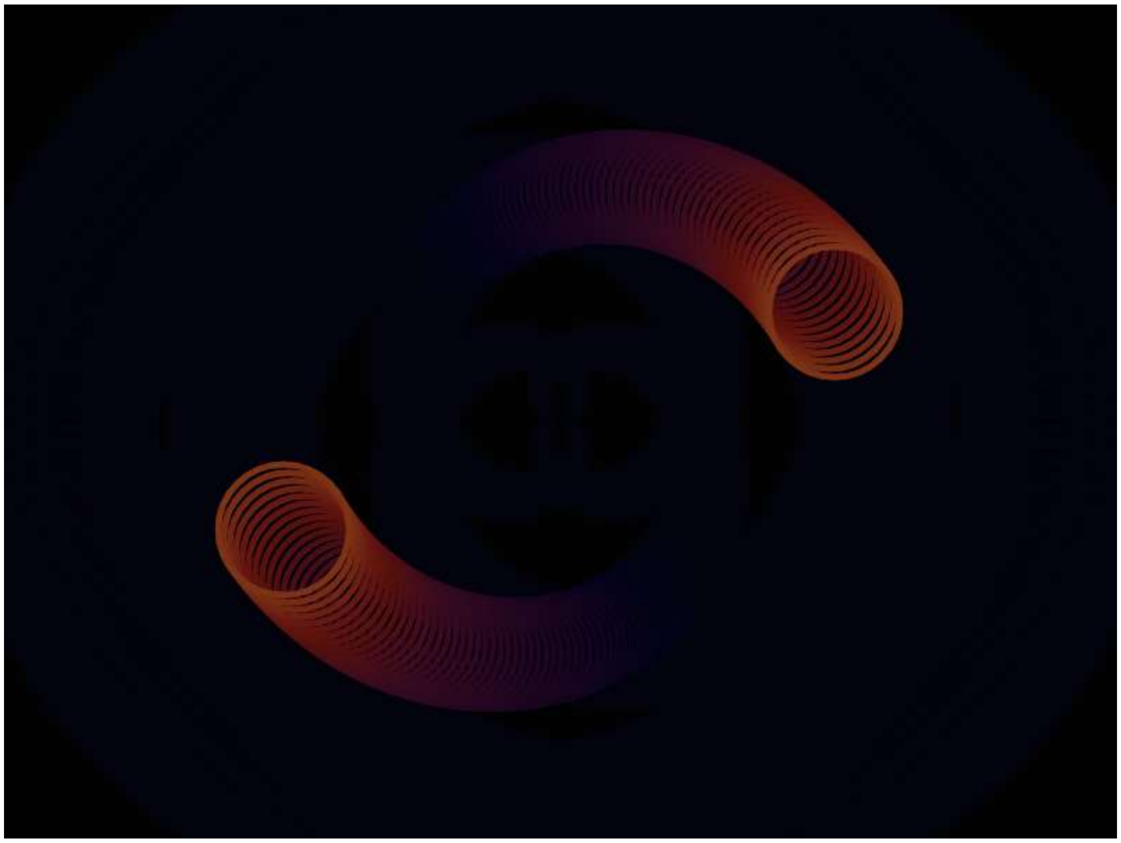
Trigonometry in CSS and JavaScript: Beyond Triangles

In part three of our series we'll look at how to create more interesting shapes with trigonometry, and how to draw them with the Canvas API.

By Michelle Barker in Tutorials on June 4, 2021



In the previous article we looked at how to clip an equilateral triangle with trigonometry, but what about some even more interesting geometric shapes?

This article is the 3rd part in a series on Trigonometry in CSS and JavaScript:

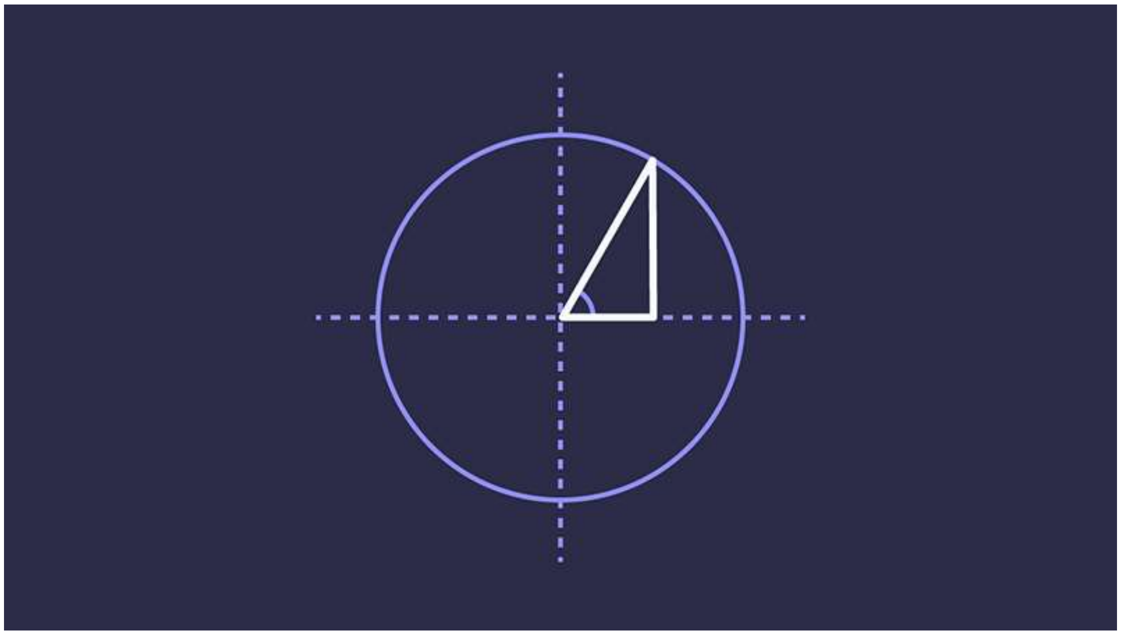
1. Introduction to Trigonometry
2. Getting Creative with Trigonometric Functions
3. **Beyond Triangles** (this article)

