6

Chapter Six

Emergence

I live in Brighton, on the southern coast of England. I moved here after a lifetime of living in urban environments; so I was, and remain, enraptured by the view to the south, where there is nothing but sea and sky. I used to walk home past Brighton's West Pier, a once-beautiful Victorian pleasure palace, abandoned since the 1970s and burned to a skeleton in 2003. It's now home to many thousands of starlings, and every night shortly before the sun goes down they perform a stunning murmation, circling in complex formations before settling to roost (see figure 6.1).

I find these formations mesmerizing, because they appeal to both sides of my brain. One hemisphere appreciates the natural aesthetic beauty of these creatures in motion and the shapes they form, but there is also the mathematical beauty of the flocking algorithm they are following, that challenges my logical side. Each bird in the formation is obeying simple instincts, oblivious to the higher-level pattern the flock is creating. Their behavior is defined by a small set of rules—orienting themselves in relation to their immediate neighbors, seeking the center of the flock, avoiding the paths of other birds—but these simple low-level rules, when applied en masse, create patterns of abstract beauty on a higher level.

6.1 Emergence defined

This phenomenon, whereby a simple rule set at a low level creates organized complexity on a higher level, is called *emergence*. The term was first coined in the mid-19th Century by George Henry Lewes, the English psychologist, philosopher, and sometime lover of George Eliot, in a paper on the mind; but notions of the whole being greater than the sum of its parts can be dated back as far as Aristotle. In recent decades, the concept of emergence has found a new relevance and wider popularity through its application to Complexity theory, first by John Holland in his book *Emergence* (1989), and later for a more popular audience by Steven Johnson in a 2001 book of the same name.

In a nutshell, *emergence* is the observation of how complex and coherent patterns can arise from a large number of small, very simple interactions. The classic example is the ant colony, an organism that has clearly defined, logical and coherent behaviors observable on two different scales. When we study the patterns of each ant, we see that each one has needs, abilities, and pheromone responses that define its behavior as an individual insect. But when we study the collective behavior of the colony, we again see sophisticated behavioral patterns; the colony operating like a city, with factories, defenses, and waste-disposal facilities. What is remarkable is the fact that these macro patterns aren't formed through any central design or intent: they're nothing more than *byproducts* of the local self-interested behaviors of the individuals collectively. These behaviors, seemingly insignificant on the micro level, form a more complex macro-organism when viewed collectively.

This is the principle we'll be trying to harness in the next few chapters. When you create a large number of small agents with simple behaviors, collectively they can become capable of producing a more complex, emergent, behavior on the macro level.

6.1.1 Ant colonies and flocking algorithms

The common myth, still prevalent today among lazier minds (including the makers of animated films) is that ant colonies operate in a hierarchical organization, with the queen at the top of a chain of command. With our human-centric perspective, this is the type of organizational patterns we find ourselves seeking; but the myth has been debunked in studies as far back as the 1970s. No central organization defines the structure and operation of an ant colony. All these global-level patterns and structures arise as the result of a large number of local-level micro-decisions. Design, hierarchy, pacemakers, and any other form of higher-level influence have no part in it.

Figure 6.2

Blonde Boids (Abandoned Artwork #4) was my adaptation of Dan Shiffman's adaptation of Craig Reynolds Boids code.

The role of the queen is as perfunctory as that of every other ant with a job to perform within the greater whole. There is no trickle-down command; the ants' organization is from the bottom up. It is grass-roots. Emergent.

Similarly, the murmating starlings aren't following a leader as they swirl around the West Pier. The flock is made up of individuals, with individual behavioral impulses. Each starling looks only to its immediate surroundings and the movement of its closest neighbors to define the speed and orientation of its flight. This effect has been modeled in computing, most famously by Craig Reynolds in his *Boids* algorithm, first published in 1987. Reynolds discovered that to produce a realistic flocking simulation in code, he needed only three rules:

- *Separation*—Steer to avoid your immediate neighbors.
- *Alignment*—Steer to align with the average heading of your immediate neighbors.
- *Cohesion*—Steer toward the average position of your immediate neighbors.

More complex rules can be added (object avoidance, for example), but these three are all that are required to achieve a complexity comparable with the West Pier starlings. Dan Shiffman (and others) have created Boids implementations in Processing, which are fun to adapt. I won't reproduce the code here, because it's a little lengthy and more complex than required for the purposes of this chapter; but you can copy it from http://processing.org/learning/topics/flocking.html if you want to play with it. One example of my experiments with this code is shown in figure 6.2, and there are more sophisticated adaptations among the color images in the introduction.

