Chapter Four

4

The Wrong Way to Draw a Circle

To most, *trigonometry* is a scary word. I'll be the first to confess that it utterly bewildered me at school. I remember thinking to myself that I would *never* use this crap in the real world. And I was right—for a couple of years at least.

Until sometime in my late twenties, I found myself making Flash games for a living, and more and more I found bits of half-remembered trig proving useful.

In this chapter, we'll play with unconventional ways of drawing circles, for which we'll need a little trig. In the same way we wandered away from the useful convenience of the **line** function in the last chapter, we won't rely on Processing's **ellipse** function here. Instead, by working out the math ourselves, we'll be able to tease out a more nonmechanical, imprecise way of drawing this one shape. Many of my own works have been based on this single idea—rotational drawing algorithms—and so I'll offer one of them as a case study later in the chapter.

4.1 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've ever really needed to know fits quite easily onto the back of a postcard. I scribbled the arcane markings you see in figure 4.1 sometime during my coding adolescence (a period I don't think I've ever really left). I've had this card tucked into a book in my office for more than 10 years now, and I've never needed much more (although the book it's tucked into has changed many times).

We'll use a little of that trig now, as in the same way we deconstructed the drawing of a line, we vandalize the circle.

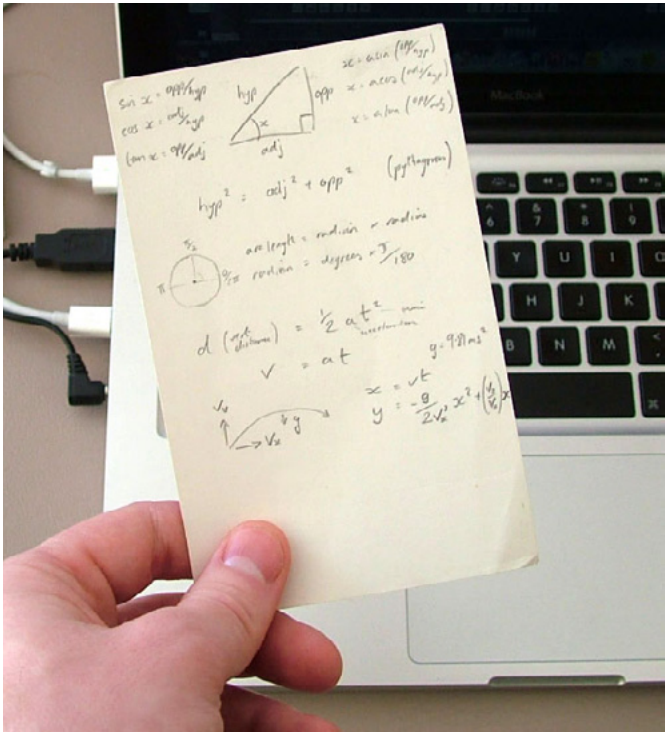

Figure 4.1
All the trig I've ever needed fits on the back of a postcard.

4.1.1 Drawing your first circle

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

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

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

Knowing the center point of the circle, the hypotenuse between the center 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 center point. By jigging the equations, you can calculate the coordinates of any point on the circumference of a circle relative to the angle as follows:

```
X = centerX + (radius * cos(angle));
Y = centerY + (radius * sin(angle));
```

Figure 4.2 shows the circle drawn using the code in listing 4.1. To highlight the math in action, I've marked the steps of the loop as points.

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

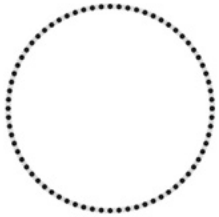

Figure 4.2
Drawing a circle using trigonometry: the output of listing 4.1

Radians and degrees

Angles used by trigonometric functions are 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 aren't 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), but Processing has two handy conversion functions already built in - **degrees** and **radians**.