Plotting regular polygons

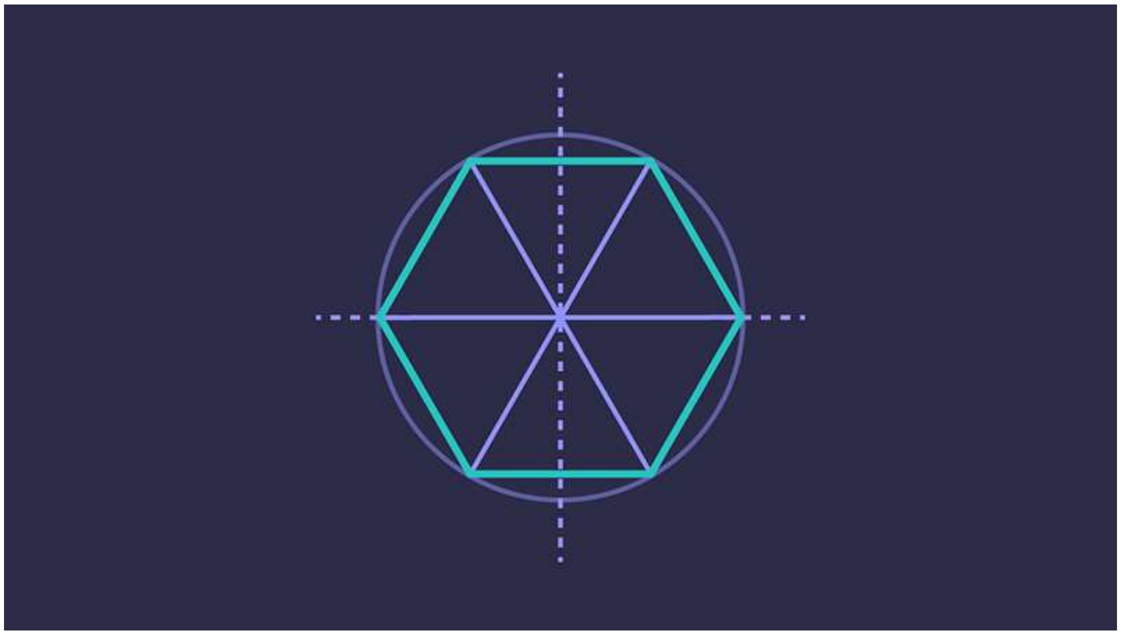
A regular polygon is a polygon with all equal sides and all equal angles. An equilateral triangle is one, so too is a pentagon, hexagon, decagon, and any number of others that meet the criteria. We can use trigonometry to plot the points of a regular polygon by visualizing each set of coordinates as points of a triangle.

Polar coordinates

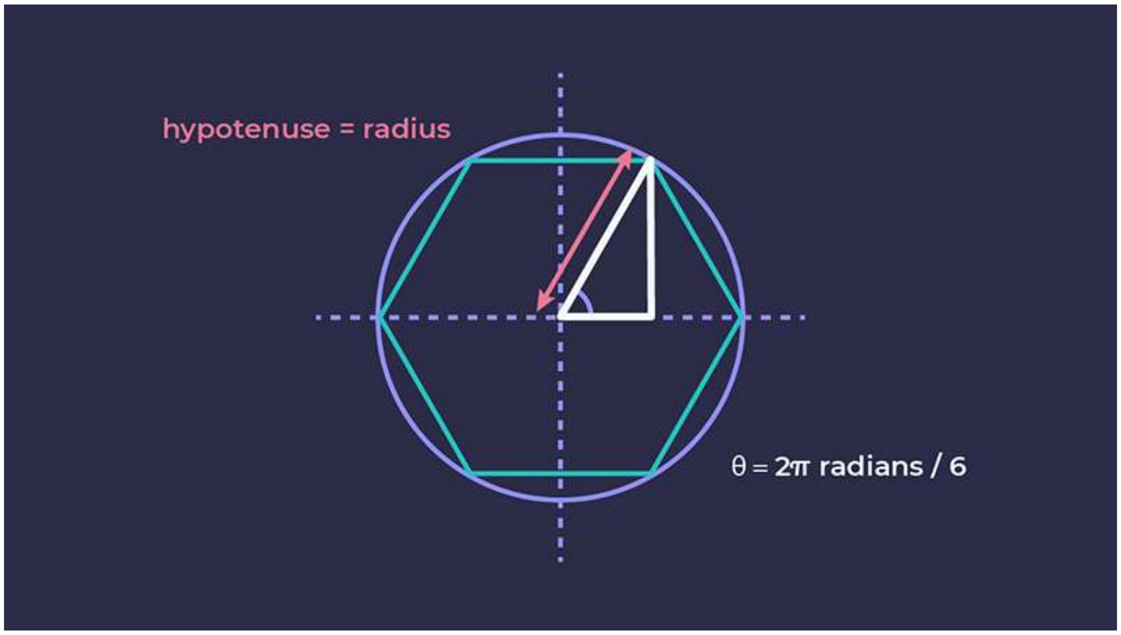
If we visualize a circle on an *x/y* axis, draw a line from the center to any point on the outer edge, then connect that point to the horizontal axis, we get a triangle.



