shrinking the square to a point.

When s=1, the scaling factor becomes 1 so that the square gets back to its original size.

For any values of s in between, the square gets scaled to some fraction between 0 and 1.

A4 — Interpolating colours



the vertex shader now includes an attribute `vec4 colour` and a `varying lowp vec4 colour_var`. The colour attribute is passed from the JavaScript code, and `colour_var` is used to interpolate the color across the surface of the square. Inside the main function of the vertex shader, `colour_var` is set to `colour`, allowing the fragment shader to interpolate the colors between vertices.

In the fragment shader, the incoming vertex color is declared as a `varying lowp vec4 colour_var`, and the color setting is changed to `gl_FragColor = colour_var`. This setup ensures that the color is smoothly interpolated across the pixels, based on the four vertex attribute colors.

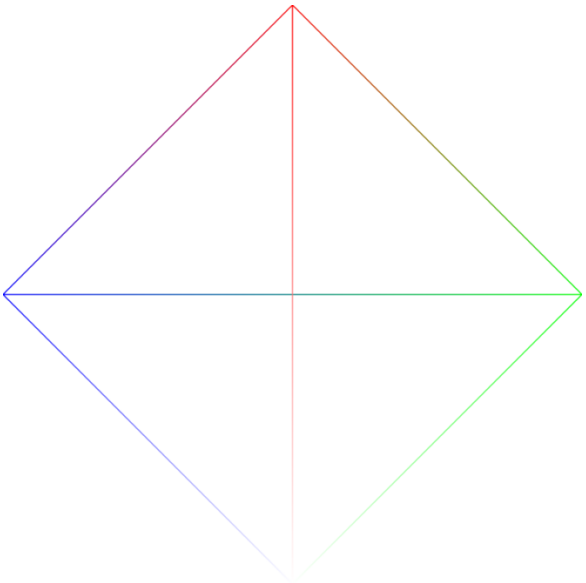
Finally, the attribute color data is connected in the render function with the following lines:

```
// connect vertex_colour attribute in shader to colour_buf
    gl.vertexAttribPointer(colour_loc, 4, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(colour_loc);
```

This enables the attributes and ensures the colors are correctly applied to the vertices.

You should see an image similar to the one provided, where the color is interpolated across the pixels, creating a smooth gradient effect based on the four vertex attribute colors.

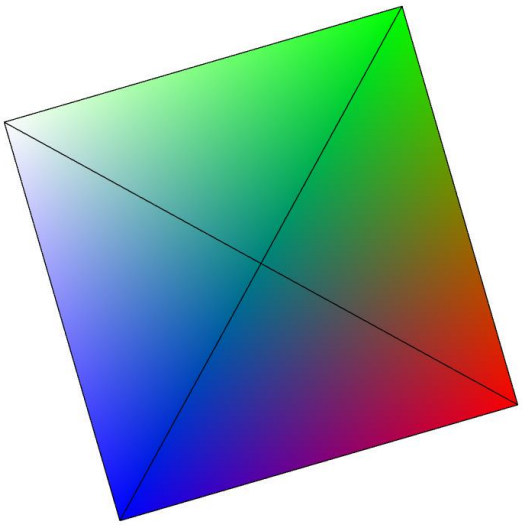
A5 — Vertex indexing



The indices array should be filled in as follows to draw all six lines of the square:

```
// A5: MODIFY BELOW
    indices = [
        0, 1,
        1, 2,
        2, 3,
        3, 0,
        0, 2,
        1, 3
    ];
```