Listing 4.1 **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) {
 float rad = radians(ang);
 x = centX + (radius * cos(rad));
 y = centY + (radius * sin(rad));
 point(x,y);
}
```

360-degree loop

Converts from degrees to radians

That's all you need. That's as hard as the trig is going to get within these pages. You now have a systematic way of drawing a circle; the next job is to begin tweaking it into something less rigid.

Figure 4.3
A few minor tweaks, and the circle becomes a spiral.

4.1.2 Turning a circle into a spiral

Now that you have greater control over the drawing, you can perform tricks like turning a circle into a spiral. To do so, you simply increase the radius as the angle turns. Note that you'll need to turn more than just 360 degrees, though. The required changes to the code are highlighted in the following listing, and the output is shown in figure 4.3.

Listing 4.2 Code changes required to turn a circle into a spiral

```
radius = 10;                                          Starts small
float x, y;
float lastx = -999;
float lasty = -999;
for (float ang = 0; ang <= 1440; ang += 5) {          Loops four times
 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;
}
```

So far, so boring. Let's add some noise.

4.1.3 Noisy spirals

Listing 4.3 increments the radius as before but also varies it by a rather large noise factor. A possible output is shown in figure 4.4.

Listing 4.3 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;
```

(continued on next page)

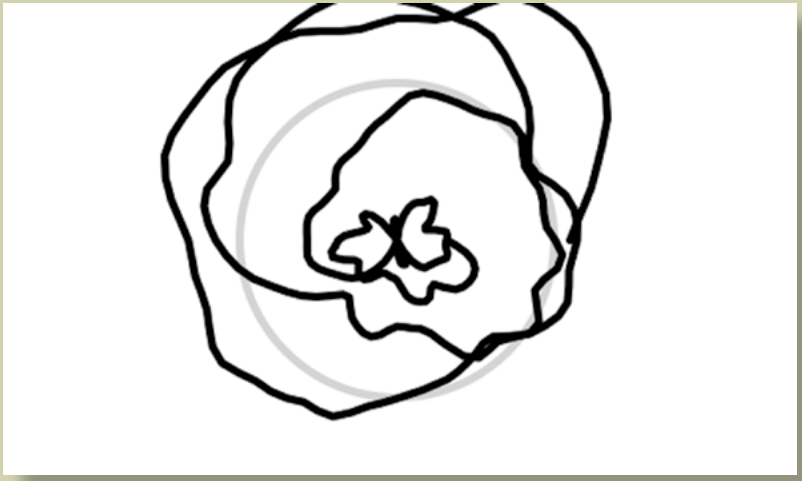

Figure 4.4
The spiral, with noise. Is that a butterfly in the middle there?

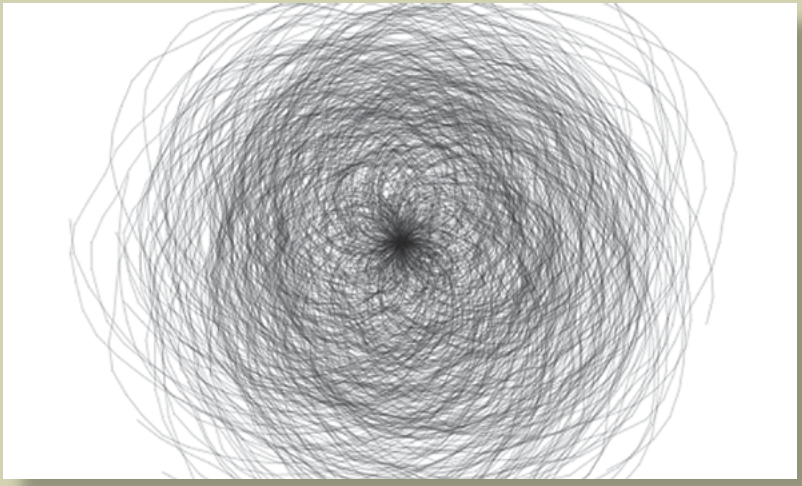

Figure 4.5
The spiral code turned up to 11

```
  radius += 0.5;                                          (Listing 4.3 continued)
  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;
}
```

I hope you agree that by the time you've reached this stage, you're starting to see something interesting emerging—something you might call *generative*. These shapes have a basis in mathematical seeds you've programmed, but there is enough randomness about the way they're drawn that they're taking 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 (see figure 4.5).

