7

Chapter Seven

Autonomy

Object-oriented programming (OOP) is a programming conceit. Objects are for organizing conceptual things into conceptual structures. An "object" can be anything: a speck within a particle system, a game character, a pure data form, or just a neat way of collecting variables and methods.

It doesn't necessarily have to correspond to anything that has a visual form or identity; it's a data holder. But you don't need to think of objects in such clinical terms. If you chose to define them so, you could give your objects feelings, aspirations, failings, and destinies. They can be active citizens in a virtual world, albeit a highly abstract one. Your objects can be more than data holders: they can be autonomous agents.

The difference between an object and an agent, in programming parlance, is that agents, specifically, observe and interact with their environment. They may have more sophisticated behaviors, such as goals, beliefs, and prejudices. They can also harbor imprecisions, self-interests, and irrationalities too, which is what makes them so interesting to work with.

This is what we mean when we speak of *autonomy*: the capability for something, whether human, monkey, software construct, robot, or supermodel, to make its own decisions. The difference between an agent and an object is the difference between an animal and a rock. You don't

necessarily have to anthropomorphize your objects to make them more interesting, but you might do so expecting the more complex, and less predictable, behavior you'd get from a creature rather than a data construct.

The most familiar single example of an autonomous agent, to you at least, is the one staring at this page. You, dear reader, with your behavior on any particular day defined by a complex mix of biology, psychology, and the comfortableness of your footwear, are an autonomous object, creating interesting patterns in the data you create. Most everything you do these days leaves a data trail: every purchase you make, every point you earn on your store card, every link you click, and every journey you take. The cards in your wallet are writing your economic autobiography, your tweets and SMSs are writing an ASCII diary as part of a social map, and the phone in your pocket is drawing a GPS picture of your daily psycho-geography.

In the second half of this chapter, we'll look at the concept of the agent further, discussing some particularly creative examples from both within and without our electronic boxes, including ways of tapping into the available mass of human-generated data for artistic use. But first, to familiarize you with the concept of autonomy and the emergent complexity of this breed of object, we'll play one of the early parlor games of computer science: cellular automata.

7.1 Cellular automata

We demonstrated in the previous chapter how easy it is to stumble across emergent complexity without particular intention. With the help of a cellular automata (CA) grid, you'll learn a more structured way of observing the phenomenon. In the 1970s, the field of computer science was obsessed with CA. Super-computers were employed to churn through iterations of John Conway's *Game of Life* (see section 7.1.2) over periods of weeks. Today, that kind of computing power is available in our mobile phones, so we can easily simulate cellular automata in a lightweight Processing sketch.

A 2D[1] CA is a grid of cells (see figure 7.1), each of which has only two states: on and off, black or white, alive or dead. Each cell has limited local knowledge, only able to see its eight immediate neighbors. In a series of cycles, each cell decides its next state based on the current states of its surrounding cells.

1. There are one-dimensional and multidimensional variants of CA, but for our purposes we'll stick to two dimensions.

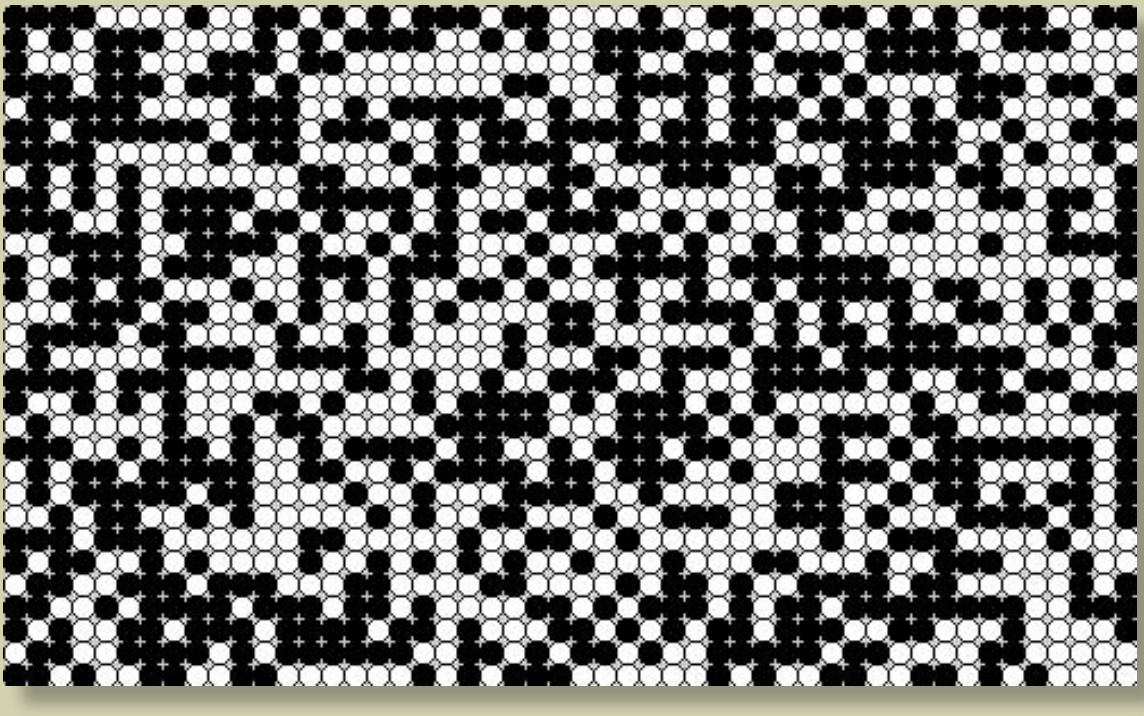

Figure 7.1
A cellular automata grid

7.1.1 Setting up the framework

The best way to demonstrate is by example. The following listing creates a grid of cells with on/off states and a reference to their immediate neighbors.

Listing 7.1 CA framework

```
Cell[][] _cellArray;
int _cellSize = 10;
int _numX, _numY;

void setup() {
 size(500, 300);
 _numX = floor(width/_cellSize);
 _numY = floor(height/_cellSize);
 restart();
}

void restart() {
 _cellArray = new Cell[_numX][_numY];
 for (int x = 0; x<_numX; x++) {
  for (int y = 0; y<_numY; y++) {
   Cell newCell = new Cell(x, y);
   _cellArray[x][y] = newCell;
  }
 }


 for (int x = 0; x < _numX; x++) {
  for (int y = 0; y < _numY; y++) {

   int above = y-1;
   int below = y+1;
   int left = x-1;
   int right = x+1;

   if (above < 0) { above = _numY-1; }
   if (below == _numY) { below = 0; }
   if (left < 0) { left = _numX-1; }
   if (right == _numX) { right = 0; }
```