A6 — Drawing an outline

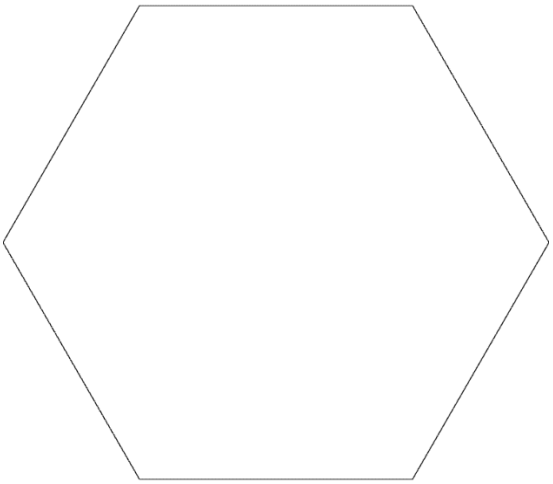


```
// A6: APPEND SIX BLACK VERTICES

for (let i = 0; i < 6; i++) {
    colours.push([0.0, 0.0, 0.0, 1.0]);
}
```

```
// connect vertex_colour attribute in shader to colour_buf

gl.vertexAttribPointer(colour_loc, 4, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(colour_loc);



// draw triangle strip

let num_strip_vertices = vertices.length;
gl.drawArrays(gl.TRIANGLE_STRIP, 0, num_strip_vertices);


// gl.drawArrays(gl.TRIANGLE_STRIP, 0, 3);


let black_offset = vertices.length * 4 * 4;


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

My guess is that the lines appear constant in color because the interpolation occurs over very short distances between vertices that have been assigned the same color value. This means there is no visible gradient along the length of each line segment, making the lines appear as solid colors.

B1 — Drawing a hexagon



```
vertices = [];
// B1: ADD CODE HERE

indices = [];
// B1: ADD CODE HERE


const num_vertices = 6;

// Generate vertices
for (let k = 0; k <= num_vertices; k++) {
    let t = (k / num_vertices) * 2.0 * Math.PI;
    let P = [Math.cos(t), Math.sin(t)];
    vertices.push(P);
}

// Generate indices for gl.LINE_STRIP
for (let i = 0; i < num_vertices; i++) {
    indices.push(i);
}
indices.push(0); // Close the hexagon by connecting the last vertex to the first
```

To begin, I chose the number of vertices (num_vertices) of our hexagon which was six.

Then we look for coordinates of those vertices on that unit circle. A unit circle is a circle in a plane whose center is at the origin, (0,0) and a radius of 1 unit.

For any vertex, an angle t is calculated that indicates what position on the circle the vertex will be in. This angle runs from zero to basically two pi where full circle is subtended.

With the angle t determined, we then find the **x** and **y** coordinates of the vertex by applying the cosine and sine functions respectfully.

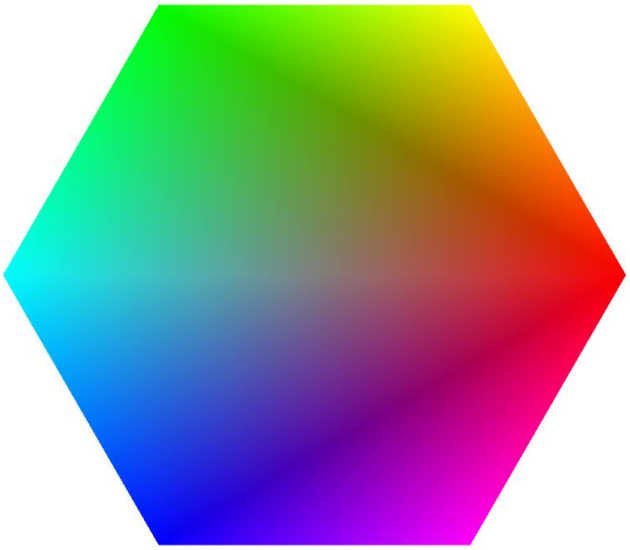
This involves storing the vertices coordinates in the vertex's arrays.

Then we connect the vertices in the order which the hexagon will be drawn. This is where the indices array comes in.

The indices array does the function of connecting the vertexes to create the hexagon.

Finally, I added the first vertex index again at the end of the indices array to close the hexagon, making sure the last vertex connects back to the first one.

B2 — Colouring the hexagon