In this version, you use a **for** loop to draw the same spiral 100 times. You also lighten the line weight; and for each spiral, you randomize the stroke color, the alpha, the starting angle, the end angle, and the angle step. The full code is in the following listing.

Listing 4.4 100 spirals, with noise

```
size(500,300);
background(255);
strokeWeight(0.5);                                        Finer line
smooth();

int centX = 250;
int centY = 150;

float x, y;
for (int i = 0; i<100; i++) {                             Draws 100 spirals
  float lastx = -999;
  float lasty = -999;
  float radiusNoise = random(10);
  float radius = 10;                                      (continued on next page)
```

```
  stroke(random(20), random(50), random(70), 80);
```
(Listing 4.4 continued)

Randomizes spiral aspects

```
  int startangle = int(random(360));
  int endangle =  1440 + int(random(1440));
  int anglestep =  5 + int(random(3));
  for (float ang = startangle; ang <= endangle; 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;
  }
}
```

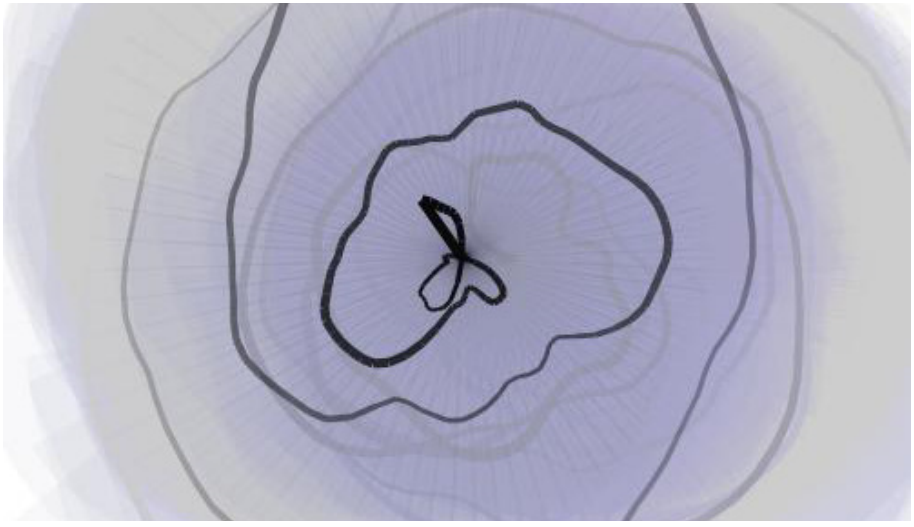

Figure 4.6
Spiral Stairs (2009)

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. I *do* hope you can see how the application of noise and/or randomness to a systematic drawing process can lead us toward aesthetically interesting results (see figure 4.6, for example).

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

4.1.4 Creating your own noise, revisited

To stop you from getting too dependent on Perlin noise, useful though it is, let's try the circle drawing once more, but this time produce your own naturalistic variance. In the last chapter, you created a custom **random** function for the drawing of a line. Now you can take that a step further and adapt that into a custom **noise** function, which calculates a return value according to the seed value it's passed; you can then apply this function to your circle.

To do this, you'll have to organize things a little first. In listing 4.5, you take the circle-drawing code from listing 4.1 and do the following:

- Rewrite it into function blocks.
- Add the **customNoise** function.

