Non-stinking kaleidoscope in Vuo

Part 1 - Lines (damn lines, and statistics!)

A kaleidoscope is perhaps my earliest memory of an abstract visual. Instant gratification from three mirrors and some cheap plastic «jewelry». Amazing! Today though, it is more of a staple for the badly planned gig, someone inexperienced, stock clips, and perhaps something that is heavily linked to funny-smelling people inevitably clothed in tie-dye garments with matching copper-pyramid hats. Individuals in selected VJ forums has even more than once (at least once) posted NO KALEIDOSCOPES!!! when soliciting for content creators. With the previously mentioned prerequisites - who can really blame them?

So what can we do? Luckily the problem seems to lie with application, and not technique. Phew..

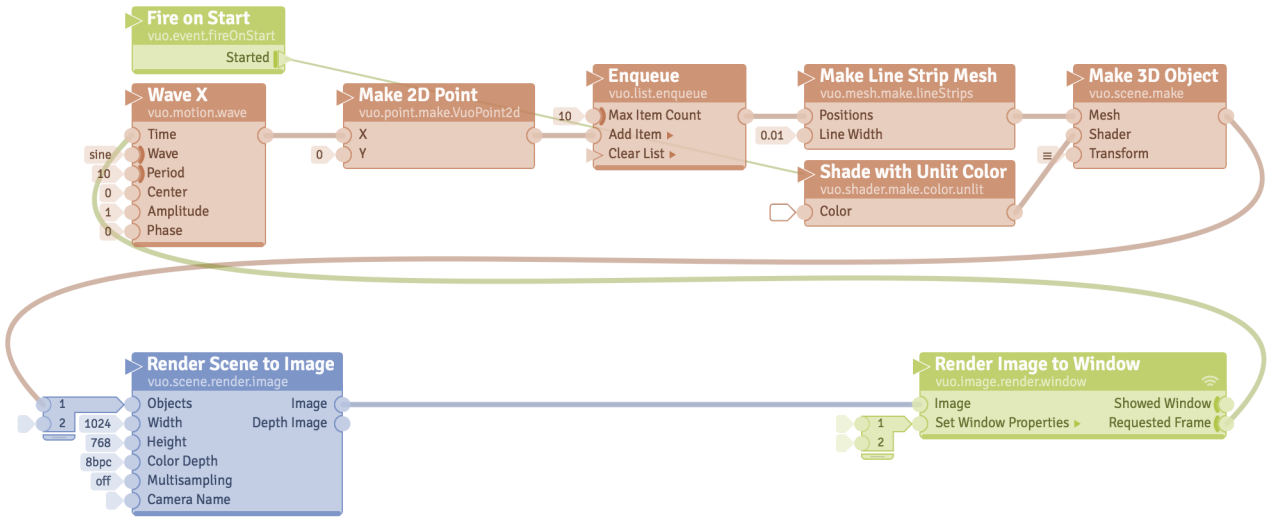
First we need to define the most basic of kaleidoscope stinking. This could be subjective, but I think the lowest threshold will be that it is slapped onto something without care. If you're served soup, and it gets splashed over the table and your clothes by a waiter that is seemingly on drugs and says «it's cool, look at the colors, whoa…», you would probably not think much of either the waiter *or* the soup, no matter how delicious the concept of soup is (I just had soup). If this also were the only way you had been served soup your entire life, you would be excused for screaming NO SOUP!!! every time a meal were on the horizon (both literally and figuratively). So this will be my starting point: Make content *for* the kaleidoscope!

In making the Kaleidoscope look good however, we will focus on the image that goes into it. The kaleidoscope in itself basically just splits images up into triangles, so for simplicity and clarity in figuring out what happens, we'll start out with a basic line.

