The Mandalizer: Theory and Implementation

Mandalas

Mandalas as a form of decoration or spiritual art have been around for thousands or tens of thousands of years. They are best known as religious symbols from Hinduism, Buddhism, Jainism and Shintoism (the word itself is from the Sanskrit, and means "circle")[1]. Similar circular and spiral forms existed in rock carvings from many places throughout the ancient world: Aztec calendar stones[2], Newgrange kerb

1 https://en.wikipedia.org/wiki/Mandala
2 https://en.wikipedia.org/wiki/Aztec_sun_stone

stones[3], North American petroglyphs[4], African petroglyphs[5], and so on. In more recent years, mandalas were popularized by Carl Jung as secular/personal symbols[6]. They're also common in new-age spirituality and art.

So what defines a mandala? Depending on the source, you'll get different definitions. The primary commonality involves concentric circles or patterns.

What's This, Then?

The Mandalizer is a program to create visually-interesting patterns. It comes less from religious, spiritual, or psychological origins than aesthetic ones, which in no way limits the applications of the output to religious, spiritual, or psychological uses!

The Mandalizer Approach

The Mandalizer is written in fairly standard JavaScript using HTML 5 Canvas as its output.

It works on the principle of multiple layers or "stages" that are rotated around the center point. For each layer, a type of decoration is chosen, bounds are calculated, and the stage is repeatedly rendered around the circle (see Illustration 1: A single stage of a mandala).

Optional enhancements are may also be added. An "underlay" (a color or texture applied to the stage), may be rendered before the main decoration for a stage, and an "overlay" design element may be applied over the center or edges of the stage.

Types of decoration started with simple things like lines and fills, and eventually added more complex things like rows of variable-size circles or petal shapes. These were loosely inspired by henna tattoo art, thangka backgrounds, and whorls from banknotes and fancy certificates.

Many aspects of the generated mandalas involve randomness[7]: random types of decorations, random sizes, colors, fill styles, etc. To make things more interesting, a date-to-random-number-seed approach is used, so a given date will always yield the same random number sequence.

3 https://www.worldheritageireland.ie/bru-na-boinne/built-heritage/art/
4 https://www.newmexico.org/places-to-visit/regions/southeast/three-rivers-petroglyph-site/
5 https://www.megalithic.co.uk/article.php?sid=30876
6 https://www.ancient.eu/mandala/
7 Or pseudo-randomness, to be precise.

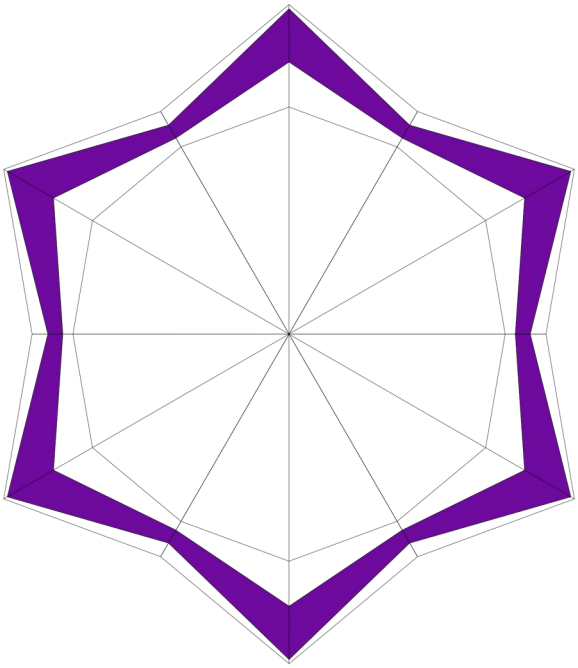

Illustration 1: A single stage of a mandala

Some Math

Basic Stage Geometry

Each stage is defined by a JavaScript class representing the chevron shape that gets repeated to form a concentric layer of the mandala. All stages are defined by a small set of radii: the inner and outer radii at the center and at one edge (since it's symmetrical about the center). These are shown as `stage.e0`, `stage.e1`, `stage.c0`, and `stage.c1` in Illustration 2: Description of a "stage". Note that the stage does not maintain any information about the angle – this allows us to use the same stages to render the mandala with varying symmetry.