Listing 4.5 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);
```

(continued on next page)

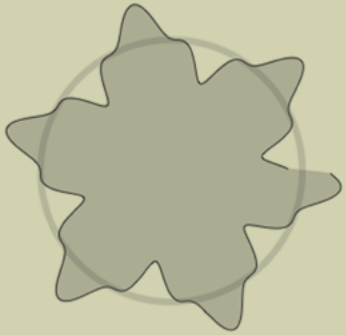

Figure 4.7
The output of listing 4.5: a custom variance to the circumference, based on sine

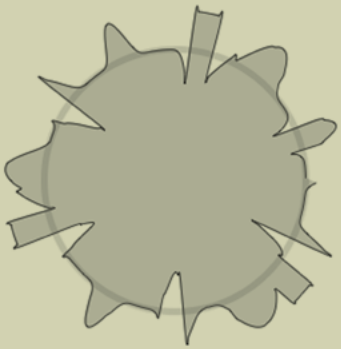

Figure 4.8
Another custom variance based on powers of sine

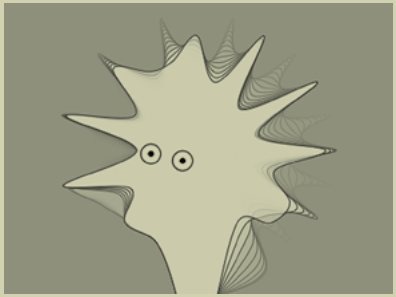

Figure 4.9 You can see the source code for this animation at http://abandonedart.org/?p=549. It uses a simple custom **noise** function much like you've created in this section.

(Listing 4.5 continued)

```
  float x, y;
  float noiseval = random(10);
  float radVariance, thisRadius, rad;
  beginShape();
  fill(20, 50, 70, 50);
  for (float ang = 0; ang <= 360; ang += 1) {

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

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

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

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

Randomizes start point

Returns value -1 to +1

You make one other crucial change in listing 4.5. Instead of drawing lines back to the previous point, as you did before, this time you use **beginShape** and **endShape** to draw an enclosed area to which you apply a **fill**. You add points to the shape with **curveVertex(x,y)**. The result is something a little like figure 4.7.

For this first example, **customNoise** returns the value of sine cubed, which gives you a regular repeating variance. Now that you have the structure, you can experiment with the mathematics in your new function. Let's try, for example, using the parameter value to determine the power to which you raise sine:

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

Using this function, you end up with something like figure 4.8.

Okay, it doesn't quite have the beauty of Perlin noise, so I don't think this one will win the Oscar. But you now have a custom variance you can call your own, which you can apply to lines, shapes, movement, or anything else you might apply **noise** to.

The next stage is to continue throwing whatever mathematics you know into the function and seeing what other glorious messes the routine can create (see, for example, figure 4.9). It doesn't matter if your math is poor—you can achieve plenty 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 11, and see if you can surprise yourself.

In the next section, I'll walk you through my process in creating one of my own Perlin noise–based works. There are a thousand creative things you can do with a set of numbers in a naturally varying sequence (we'll look at a few more in the next chapter), but I hope a single practical example at this point will be sufficient to inspire you to apply the techniques of this chapter and the last to your own ideas.

4.2 Case study: *Wave Clock*

My *Wave Clock* (see figure 4.10) is a nice, simple example of using Perlin noise to produce a natural-looking form. I can't say I was working toward any particular goal at the time I fashioned the system, but I can still recount the steps I took so you can gain some appreciation of my process.

I began by plotting a circle, exactly as you did in the previous section, and then drawing a line from each point on the circumference to its opposite point. I calculated the opposite point by adding pi to the current angle, which, if you're using radians, rotates it 180 degrees. I then extrapolated the two circumference points using the same sine and cosine functions:

```
float rad = radians(_angle);

float x1 = centerX + (_radius * cos(rad));
 float y1 = centerY + (_radius * sin(rad));

float opprad = rad + PI;
float x2 = centerX + (_radius * cos(opprad));
 float y2 = centerY + (_radius * sin(opprad));

 line(x1, y1, x2, y2);
```