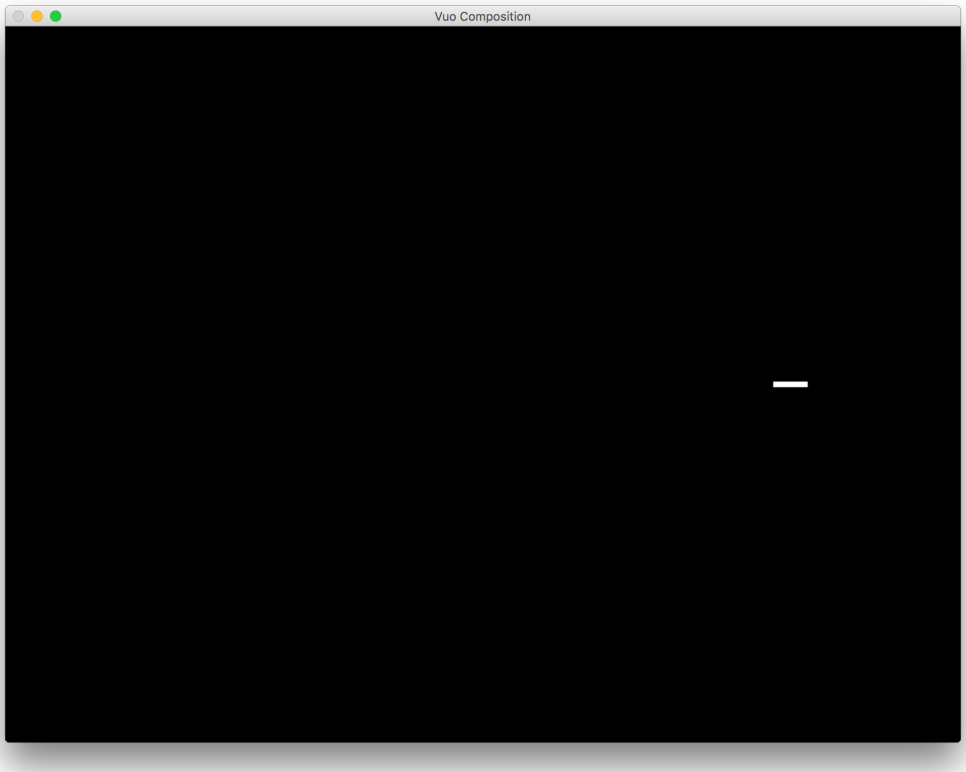
To get a basic line, you can either use the **Make Line Mesh** or the **Make Line Strip Mesh** nodes. The first will create a dotted line segment, and the latter a connected line. For this tutorial I will use the **Make Line Strip Mesh** node. The *Positions* input of the node expects a list of 2d Points. To generate this, we'll need a few more nodes starting with a **Wave** node. Renaming this to **Wave X** and connecting it to the *X* port of a **Make 2d Point** node generates our basic moveable 2d point. To get it to become a list is no harder matter than connecting the output of the **Make 2d Point** node to an **Enqueue** node. This will collect an amount of values specified by the *Max Item Count* port of the **Enqueue** node and generate a list from it.

Getting a meaningful output from the **Make Line Strip Mesh** node means connecting it further to a **Make 3D Object** node. This node also needs a *Shader* connected. I'm using the **Shade With Unlit Color** node for simplicity, triggered by a **Fire On Start** node. This is then connected to a **Render Scene to Image** node, which is finally connected to the **Render Image to Window** node. *The Requested Frame* port of the **Render Image to Window** node is connected to the *Time* input of the **Wave X** node.

I color all my nodes based on function for clarity, but you can of course color it by your own desire - or not care at all. You should however, by following these steps, end up with a composition looking somewhat like this:



If you then run the composition, you'll end up with a tiny line moving across the screen on the X-axis. Remember that the length is decided by the *Max Item Count* of the **Enqueue** node, and try adjusting it.



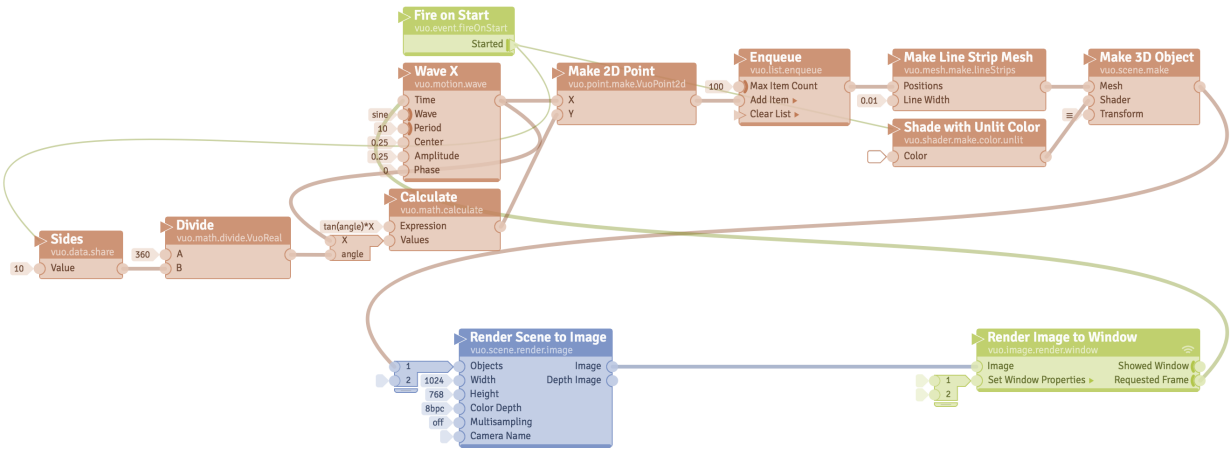
The **Make Kaleidoscope Image** node, which is what we'll produce our kaleidoscope from, works by taking an angled cutout of the input *Image*, and dragging out the edges that happens to be within the visible area (kinda cheat-y, but this works to our advantage). The base edge of the visible area/cutout always goes from X,Y = 0,0 to 1,0. You can set the *Sides* input by a real number. Further, it has *Blade* and *Image Center* inputs, that can be left at 0,0 for this application. The last input is the *Reflect Odd Sides* which flips and mirrors the original picture. This can make for some interesting variations further down the line.