If we repeatedly rotated the line at equal intervals six times around the circle, we could plot the points of a hexagon.



But how do we get the *x* and *y* coordinates for each point? These are known as cartesian coordinates, whereas polar coordinates tell us the distance and angle from a particular point. Essentially, the radius of the circle and the angle of the line. Drawing a line from the center to the edge gives us a triangle where *hypotenuse* is equal to the circle's radius.

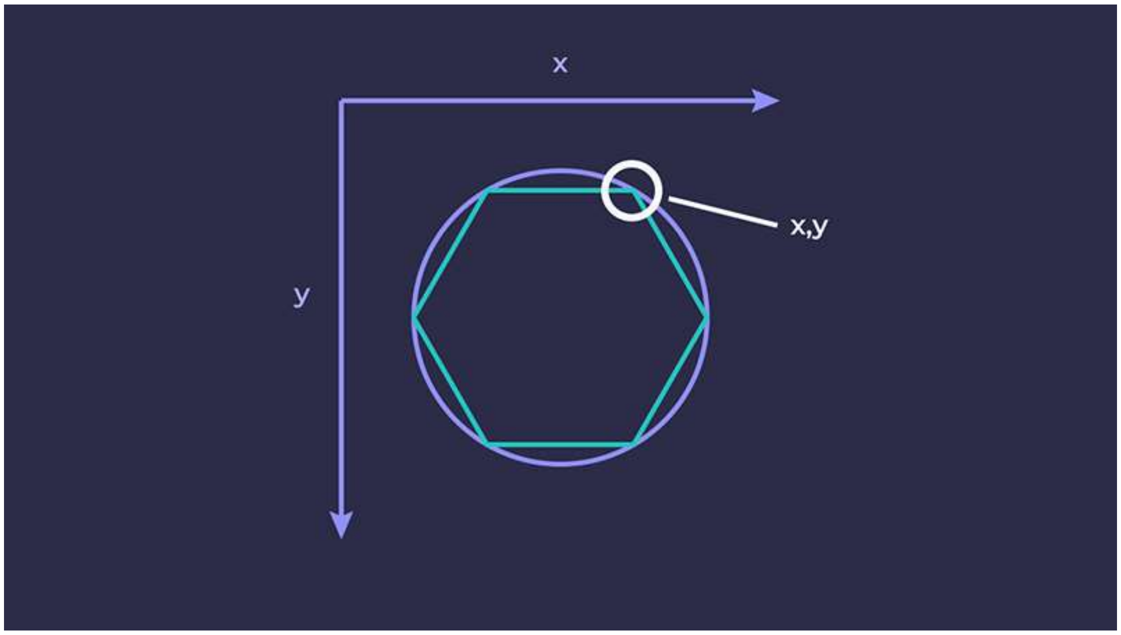


We can get the angle in degrees by diving 360 by the number of vertices our polygon has, or in radians by diving 2pi radians. For a hexagon with a radius of 100, the polar coordinates of the uppermost point of the triangle in the diagram would be written (100, 1.0472rad) *(r, θ)*.