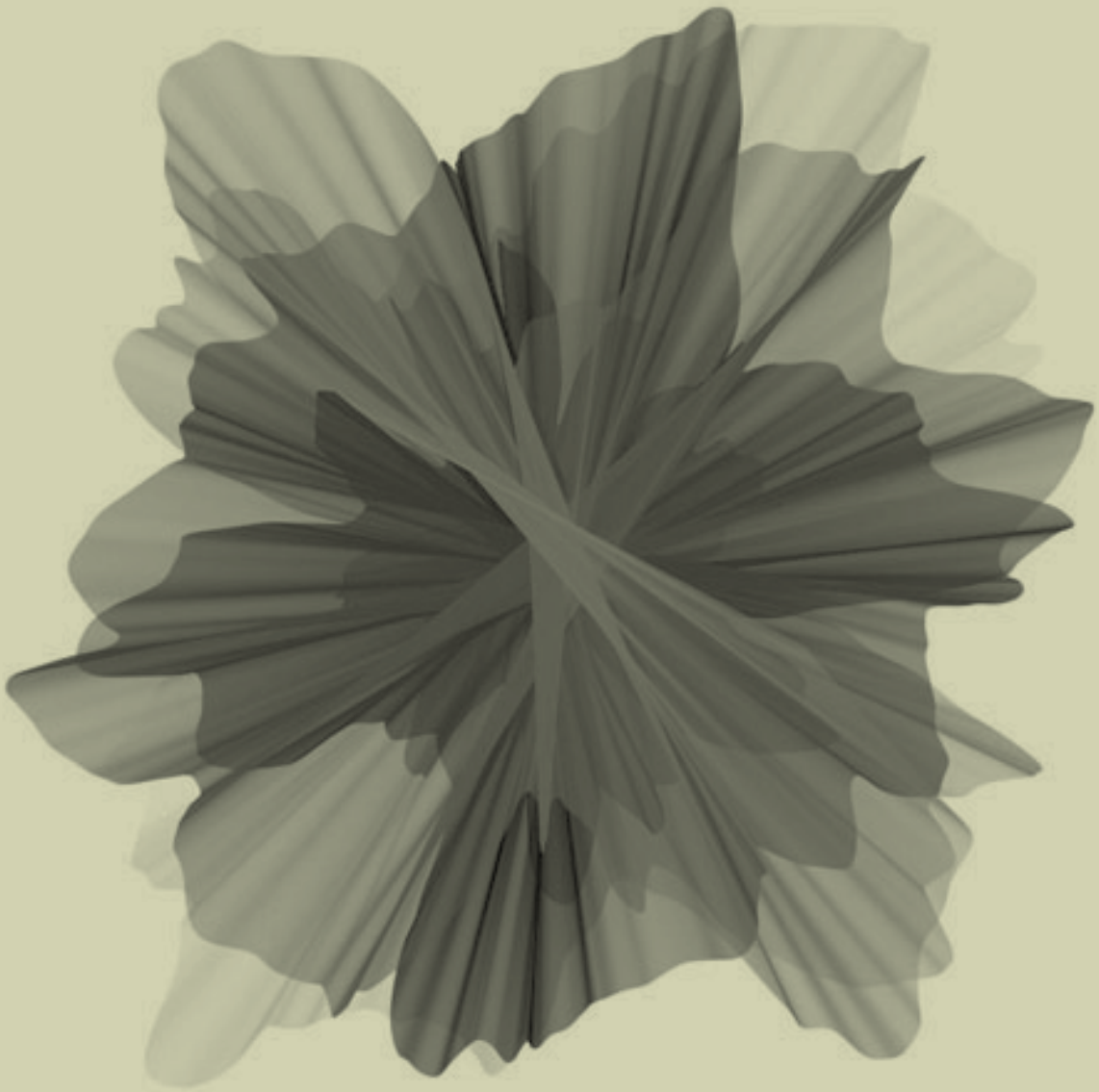

Figure 4.10
Wave Clock (2009)

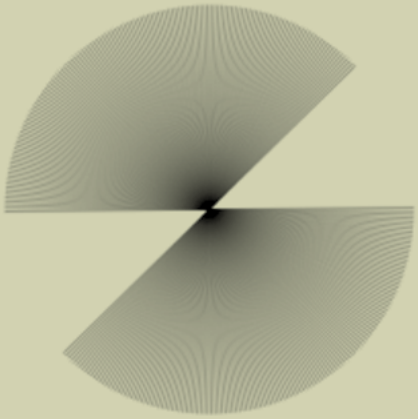

Figure 4.11

A circle constructed by drawing lines from a point on the circumference to its opposite edge