Creates grid of cells

Loop tells each object its neighbors

Gets locations to each side

Wraps locations at edges

```
    _cellArray[x][y].addNeighbour(_cellArray[left][above]);
    _cellArray[x][y].addNeighbour(_cellArray[left][y]);
    _cellArray[x][y].addNeighbour(_cellArray[left][below]);
    _cellArray[x][y].addNeighbour(_cellArray[x][below]);
    _cellArray[x][y].addNeighbour(_cellArray[right][below]);
    _cellArray[x][y].addNeighbour(_cellArray[right][y]);
    _cellArray[x][y].addNeighbour(_cellArray[right][above]);
    _cellArray[x][y].addNeighbour(_cellArray[x][above]);
   }
  }
}

void draw() {
 background(200);

 for (int x = 0; x < _numX; x++) {
  for (int y = 0; y < _numY; y++) {
   _cellArray[x][y].calcNextState();
  }
 }

 translate(_cellSize/2, _cellSize/2);

 for (int x = 0; x < _numX; x++) {
  for (int y = 0; y < _numY; y++) {
   _cellArray[x][y].drawMe();
  }
 }
}

void mousePressed() {
 restart();
}

//================================== object

class Cell {
 float x, y;
```

Passes refs to surrounding locs

Calculates next state first

Draws all cells

(continued on next page)

(Listing 7.1 continued)

On or off

Randomizes initial state

```
boolean state;
boolean nextState;
Cell[] neighbors;

Cell(float ex, float why) {
 x = ex * _cellSize;
 y = why * _cellSize;
 if (random(2) > 1) {
  nextState = true;
 } else {
  nextState = false;
 }
 state = nextState;
 neighbors = new Cell[0];
}

void addNeighbor(Cell cell) {
 neighbors = (Cell[])append(neighbors, cell);
}

void calcNextState() {
  // to come
}

void drawMe() {
 state = nextState;
 stroke(0);
 if (state == true) {
  fill(0);
 } else {
  fill(255);
 }
 ellipse(x, y, _cellSize, _cellSize);
}

}
```

It's a lengthy starting point, but nothing you haven't encountered before. At the bottom of the script you define an object, a **Cell**, which has x and y positions and a **state**, either on or off. It also has a holder for its next state, **nextState**, the state it will enter the next time its **drawMe** method is called.

In **setup**, you calculate the number of rows and columns based on the width and height and the size you want the cells; then, you call the **restart** function to fill a 2D array with this grid. After the grid is created, there is a second pass through the array to tell each of the cells who its neighbors are (above, below, left, and right of them).

The **draw** loop then does a double pass through the array. The first pass triggers each cell to calculate its next state, and the second pass triggers each to make the transition and draw itself. Note that this needs to be done in two stages so each cell has a static set of neighbors on which to base its next state calculation.

Now that you have the grid of cells (as shown in figure 7.1), you can begin giving the cells their simple behaviors. The best-known CA is John Conway's *Game of Life*, so that is the one we'll start with.

Multidimensional arrays

A one-dimensional array, as you saw in chapter 6, is a data structure for storing a list of elements. You define one as follows:

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

But if you want to store more than a list, if you have a matrix of elements to keep track of, you can add an extra dimension to the array. A 2D array is initialized like so:

```
int[][] twoDimArray = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

Essentially, you're defining an array of arrays. In listing 7.1, you can see an example of creating and iterating through a 2D array.

If you want more dimensions, you can have them. You're free to add as many dimensions as you can get your head around:

```
int[][][] threeDimArray;
int[][][][] fourDimArray;
...
```

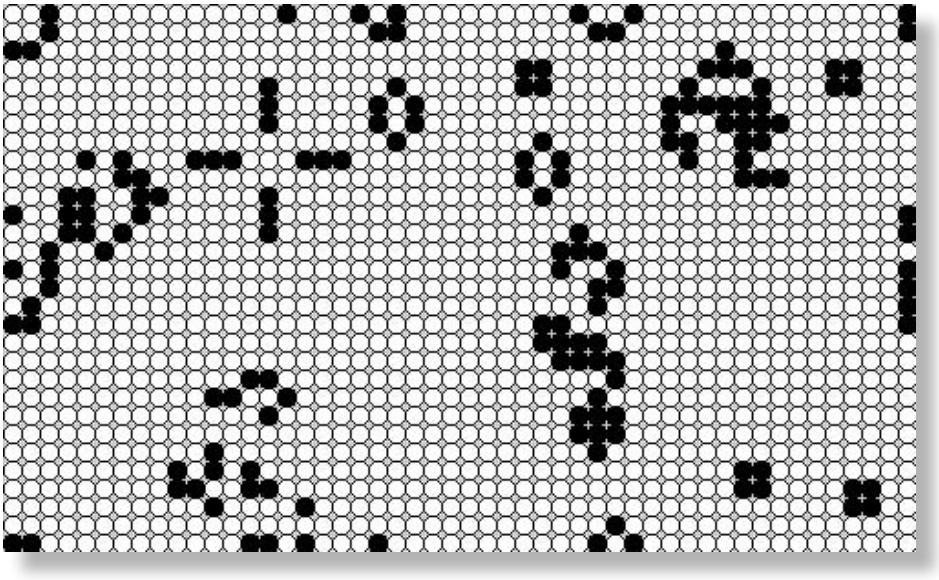

Figure 7.2
Conway's Game of Life after 100 iterations

7.1.2 The Game of Life

The popularity of Conway's Game of Life (GOL) is mostly due to the relevance it has found outside the field of mathematics. Biologists, economists, sociologists, and neuroscientists, amongst others, have discussed at length the parallels between the behavior of CA obeying GOL rules and the results of studies in their respective fields. It's essentially a simple computational form of artificial life, a mathematical petri dish teeming with autonomous agents.

The rules for GOL are as follows:

- Rule 1: If a live (black) cell has two or three neighbors, it continues to live. Otherwise it dies, of either loneliness or overcrowding.
- Rule 2: If a dead cell has exactly three neighbors, a miracle occurs: it comes back to life.

You write this in code as follows. Complete the **calcNextState** function in accordance with the following listing.

Figure 7.3

Triangle height and rotation depend on the age of a cell. Triangle width is the neighbor count. The color of the shape is mixed using a combination of the two.

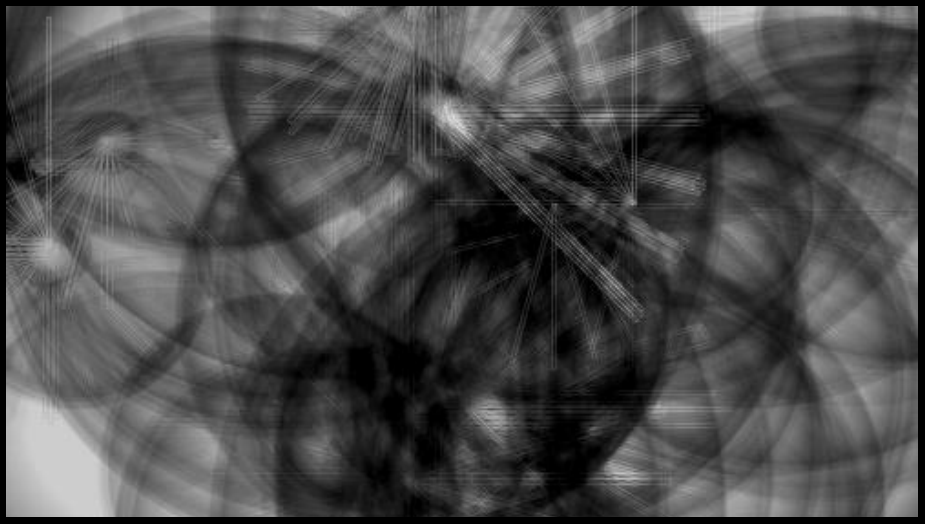

Figure 7.4

Circle size is determined by neighbor count. Lines and their rotations are determined by cell age.

Listing 7.2 Calculating the next state using GOL rules