Knowing this, we can start setting up our line to do something more interesting than trudging along it's narrow path in life. Limiting the X-axis of the line is simple, but as we're using a **Wave** node (which by the way can be substituted by a **Curve** node with a slightly different configuration) we have to account for the values oscillating between positive and negative values. So to get the point to limit itself to X min = *0* and X max = *0.5* we need to set the *Center* of the **Wave X** to *0.25* and the *Amplitude* to *0.25*.

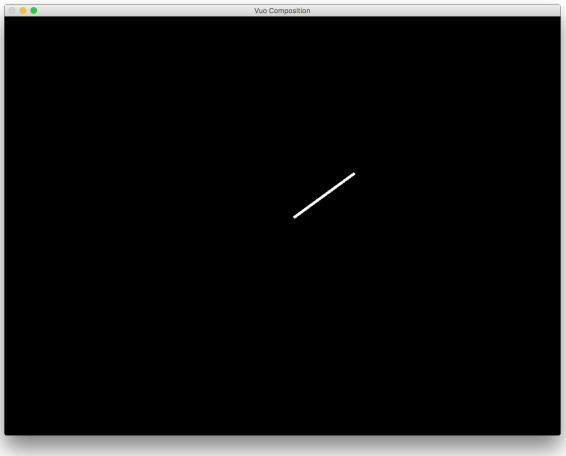
With that out of the way, the next step would be to look at the Y-axis. As we are working with triangles, we should know from elementary school math how to find the height (or length?) of the opposite side from the angle and the adjacent side. I didn't, so I had to scratch my head, turn to google, and look at some Khan academy videos. Turns out that the only thing we strictly need to do is multiplying the tangent of the angle with the X-position (I still don't understand exactly what a «tangent» is, it's kinda greek to me). You can use discrete nodes for this, but I find it is a lot more tidy to just use the **Calculate** node to do more involved calculations.

The other part is to find the angle. This one is easy, just divide 360 by the amount of sides in the kaleidoscope, and we should be good to go. First we add a **Share Value** node, rename it **Sides** and trigger it from the **Fire on Start** node. I like to set this to use integer inputs (right-click *Value* -> Set Data Type -> Integer) since it then prevents the kaleidoscope from getting a weird slice (but this is again totally your prerogative). Add a **Divide** node and enter *360* at the *A* port, and connect the **Sides** output to the *B* port. Then add in a **Calculate** node, and type in (or copy/paste) the *Expression* *tan(Angle)*X*. The *Values* input of the **Calculate** node should now read *Angle* and *X*. Connect the output from **Divide** to the *Angle* input, and the output from **Wave X** to the *X* input of the **Calculate** node.

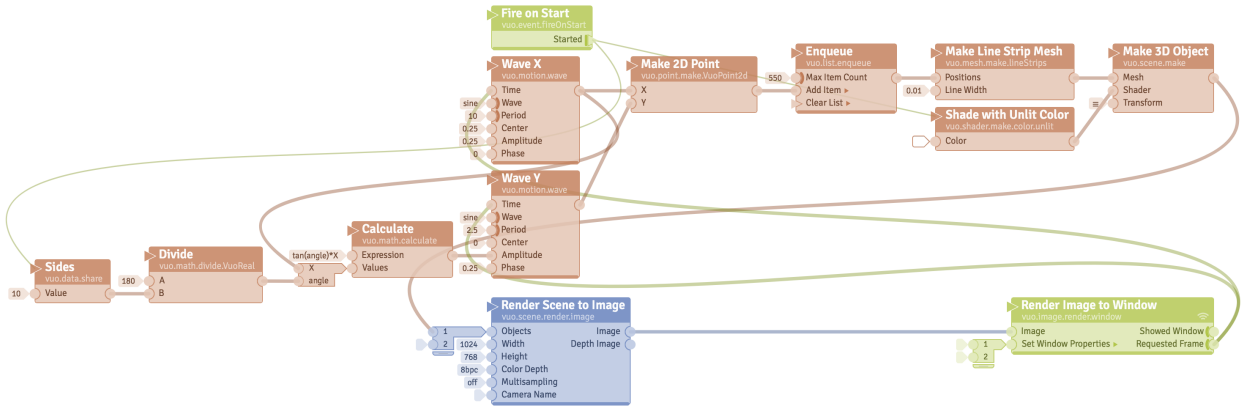
The composition should now look somewhat like this:



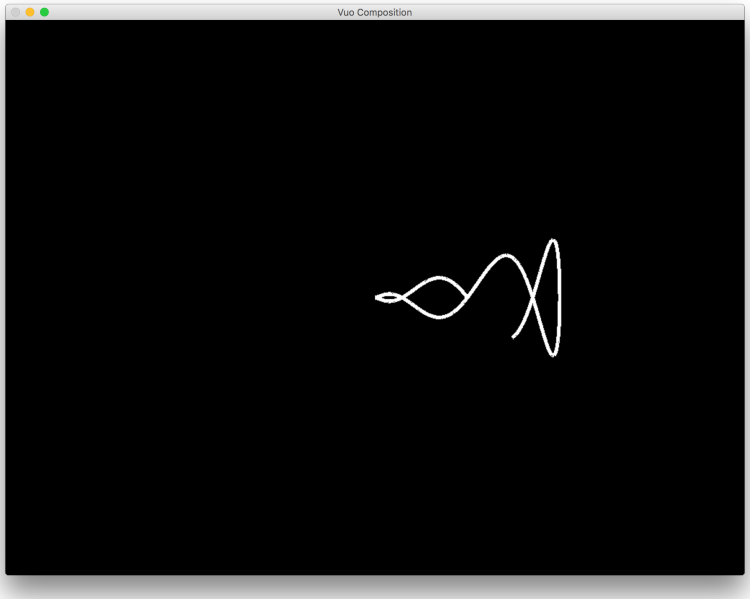
And output the line, now traveling at an angle decided by the **Sides** and **Divide** nodes:



To animate the Y-axis, we add another **Wave** node and name it **Wave Y** this time. Then connect the *Requested Frame* from the **Render Image to Window** node to its *Time* port. Leave the *Center* input at *0*, and connect the output from the **Calculate** node to the *Amplitude* port. Remembering that we have to account for the **Wave** nodes oscillation between positive and negative numbers, we should divide the angle by 2, but to make things a bit simpler, we can just change the *A* input of the **Divide** node from 360 to 180. To make things a bit more exciting as well, adjust the *Phase* input of the **Wave Y** node to 0.25. The composition should now look something like this:



And output a more satisfying line:

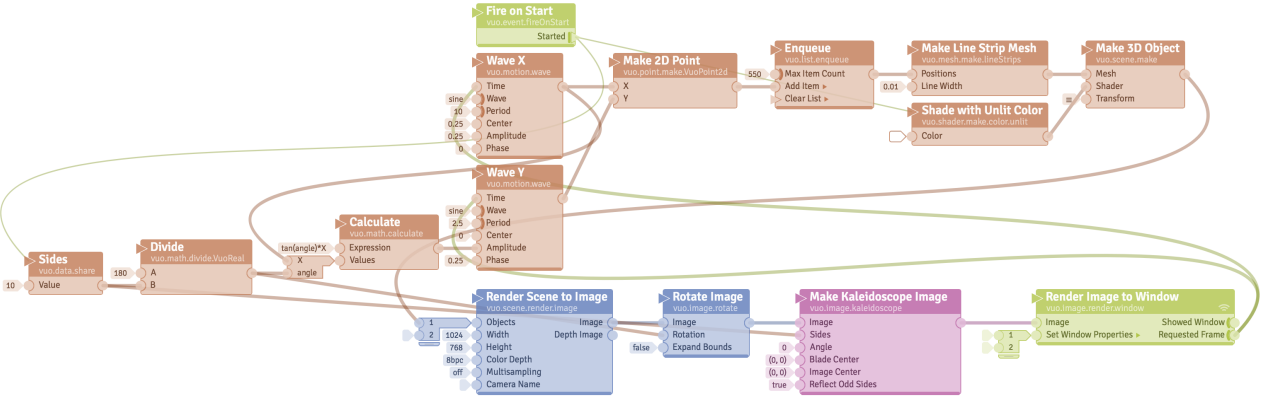


Now that our slice is ready, we can start putting it into the kaleidoscope. However, a small adjustment needs to be made before it fits correctly. As the kaleidoscope will cut anything below Y = 0.0, and the image half of the time is on the negative side, we will have to rotate the image by some angle. The angle we need to rotate it by is luckily very convenient - it is the output of the **Divide** node.