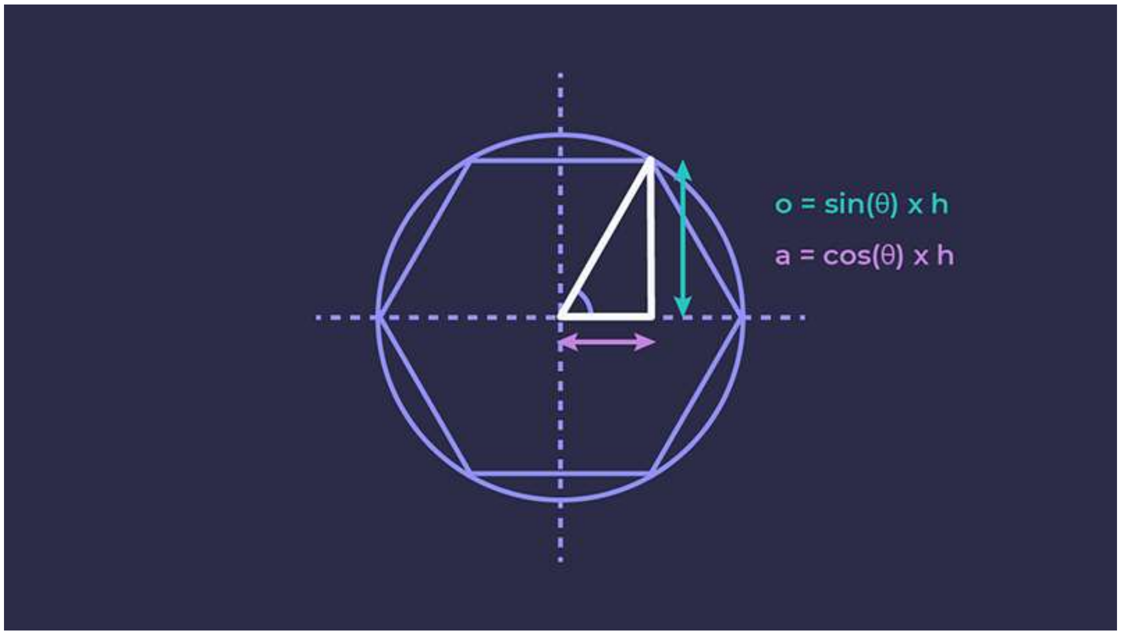
An infinite number of points would enable us to plot a circle.

Polar to cartesian coordinates

We need to plot the points of our polygon as cartesian coordinates – their position on the *x* and *y* axis.



As we know the radius and the angle, we need to calculate the *adjacent* side length for the *x* position, and the *opposite* side length for the *y* position.



Therefore we need *Cosine* for the former and *Sine* for the latter:

```
adjacent = cos(angle) * hypotenuse
opposite = sin(angle) * hypotenuse
```

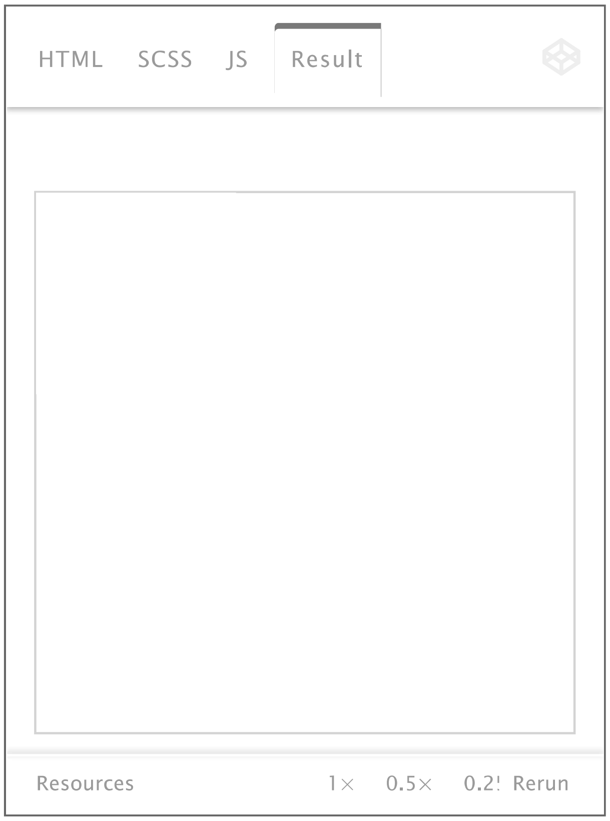
We can write a JS function that returns an array of coordinates:

```
const plotPoints = (radius, numberOfPoints) => {

    /* step used to place each point at equal distances */
    const angleStep = (Math.PI * 2) / numberOfPoints

    const points = []

    for (let i = 1; i <= numberOfPoints; i++) {
        /* x & y coordinates of the current point */
        const x = Math.cos(i * angleStep) * radius
        const y = Math.sin(i * angleStep) * radius

        /* push the point to the points array */
        points.push({ x, y })
    }

    return points
}
```

We could then convert each array item into a string with the *x* and *y* coordinates in pixels, then use the `join()` method to join them into a string for use in a clip path:

```
const polygonCoordinates = plotPoints(100, 6).map(({ x, y }) =
        return `${x}px ${y}px`
    }).join(',')
```

```
shape.style.clipPath = `polygon(${polygonCoordinates})`
```



This clips a polygon, but you'll notice we can only see one quarter of it. The clip path is positioned in the top left corner, with the center of the polygon in the corner. This is because at some points, calculating the cartesian coordinates from the polar coordinates is going to result in negative values. The area we're clipping is outside of the element's bounding box.