```
// Colours for each vertex
colours = [
    [1.0, 0.0, 0.0, 1.0], // Red
    [1.0, 1.0, 0.0, 1.0], // Yellow
    [0.0, 1.0, 0.0, 1.0], // Green
    [0.0, 1.0, 1.0, 1.0], // Cyan
    [0.0, 0.0, 1.0, 1.0], // Blue
    [1.0, 0.0, 1.0, 1.0], // Magenta
    [1.0, 0.0, 0.0, 1.0]  // Red again to close the loop
];
```

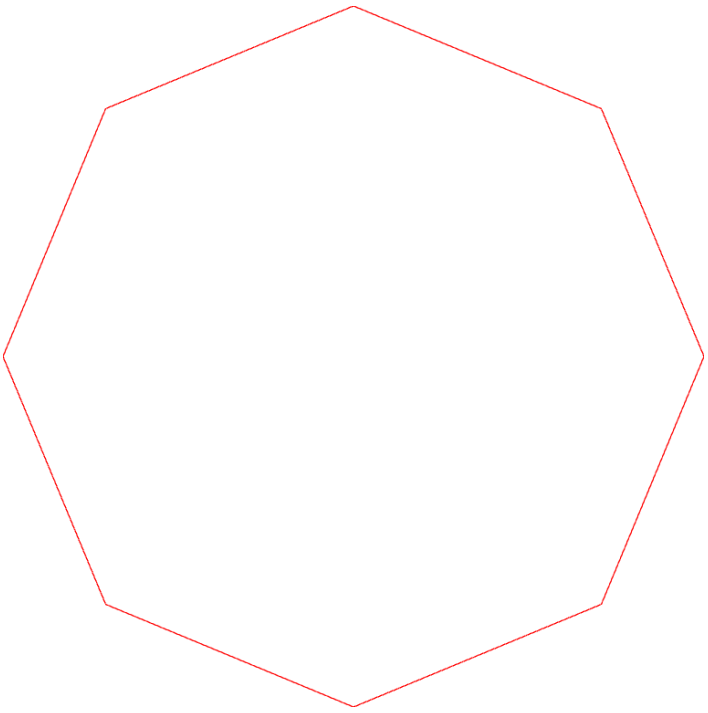
```
// B2: MODIFY CODE HERE
//gl.drawElements(gl.LINE_STRIP, num_line_vertices, gl.UNSIGNED_SHORT, 0);

gl.drawElements(gl.TRIANGLE_FAN, indices.length, gl.UNSIGNED_SHORT, 0);
```

In this part we want each vertex of the hexagon to have a specific color, going anticlockwise from the right-most vertex. So I defined the colors in the colours array in the order: red, yellow, green, cyan, blue, magenta.

Then I changed the drawing mode to TRIANGLE_FAN. This mode connects the center vertex to each of the outer vertices, forming triangles.

The gl.drawElements function uses the indices array to draw the hexagon.

C1 — Drawing an octagon



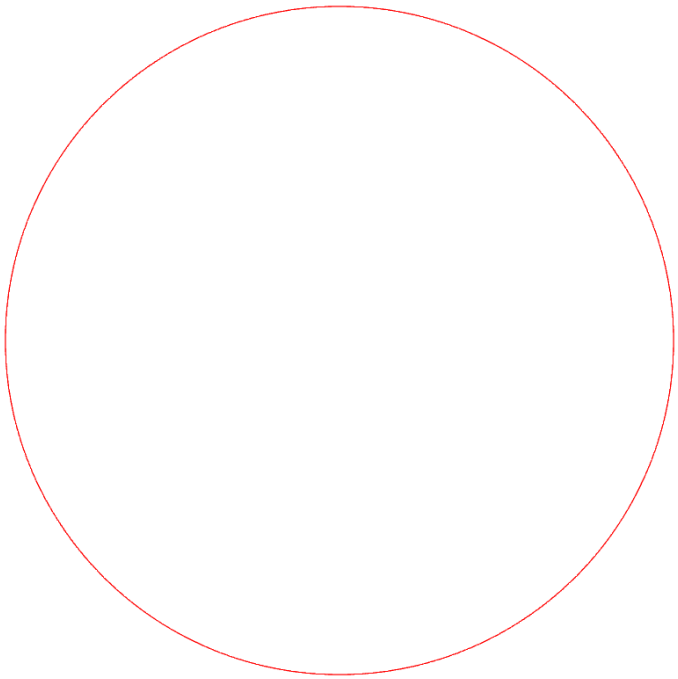
Added code :

```
// data for attributes

    vertices = [];

    colours = [];

    for (let i = 0; i < num_vertices+1 ; i++)
{
        colours.push([1.0, 0.0, 0.0, 1.0]);
    }


    indices = [];
    // C1, C3, C4, C5: ADD CODE HERE
```

```
    // Generate vertices
    for (let k = 0; k <= num_vertices; k++) {
     let t = (k / num_vertices) * 2.0 * Math.PI;
     let P = [0.99 * Math.cos(t), 0.99 * Math.sin(t)];
     vertices.push(P);
}

