CG lab 1

Name: James Davies (200682354)

October 23, 2023

All rendered images included at the end of report.

1 Exercise A

1.1 A0

This image shows the starting point for the animation, the default being theta_step variable equal to 0.0. The drop down menu has four options, with pause being the 'default' value associated with the default theta_step value. All of these options in the menu are chosen via a switch statement in rotating-square.js. The actual rotation occurs via addition of the step value to the rotation angle (also beginning at value 0.0), or 'theta += theta_step'. This updating of the rotation value in theta occurs every time the frame is rendered. This value is then passed to the vertex shader via the API function call below this addition (gl.uniform1f()). Within rotating-shape-vert.glsl, the sine and cosine values of theta are calculated and then used to set x and y values, which dictate the actual rotating of the square.

1.2 A1

Within the JS render function, the changing of the number of strip vertices from 4 (length of array) to 3, shows that the square is actually made up of two triangles, or a triangle mesh. These two triangles have 2 shared vertices, so when the number of vertices passed to the drawArrays function is reduced by one, then we are unable to render the full square. The effect is a halving of the rendered shape, i.e. square to a triangle.

1.3 A2

In the fragment shader, we can declare two global variables (red/green) of vec3 type. Within the main method we can then initialize them as red and green within the RGB vector representation. Then further assigning vec4(red + green, 1.0) to the gl_FragColour to specify the final colour information.

1.4 A3

After adding the two lines within the vertex shader, the effect is the square beginning at half original size, then oscillating between full size to nothing whilst still rotating. An affine transformation is performed, as lines and parallelism are preserved, but distance is not as the effect is periodic dilation and expansion.

1.5 A4

To create the effect of smooth colour transitions when shading rendered primitives, we need to use varyings within the shaders. First creating a colour variable in the form of a vec4 type, and then creating a varying of vec4 type. Within main() in the vertex shader we can now set the variable colour_var = colour, which allows for the current vertex to contribute to the varying. Within the fragment shader, we can then declare/initialize another vec4 varying colour_var and set the colour setting FragColor to use colour_var as input. The attribute colour data isn't connected yet, and we must use the vertexAttribPointer API function to bind the buffer colour_buf to the vertex shader attribute.

1.6 A5

By changing the rendering mode from TRIANGLE_STRIP to LINES/drawArrays to drawElements, whilst also keeping the same colour interpolation from A4, we create a rendering of the outline of the shape rather than 2 fully shaded primitives. The given indices array creates just one line, and then filling in the rest of the data (0, 2, ...) ensures that the rest of the shapes outline is rendered rather than the one line.

1.7 A6

Extending the exercise from A5, we can colour in the vertices black by adding RGBA data to the colour array and specifying a variable (black_offset) to pass to vertexAttribPointer() to set where the shader should actually start from in the array. This offset is the number of bytes (in this case 64), specified by number of vertices * length of the RGBA vector given * number of bytes in float32. Each of these values is 4, so we have 4*4*4=64.

2 Exercise B

2.1 B1

The generation of the vertices[] array is via a for loop, which first states that we will be iterating over the loop 6 times, from k=0 to 5 with the condition being 0 less than num_vertices. We then create and initialize a variable t to define each angle as we move around the rendered shape (1/6, 2/6, ...). The angular variable t can then be used to specify points in the clip space using the cosine/sine of this angle, by calling the Math module and .cos() and .sin() respectively. As we are using LINE_STRIP to pass to .drawElements, we have to also set duplicate data within the indices data.

2.2 B2

In B2 we change the rendering from an outline to a fully coloured shape, via using TRIANGLE_FAN to pass to .drawElements(). This ensures that using the shaders we can interpolate a colour spectrum across the 6 rendered primitives. In order to render these primitives we have to first change the number of vertices from 6 to 7, and specify a starting point within the clip space. In vertices[0] we can initialize an x/y vector of 0, 0 for centering. The values within the for loop used to create angular/positional data can then be adjusted to ensure that we are still rendering a hexagon (i.e number of vertices still 6, with an added vertex in the centre of the shape). TRIANGLE_FAN can then be utilised properly, as we create the indices array and subsequently create the 6 primitives.

3 Exercise C

3.1 C1

See B1 for explanation of drawing code within for loop. Within the indices array, as we are using .drawArrays() and .LINE_STRIP, we have to take into account that the final set of indexed data needs to be duplicated to finish the rendered shape. In this case we have to set the indices between vertex 0 and 1 twice. In order to set up for the vertex colouring in C4, we can streamline the vertex colouring early on, and use colours.push([1.0, 0.0, 0.0, 1.0]) to specify that for each vertex rendered we can colour it red. The code within the .push() function can then be swapped out for the RGBA colour wheel function.

3.2 C2

Increasing the vertex loop to include 1000 vertices instead of 8, creates the appearance of a continuous line rather than individual and discrete vertices; a circle is created.

3.3 C3

From the rendering of the circle we can now nest the for loop creating the vertices of the rendered shape within a while loop. The condition of this while loop uses the scale parameter s (!while s = 1), which alongside ”s += 0.001” at the end of the loop ensures that we sweep from 0 to 1 as specified in the requirements. The actual use of this scale parameter is to multiply each of the x, y coordinates during their initialization which ensures the spiral begins at 0 (centre of clip space) and ends at the edge of the canvas (0.99). To ensure we have 16 cycles, within the initialization of the variable t we can use k/numvertices * 2*pi*16.

