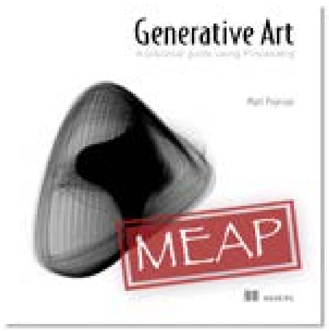
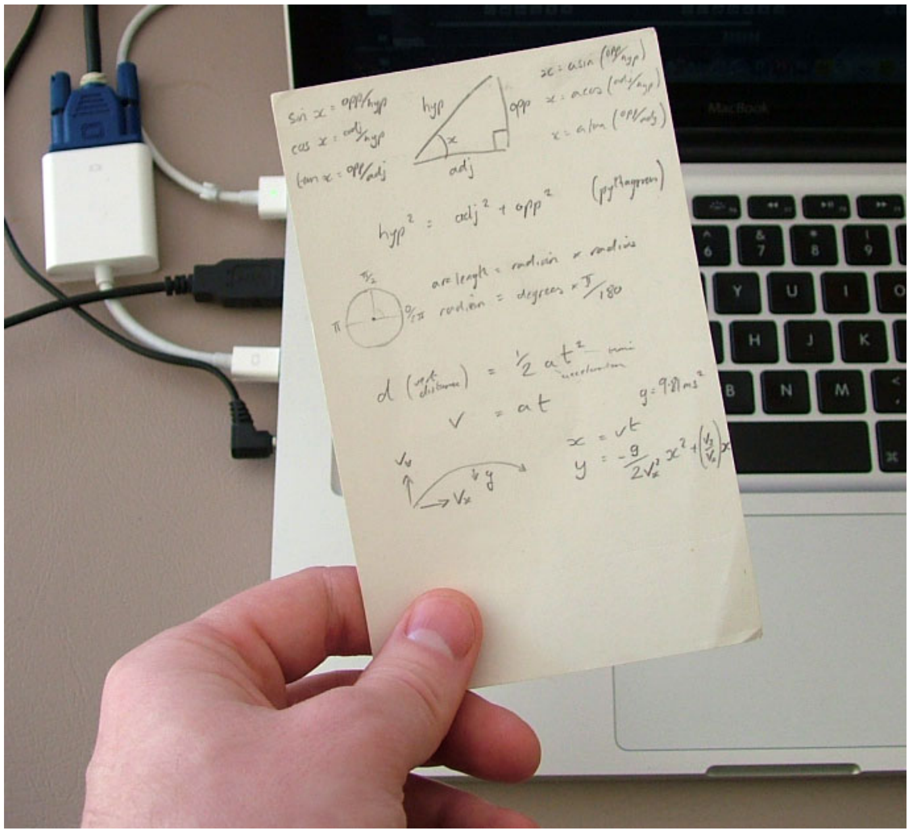
MANNING PUBLICATIONS



Rotational Drawing

It didn't take much googling to discover that, after subjecting my poor adolescent brain to two years of A-level mathematics, all the trig I have ever really needed to know could fit quite easily onto the back of a postcard. I scribbled the arcane markings (figure 1) sometime during my coding adolescence (a period I don't think I've ever really left). I have had this postcard tucked into a book in my office for over ten years now and I have never needed much more, although the book it was tucked into has changed many times.



$\sin x = opp/hyp$

$\cos x = adj/hyp$

$\tan x = opp/adj$

$x = a\sin(opp/hyp)$

$x = a\cos(adj/hyp)$

$x = a\tan(opp/adj)$

$hyp^2 = adj^2 + opp^2$ (pythagoras)

arc length = radian × radius

radian = degrees × $\pi/180$

d (distance) $= \frac{1}{2}at^2$

$v = at$

$g = 9.81\,ms^{-2}$

$x = vt$

$y = -\frac{g}{2V_x^2}x^2 + \left(\frac{V_y}{V_x}\right)x$

hyp
opp
adj
x
$\pi/2$
$0/2\pi$
π
acceleration
V_y
V_x
g

Figure 1 All the trig I have ever needed fits onto the back of a postcard.

We're going to use a little of that trig now as we vandalize a circle.