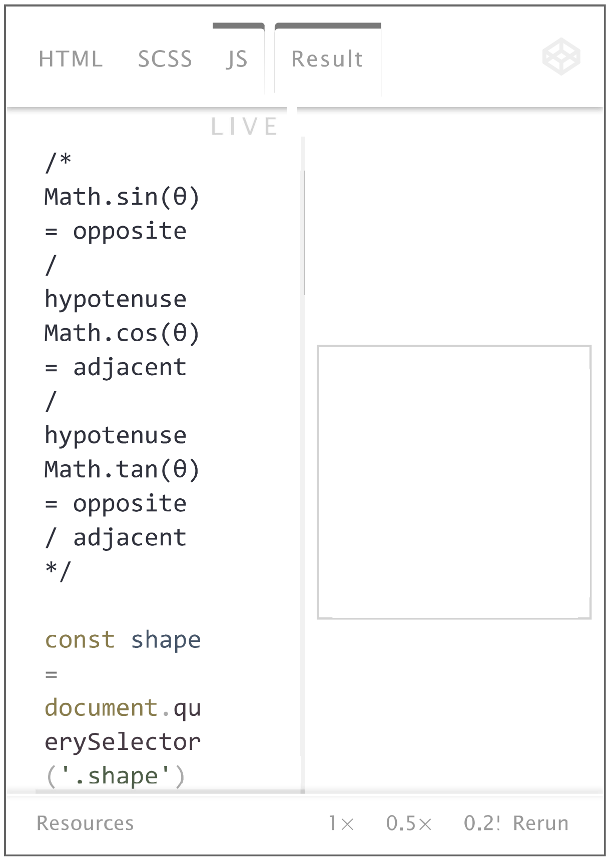
To position the clip path centrally, we need to add half of the width and height respectively to our calculations:

```
const xPosition = shape.clientWidth / 2
const yPosition = shape.clientHeight / 2

const x = xPosition + Math.cos(i * angleStep) * radius
const y = yPosition + Math.sin(i * angleStep) * radius
```

Let’s modify our function:

```
const plotPoints = (radius, numberOfPoints) => {
    const xPosition = shape.clientWidth / 2
    const yPosition = shape.clientHeight / 2
    const angleStep = (Math.PI * 2) / numberOfPoints
    const points = []

    for (let i = 1; i <= numberOfPoints; i++) {
        const x = xPosition + Math.cos(i * angleStep) * radius
        const y = yPosition + Math.sin(i * angleStep) * radius

        points.push({ x, y })
    }

    return points
}
```

Our clip path is now positioned in the center.

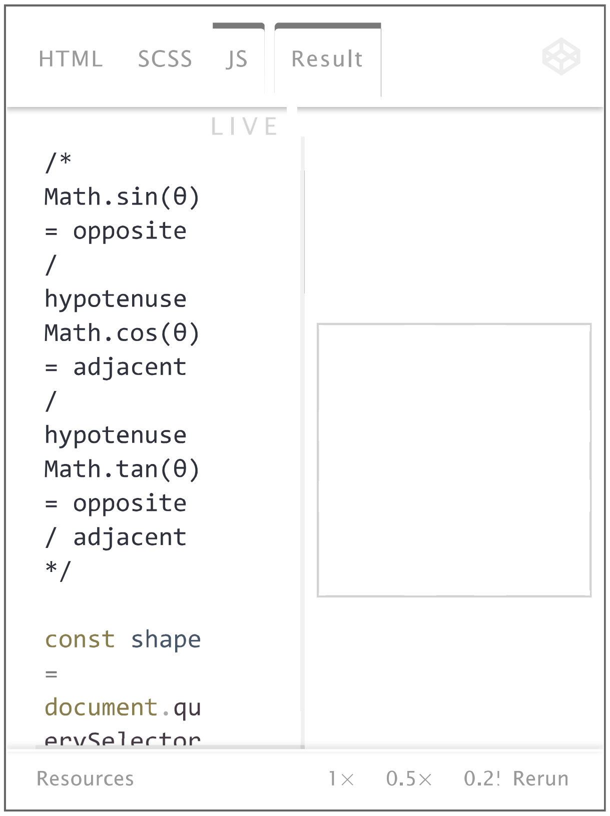


```
/*
Math.sin(θ)
= opposite
/
hypotenuse
Math.cos(θ)
= adjacent
/
hypotenuse
Math.tan(θ)
= opposite
/ adjacent
*/

const shape
=
document.qu
erySelector
('.shape')
```