3.4 C4

We now use the function rgba_wheel() to colour the spiral we have just created. As the function returns a RGBA vector, we can pass the function directly to the JS array function .push(). This allows for RGBA colouring during the rendering of each of the vertices within the loop.

3.5 C5

The 'squircle' exercise involves manipulating the spiral from C3/C4 and uses parametric equations to create a curved square, a type of superellipse. We do this by creating a new function which takes an angular parameter (t), and n which serves as the exponent for the curve. This function then returns a vector/array to serve as positional coordinates. The pseudocode given in the lab sheet is converted to functional JS via multiple function calls to the Math module, specifically: .abs(), .cos(), .sin(), .sign() and .pow().

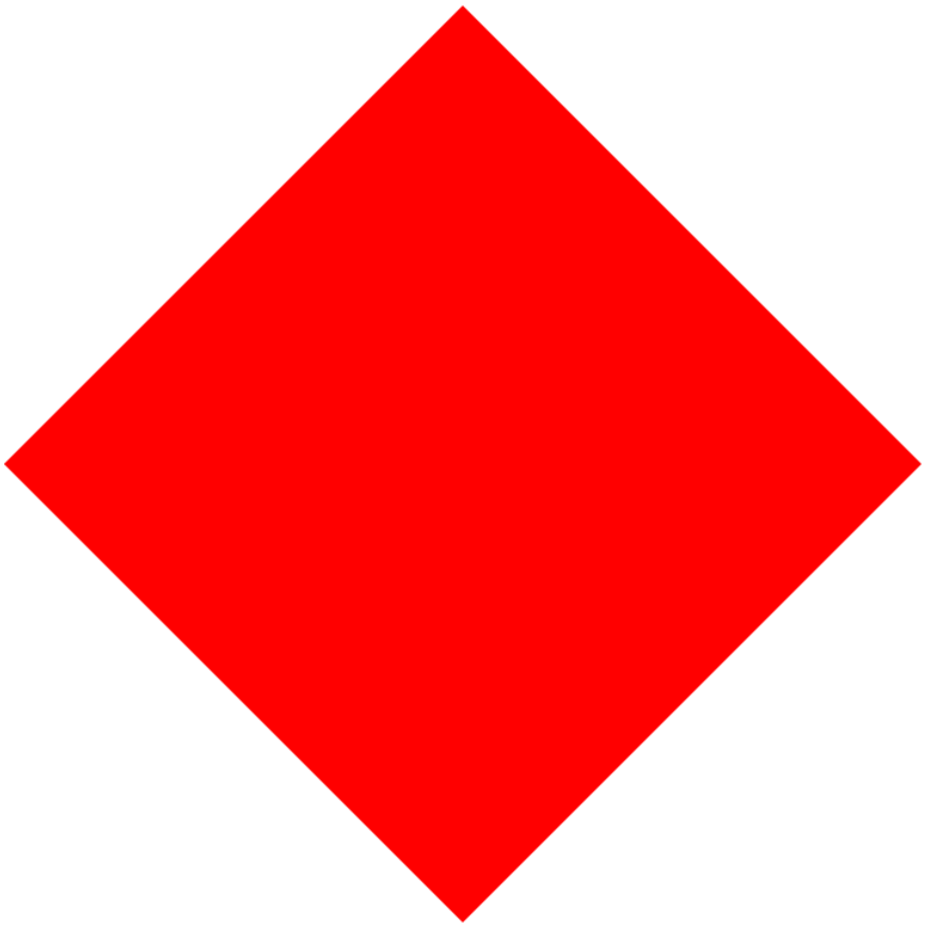

Figure 1: A0

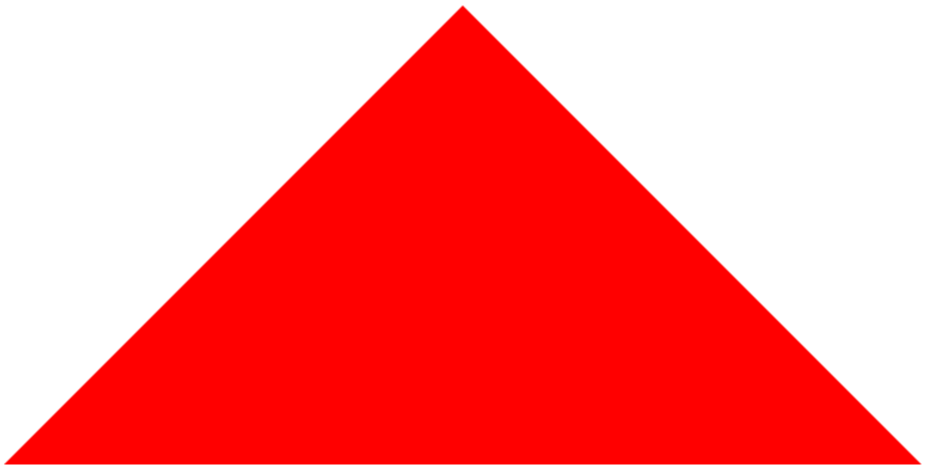