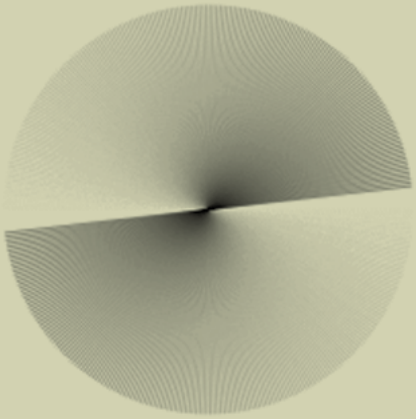

Figure 4.12

Fade effect added by incrementing the stroke color as the line rotates

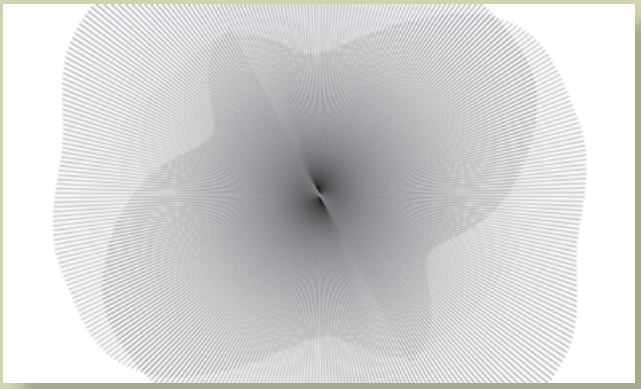

Figure 4.13

Perlin noise variance added to the radius value

Figure 4.14

Perlin noise variance added to the rotation speed/direction

Figure 4.11 shows the output.

This produced a nice, smooth circle. To give it a fade, I added a few lines to vary the stroke color. I defined a **strokeCol** variable, started it at 255 (white), and decremented it by 1 every frame until it reached 0 (black). Then, I reversed the process back to 255 *ad infinitum*. The result is shown in figure 4.12.

Next, I made the length of the connecting line change by varying the radius with a noise value as it rotated (see figure 4.13).

```
_radiusnoise += 0.005;
_radius = (noise(_radiusnoise) * 550) +1;
```

Then, to give it a further spin, I varied the density of the line packing by adding a second noise variance to the angle of rotation. I allowed the angle to increase or decrease so the rotation could reverse. This gave the effect you can see in figure 4.14:

```
_angnoise += 0.005;
_angle += (noise(_angnoise) * 6) - 3;
if (_angle > 360) { _angle -= 360; }
if (_angle < 0) { _angle += 360; }
```

Finally, for one extra touch of non-linearity, I wobbled the center of the circle slightly, again using a noise function to move it plus or minus 10 pixels in the x or y direction:

```
_xnoise += 0.01;
_ynoise += 0.01;
float centerX = width/2 + (noise(_xnoise) * 100) - 50;
float centerY = height/2 + (noise(_ynoise) * 100) - 50;
```

The final code for this system is in the following listing.

Listing 4.6 Final code for *Wave Clock*

```
float _angnoise, _radiusnoise;
float _xnoise, _ynoise;
float _angle = -PI/2;
float _radius;
float _strokeCol = 254;
int _strokeChange = -1;



void setup() {
```