Star polygons

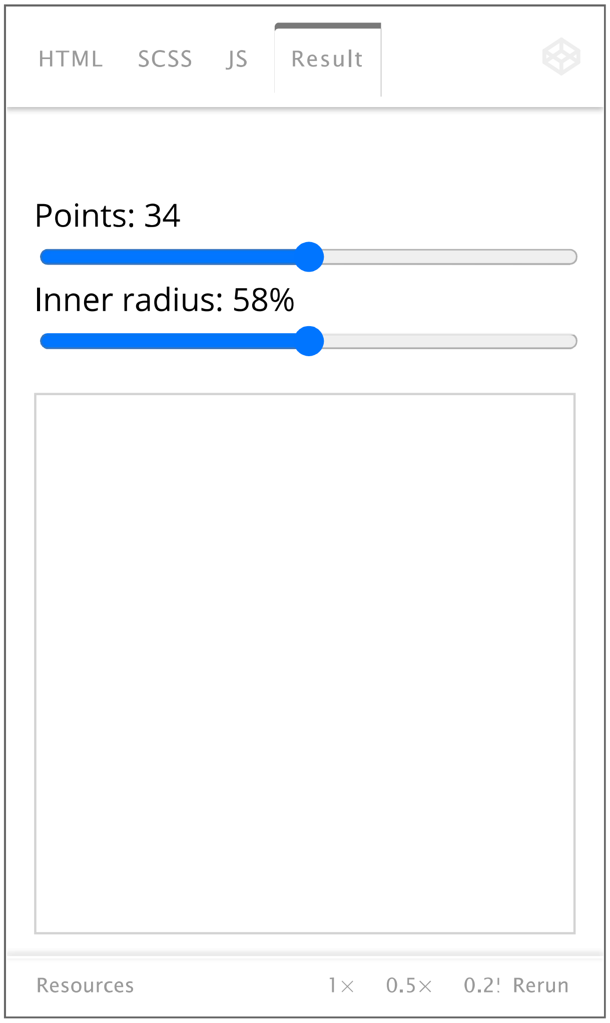
The types of polygons we've plotted so far are known as *convex* polygons. We can also plot *star* polygons by modifying our code in the `plotPoints()` function ever so slightly. For every other point, we could change the radius value to be 50% of the original value:

```
/* Set every other point's radius to be 50% */
const radiusAtPoint = i % 2 === 0 ? radius * 0.5 : radius

/* x & y coordinates of the current point */
const x = xPosition + Math.cos(i * angleStep) * radiusAtPoint
const y = yPosition + Math.sin(i * angleStep) * radiusAtPoint
```



```
/*
Math.sin(θ)
= opposite
/
hypotenuse
Math.cos(θ)
= adjacent
/
hypotenuse
Math.tan(θ)
= opposite
/ adjacent
*/

const shape
=
document.qu
erySelector
```

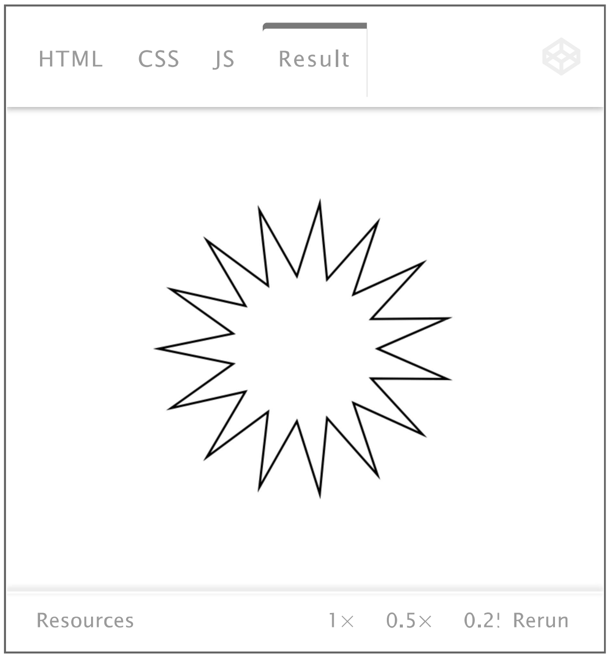
Here's an interactive example. Try adjusting the values for the number of points and the inner radius to see the different shapes that can be made.



Drawing with the Canvas API

So far we've plotted values to use in CSS, but trigonometry has plenty of applications beyond that. For instance, we can plot points in exactly the same way to draw on a `<canvas>` with Javascript. In this function, we're using the same function as before (`plotPoints()`) to create an array of polygon points, then we draw a line from one point to the next:

```
const canvas = document.getElementById('canvas')
const ctx = canvas.getContext('2d')

const draw = () => {
    /* Create the array of points */
    const points = plotPoints()
    /* Move to starting position and plot the path */
    ctx.beginPath()
    ctx.moveTo(points[0].x, points[0].y)

    points.forEach(({ x, y }) => {
        ctx.lineTo(x, y)
    })

    ctx.closePath()

    /* Draw the line */
    ctx.stroke()
}
```



Spirals

We don't even have to stick with polygons. With some small tweaks to our code, we can even create spiral patterns. We need to change two things here: First of all, a spiral requires multiple rotations around the point, not just one. To get the angle for each step, we can multiply *pi* by 10 (for example), instead of two, and divide that by the number of points. That will result in five rotations of the spiral (as 10*pi* divided by two is five).

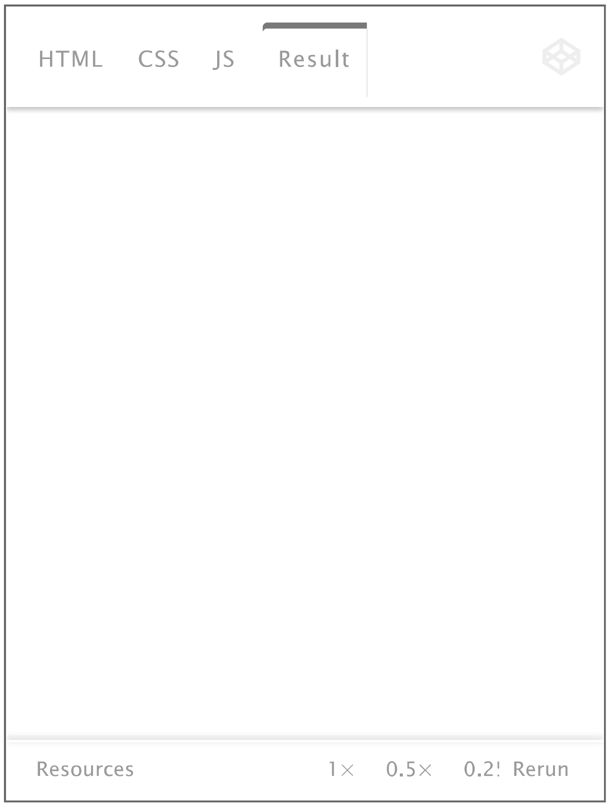
```
const angleStep = (Math.PI * 10) / numberOfPoints
```