It's difficult to get far with flocking algorithms, and other forms of emergent code, without a basic understanding of object-oriented approaches to programming. This is the main thrust of this chapter and something we'll get on to in section 6.2. But before we do, I want to give you a little more context on what emergence means from a sociological perspective.

6.1.2 Think locally, act locally

The concept of emergence, particularly in relation to both the natural world and society, has necessitated a steady political shift upon its path to wider, mainstream acceptance. The concept was extremely left-field in the 1950s (Alan Turing's final work before his death in 1954 was on the subject of morphogenesis: the way patterns formed in nature through emergent rules), groundbreaking in the 1970s and 80s, and the subject of excited discussion in the 1990s. It's only now becoming more comfortable and familiar in the web-savvy 21st century, because the internet has played no small part in changing the way we view other forms of organization.

The idea that a self-organized collective—be it the single cells of a slime mold, the ants of a colony, the human inhabitants of a city, or the globally distributed users of the web—can organize themselves into coherence is understandable and acceptable in a world where we conduct much of our intellectual discourse via the distributed nodes of the internet. Most of us can grasp how a network like Twitter, for example, can have a significant sociological meaning, even though each of its single 140-character tweets is near meaningless.[1] Each user's tweeting is either entirely self-interested or influenced by a small local group of a few hundred followers. But the meta-patterns of this chatter, when multiplied by hundreds of thousands, shape themselves into cultural shifts and global opinions. All we need is the right perspective, tools, or visualizations to be able to appreciate them.

Emergent networks like this have incredible but subtle strength and resilience. Their power is in their distribution. Each node is too small to be meaningful individually, so it's difficult to target, influence, or attack an emergent body. The internet avoids the vulnerabilities of a centralized organization in the same way a slime mold does: if a single node were to be lost or damaged, the greater whole would feel little effect. Distributed file-sharing networks share this principle, resilient to legislation because there is no central point to target.

1. The Twitter protocol itself may be seen as an emergent phenomenon. Its syntax (hashtags, @ addresses, and so on) were never designed by its creators but have been adopted from the bottom up, through their use by the collective.

Clearly, there are political connotations here. Emergence may suggest that, if it works so well for the natural world, central organization in the human world — governments, institutions — may be largely redundant. Ants follow local pheromone cues, oblivious to the effect of their decisions on the colony as a whole, just as humans give their local concerns (wealth, status, security, and so on) priority over the needs of the macro-organisms they form (groups, cities, countries). While our grander objectivity acknowledges that our car use is having devastating environmental effects on a global scale, it doesn't influence many to sacrifice the convenience a car allows them to get to the supermarket a little quicker. Even an understanding of the greater whole and how the effect works doesn't change the micro-behavior.

Many newer, more radical groups have acknowledged and embraced emergence as an organizing principle. Anti-globalization and environmental protest groups, for example, have explicitly modeled themselves after distributed, self-organized systems, as have the champions of file sharing.

There is a theory from the realms of neuroscience that argues that perhaps consciousness itself — the sense of self that is interpreting these words — is an example of emergence, being a macro-level coherence formed from the micro-interactions between firing synapses within our brains. Our whole sense of being may be an emergent effect.

It's a concept with many applications and theories across many disciplines; but for our interests, emergence is of relevance as a way of generating organic artworks. It's relevant first because emergence dictates how many forms of the natural world organize and construct themselves. If you wish to resonate with this aesthetic, you should pay attention to how it works. Second, within a methodology whereby you work with the simple logic of the programming language, where you encourage artworks to grow from logical seeds, and you wish to evolve a complexity sufficient to be visually interesting, the principles of emergence seem to be an ideal tool for getting complex results from simple code.

Until now, you've relied heavily on randomization and noise functions as ways of introducing unpredictability to your creations; but you'll discover that you don't always need such an obvious approach. You can program algorithms with a simplicity your animal brain can understand, but which through minimal abstraction can develop a complexity beyond what you could predict.