(continued on next page)

```
  size(500, 300);                                                    (Listing 4.6 continued)
  smooth();
  frameRate(30);
  background(255);
  noFill();

  _angnoise = random(10);
  _radiusnoise = random(10);
  _xnoise = random(10);
  _ynoise = random(10);
}

void draw() {

  _radiusnoise += 0.005;
  _radius = (noise(_radiusnoise) * 550) +1;

  _angnoise += 0.005;
  _angle += (noise(_angnoise) * 6) - 3;
  if (_angle > 360) { _angle -= 360; }
  if (_angle < 0) { _angle += 360; }

  _xnoise += 0.01;
  _ynoise += 0.01;
  float centerX = width/2 + (noise(_xnoise) * 100) - 50;
  float centerY = height/2 + (noise(_ynoise) * 100) - 50;

  float rad = radians(_angle);
  float x1 = centerX + (_radius * cos(rad));
  float y1 = centerY + (_radius * sin(rad));

  float opprad = rad + PI;
  float x2 = centerX + (_radius * cos(opprad));
  float y2 = centerY + (_radius * sin(opprad));

  _strokeCol += _strokeChange;
  if (_strokeCol > 254) { _strokeChange = -1; }
  if (_strokeCol < 0) { _strokeChange = 1; }
  stroke(_strokeCol, 60);                                  (continued on next page)
```

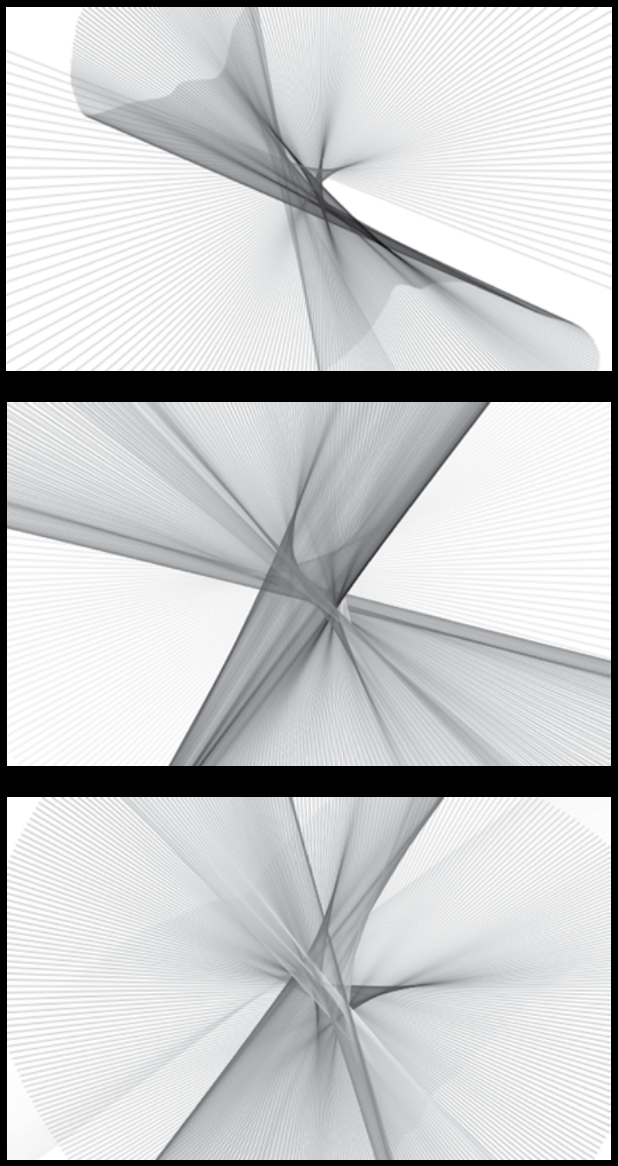

Figure 4.15
Outputs from the completed system: *Wave Clock* (2009)

```
    strokeWeight(1);                                    (Listing 4.6 continued)
    line(x1, y1, x2, y2);
  }
```

Figure 4.15 shows a few possible outputs.

4.3 Summary

In this chapter, we've 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 established in these last two chapters is as follows:

- Deconstruct a machine-drawn shape.
- Reconstruct it with some form of unpredictability.

You can apply this simple technique to make just about anything more interesting.

There is plenty of further fun to be had playing with circles—they're one shape it's hard to tire of. But you may now want to try other computer-drawn forms. I'll leave it with you experiment.

In the next chapter, before we move on to even cooler organic methodologies in part 3 of the book, we'll take noise into a few extra dimensions and also see what happens when we use it to manipulate both time and space.