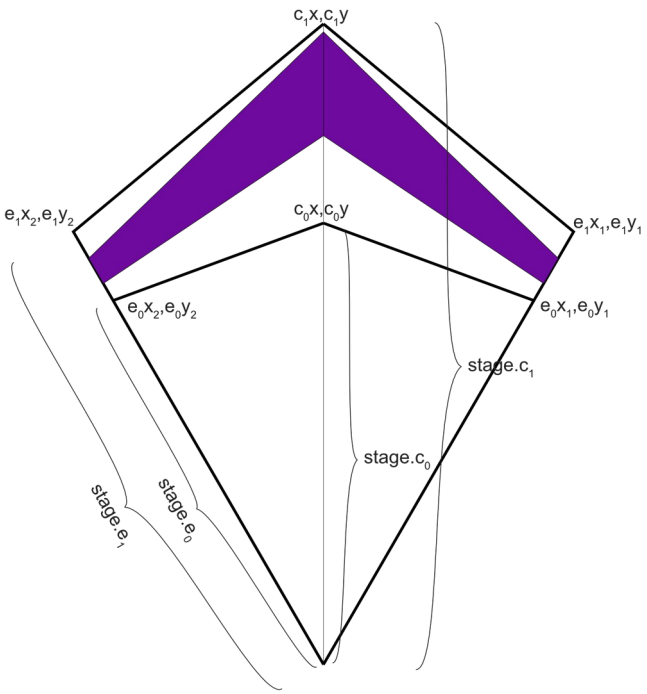

Illustration 2: Description of a "stage"

The stage object populates the radii variables in their `next()` method. In this method, they can also populate arbitrary variables to be tracked, like random coin tosses for whether the shape is filled, whether to use a gradient, etc.

The angle of each stage is computed based on the user's selected symmetry number. In other words, if the user wants 5-fold symmetry, the angle is $2\pi/5$.

When the concentric layer is being rendered, a stage object's `render()` method gets called for each degree of symmetry. There's a set of utility functions that compute the bounds of a stage for a given step number and angle: these values are (somewhat arbitrarily) labeled by edge and axis as shown in Illustration 2: Description of a "stage".

These bounds are pretty straight-forward trigonometry:

```
c₀x = w/2 + stage.c₀ * cos(angle * step)
c₀y = h/2 + stage.c₀ * sin(angle * step)
e₀x₁ = w/2 + stage.e₀ * cos(angle * (step + 0.5))
e₀y₁ = h/2 + stage.e₀ * sin(angle * (step + 0.5))
```

etc,

where `w` is the width of the canvas, and `h` is the height of the canvas.

Advanced Stage Geometry

Some stages want to do fancy things that involve filling in patterns that fit in radial slices ("segments") of the stage. Finding the bounds of these segments is still just trigonometry, but a lot more

complicated[8]. We have a similar utility function that will compute the bounds of each segment, so a stage can iterate through them when rendering itself.

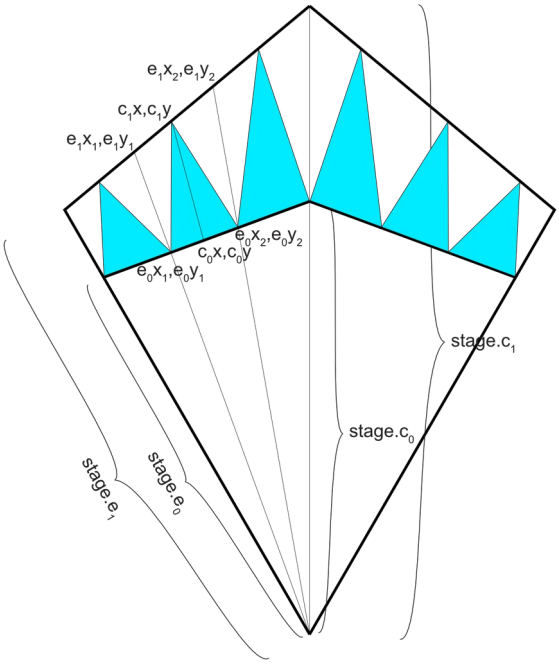

Illustration 3: Example of stage segments

The math for the utility function has to project along the inner and outer edges of the stage for each step. Since the stage itself only tracks the radii mentioned before, and the system has an angle for the width of the stage, we have a lot of unknowns to solve for.

To abstract a bit, we look at what it would take to find the point (e_0x_2, e_0y_2) on Illustration 3: Example of stage segments. We don't know a lot of things directly. What we know is shown in Illustration 4: Known and Unknown in black; what we don't know directly is shown in red. To save space in the equations, we're abbreviating `stage.c`$_0$ as `C`, and `stage.e`$_0$ as `E`. To get to the point in question, we want to do simple trigonometry with length `x` and angle γ. How do we get those?