Over the coming pages we'll explore emergence in code. In the section to follow, you'll stumble across it as an accidental byproduct of simple programming. Then, in chapter 7, you'll create it again via the more organized approach of cellular automata, among other ideas. Chapter 8, on the subject of fractals, then builds on this way of programming, adding the further concept of self-similarity. Throughout the chapters in this final part of the book, you'll see copious examples of code that mirrors the organizational principles of the natural world.

6.2 Object-oriented programming

There is a mission to the rest of this chapter: we're going in search of the emergent complexity I just described. But in order to do this through the medium of code, we first need several new tools. In the next few pages, you'll learn the basics of object-oriented programming (OOP). If you're a seasoned programmer, you're probably already familiar with this concept—very few languages above a certain power and sophistication don't accommodate object-oriented structures. But even if you're on familiar ground, please bear with me, because OOP is only half of what I'm trying to illustrate.

If you've never heard the term before, OOP is just a small conceptual leap in how you think about coding. Rather than giving the machine the task of dealing with a linear progression of instructions, which is what you've done so far, you think in terms of creating *objects* in memory: self-contained data boxes that you can then use within the familiar linear programming style.

This conceptual shift is sometimes a little difficult for beginners, but it's nothing to be scared of. All it's really going to involve initially is drawing some circles and seeing where it takes you. Remember that what you're really doing in this chapter is seeking emergent complexity, and I guarantee it won't be long until you see some. First, let's look at classes and instances.

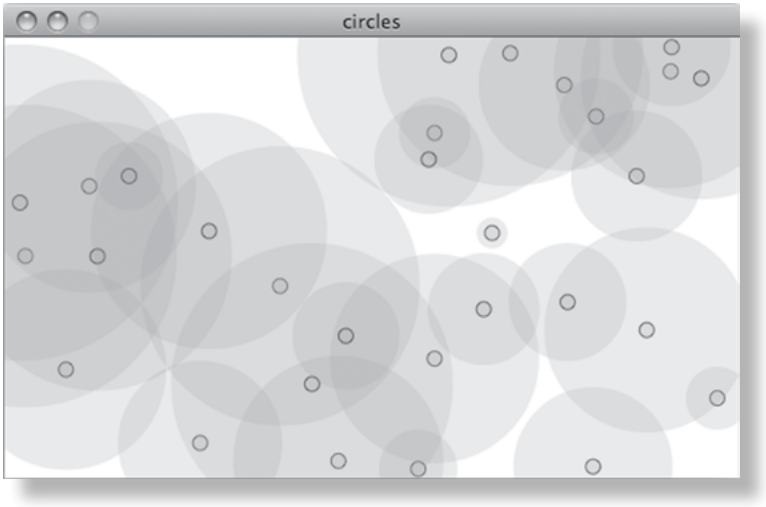

Figure 6.3
The output of listing 6.1: randomly generated circles

6.2.1 Classes and instances

Let's start simple. The following listing creates a script that draws a handful of randomly placed circles every time the mouse is clicked.

Listing 6.1 Code to draw a few circles at the click of the mouse

```
int _num = 10;

void setup() {
  size(500,300);
  background(255);
  smooth();
  strokeWeight(1);
  fill(150, 50);
  drawCircles();
}

void draw() {

}

void mouseReleased() {
  drawCircles();
}

void drawCircles() {
  for (int i=0; i<_num; i++) {
    float x = random(width);
    float y = random(height);
    float radius = random(100) + 10;
    noStroke();
    ellipse(x, y, radius*2, radius*2);
    stroke(0, 150);
    ellipse(x, y, 10, 10);
  }
}
```

I hope you can follow what's happening here. The script creates 10 circles at random positions on the screen every time you click the mouse. Try it a few times to get something similar to figure 6.3.

So far, the circles are pretty unsophisticated. They're thrown at the screen and left there, and you don't have any way to do anything further with them. If you reapproach the circles and define them in a more organized way, you'll find they're capable of a little more life. Instead of just telling Processing to draw a circle, let's instead create a circle *object*. The circle object will encapsulate everything there is to know about itself, which at the moment isn't much more than the x,y of its center point, and its radius.

An object, in programming, isn't too dissimilar from the concept of an object in the real world. Imagine a rock: this is an object. It has *properties*—color, weight, roughness—just as the circle object has an x,y and a radius. There are also things you can do with the rock: it has *functions*; it can be thrown, it can be hit with a hammer, it can be placed on a desk to use as a paperweight. It can also be disposed of before it becomes too convoluted a metaphor; let's do that and look instead at how you model an object in Processing.