// Generate indices for gl.LINE_STRIP
for (let i = 0; i < num_vertices; i++) {
    indices.push(i);
}
indices.push(0); // Close the octagon by connecting the last vertex to the first
```

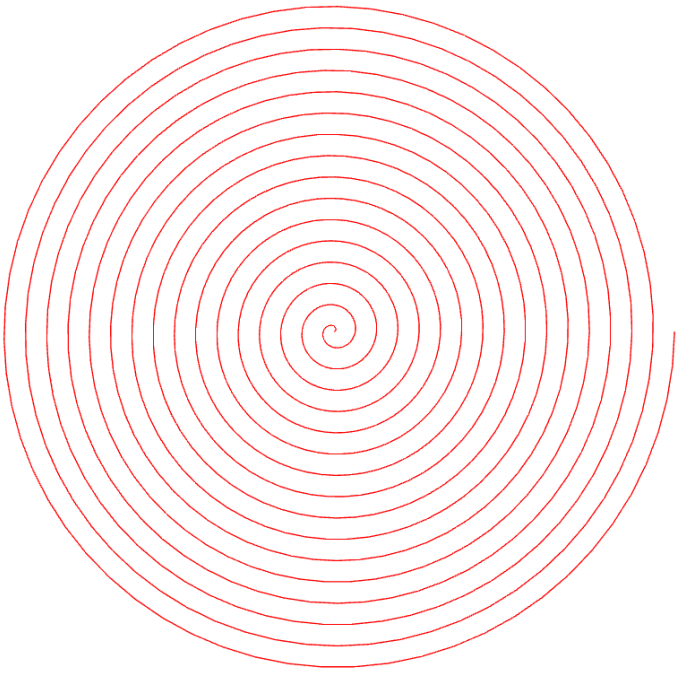
What I did in this part, was that I added the same code from the B section of the lab and the modifications that I made was to change the value of num_vertices from 6 to 8 for octagon shape and I added a for loop to create the colors arrays for drawing lines and the remaining stuff are the same as it goes.

C2 — Drawing a circle



For this part the only change that I made was to increase the value of num_vertices from 8 to 1000 which kind of resembles the circle because of the large number of vertxes.

C3 — Drawing a spiral



Code :

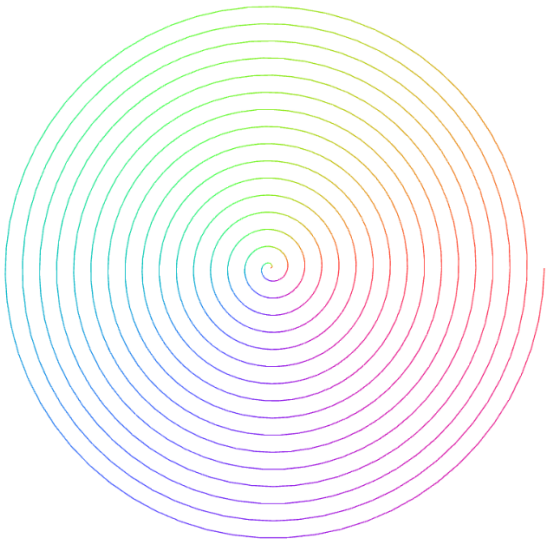
```
for (let k = 0; k <= num_vertices; k++) {
       let cycles = 16; // 16 cycles for the spiral
       let s = k / num_vertices; // scale parameter from 0 to 1
       let angle = cycles * 2 * Math.PI * s;   // Angle increases faster for spiral
       let t = cycles * 2.0 * Math.PI * s; // 16 cycles
       let x = 0.99 * s * Math.cos(t);
       let y = 0.99 * s * Math.sin(t);
       vertices.push(x,y);
   }
```

Here are the key changes I made:

1. Added a scale parameter s that goes from 0 to 1 as k goes from 0 to num_vertices.
2. Added the angle parameter to go 16 times faster, covering 16*2π by the end of the loop.
3. Added the x and y parameter for the scale from 0 to 1
4. And pushed the x and y to the vertices

This modification will create a spiral that starts from the center (when s = 0) and grows outward, completing 16 full rotations before reaching its maximum radius (0.99) at the edge of the canvas.

C4 — Colouring the spiral



Code :