```
void calcNextState() {
  int liveCount = 0;
  for (int i=0; i < neighbours.length; i++) {
   if (neighbors[i].state == true) {
    liveCount++;
   }
  }

  if (state == true) {
    if ((liveCount == 2) || (liveCount == 3)) {
     nextState = true;
    } else {
     nextState = false;
    }
  } else {
    if (liveCount == 3) {
     nextState = true;
    } else {
     nextState = false;
    }
  }
}
```

This code first counts the number of neighbors alive and then applies GOL rules. If you run it for a few seconds, you begin to see what looks like a microscope view of abstract organisms (see figure 7.2). Some parts bubble and then fade out; others achieve a looping stability. There has been much study of this game, identifying the mathematical "life" that forms: gliders, toads, boats, blinkers, beacons, and so on. The GOL Wikipedia page has many animated examples of the complex shapes and patterns that arise from these simple rules: see http://en.wikipedia.org/wiki/Conway's_Game_of_Life#Examples_of_patterns.

It would be very easy for us to get sidetracked here, because what you've written is essentially an AI program. Artificial intelligence is a field that, admittedly, has huge potential for the creation of generative art, but it's far beyond the scope of this book.[2] We'll keep our focus relatively shallow and toy with the possibilities for each of these cells, with its local, blinkered behavior, to render itself to the screen in visually interesting ways.

2. If this subject interests you, I suggest that you seek out Douglas Hofstadter's Pulitzer Prize winning *Gödel, Escher, Bach* (1979) as an entertaining, not too scary, starting point.

A simple black or white dot is only a starting point. Just as you did with multidimensional Perlin noise back in chapter 5, where you used the noise value as rotation, shape, size, and alpha values, you can try similar rendering experiments using a cell's properties. In the two examples in figures 7.3 and 7.4, I've used the number of live neighbors a cell has, and the number of frames it has remained alive, to determine the size, rotation, and color of a shape drawn at that cell's location. You may like to try similar experiments.

GOL is only one of many potential rule sets. In addition to experimenting with visualizations, I'd encourage you to try inventing your own. In the following pages, I'll demonstrate two more well-known patterns, and one invented behavior, each of which produces forms that have analogues in the natural world.

7.1.3 Vichniac Vote

The first pattern, the Vichniac Vote (after Gerard Vichniac who first experimented with it), is a lesson in conformity. Each cell is particularly susceptible to peer-group pressure and looks to its neighbors to observe the current trend. If the cell's color is in the majority, it remains unchanged. If it's in the minority, it changes. To ensure that there is an odd number of cells on which to make the decision, the cell includes its own current state in addition to its eight neighbors in making the calculation.

To see the vote in action, rewrite the **calcNextState** function using the code from the following listing. To create an artificial instability in the algorithm, you'll swap the rules for four and five neighbors; otherwise, it settles quickly into a static pattern.

Listing 7.3 The Vichniac Vote rule

```
void calcNextState() {
  int liveCount = 0;
  if (state) { liveCount++; }
  for (int i=0; i < neighbors.length; i++) {
    if (neighbors[i].state == true) {
      liveCount++;
    }
  }

  if (liveCount <= 4) {
    nextState = false;
  } else if (liveCount > 4) {
    nextState = true;
  }
```

Counts neighbors, including me

Am I in the majority?

(continued on next page)

(Listing 7.3 continued)

```
    if ((liveCount == 4) || (liveCount == 5)) {
      nextState = !nextState;
    }
  }
```

Swaps rules for 4 and 5

The result is shown in figure 7.5. You can see how the clustering effect creates patterns similar to the hide of a cow, or the contours of a landscape seen from above.

With the Vichniac Vote, the rules you apply are sociological —agents succumbing to the peer-group pressure from their neighbors—but the results have an aesthetic more familiar from biology or geology. Might this imply that common computational principles underpin these disciplines? Before we've finished with this chapter and the next, you'll see plenty more algorithmic systems that may support this hypothesis. Brian's Brain, for example.

7.1.4 Brian's Brain

This is a three-state cellular automaton, meaning a cell can be in one more condition, apart from on or off. The states of a Brian's Brain CA are *firing*, *resting*, and *off*. It's designed to mimic the behavior of neurons in the brain, which fire and then rest before they can fire again. The rules are as follows:

- If the state is *firing*, the next state is *resting*.
- If the state is *resting*, the next state is *off*.
- If the state is *off*, and exactly two neighbors are firing, the state becomes *firing*.

You'll need to modify the cell class for this, as in the following code. Note that you have to change the type of **state** to **int**, so it can have more than two values. The integers 0, 1, and 2 indicate *off*, *firing*, and *resting*, respectively.

Listing 7.4 Cell object modified for the three-state Brian's Brain behavior

```
class Cell {
  float x, y;
  int state;
  int nextState;
  Cell[] neighbors;

  Cell(float ex, float why) {
    x = ex * _cellSize;
```

(continued on next page)

State 1, 2, or 0

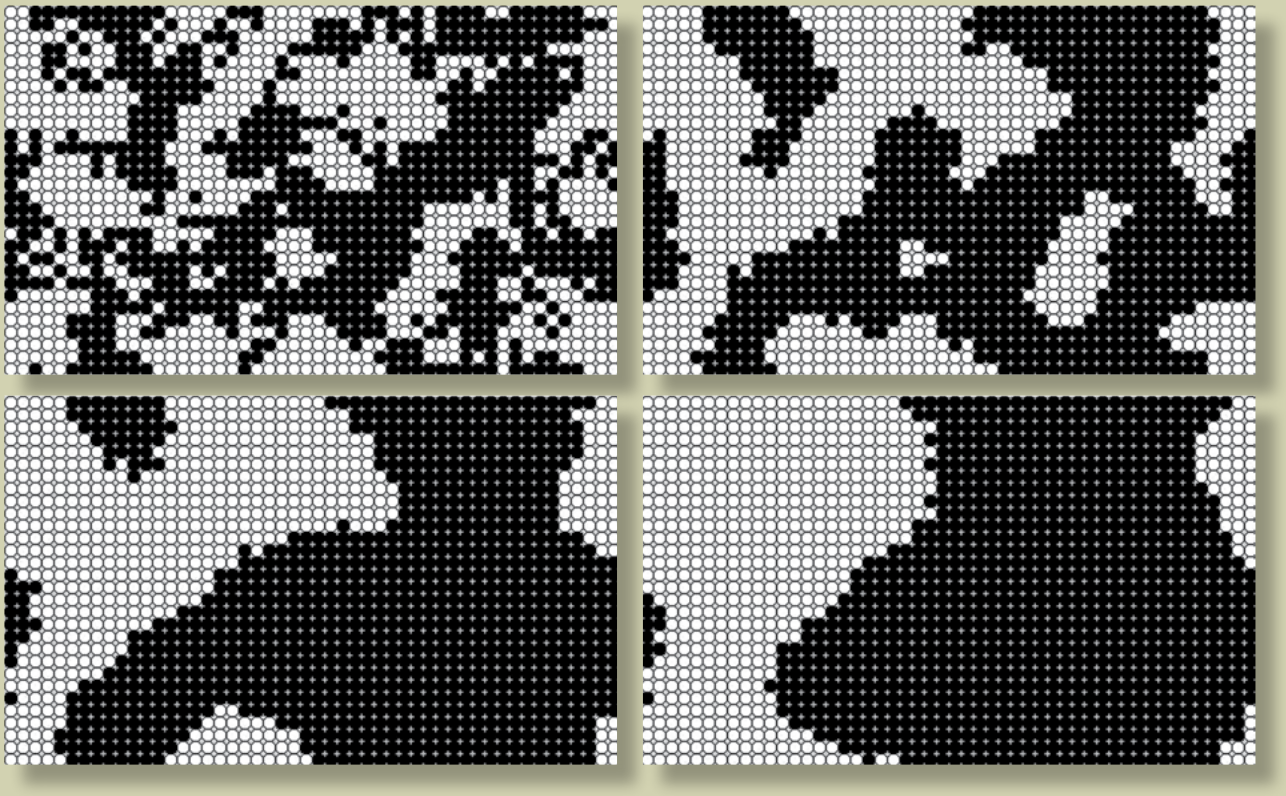

Figure 7.5

The Vichniac Vote rule after 3 iterations, 11 iterations, 41 iterations, and 166 iterations