Defining an object is very similar to defining a function. You use the **class** keyword to specify that this is a reusable object you're defining. Add the contents of the next listing to the bottom of your sketch.

Listing 6.2 **A Circle class**

```
class Circle {
  float x, y;
  float radius;
  color linecol, fillcol;
  float alph;

  Circle () {
    x = random(width);
    y = random(height);
    radius = random(100) + 10;
    linecol = color(random(255), random(255), random(255));
    fillcol = color(random(255), random(255), random(255));
    alph = random(255);
  }
}
```

Object properties

Object constructor

Object color and alpha

What you have done here is not define a single object, but a class of all objects of that type. Think of it as a template that you can reuse. You don't need to make lots of different circles; you can just create an *instance* of the **circle** class, an object constructed according to the template you've defined.

Classes and instances

The difference between *classes* and *instances of those classes* is important. If you can digest this one concept, you've cracked OOP.

A *class* is a definition for a collection of objects. For example, if we define a class **80s English indie band**, it describes a number of objects with shared characteristics (guitars, backcombed hair, maudlin lyrics. and so on). We can make an instance of that class called **The Smiths** and a second instance called **Echo and The Bunnymen**. Both are independent of each other but can share common functionality; **recordAlbum()**, **featureInJohnHughesFilm()**, **splitAcrimoniously()**, etc.

The class is a template: it defines only the parameters these objects share, functions they can perform, and logic for how an object can be constructed. A class becomes an object only when you make an *instance* of it.

In one of our lazy human languages, you might say you once saw an **80s English indie band**, but what you mean is that you once saw one or more *instances* of **80s English indie band**. You can't get drunk and dance to a conceptual definition, no matter how post-modern you think you are.

Note that so far, this is purely conceptual. You haven't drawn a circle to the screen—you haven't even created a circle yet. You've just defined the template for it. To create an actual object from this template, an *instance* of the class, you call the constructor by using the **new** keyword, as demonstrated in this new version of the **drawCircles** function:

```
void drawCircles() {
 for (int i=0; i<_num; i++) {
  Circle thisCirc = new Circle();
 }
}
```

Fine so far, but you're still not actually drawing any circles, just creating them conceptually in memory. You've only defined the *properties* (variables) and the *constructor* (the function called when an object of that class is created) of the class so far. If you want to do something with the class, you need to add some *methods* (functions) to it, as per the following listing.

Listing 6.3 **The Circle class again, with its first method**

```
class Circle {

 float x, y;
 float radius;
 color linecol, fillcol;
 float alph;

 Circle () {
  x = random(width);
  y = random(height);
  radius = random(100) + 10;
  linecol = color(random(255), random(255), random(255));
  fillcol = color(random(255), random(255), random(255));
  alph = random(255);
 }

 void drawMe() {
  noStroke();
  fill(fillcol, alph);
  ellipse(x, y, radius*2, radius*2);
  stroke(linecol, 150);
  noFill();
  ellipse(x, y, 10, 10);
 }
}
```

Properties

Constructor

Object method

Now, update the **drawCircles** function to call this method after you've created the object:

```
void drawCircles() {
 for (int i=0; i<_num; i++){
  Circle thisCirc = new Circle();
  thisCirc.drawMe();
 }
}
```

Et voilà, you have some circles, much as in figure 6.3. Great. This may seem like an awful lot of effort to get to where you were a few pages back. But when you start playing with the new structure, you'll see what you've gained by encapsulating the circles as objects. Now you can

give each of the circles actions, behaviors, and lives of their own. You could also give them feelings and opinions if you chose to, but we'll save that for chapter 7. For now, we'll concentrate on making them drift around and interact with each other.

Let's make another few updates to the code. In listing 6.4, you do the following:

- Create an array to store all the circles.
- Give each circle a movement factor in the x and y directions.
- Give each circle an **updateMe** method, called every frame by the **draw** loop.