8 It's been many years since the author had a math class, and these equations took embarrassingly long to figure out.

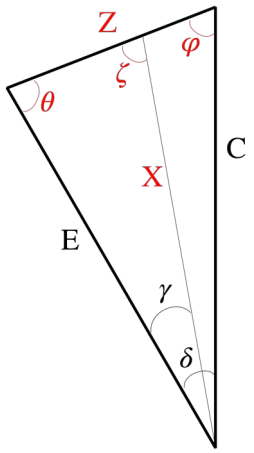

Illustration 4: Known and Unknown

To solve this, we start with the law of sines, which tells us $\frac{C}{\sin(\theta)}=\frac{Z}{\sin(\delta)}=\frac{E}{\sin(\phi)}$ and law of cosines says $Z^2=E^2+C^2-2EC\cos(\delta)$

so $Z=\sqrt{(E^2+C^2-2EC\cos(\delta))}$

Originally, we tried solving for θ using the law of sines: $Z\sin(\theta)=C\sin(\delta)$ so

$\theta=\sin^{-1}(\frac{C\sin(\delta)}{Z})=\sin^{-1}(\frac{C\sin(\delta)}{\sqrt{(E^2+C^2-2EC\cos(\delta))}})$

This ends up failing if θ is greater than 90°.

But since we have Z, we can solve for θ using law of cosines: $C^2=Z^2+E^2-2EZ\cos(\theta)$

so $C^2-Z^2-E^2=-2EZ\cos(\theta)$ and $\cos(\theta)=\frac{-C^2+Z^2+E^2}{2EZ}$ and $\theta=\cos^{-1}(\frac{-C^2+Z^2+E^2}{2EZ})$

Hooray! This works even when θ exceeds 90°.

Since θ is the same for both pieces of the triangle, $\zeta=180-\gamma-\theta$

And again using law of sines, $\frac{E}{\sin(\zeta)}=\frac{X}{\sin(\theta)}$ or $X=\frac{E\sin(\theta)}{\sin(\zeta)}$

Now we substitute a whole lotta stuff:

$$X = \frac{\frac{EC\sin(\delta)}{\sqrt{(E^2+C^2-2EC\cos(\delta))}}}{\sin(180-\gamma-\sin^{-1}(\frac{C\sin(\delta)}{\sqrt{(E^2+C^2-2EC\cos(\delta))}}))}$$

or

$$X = \frac{EC\sin(\delta)}{\sqrt{(E^2+C^2-2EC\cos(\delta))}\sin(180-\gamma-\sin^{-1}(\frac{C\sin(\delta)}{\sqrt{(E^2+C^2-2EC\cos(\delta))}}))}$$

There must be simpler ways of computing this, but this is how I did it.

If you look at the code, you can see how this gets implemented in the `segments()` function, which converts the handful of angles and radii to a data structure of points like the one shown in Illustration 3: Example of stage segments.

A couple of other helper functions can take a stage, and divide it up into the subsegments either by count, or by a fixed width along the edge.

Code and Observations

Stages, overlays, and underlays are all defined as JavaScript classes. Stages have a `next()` method, which gets passed the previous stage. It computes whatever boundaries based on the previous layer's values, and any random parameters it needs. Stages, overlays, and underlays all have `render()` methods, which they use to paint themselves onto the Canvas.

Overall, the code's not especially well laid out or organized. HTML controls and various runtime parameters were hand-coded, which seems like a poor idea in retrospect. In fact, creating the Mandalizer without some framework or library like jQuery or vue.js was weird and shortsighted.

The stage model and the main rendering loop of the code are not really flexible enough. Looking at henna hand-art, these mandalas should support a degree of recursion, so stages could contain series of smaller mandalas. Also, they should support time-references, so the animation could go beyond the radial drawing, but could actually support a zoom-like effect of diving into an infinitely growing mandala. Remember those Mandelbrot-set programs that animated the color table or created videos of diving into the set? Had I planned better, these mandalas could do those things too.

But there you go. The best way to code something good is to rewrite it a few months or years after the first version.

Similarly, were I to do this again, I wouldn't use Canvas. After all, modern browsers support SVG natively, and the vector output would be usable in many other contexts. Want to print a mandala on a T-shirt? It scales to that size without problem. Want to print on a billboard? No problem! Want to import it into a design program and tweak it? SVG would allow all of that.

All of these criticisms, valid though they are, don't detract from the fact that the Mandalizer creates some pretty amazing images. Some of the results are far more beautiful than I envisioned when I first started work on the project.

License