Trigonometry

Let's start with the easy way of drawing a circle:

```
ellipse(x, y, diameter, diameter);
```

All we need for this method is a centre point, width, and height; width and height both being equal to the diameter of a circle. But there is a lot more fun to be had if we break it down a little. To do so, we're going to use two equations from my postcard cheat-sheet, `sin x = opp/hyp` and `cos x = adj/hyp`. The sine of the angle `x` is equal to the opposite over the hypotenuse. The cosine of the angle `x` is equal to the adjacent over the hypotenuse.

Knowing the centre point of the circle, the hypotenuse between the centre and a point on the circumference is the radius. The opposite and adjacent then correspond to the difference in `X` and `Y` positions from the centre point. By jigging the equations, we can see that the coordinates of any point on a circumference of a circle can be calculated relative to the angle as follows:

```
X = center x + (radius * cos(angle));
Y = center y + (radius * sin(angle));
```

NOTE: Radians and degrees

Angles used by trigonometric functions are represented in radians, not degrees. One radian is equal to 180/π, so a full revolution of a circle (360 degrees) is 2π radians. If you are not comfortable with radians, you can work in degrees and convert before calling a trigonometric function. The formula is on my postcard, `radian = degrees * (180/π)`.

Figure 2 shows the circle drawn using the code in listing 1. To highlight the math in action we have marked the steps of the loop as points.

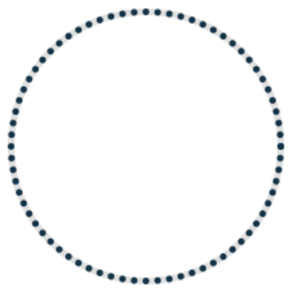


Figure 2 Drawing a circle using trigonometry, the output of listing 1

To complete the circle we would draw lines connecting these points. Increments of 5 degrees will be smooth enough for this example, but we can decrease this increment later, if we want.

Listing 1 Drawing a circle, first the easy way, then using trigonometry

```
size(500,300);
background(255);
strokeWeight(5);
smooth();
```

```
float radius = 100;
int centX = 250;
int centY = 150;

stroke(0, 30);
noFill();
ellipse(centX,centY,radius*2,radius*2);

stroke(20, 50, 70);
float x, y;
float lastx = -999;
float lasty = -999;
for (float ang = 0; ang <= 360; ang += 5) { #A   float rad = radians(ang);   #1
  x = centX + (radius * cos(rad));
  y = centY + (radius * sin(rad));
  point(x,y);
}
```

#A 360 degree loop
#1 Angles have to be converted from degrees to radians for trig calculations

That's all we need. That's as hard as the trig is going to get within these pages. We've now got a systematic way of drawing a circle, so the next job is to start tweaking it into something less rigid.

Noisy spirals

Now that we have a greater control over the drawing, we can perform tricks like turning our circle into a spiral. We simply increase the radius as the angle turns. Note that we'll need to turn more than just 360 degrees though. The required changes to the code are highlighted in listing 2, with the output in figure 3.

Listing 4.2 Code changes required to turning our circle into a spiral

```
radius = 10;        #1
float x, y;
float lastx = -999;
float lasty = -999;
for (float ang = 0; ang <= 1440; ang += 5) {      #2
  radius += 0.5;
  float rad = radians(ang);
  x = centX + (radius * cos(rad));
  y = centY + (radius * sin(rad));
  if (lastx > -999) {
    line(x,y,lastx,lasty);
  }
  lastx = x;
  lasty = y;
}
```

#1 Starts small
#2 Loops around the 360 degrees 4 times

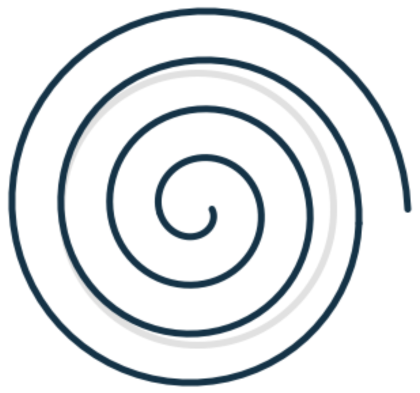


Figure 3 A few minor tweaks and our circle becomes a spiral

So far, so boring, so let's add some noise. In listing 3, we increment the radius as before but vary it by a rather large noise factor as well.