Listing 6.4 Object-oriented circle-drawing code with movement

```
int _num = 10;
Circle[] _circleArr = {};                          Define array of circles
void setup() {
 size(500,300);
 background(255);
 smooth();
 strokeWeight(1);
 fill(150, 50);
 drawCircles();
}

void draw() {
 background(255);
 for (int i=0; i<_circleArr.length; i++) {
  Circle thisCirc = _circleArr[i];
  thisCirc.updateMe();
 }
}

void mouseReleased() {
 drawCircles();
}

void drawCircles() {
 for (int i=0; i<_num; i++) {
  Circle thisCirc = new Circle();
  thisCirc.drawMe();
  _circleArr = (Circle[])append(_circleArr, thisCirc);   Add object to array
 }
```

(continued on next page)

```
}                                                                    (Listing 6.4 continued)


//================================= objects

class Circle {

 float x, y;
 float radius;
 color linecol, fillcol;
 float alph;
 float xmove, ymove;

 Circle () {
  x = random(width);
  y = random(height);
  radius = random(100) + 10;
  linecol = color(random(255), random(255), random(255));
  fillcol = color(random(255), random(255), random(255));
  alph = random(255);
  xmove = random(10) - 5;
  ymove = random(10) - 5;
 }

 void drawMe() {
  noStroke();
  fill(fillcol, alph);
  ellipse(x, y, radius*2, radius*2);
  stroke(linecol, 150);
  noFill();
  ellipse(x, y, 10, 10);
 }

 void updateMe() {
  x += xmove;
  y += ymove;
  if (x > (width+radius)) { x = 0 - radius; }
  if (x < (0-radius)) { x = width+radius; }
  if (y > (height+radius)) { y = 0 - radius; }
  if (y < (0-radius)) { y = height+radius; }
```

Steps to move every frame

Random step

Move every frame

Wrap position at stage edges

```
    drawMe();
  }
}
```

Draw it

Now the circles are moving around. The **updateMe** method changes each object's x and y positions and wraps it if it's drifted offscreen. The **drawMe** method then renders the instance to the screen.

Using Arrays

An *array* is a list of objects of a certain type. If you define an array as

```
int[] numberArray = new int[5];
```

you're saying that you want a list of five items, all of type **int**.
You can specify what those five items are when you define the array using the braces syntax:

```
int[] numberArray = {1, 2, 3, 4, 5};
```

However you define it, you can add items to each position like so:

```
numberArray[2] = 3;
```

This places a 3 as the third item, not the second. The first position is index 0, so the third item is index 2.
An empty array, with no items, is defined as follows:

```
int[] numberArray = {};
```

To add an extra slot to an array and place data in that slot (as you do in listing 6.4), you use **append**. Note that you need to cast the new array returned by this command to the correct type, such as **int[]**:

```
numberArray = (int[])append(numberArray, 6);
```

Notice how, when you click to add more circles, the previous circles stay onscreen unchanged. You're still clearing the canvas every frame (you call **background(255)** in the **draw** frame loop); but because the objects remain in memory, they don't go anywhere, so you can redraw them whenever you need them. Even if you didn't render them to the screen every frame, they would still be there in memory, following their paths invisibly.

If you click several times, bringing the circle count into the hundreds, then thousands, you'll notice that the performance of the movie starts to suffer. Depending on the speed of your machine, you can probably comfortably manage a few hundred circles independently moving around the screen before you see any sign of the animation slowing down, but you should start to get an appreciation of the limits of your processor.

You have the object-oriented structure, and you have an application capable of generating many, many simple objects. Next, lets look at how you might tweak this system and tease out some emergent patterns.

6.2.2 Local knowledge (collision detection)

An instance created from the **Circle** class has what we might call *self-awareness*. It knows everything there is to know about itself, but nothing about the environment it's in or how it's being used by the script. If this was one of the murmating starlings, it would be flying blind, unable to see its flock-mates. You can change this though, and give the objects created from the class some rudimentary senses.

You have all the circle objects in an array; so by looping through this array, which is available globally (to all objects), each object can test itself against every other object (regardless of whether that object is onscreen or not). For this example, you'll test to see if the current circle is overlapping any other circle. This is called, in gaming parlance, *collision detection.*

Collision detection (and *collision correction* — making an object react realistically to a collision) is one of the dark arts of games programming. To have a realistic physics systems, objects need to be able to have an awareness of the environment around them and its inhabitants, and be able to react to their surroundings. As environments get more complex, and the number of inhabiting objects increases, this becomes quite a processor intensive action to perform every frame loop, especially if objects are of peculiar shapes.