Figure 2: A1

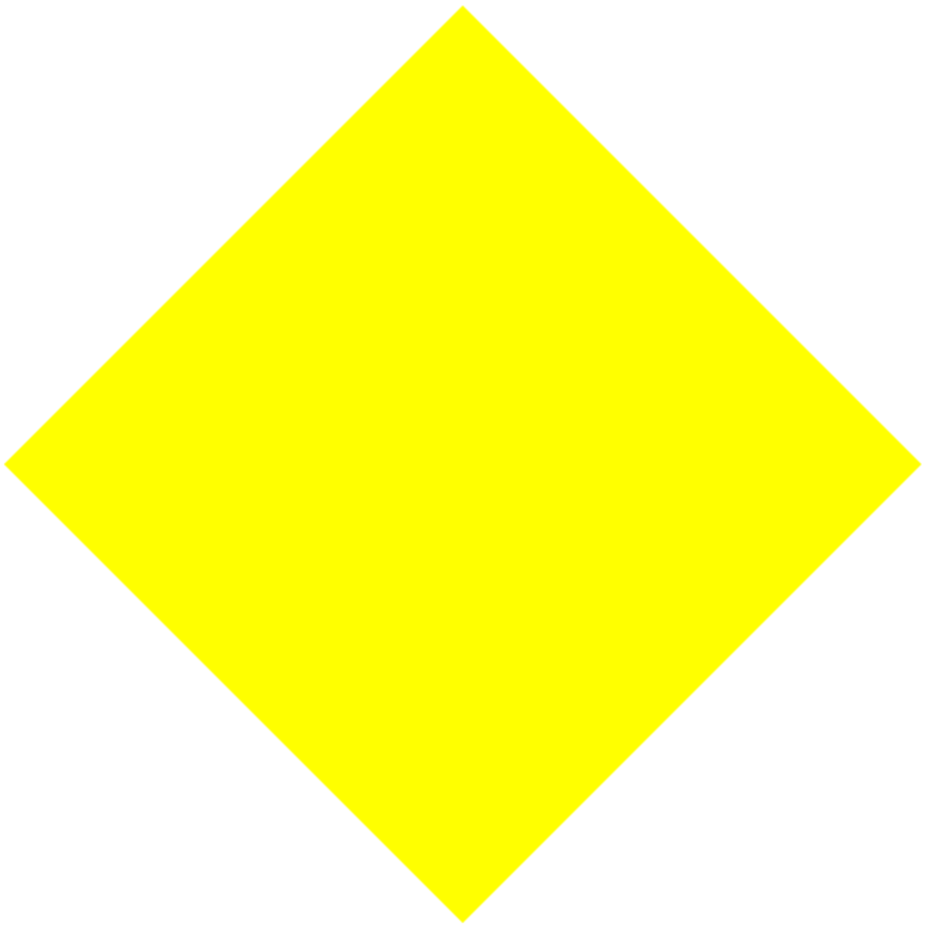

Figure 3: A2

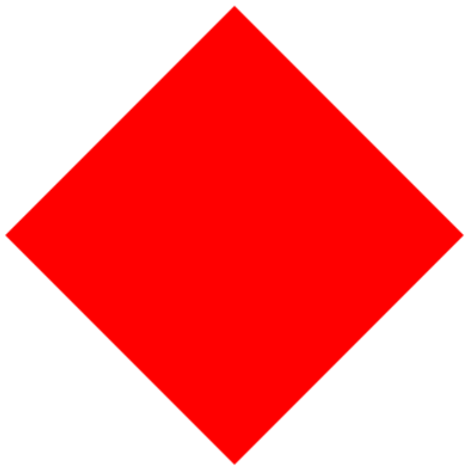

Figure 4: A3

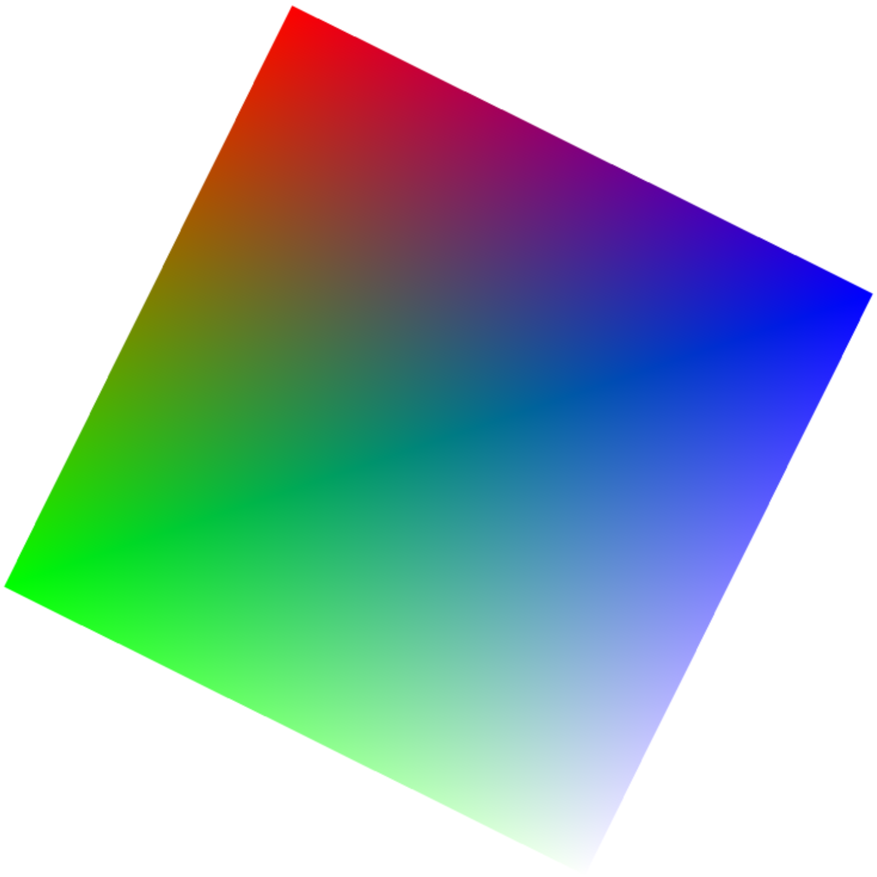

Figure 5: A4

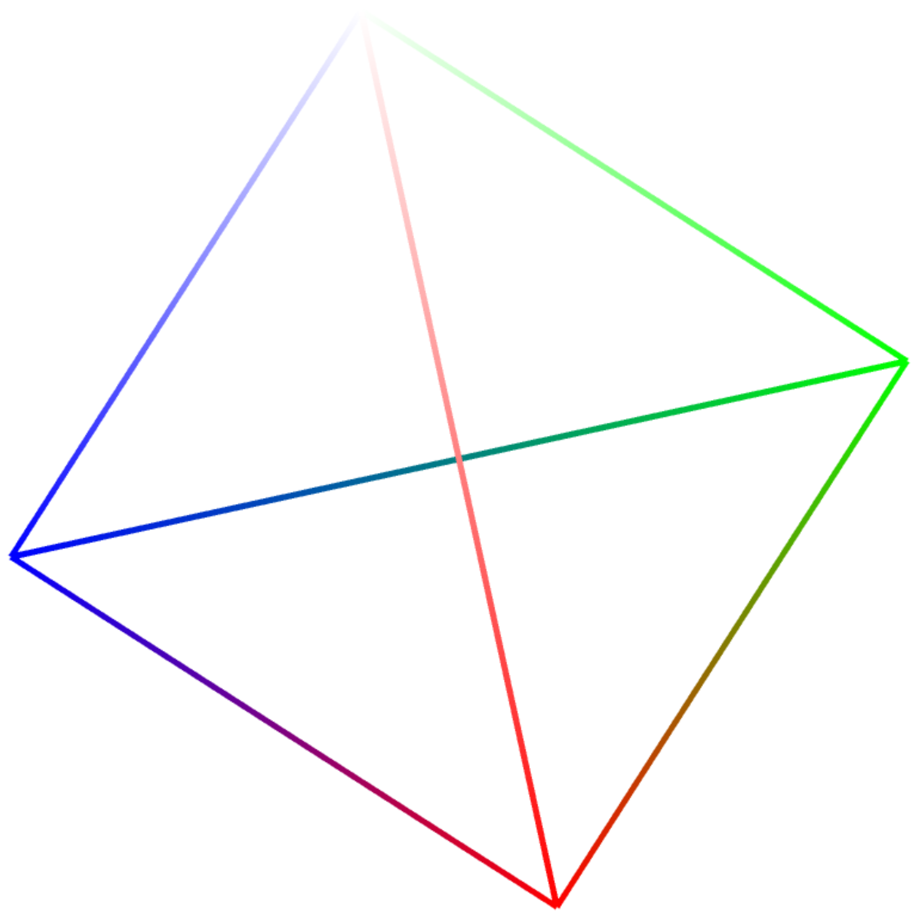