```
  y = why * _cellSize;                                          (Listing 7.4 continued)
  nextState = int(random(2));
  state = nextState;
  neighbors = new Cell[0];
 }

 void addNeighbor(Cell cell) {
  neighbors = (Cell[])append(neighbors, cell);
 }

 void calcNextState() {
  if (state == 0) {
   int firingCount = 0;
   for (int i=0; i < neighbors.length; i++) {
    if (neighbors[i].state == 1) {
     firingCount++;
    }
   }
   if (firingCount == 2) {
    nextState = 1;
   } else {
    nextState = state;
   }
  } else if (state == 1) {
   nextState = 2;
  } else if (state == 2) {
   nextState = 0;
  }
 }

 void drawMe() {
  state = nextState;
  stroke(0);
  if (state == 1) {
   fill(0);
  } else if (state == 2) {
   fill(150);
  } else {
   fill(255);
  }
  ellipse(x, y, _cellSize, _cellSize);
```

Counts firing neighbors

If two neighbors are firing, fire too

Else, don't change

If just fired, rest

If rested, turn off

Firing = black

Resting = grey

Off = white

```
  }

}
```

The results look something like figure 7.6. Spaceships (the CA term, not mine) move across the plane in horizontal, vertical and sometimes diagonal directions.

Does this behavior reflect the way your thoughts form from the electronic bursts of firing synapses? Is what you're seeing a mathematical model of consciousness itself? Perhaps; but for our purposes, for the time being at least, it's just another emergent behavior that you can use as the basic for a generative visualization.

Just one more example before we move on. Not one of the famous rule sets, but a custom behavior instead. The patterns it produces should be quite familiar though.

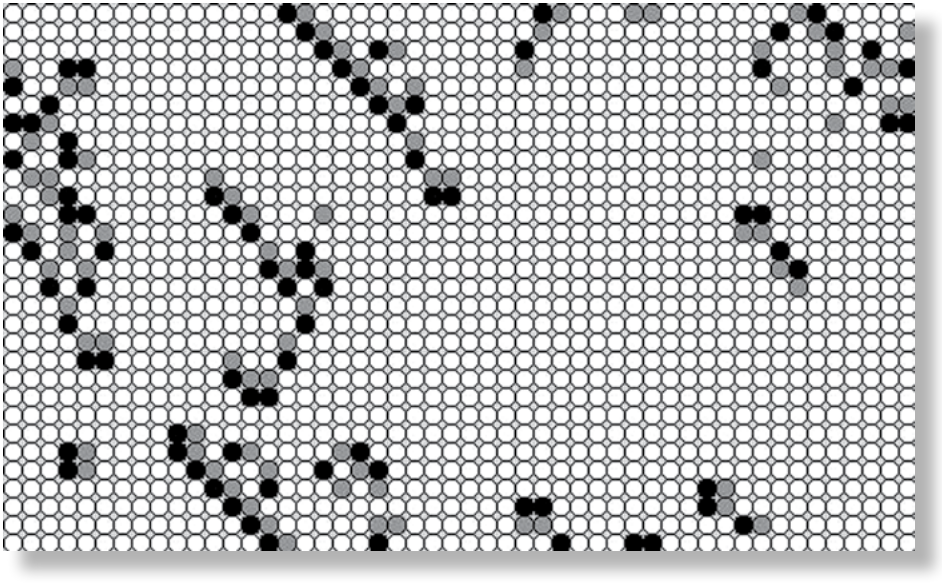

Figure 7.6
The *Brian's Brain* behavior after 19 iterations

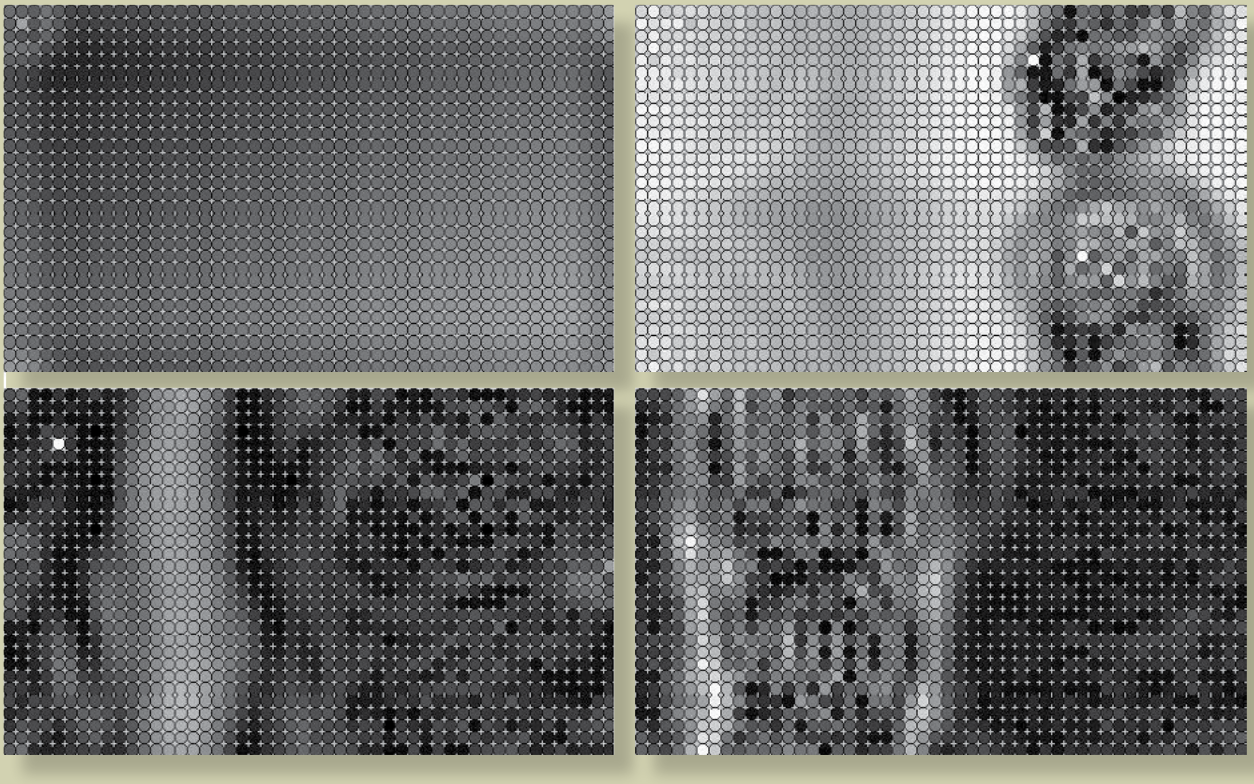

Figure 7.7

The customized wave rule after 6, 26, 49, and 69 iterations

7.1.5 Waves (averaging)

This final pattern is an example of a CA with a continuous behavior. There is no reason a cell has to be limited to two or three distinct values: its state can vary across a range of values. In this example, you'll use 255 values, the grayscale from white to black. This is a custom behavior, so it doesn't have a name, but I've based it on a standard physics model—averaging—whereby a chaotic state settles into a steady one thorough the influence of its neighbors.

The rules are as follows:

- If the average of the neighboring states is 255, the state becomes 0.
- If the average of the neighboring states is 0, the state becomes 255.
- Otherwise, new state = current state + neighborhood average – previous state value.
- If the new state goes over 255, make it 255. If the new state goes under 0, make it 0.

If you were adhering to the physics model, you'd need only the third of these rules. I've added the other rules for instability, to stop it from settling. The code is in the next listing. The output is shown in figure 7.7.

Listing 7.5 Custom wave-like behavior