Listing 3 Adding noise to the spiral

```
radius = 10;
float x, y;
float lastx = -999;
float lasty = -999;
float radiusNoise = random(10);
for (float ang = 0; ang <= 1440; ang += 5) {
  radiusNoise += 0.05;
  radius += 0.5;
  float thisRadius = radius + (noise(radiusNoise) * 200) - 100;
  float rad = radians(ang);
  x = centX + (thisRadius * cos(rad));
  y = centY + (thisRadius * sin(rad));
  if (lastx > -999) {
    line(x,y,lastx,lasty);
  }
  lastx = x;
  lasty = y;
}
```

A possible output can be seen below (figure 4).

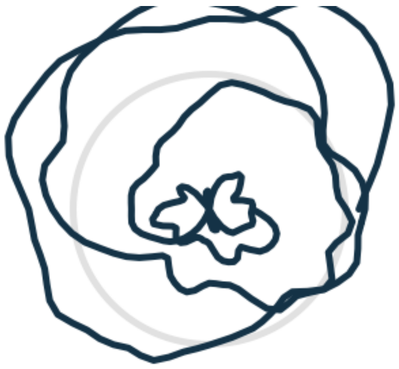


Figure 4 Our spiral with noise. Is that a butterfly in the middle there?

Okay, now I would hope you agree that, by the time we have got to this stage, we began seeing something interesting emerging, something we might call "generative". These shapes have a basis in mathematical seeds we have programmed, but there is enough randomness about the way they are drawn to have taken on a form of their own.

The next step—and I firmly believe that, if in doubt, this should *always* be your next step—is to multiply it all by 100 (figure 5).

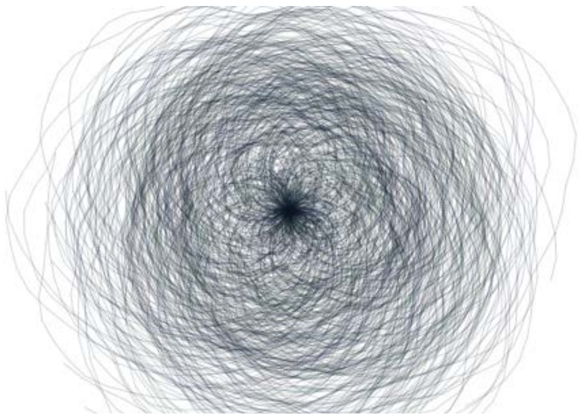


Figure 5 Our spiral code turned up to 11

In this version, we have used a `for` loop to draw the same spiral 100 times. We have also lightened the line weight and for each spiral randomized the stroke colour, the alpha, the starting angle, the end angle, and the angle step. The full code is in listing 4.

Listing 4 100 spirals, with noise

```
size(500,300);
background(255);
strokeWeight(0.5);    #1
smooth();

int centX = 250;
int centY = 150;

float x, y;
for (int i = 0; i<100; i++) {    #2
  float lastx = -999;
  float lasty = -999;
  float radiusNoise = random(10);
  float radius = 10;
  stroke(random(20), random(50), random(70), 80); #3
  int startangle = int(random(360)); #4
  int endangle =  1440 + int(random(1440)); #5
  int anglestep =  5 + int(random(3)); #6
  for (float ang = startangle; ang <= 1440 + random(1440); ang += anglestep) {
    radiusNoise += 0.05;
    radius += 0.5;
    float thisRadius = radius + (noise(radiusNoise) * 200) - 100;
    float rad = radians(ang);
    x = centX + (thisRadius * cos(rad));
    y = centY + (thisRadius * sin(rad));
    if (lastx > -999) {
      line(x,y,lastx,lasty);
    }
    lastx = x;
    lasty = y;
  }
}
```

#1 Makes the line finer
#2 Draws 100 of these spirals
#3 Randomizes the colour and add a bit of transparency
#4 Randomizes where the spiral starts
#5 Randomizes where the spiral ends
#6 Randomizes the increments

It's a mess, but it's an interesting mess. I wouldn't print this on a greeting card quite yet, but it's a start. Although I would hope you can see from the process how the application of noise and/or randomness to a systematic drawing process can lead us towards aesthetically interesting results (figure 6).