Figure 6: A5

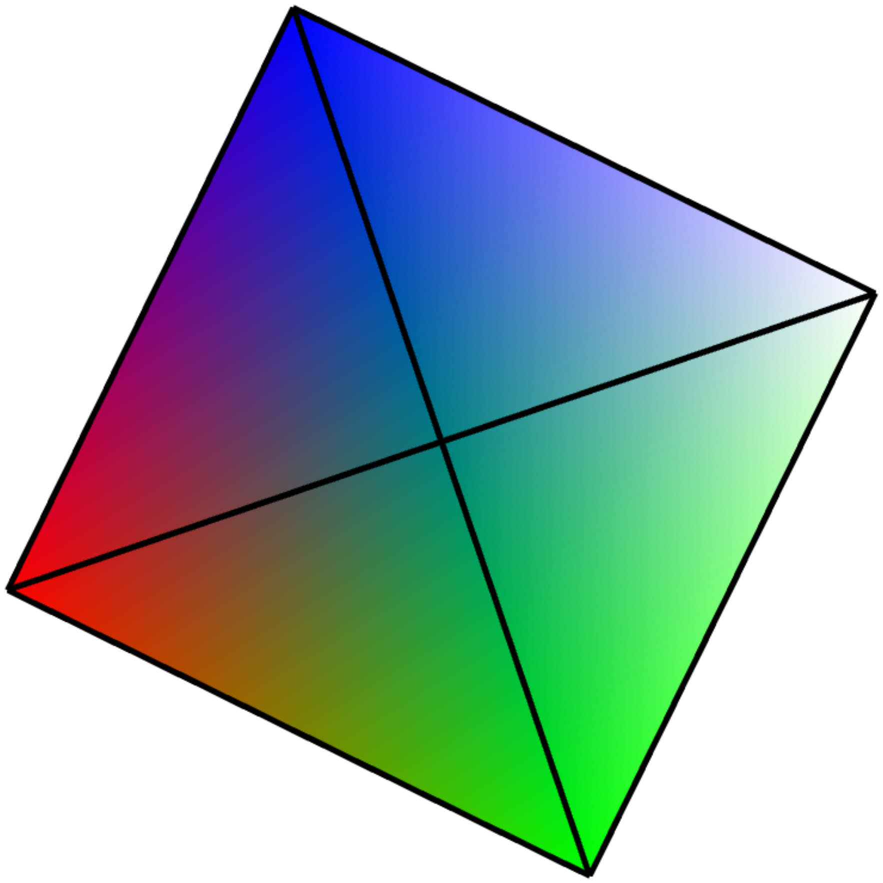

Figure 7: A6

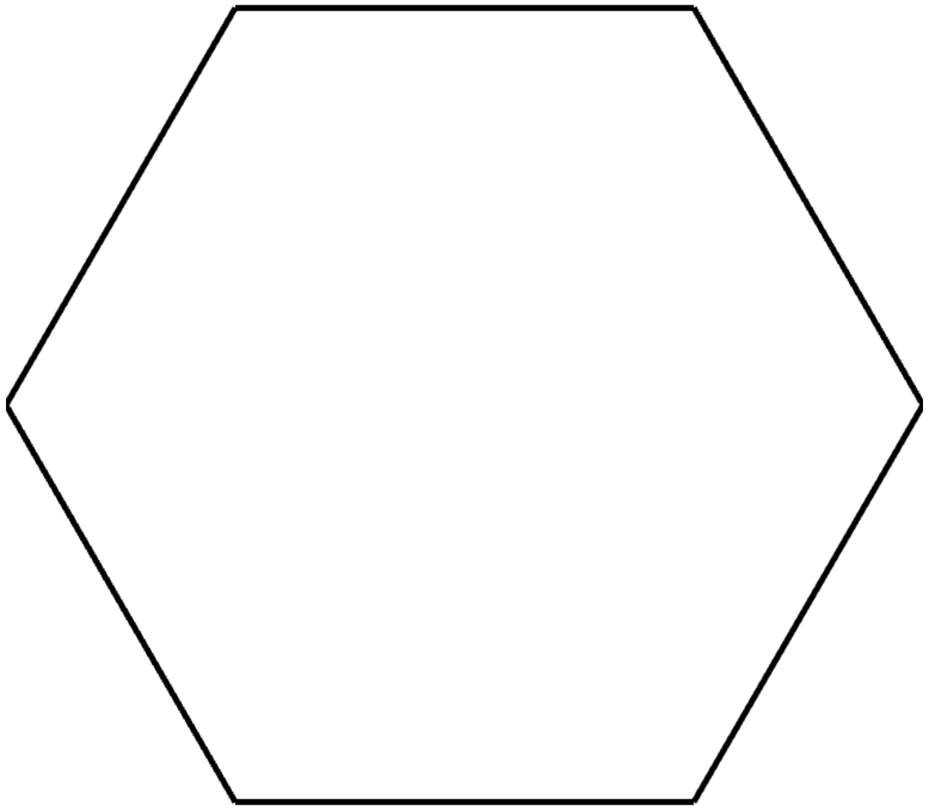
Figure 8: B1

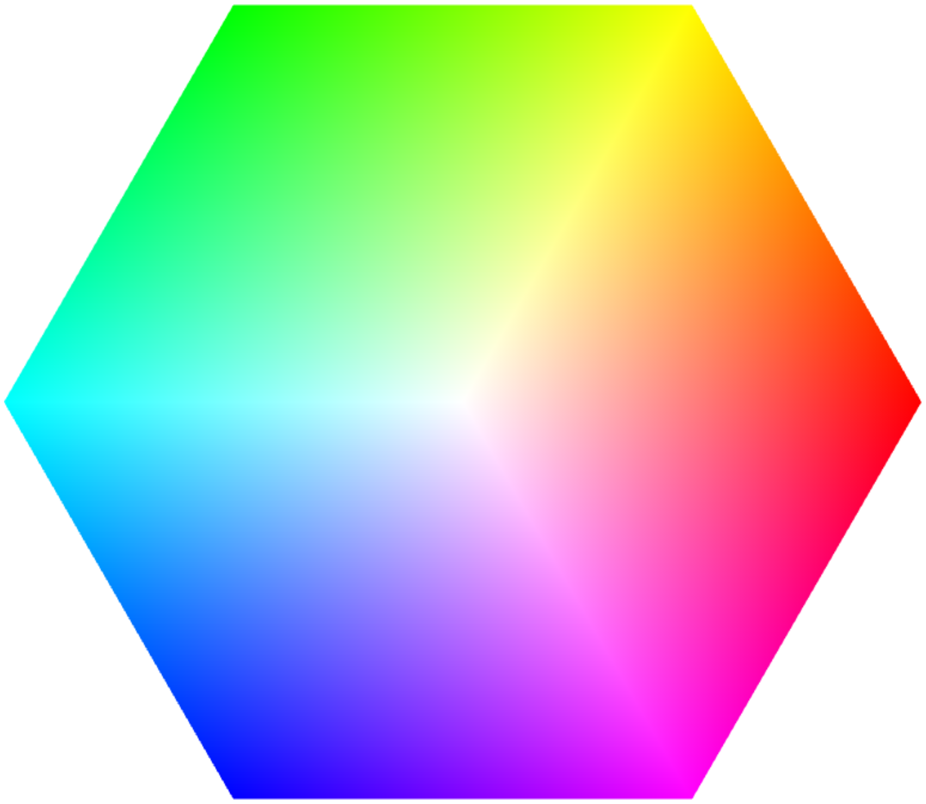

Figure 9: B2

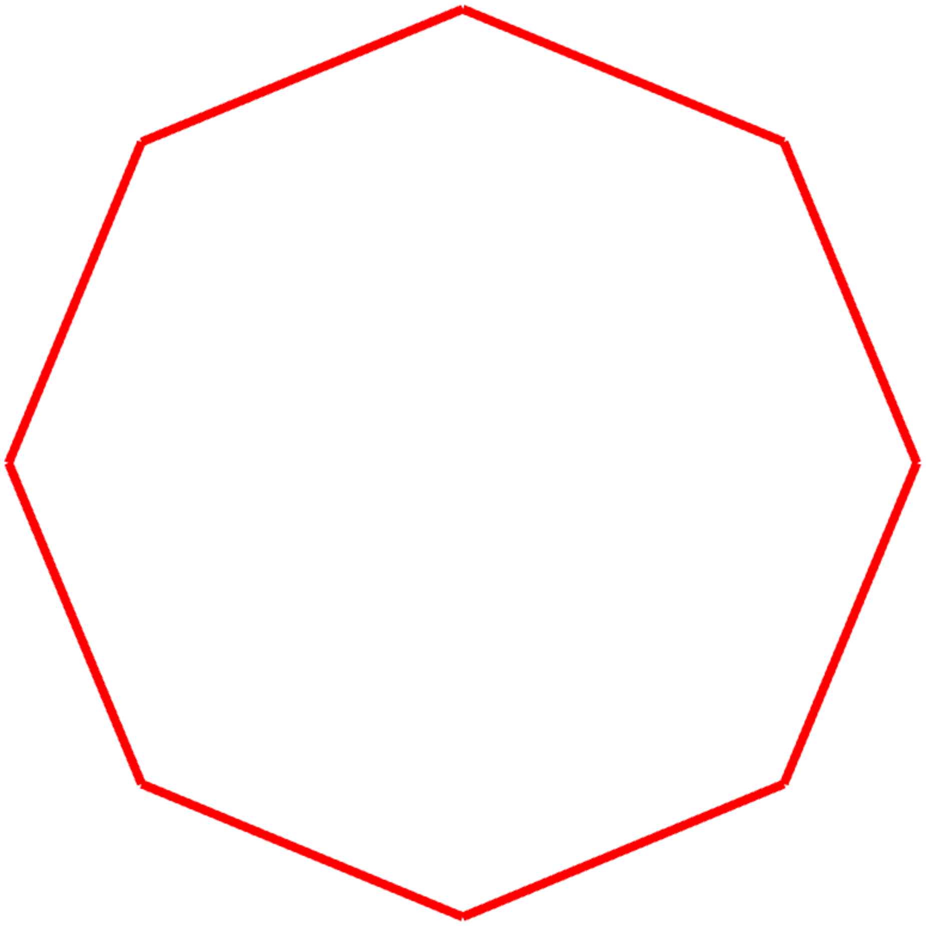
Figure 10: C1