```
class Cell {
 float x, y;
 float state;
 float nextState;
 float lastState = 0;
 Cell[] neighbors;

 Cell(float ex, float why) {
  x = ex * _cellSize;
  y = why * _cellSize;
  nextState = ((x/500) + (y/300)) * 14;
  state = nextState;
  neighbors = new Cell[0];
 }

 void addNeighbor(Cell cell) {
  neighbors = (Cell[])append(neighbors, cell);
 }

 void calcNextState() {
```

Creates initial gradient

(continued on next page)

(Listing 7.5 continued)

```
    float total = 0;
    for (int i=0; i < neighbours.length; i++) {
      total += neighbors[i].state;
    }
    float average = int(total/8);

    if (average == 255) {
     nextState = 0;
    } else if (average == 0) {
     nextState = 255;
    } else {
     nextState = state + average;
     if (lastState > 0) { nextState -= lastState; }
     if (nextState > 255) { nextState = 255; }
     else if (nextState < 0) { nextState = 0; }
    }

    lastState = state;
  }

  void drawMe() {
   state = nextState;
   stroke(0);
   fill(state);
   ellipse(x, y, _cellSize, _cellSize);
  }

}
```

Calculate neighborhood average

Stores previous state

Uses state value as fill color

The most obvious comparison with this behavior is a liquid. Like rain on a shallow pool of water, it churns and varies.

The reason for this example is to emphasize how you can tweak not only the rules but also the framework itself. You don't need a grid, you don't need circles, and you don't need a limited set of grayscale values: your system can be as subtle and sophisticated as you care to make it (see figure 7.8). In the next section, we'll look at a few further simulation and visualization ideas that step far outside the ordered gird but still utilize agents, with their limited local knowledge, for emergent effects. We'll also look at the potential in exploiting one very familiar, oblivious data object: you.

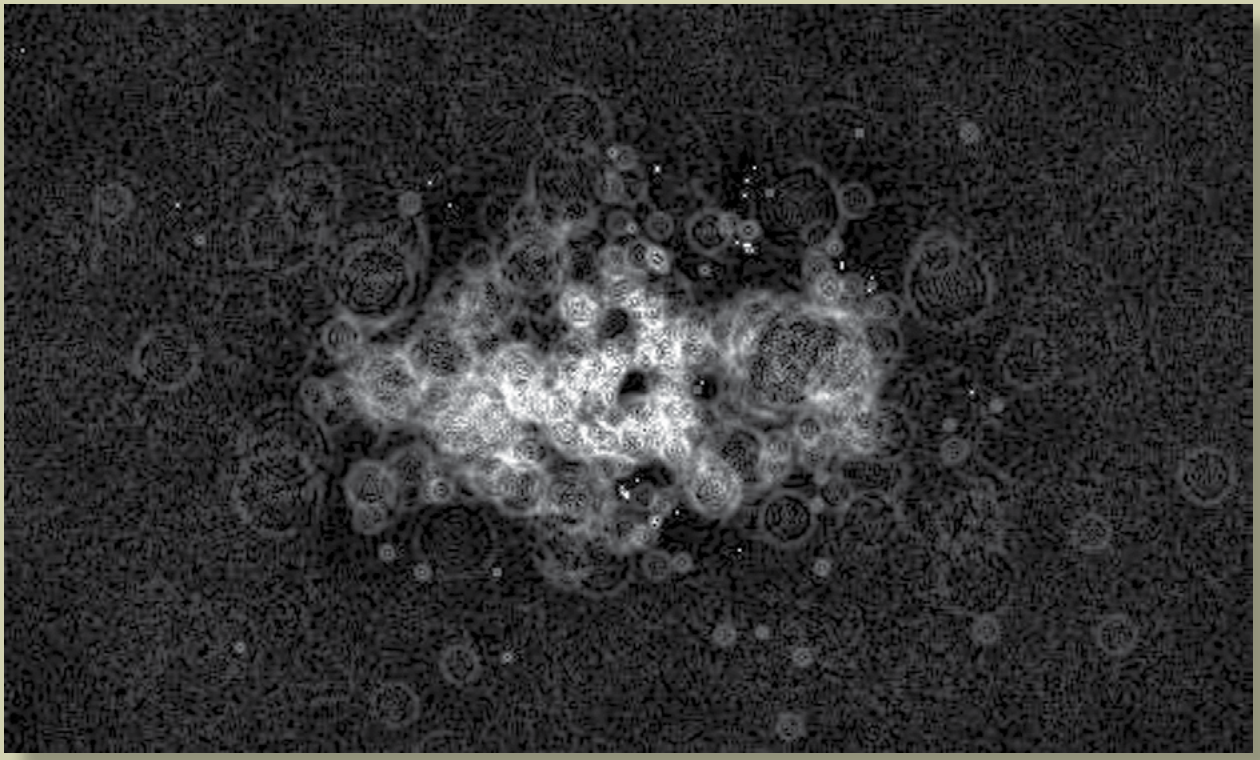

Figure 7.8

Single-pixel-sized cells produce more subtle (but more processor-intensive) patterns.

7.2 Simulation and visualization

CAs may be of only limited interest to us visually. The real purpose of your experiments so far has been to drum in the agent-oriented approach, and the ways you can study the patterns that objects unwittingly create. CAs, for all their nerdy amusement value, may be only the simplest of possible systems you can visualize this way.

The coding part of this chapter is finished now, but there is plenty of scope to develop these approaches in more meaningful directions. For the remaining pages of this chapter, we'll discuss a few ideas.

7.2.1 Software agents

When you define a class in code, as you have a number of times now, you give it properties and methods: **x**, **y**, **height**, **width**, **alpha**; **update**, **draw**, **calcWidth**, and so on. But an object's properties and methods may just as easily be as follows:

```
class Agent {
 String political_leaning = "right";
 int num_years_in_education = 12;
 int salary = 30000;
 Agent[] neighbors;
 Agent[] coworkers;
 Agent[] friends;

 Agent() {
  num_years_in_education = 11 + int(random(8));
  if ((num_years_in_education < 13) {
    if (random(1) > 0.8)) {
       political_leaning = "right";
    } else {
     political_leaning = "left";
    }
  } else {
     if (random(1) > 0.3) {
       political_leaning = "right";
     } else {
```

```
        political_leaning = "left";
        }
    }
  }
}
```

If you give an agent real-world attributes, suddenly the simulations can take on more meaningful interpretations. The interactions of these simplified souls can say something about their less simple analogues: us. For example, you can query the thoughts or feelings of an agent, perhaps something like this:

```
boolean isHappy() {
if ((political_leaning == "right") {
 if (salary > 60000)) {
    return true;
  } else {
     return false;
  }
} else {
  if (friends.length > 25) {
    return true;
  } else {
    return false;
  }
 }
}
```

Given an infinite number of coders at an infinite number of MacBooks, you might follow this to its logical extreme and code an entire human being, and pack a bunch of them into a CA grid to see how they got on. Or, more interestingly, you could code up an artificial city environment to house them, and see the patterns they traced. But a human, or near-human, level of intelligence isn't needed for interesting agents, as you've already seen. It isn't the sophistication of the objects that produces the complexity; it's their large numbers and the way they interact with each other.

One of my favorite examples of an analogous agent-oriented system is Jeremy Thorp's *Colour Economy* (2008). He constructed a system whereby a number of colored cubes are given the ability to trade their own color value in the pursuit of profit (see figure 7.10). The set of rules each cube followed was the same, but each cube had different leanings and idiosyncrasies influencing its self-interested behavior; these collectively defined the unstable economics of the system.

Thorp visualized the cubes and their trades in both two- and three-dimensional views. There are some great videos of the system (built in Processing) in action on Jeremy's blog: http://blog.blprnt.com/blog/blprnt/the-colour-economy-the-gap-between-the-rich-and-the-poor.

The interplay between Thorp's cubes, although not smart enough to be considered true artificial intelligence, has sufficient complexity to model a system with obvious cultural parallels. His piece was more than a nice aesthetic: it was also a sociopolitical discussion point, commenting on consumerism, free trade, and the gap between rich and poor.

When you begin down a path of humanizing agents, inevitably you'll hit upon the obvious shortcut.

7.2.2 Human agents