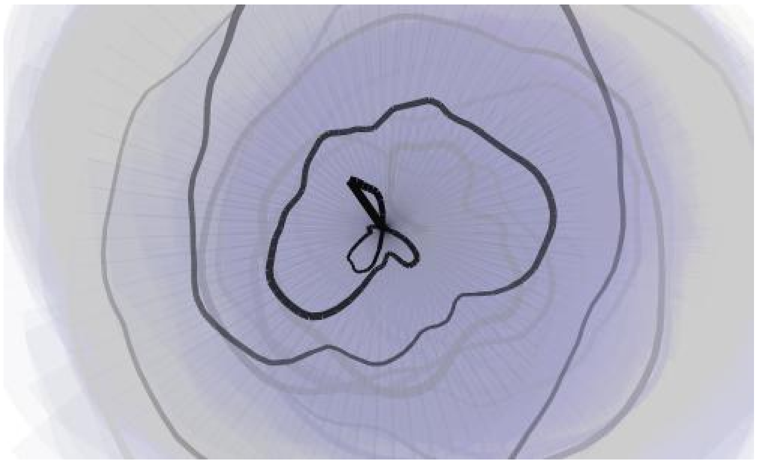


Figure 6 Spiral Stairs (2009)

Through this process of abstracting a mathematically constructed shape, we move the visual from order towards chaos. As long as we don't go too far toward the disordered end of the spectrum, the abstraction stays interesting. And if it is interesting, we can call it art.

Creating our own noise

To stop us getting too dependent on Perlin Noise, useful though it is, let us try our circle drawing once more, but this time producing our own naturalistic variance.

Perlin Noise

There aren't many programming awards that your mom has heard of. So upon the day Ken Perlin bought home an Oscar for an algorithm he had created, I'd imagine it was a rare case of a programmer's mother phoning round all her friends and praising his decision to devote his life to communing with the calculating machines.

The Technical Achievement Oscar Ken rather belatedly received in 1997 was for his work on procedural textures developed as part of the graphics for the science fiction film *TRON* (1982). *TRON* (figure 1) was a pioneering movie both in subject matter and production. It made a hero of a video gamer, at a time when video games were seen as an idiotic frivolity, and much of the film took place in *virtual reality*, two years before William Gibson's *Neuromancer* was credited with popularizing the concept of cyberspace. Most significantly though, it was one of the first films to rely heavily on computer graphics for its special effects.

Ken's job was to develop textures to give 3D objects a more natural look, so he was looking for ways of generating randomness in a controlled fashion. The function he came up with to do this—*Perlin Noise*—is really just another pseudo-random function, but it is a pretty good one, one specifically attuned to naturally looking visuals. If you are interested in the logic behind it, you can view the code along with examples of Ken's applications of it on his website at http://mrl.nyu.edu/~perlin/.

By the late eighties, Ken Perlin's noise functions were pretty much ubiquitous. It is probably safe to say that, if you have ever played a graphical console game, or seen a computer-animated film, or seen CGI used in any movie made in the last twenty years, you have seen Ken's work in action. Now, with *Processing,* you have Ken's algorithms encapsulated into a simple function, `noise()`, enabling you to harness the same naturalistic variance you have seen on the big screen in your own creations.

We will take a custom `random()` function for the drawing of a line and adapt it into a custom `noise()` function, one that calculates a return value according to the seed value it is passed, which we can then apply to our circle.

To do this we're going to have to organize things a little first. In listing 5, we take the circle drawing code from listing 1 and:

1. Rewrite it into function blocks.
2. Add the `customNoise()` function.

Listing 5 A circle, drawn with a custom noise function

```
void setup () {
  size(500,300);
  background(255);
  strokeWeight(5);
  smooth();

  float radius = 100;
  int centX = 250;
  int centY = 150;

  stroke(0, 30);
  noFill();
  ellipse(centX,centY,radius*2,radius*2);

  stroke(20, 50, 70);
  strokeWeight(1);
  float x, y;
  float noiseval = random(10); #1

  beginShape();
  fill(20, 50, 70, 50);
  for (float ang = 0; ang <= 360; ang += 1) {

    noiseval += 0.1;
    float radVariance = 30 * customNoise(noiseval);

    float thisRadius = radius + radVariance;
    float rad = radians(ang);
    x = centX + (thisRadius * cos(rad));
    y = centY + (thisRadius * sin(rad));

    curveVertex(x,y);
  }
  endShape();
}

float customNoise(float value) {    #A
   float retValue = pow(sin(value), 3);
   return retValue;
}
```

#1 Randomizes the start point
#A Returns value -1 to +1

There is one other crucial change we have made in listing 5. Instead of drawing lines back to the previous point, this time we are using `beginShape()` and `endShape()` to draw an enclosed area to which we have applied a `fill()`. Points are added to the shape with `curveVertex(x,y)`. This results in something a little like figure 7.