There are entire books devoted to this topic, and for our needs we don't require a realistic physics simulation, so we'll keep the example simple. When you're dealing with circles, you just need a bit of trig to check the distance between two points; and Processing has a built-in **dist** function that does the calculation for you. If you check the distance between the centers of two circles and subtract their radii, if anything is left, you know the two circles aren't touching. This calculation requires a few extra lines in the **updateMe** function, as you can see in the following listing.

Listing 6.5 Circle class's updateMe method with added collision detection

```
void updateMe() {
  x += xmove;
  y += ymove;
  if (x > (width+radius)) { x = 0 - radius; }
  if (x < (0-radius)) { x = width+radius; }
  if (y > (height+radius)) { y = 0 - radius; }
  if (y < (0-radius)) { y = height+radius; }
```

```
    boolean touching = false;
    for (int i=0; i<_circleArr.length; i++) {
     Circle otherCirc = _circleArr[i];
     if (otherCirc != this) {
      float dis = dist(x, y, otherCirc.x, otherCirc.y);
      if ((dis - radius - otherCirc.radius) < 0) {
       touching = true;
       break;
      }
     }
    }
    if (touching) {
     if (alph > 0) { alph--; }
    } else {
     if (alph < 255) { alph += 2; }
    }

    drawMe()
}
```

Loops through other circles

Don't test current circle

Calculates distance between circles

The loop you've added checks through every other circle in the array and tests how far away it is. If it's overlapping, the **touching** flag is set to **true**, and it breaks out of the loop. The **if** statement then applies an effect depending on whether the circles are intersecting. If circles are overlapping, they fade. They increase in opacity if they're in the clear. This should mean that smaller circles, which overlap other circles less as they move around, tend toward solidity, whereas larger circles become more transparent. Run the code to see it in action.

I'm sure you can come up with something more visually interesting to do with this behavior, but before I leave you to play around with it, let's try a little something. Where circles collide, let's draw another circle at the midpoint between them.

6.2.3 Interaction patterns

The only potentially tricky bit here is calculating the midpoint between two circles that could be anywhere on (or off) screen. You need to ensure that all bases are covered. If the x position of the current circle is greater than the x position of the intersecting circle, the midpoint x is

```
x + (otherCirc.x – x)/2
```

If it's less, the midpoint is

```
otherCirc.x + (x - otherCirc.x)/2
```

This is one way of doing it, but there is a neater way that will cover both possibilities with a single line of code. The average of the two x values will give us the midpoint between them, you can calculate this by adding them together and dividing by 2.

```
(x + otherCirc.x)/2
```

The same applies in the y direction. When you have the x and y position, you can mark it by drawing a circle at that point, with the radius of the circle dictated by the size of the overlap. The updated method is shown next.

Listing 6.6 **Circle class's updateMe method again, now with circles**

```
void updateMe() {
  x += xmove;
  y += ymove;
  if (x > (width+radius)) { x = 0 - radius; }
  if (x < (0-radius)) { x = width+radius; }
  if (y > (height+radius)) { y = 0 - radius; }
  if (y < (0-radius)) { y = height+radius; }

  for (int i=0; i<_circleArr.length; i++) {
    Circle otherCirc = _circleArr[i];
    if (otherCirc != this) {
      float dis = dist(x, y, otherCirc.x, otherCirc.y);
      float overlap = dis - radius - otherCirc.radius;
      if (overlap < 0) {
        float midx, midy;
        midx = (x + otherCirc.x)/2;
        midy = (y + otherCirc.y)/2;
        stroke(0, 100);
        noFill();
        overlap *= -1;                4
        ellipse(midx, midy, overlap, overlap); 5
      }
    }
  }

  drawMe();
}
```