The Apollo 17 Mission, the last (to date) of NASA's Moon landings, produced one of the most important photographs of the 20th century. The shot of Earth shown in figure 7.9, taken as the mission left Earth's orbit on its way out, was the first time the hairless apes, in their masses, had seen their planet like this. Before this photograph, we had highly detailed maps of the world and knew pretty much what it would look like when we got up there, but this was the first time we gained the perspective to see it for ourselves from so far away. It was a great leap in our anthropic subjectivity.

Figure 7.9
The "blue marble" photograph of Earth taken by Apollo 17, December 1972

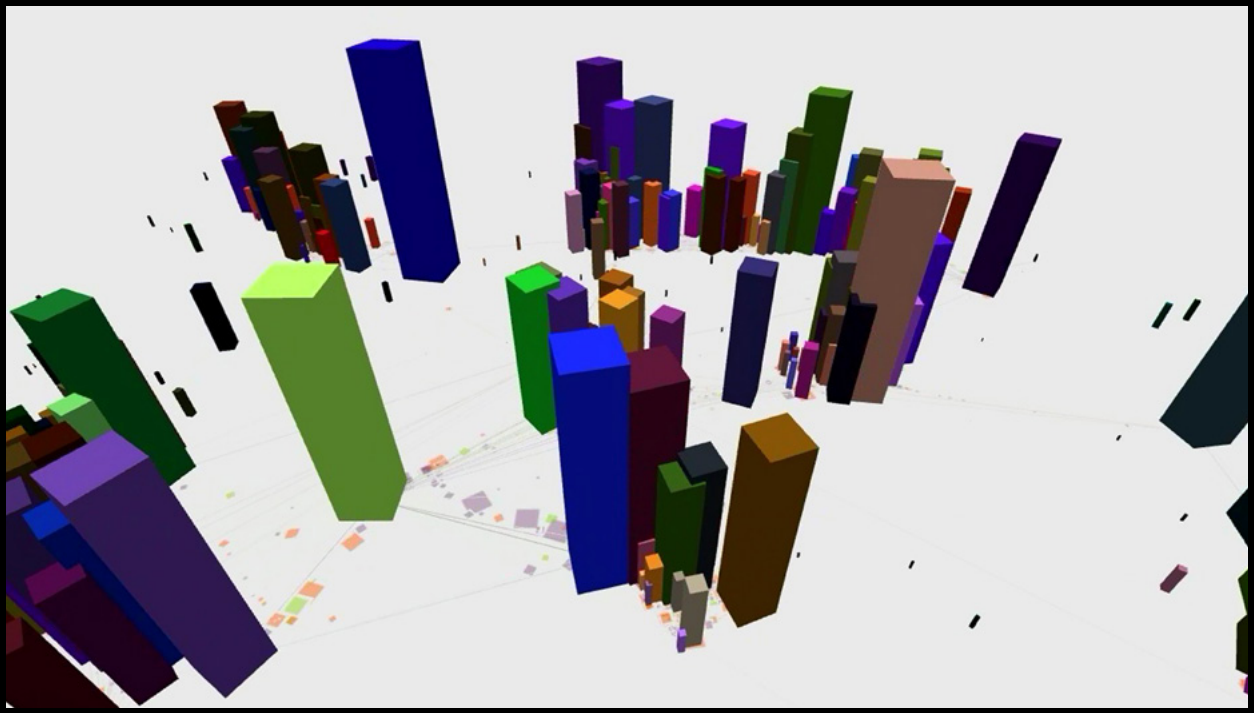

Figure 7.10
Colour Economy by Jeremy Thorp (2008)

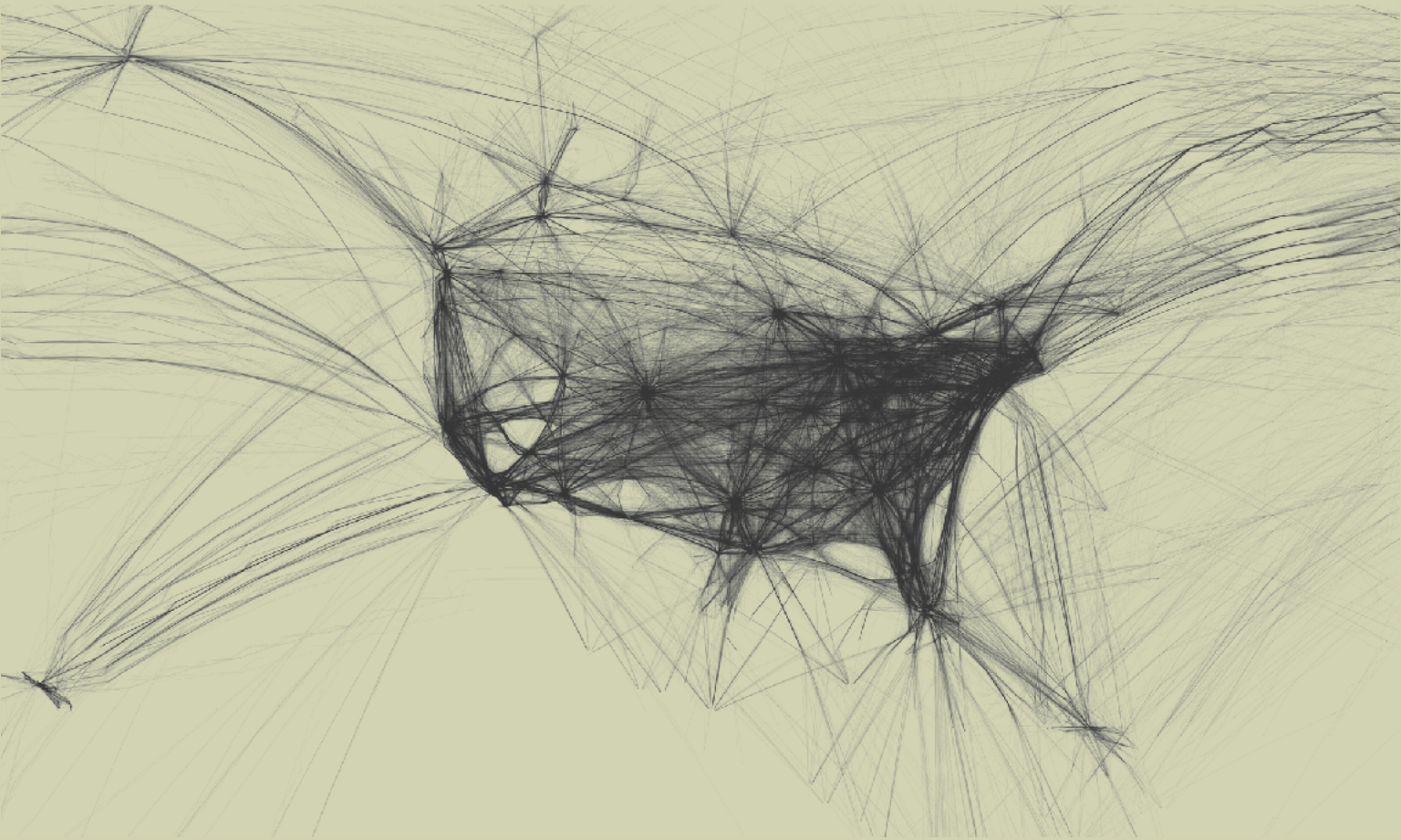

Figure 7.11

Aaron Koblin's *Flight Patterns* (2005): visualization of the flight paths of aircraft crossing North America

To the Google maps generation, this meta-view of the human race is becoming commonplace. It took only 40 years before the power of satellite imagery was on every laptop. It's one of those minor miracles we quickly take for granted when information technology accelerates at such speed. Couple this with a related technology, Global Positioning Systems (GPS), and you have the potential for a new form of unwitting artistic expression.

Your everyday movements leave a trail on the surface of the planet, and this trail draws a pattern. If you're a commuter, your GPS trail probably scores deep furrows as you repeat the same pattern day in day out; but on the weekend, your random psychogeography may create a more complex ballet. Viewed en masse, your data can produce much grander expression, as seen in Aaron Koblin's visualizations of the flight paths of aircraft crossing North America (see figure 7.11 and the color images in the introduction, figure i.18).

If you recall the opening chapters of this book, where I defined what generative art is (and isn't), one of the essential criteria was that the systems need to be autonomous: independent of outside control, free of any guiding hand. When you involve human beings with a generative system, their natural instinct is to attempt to influence it, which would remove the autonomy—*but only if the human's involvement in the system is conscious.* If the human agent is acting unaware, or uninterested, of the effect their actions are having on the system, they become as valid a data source as any other autonomous object.

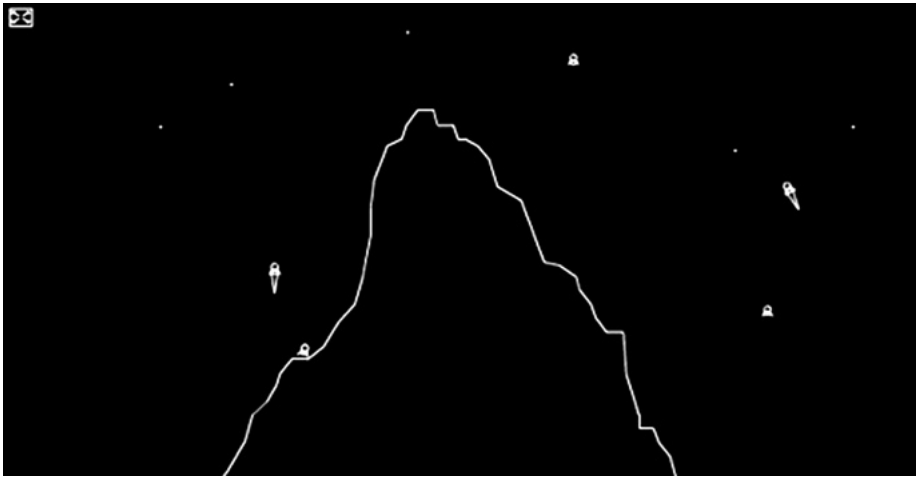

Figure 7.12
Lunar Lander, the classic arcade game re-created in ActionScript

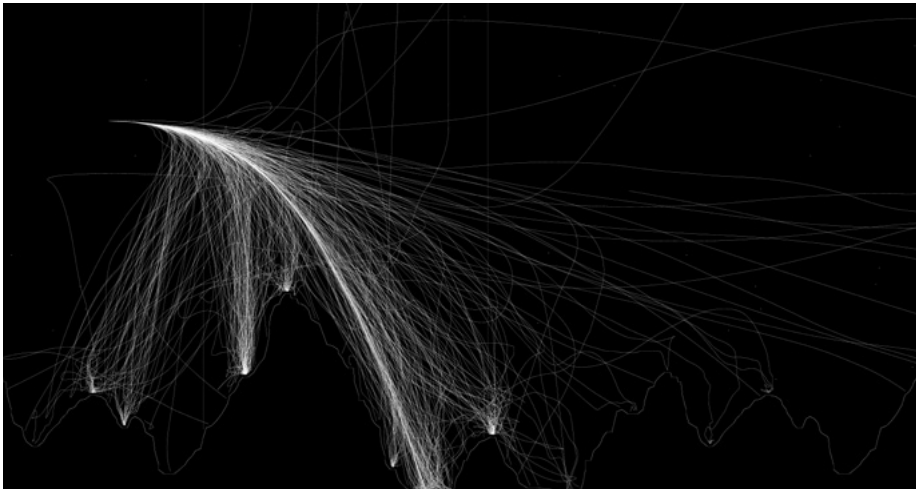

Figure 7.13
Lunar Lander Trails (2010), by Seb Lee-Delisle:
the actions of a many autonomous human agents

A fine example of this arose from a project created by my friend and colleague Seb Lee-Delisle (http://sebleedelisle.com). As an exercise in coding minimalism, he built an AS3 version of the old Lunar Lander arcade game (see figure 7.12), in less than 5 KB. Later, he used this as the basis for experiments with multiuser gaming, building a system to enable many users to play the game over the web. Seb threw his toy out there, inviting any of the great unwashed visiting his site to give it a play. But, unbeknownst to his users, he had the system record the paths they took as they maneuvered their landers safely to the lunar surface.