Secondly, instead of an equal radius for every point, we'll need to increase this with every step. We can multiply it by a number of our choosing to determine how far apart the lines of our spiral are rendered:

```
const multiplier = 2
const radius = i * multiplier
const x = xPosition + Math.cos(i * angleStep) * radius
const y = yPosition + Math.sin(i * angleStep) * radius
```

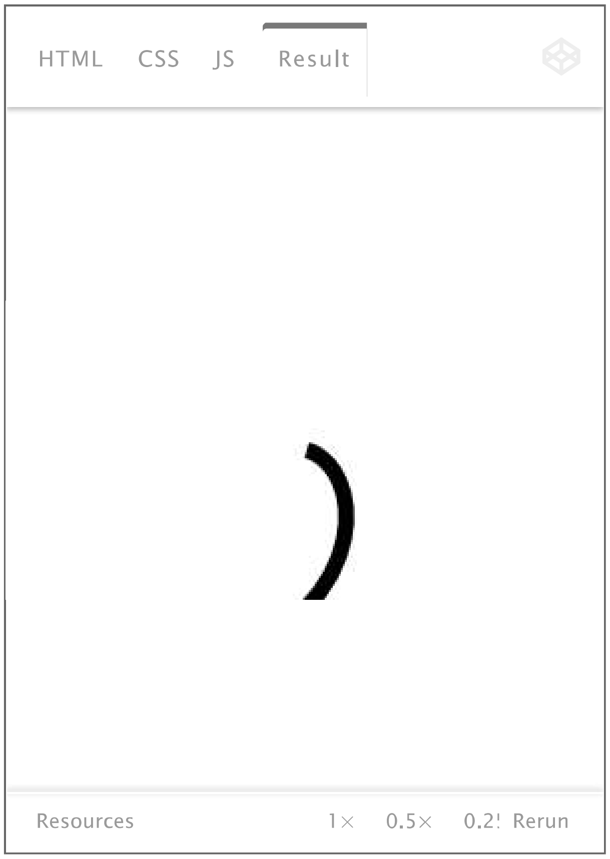
Putting it all together, our adjusted function to plot the points is as follows:

```
const plotPoints = (numberOfPoints) => {
    const angleStep = (Math.PI * 10) / numberOfPoints
    const xPosition = canvas.width / 2
    const yPosition = canvas.height / 2

    const points = []

    for (let i = 1; i <= numberOfPoints; i++) {
        const radius = i * 2 // multiply the radius to get the
        const x = xPosition + Math.cos(i * angleStep) * radius
        const y = yPosition + Math.sin(i * angleStep) * radius

        points.push({ x, y })
    }

    return points
}
```



At the moment the lines of our spiral are at equal distance from each other, but we could increase the radius exponentially to get a more pleasing spiral. By using the `Math.pow()` function, we can increase the radius by a larger number for each iteration. By the golden ratio, for example:

```
const radius = Math.pow(i, 1.618)
const x = xPosition + Math.cos(i * angleStep) * radius
const y = yPosition + Math.sin(i * angleStep) * radius
```



Animation

We could also rotate the spiral, using (using requestAnimationFrame). We'll set a `rotation` variable to *0*, then on every frame increment or decrement it by a small amount. In this case I'm decrementing the rotation, to rotate the spiral anti-clockwise