```
for (let k = 0; k <= num_vertices; k++) {
        let cycles = 16; // 16 cycles for the spiral
        let s = k / num_vertices; // scale parameter from 0 to 1
        let angle = cycles * 2 * Math.PI * s;   // Angle increases faster for spiral
        let t = cycles * 2.0 * Math.PI * s; // 16 cycles
        let x = 0.99 * s * Math.cos(t);
        let y = 0.99 * s * Math.sin(t);
        vertices.push(x,y);

        /**
         * Generates a color based on the given angle using the RGBA color wheel.
         *
         * @param {number} angle - The angle in degrees used to determine the color.
         * @returns {string} The RGBA color string corresponding to the given angle.
         */

        let color = rgba_wheel(angle);
        colours.push(...color);

    }
```

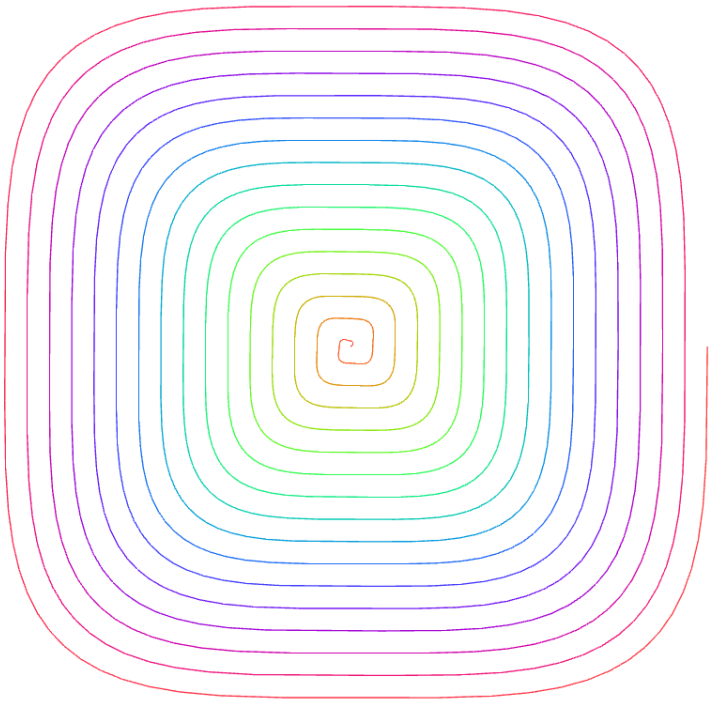
Here are the key changes that I've made:

1. I've removed the separate loop for generating colors.
2. Inside the main loop where we generate vertices, I also added the generating of colors using the rgba_wheel) function.
3. Then I pass the angle to rgba_wheel() to create the full color cycle.

This modification will create a spiral that starts from the center and grows outward, completing 16 full rotations. The color will change continuously along the spiral, creating a rainbow effect that completes one full cycle from the start to the end of the spiral.

The color at each point of the spiral will be determined by its angle, not its distance from the center. This means that each arm of the spiral will have a consistent color gradient from the center to the edge.

C5 — Drawing a squiral



Code :

```
function squircle(t, n) {
        let x = Math.pow(Math.abs(Math.cos(t)), 2/n) * Math.sign(Math.cos(t));
        let y = Math.pow(Math.abs(Math.sin(t)), 2/n) * Math.sign(Math.sin(t));
        return [x, y];
    }


    for (let k = 0; k <= num_vertices; k++) {
        let cycles = 16; // 16 cycles for the spiral
        let s = k / num_vertices; // scale parameter from 0 to 1
        let angle = cycles * 2 * Math.PI * s;   // Angle increases faster for spiral
        let t = cycles * 2.0 * Math.PI * s; // 16 cycles
        let x = 0.99 * s * Math.cos(t);
        let y = 0.99 * s * Math.sin(t);
        // Use squircle function to generate vertex position
        let P = squircle(t, 100);
```