When he traced this data out onto a composite image, the result, through no intent or design, was surprisingly beautiful (see figure 7.13).

So, it would appear that there are only two challenges in using human agents in the creation of generative art: ensuring that they remain autonomous, and collecting the data. Once, the latter would have been a daunting task, but today it's simple. Data, particularly that relating to the hairless apes and their consumption habits, is being stockpiled as if it were the most valuable commodity we have—which some may argue it is.

Most web or mobile-device users leave these data trails routinely. There are patterns in Twitter streams, web browsing, and GPS location data that can be accessed via a variety of APIs available to programmers, usually for free. For example, more recent works by Jeremy Thorp have been based on the *New York Times'* API (http://blog.blprnt.com/blog/blprnt/7-days-of-source-day-6-nytimes-graphmaker), using Processing to visualize the occurrences of key phrases over the long periods covered by the newspaper's newly digitized archives. The agents in this system are journalists, doing their jobs and writing their articles, obliviously creating meta-patterns in the commonality of their language usage.

Data visualization is an artform in itself, one with huge potential. If you want to explore this area further, I'd suggest you try one of the following options as a launching point:

- Ben Fry's book *Visualizing Data* (O'Reilly 2007) covers the topic in depth, using Processing as the principle tool.
- For the coffee-break version, drop by Jeremy Thorp's blog, particularly the post http://blog.blprnt.com/blog/blprnt/your-random-numbers-getting-started-with-processing-and-data-visualization, which talks through getting Twitter data into a GoogleDocs spreadsheet, reading that spreadsheet into Processing, and visualizing the results.
- If you want to cut straight to the pretty pictures, you'll find plenty to feast your eyes on at www.informationisbeautiful.net and www.visualcomplexity.com

7.3 Summary

The object-oriented approach to code is just the start. With this approach, you can begin realizing more sophisticated systems. Your objects can be autonomous agents, interacting with each other in an artificial environment, producing emergent complexity for you to visualize, an idea you explored through cellular automaton experiments. You then went on to learn that agents don't have to fit such rigid constructs: they don't even need to be software-based. The potential in data visualization opens up a new field of art.

You may not be feeling like such a precious little flower after this excursion—after seeing yourself as the tiniest pixel of raw data, sand in God's great egg-timer. So, next we'll refocus, from the macro to the micro, into the intricacies of another natural construct: fractals. In the final chapter, we'll examine what they are, how to make them, and what you can do with them.

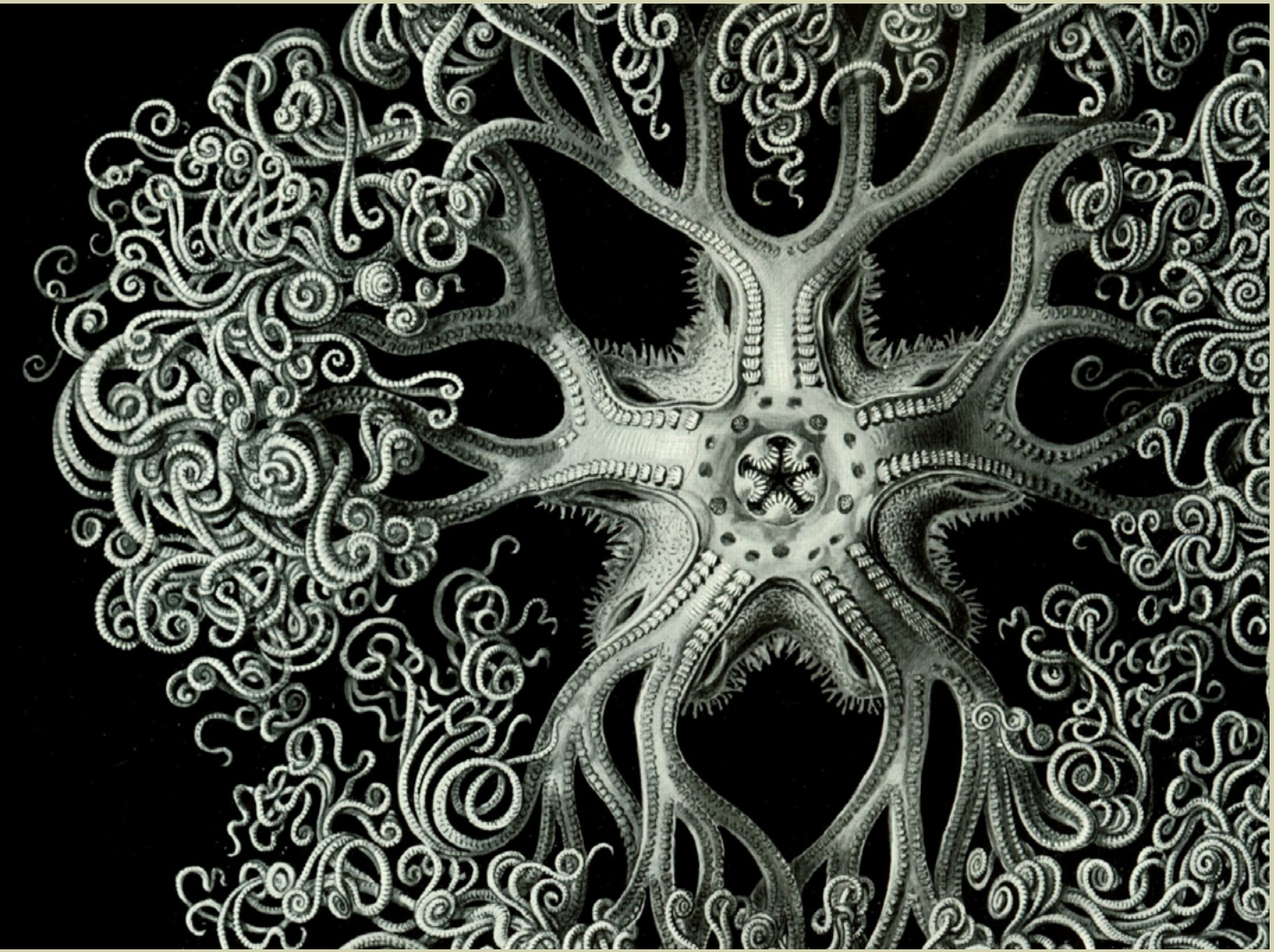

Figure 8.1

After experimenting with fractal structures, you'll start to notice self-similar recursion everywhere. This illustration is from Ernst Haeckel's 1904 book *Kunstformen der Natur (Artforms of Nature)*