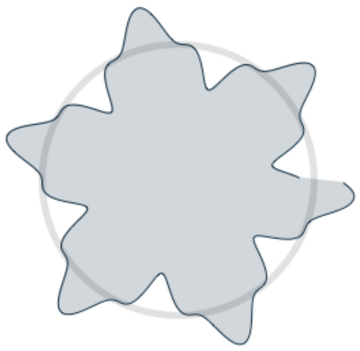


Figure 7 The output of listing 5, a custom variance to the circumference based on sine

For this first example, `customNoise()` is returning the value of sine cubed, which gives us a regular repeating variance. But, now that we have the structure, we can experiment with the mathematics within our new function. Let’s try, for example, using the seed value to determine the power we raise sine to:

```
float customNoise(float value) {
   int count = int((value % 12));
   float retValue = pow(sin(value), count);
   return retValue;
}
```

Using this function, we end up with something looking like figure 8.

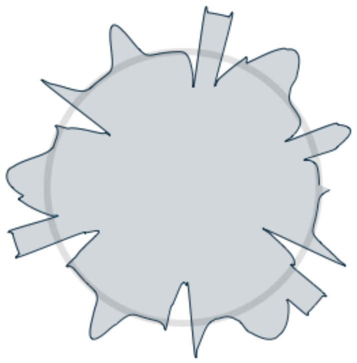


Figure 8 Another custom variance based on powers of sine

We now have a custom variance, one we can call our own, which we can apply to lines, shapes, movement, or anything else we might apply `noise()` to.

The next stage would be to continue throwing whatever mathematics you know into the function and seeing what other glorious messes our routine can create. It doesn’t matter if your math is poor; there is plenty that can be achieved with just addition, subtraction, multiplication, and division. The complexity of the math doesn’t necessarily correspond to the interestingness of the visual. Juggle some numbers, alter the shapes or the drawing style, turn it up to eleven and see if you can surprise yourself (figure 9).

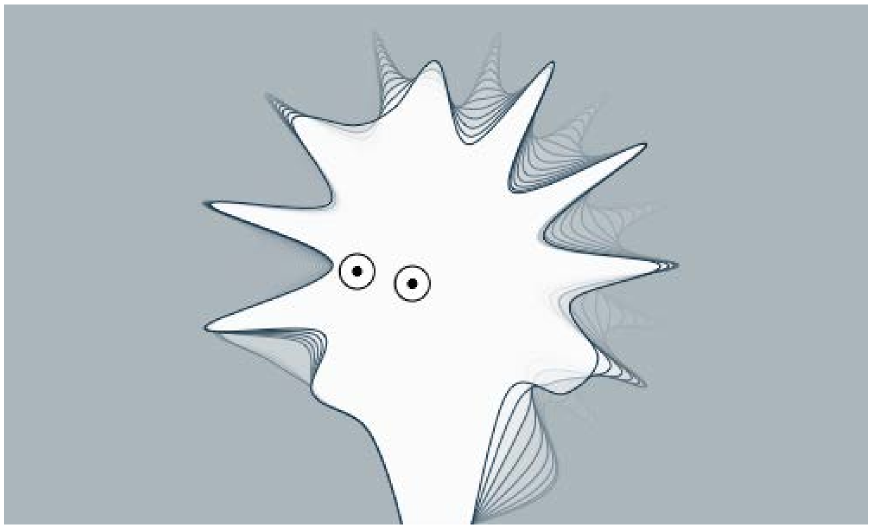


Figure 9 Schmoo (2009). You can see the source code for this at http://abandonedart.org/?p=549, It uses a simple custom noise function much like we have created in this section.

Summary

We have gone from a wonky way of drawing a line, to a complete generative system, one that churns out some rather pleasing results. The basic methodology we have established 1) deconstructs a machine-drawn shape and 2) reconstructs it with some form of unpredictability. This simple technique can be applied to make just about anything more interesting.

Last updated: June 29, 2011