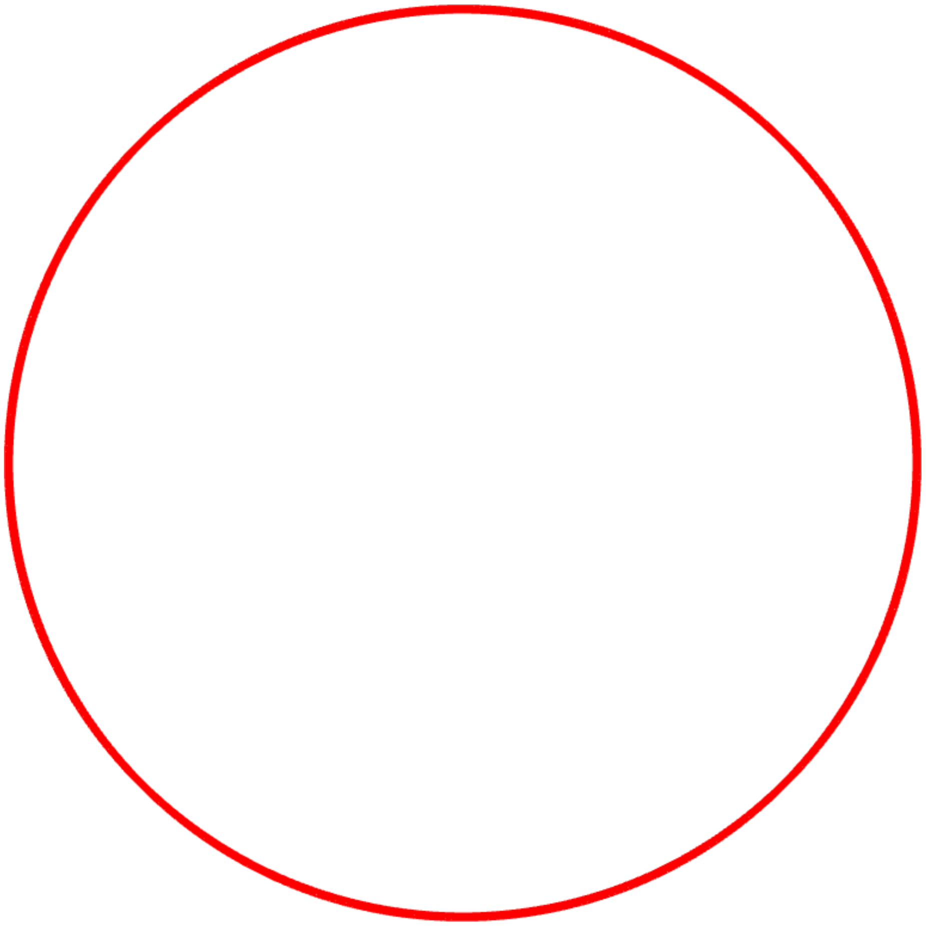

Figure 11: C2

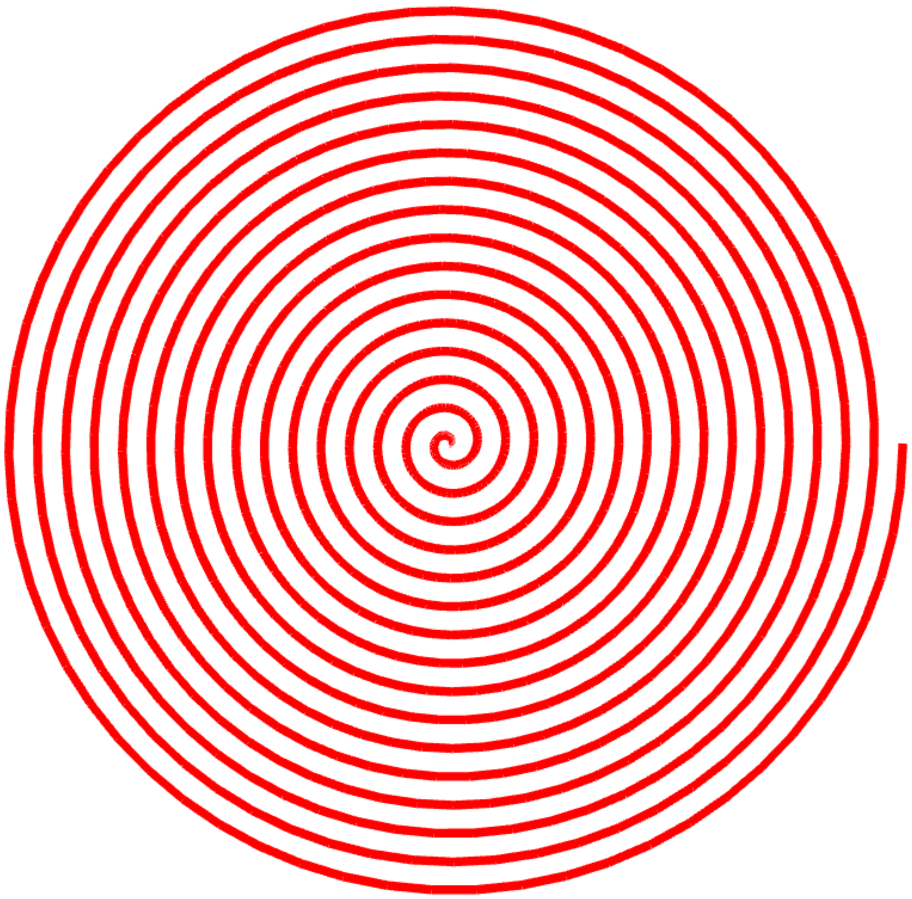

Figure 12: C3

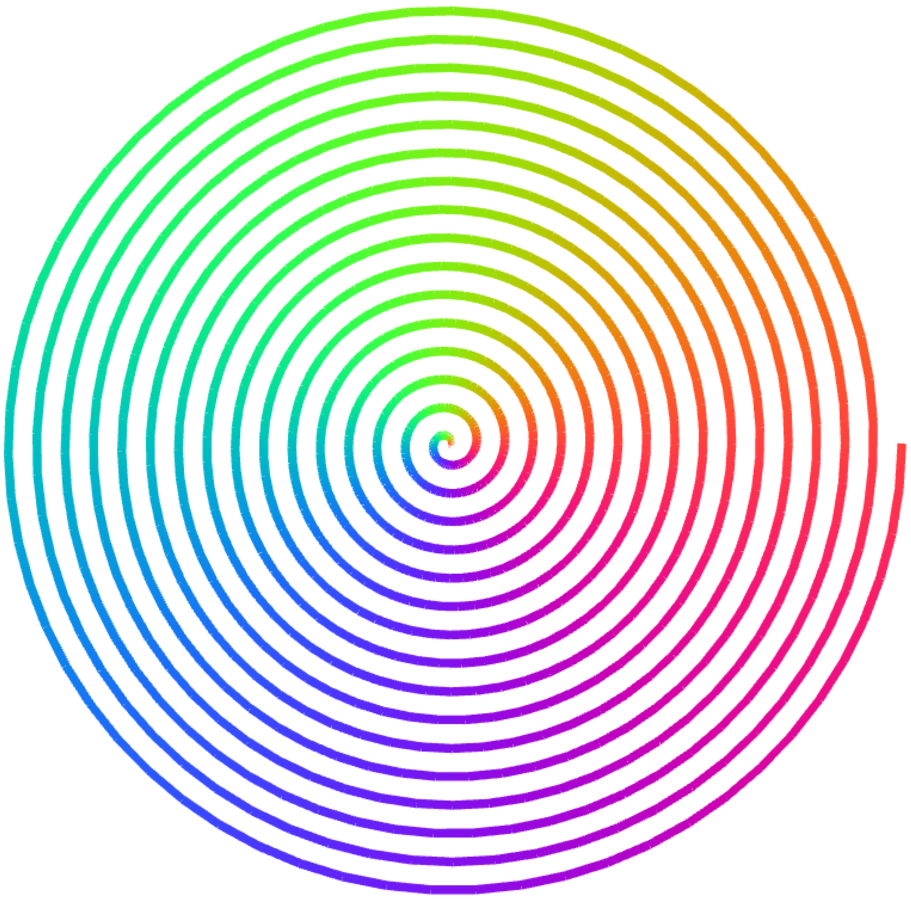

Figure 13: C4

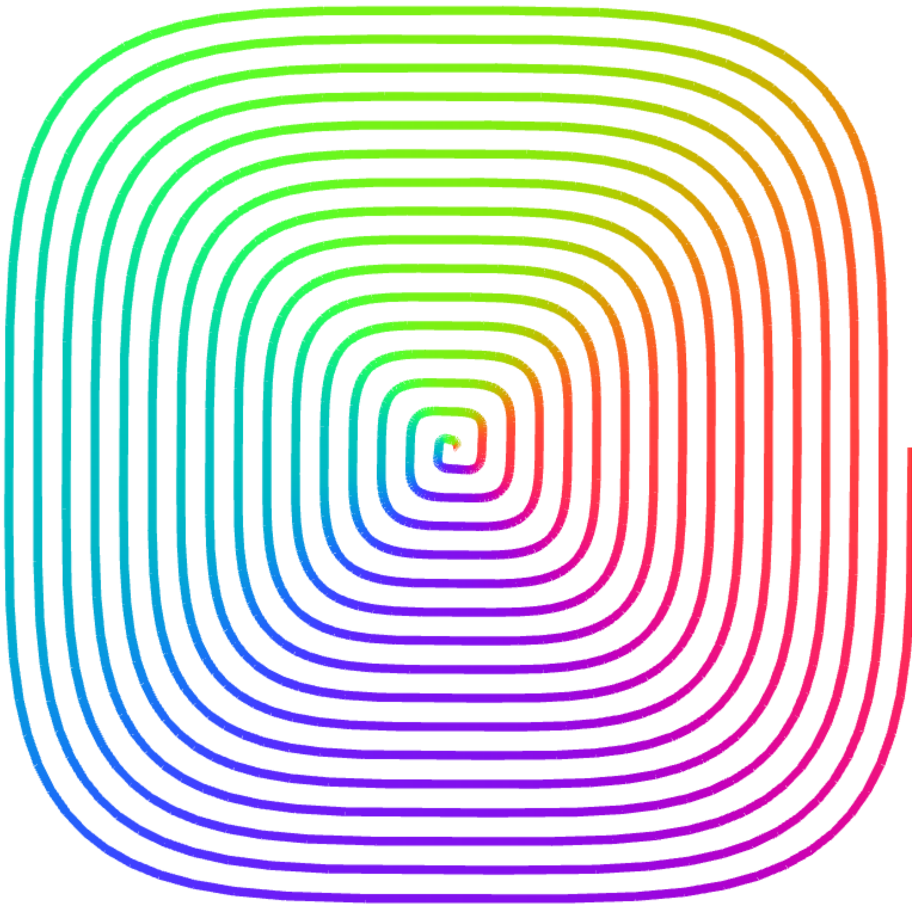

Figure 14: C5