```
        // Scale and apply the spiral effect
        vertices.push([0.99 * s * P[0], 0.99 * s * P[1]]);
        /**
         * Generates a color based on the given angle using the RGBA color wheel.
         *
         * @param {number} angle - The angle in degrees used to determine the color.
         * @returns {string} The RGBA color string corresponding to the given angle.
         */
        let color = rgba_wheel(angle);
        colours.push(...color);
    }
```

This code will create a **squiral** a spiral that follows the shape of a squircle. The squircle(t, n) function implements the parametric equations, where:

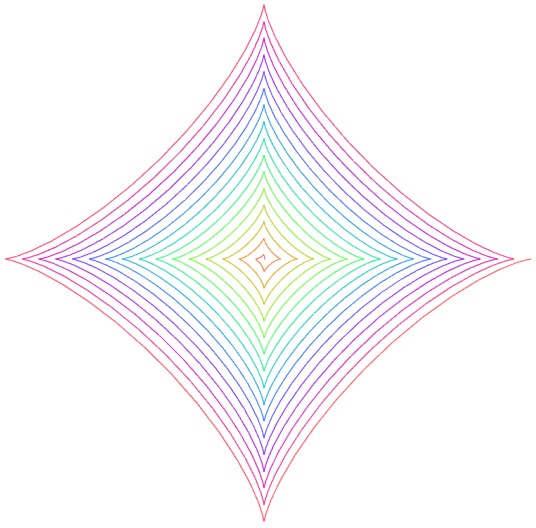
- t is the angular parameter
- n is the shape parameter that determines how "square" or "circular" the shape is

We can adjust the shape by changing the second parameter in the squircle(t, 4) call. Different values of n will affect the shape:

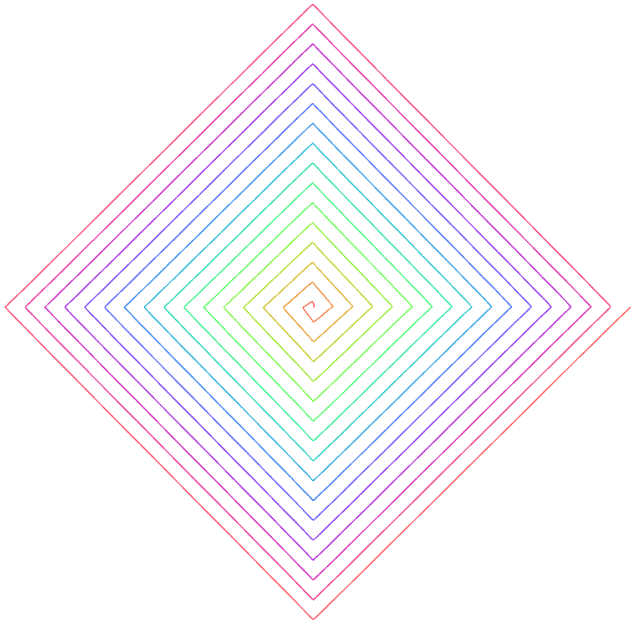
- n = 2: a perfect circle
- n = 4: This is the standard squircle, halfway between a square and a circle
- n < 2: a star-like shape with concave sides
- n > 4: This will make the shape more square-like
- n → ∞: As n approaches infinity, the shape will become more and more like a square.

Alternatively, we could test the different numbers to see how they effect the shape :

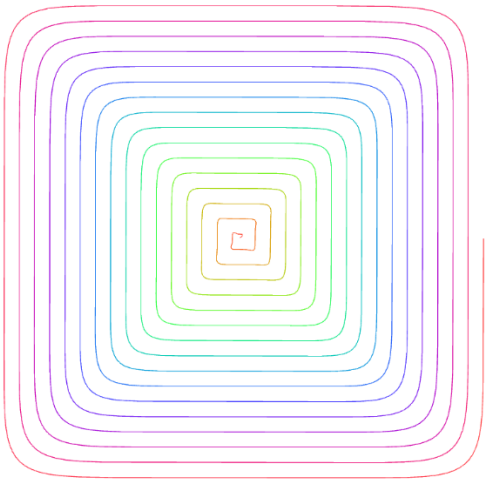
n = 0.75 :



n = 1 :



n = 10 :



n = 100 :