Calculates overlap

Negative overlap = touching

Calculates midpoint

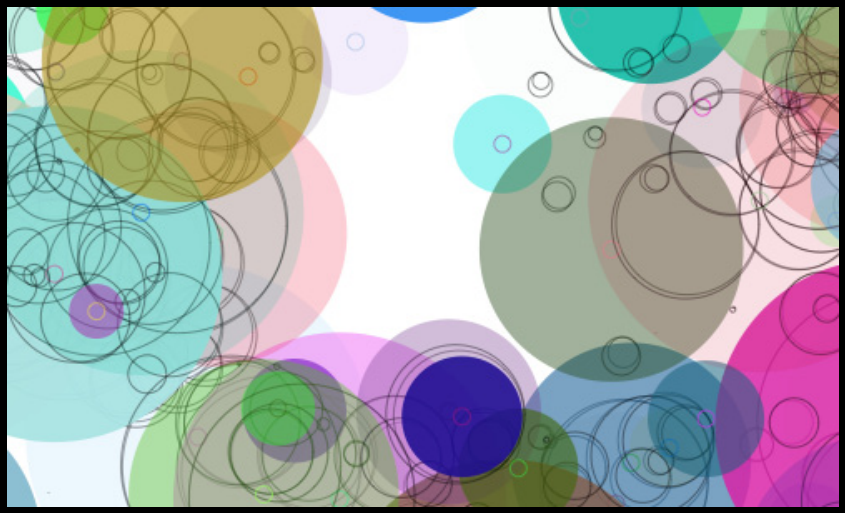

Figure 6.4

The circles are following a simple behavior you've programmed. But in the interaction between the circles, a new, emergent complexity is apparent.

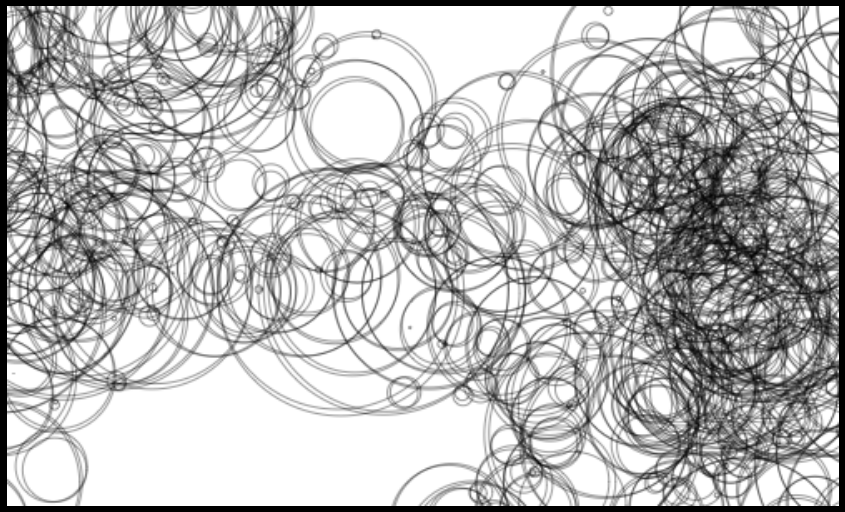

Figure 6.5

Removing the original behavior, all you have left are the emergent patterns, which are far more interesting.

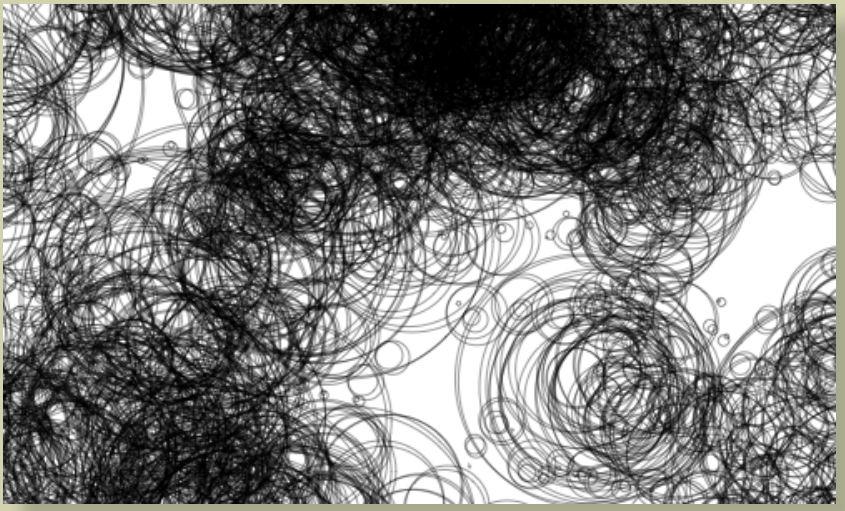

Figure 6.6

It took 170 circle objects to create this. I can only guess how many actual circles are on the screen.