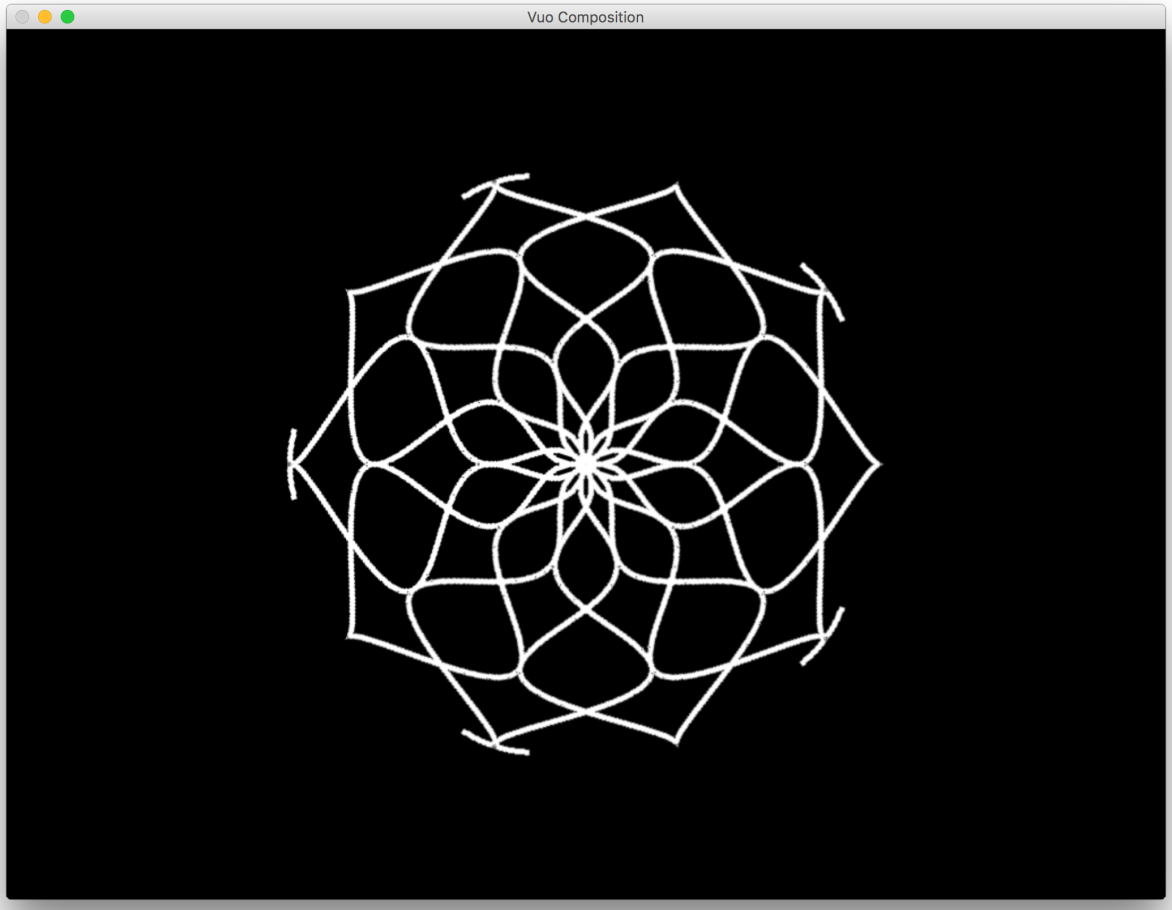
Add in a **Rotate Image** node, and connect the output from the **Render Scene to Image** node to its *Image* input. Then connect the output from the **Divide** node to the *Rotation* input.

Now, finally, add in the **Make Kaleidoscope Image** node. Connect the output from **Rotate Image** to the *Image* input, and the output from **Sides** to *Sides*.

The composition should now look like this:



And the output like this:



I hope this was interesting, and a useable primer for further exploration of kaleidoscopes and how to use & abuse them for fun and profit. In part 2 I will take a deeper look at how to dress the kaleidoscopes up nicely along with some tips and tricks to add a sense of depth.