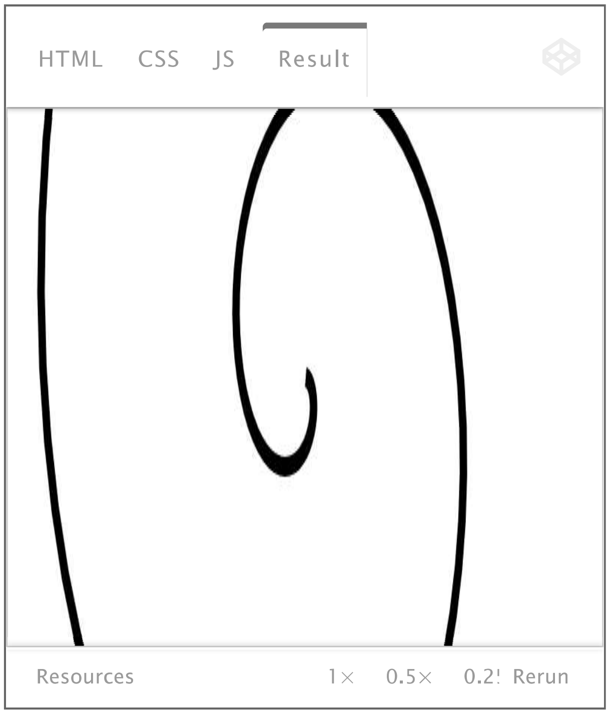
```
let rotation = 0

const draw = () => {
    const { width, height } = canvas

    /* Create points */
    const points = plotPoints(400, rotation)

    /* Clear canvas and redraw */
    ctx.clearRect(0, 0, width, height)
    ctx.fillStyle = '#ffffff'
    ctx.fillRect(0, 0, width, height)

    /* Move to beginning position */
    ctx.beginPath()
    ctx.moveTo(points[0].x, points[0].y)

    /* Plot lines */
    points.forEach((point, i) => {
        ctx.lineTo(point.x, point.y)
    })

    /* Draw the stroke */
    ctx.strokeStyle = '#000000'
    ctx.stroke()

    /* Decrement the rotation */
    rotation -= 0.01

    window.requestAnimationFrame(draw)
}

draw()
```

We'll also need to modify our `plotPoints()` function to take the `rotation` value as an argument. We'll use this to increment the *x* and *y* position of each point on every frame:

```
const x = xPosition + Math.cos(i * angleStep + rotation) * rad
const y = yPosition + Math.sin(i * angleStep + rotation) * rad
```

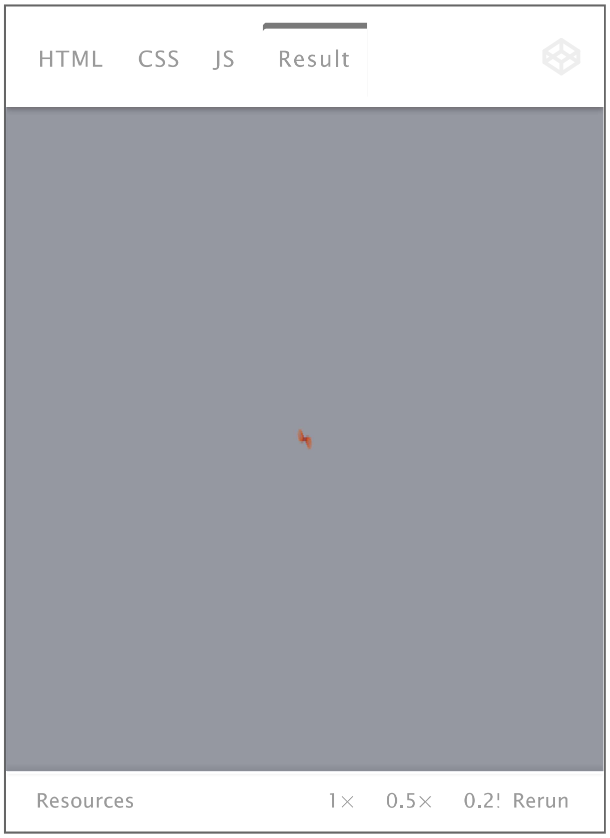
This is how our `plotPoints()` function looks now:

```
const plotPoints = (numberOfPoints, rotation) => {
    /* 6 rotations of the spiral divided by number of points *
    const angleStep = (Math.PI * 12) / numberOfPoints

    /* Center the spiral */
    const xPosition = canvas.width / 2
    const yPosition = canvas.height / 2

    const points = []

    for (let i = 1; i <= numberOfPoints; i++) {
        const r = Math.pow(i, 1.3)
        const x = xPosition + Math.cos(i * angleStep + rotatio
        const y = yPosition + Math.sin(i * angleStep + rotatio

        points.push({ x, y, r })
    }

    return points
}
```



Wrapping up

I hope this series of articles has given you a few ideas for how to get creative with trigonometry and code. I'll leave you with one more creative example to delve into, using the spiral method detailed above. Instead of plotting points from an array, I'm drawing circles at a new position on each iteration (using requestAnimationFrame).



Special thanks to George Francis and Liam Egan, whose wonderful creative work inspired me to delve deeper into this topic!

Michelle Barker

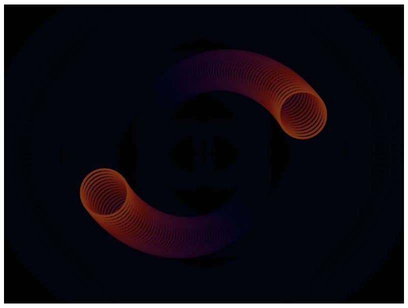
Michelle is a Lead Front End Developer at Bristol web agency Atomic Smash and author of front-end blog CSS { In Real Life }. She has written articles for CSS Tricks, Smashing Magazine, and Web Designer Magazine, to name a few. She enjoys experimenting with new CSS features and helping others learn about them.

https://michellebarker.co.uk

Twitter Instagram Codepen Github

Background Scale Hover Effect with CSS Clip-path

Crafting a Cutout Collage Layout with CSS Grid and Clip-path



Trigonometry in CSS and JavaScript: Beyond Triangles