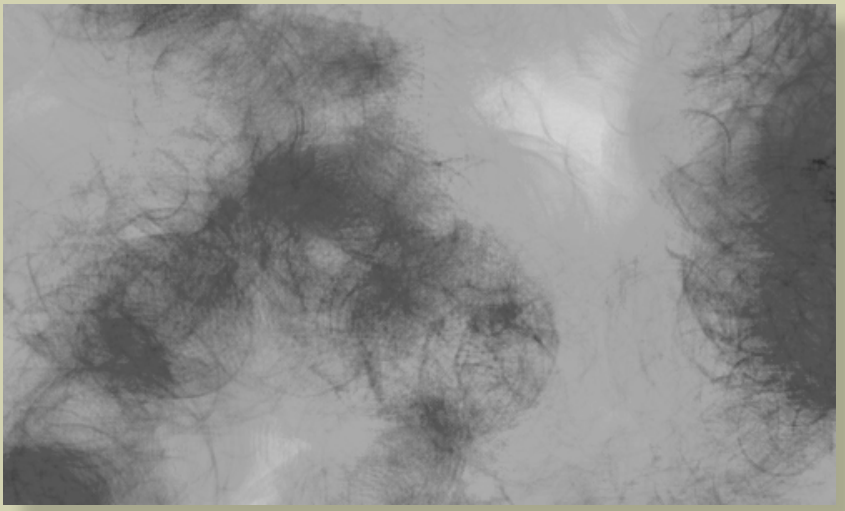

Figure 6.7

The emergent circles, abstracted further using a softer line and traces

Run that, and you'll see a whole new pattern emerging. Figure 6.4 shows the output. You're drawing the circles, which position themselves according to the simple behavior you've given them, but you've also discovered an emergent property of the *interaction between* the circles. Abstracting this behavior by only a small degree, you've stumbled upon a new layer of complexity.

This new emergent behavior is clearly much more interesting than the simple one you built in the code. Let's isolate that so you can see it more clearly. If you comment out, or remove, the call to **drawMe** in the **updateMe** function, you get rid of the original circles. The circles are still there, of course; you're just not drawing them to the screen anymore. When you've done this, you have something that looks a little like figure 6.5.

Next, add a line to the mouse-press function to trace the current number of circles to the console window, so you can see how many circles you need to produce what effect:

```
void mouseReleased() {
  drawCircles();
  println(_circleArr.length);
}
```

I kinda liked 170 (see figure 6.6).

This clustering pattern, made up of nothing but circles, isn't a product of intention. The pattern is a byproduct of the *interactions* between the behaviors you programmed, not the behaviors themselves. Typically, it's in the interactions between agents that you find the emergent patterns. A single starling follows a flight pattern that obeys a set of rules; but place that bird in a flock, and the complexity of the pattern increases several-fold, while the rule set remains unchanged.

In this example, the behaviors couldn't have been much simpler: a number of objects moving in straight lines. And the drawing style was pretty basic, too. If you want to take this further, you can follow a lot of paths. To start you off, here are a few ideas. How about:

- *Leaving traces*—Don't clear the screen between frames. Draw a transparent rectangle over everything instead (as described in 2.4.2).
- *Reducing the alpha and stroke weight*—Make the lines more subtle to turn hard edges into an organic blur (see figure 6.7).
- *Plotting more complex paths*—You've already seen in previous chapters how there are far more interesting routes to take between two points than a straight line. How about plotting movement along a curve, for example?
- *Drawing more interesting shapes*—Draw something more exciting than a simple circle. You can even import images to be the visual representations of objects.

6.3 Summary

In this chapter, we've introduced the concept of emergence, discussed examples, and then, living up to my promise, discovered one of our own: an emergent effect from the simplest of objects behaving in the simplest of manners. I hope this has demonstrated how easy it is: how emergence isn't something you need to work at, it's something you may stumble upon almost by accident. Any system of many parts above a certain (relatively low) level of complexity, will be prone to emergent complexity. The only trick is exploiting it.

We haven't relied on randomness or noise to introduce the generative element to the code in this chapter. All we've done is apply a level of abstraction sufficient to introduce an unpredictable complexity. Simple rules creating complex results.

One other achievement we've unlocked in this chapter is the mastery of a very useful programming methodology: object-oriented programming. This will open us up to more advanced coding possibilities in the remaining chapters (and beyond).

In the next chapter, we'll stick with the topic of emergent complexity but take a more structured approach to its creation. We'll take our understanding of OOP and refine it toward a definition of agent-oriented programming.