The Essential Guide to Processing for Flash Developers

Ira J. Greenberg

friendsof ED
DESIGNER TO DESIGNER™
an Apress® company

THE ESSENTIAL GUIDE TO PROCESSING FOR FLASH DEVELOPERS

Copyright © 2009 by Ira J. Greenberg

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-1979-8

ISBN-13 (electronic): 978-1-4302-1980-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

Credits

President and Publisher:
Paul Manning

Lead Editor:
Ben Renow-Clarke

Technical Reviewer:
Ian Piper

Editorial Board:
Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Coordinating Editor:
Fran Parnell

Copy Editor:
Mary Ann Fugate

Compositor:
Mary Sudul

Indexer:
BIM Indexing & Proofreading Services

Cover Image Designer:
Corné van Dooren

Cover Designer:
Kurt Krames

Chapter 7

Hacking Life

I suspect, if you're like me, that you find the idea of modeling aspects of the natural world a bit more interesting than, say, creating an accounting program (no disrespect to accountants intended). I first got excited by programming precisely because I (finally) made the connection between math and life and glimpsed the potential of code to actually model this relationship. As a painter I used to look at nature and try to deconstruct it visually: *what colors could I see in the leaves*; *how did the horizon recede into the distance*; *what made highlights and shadows appear*. When I moved to code I asked many of the same questions, but could now go much deeper, beneath the surface to the very forces that created what I was looking at: what determined a tree's branch structure; how do migrating birds organize themselves into a V; what causes the patterns in tree bark, marble, clouds, etc. And when I first began to be able to code small examples of some of these things, it was incredibly exciting—like discovering a magic box of paints. I continue to be awed and inspired by this potential of code to literally *hack* life.

Emergence and Complexity

One of the most interesting aspects of coding natural processes is that many seemingly organized, complex structures are actually created by extremely simple rules. Using really basic math, we can quickly glimpse how simple rules create vastly massive structures. For example, by beginning with the number 1 and simply doubling it 70 times, we reach the estimated number of stars in the universe, 10^{21} (according to Astrophysicist Laura Whitlock of NASA's Goddard Space Flight Center). Using some code and a few simple rules you can simulate colonies of insects organizing their surroundings, flocking and swarming behaviors, and all sorts of physical dynamic systems. The key to simulating these types of emergent phenomena is iteration. When simple rules are allowed to be executed hundreds or even thousands of times, unpredictable complex structures can emerge. One classic mathematical model that reveals this potential is Cellular Automata, which also lends itself quite well to programming.

Cellular Automata

Cellular Automata (plural of cellular automaton) or CA were first conceived of in the 1950s by mathematicians John von Neumann and Stanislaw Ulam, when both men were working at Los Alamos National Laboratory, New Mexico. At first, CA were more of a mathematical abstraction that held the promise of developing self-replicating structures—theoretically even the fundamental structures of life itself. Von Neumann and Ulam's pioneering work didn't really impact the wider scientific community until the 1970s, when their work was expanded (and popularized), due in large part to advances in and increased accessibility to computation, including the capability to more easily create computer-generated graphics.

In 1971 Princeton mathematician John Conway created arguably the most famous CA, "Conway's Game of Life," which brought CA to the popular imagination through an article written about it in Scientific American. A decade later, Steven Wolfram, founder of the Mathematica software package, continued the CA charge, eventually publishing a massive tome related to the subject: *A New Kind of Science*. For a wonderful history about CA, check out this article in the CellLab manual, by authors Rudy Rucker and John Walker: `http://www.fourmilab.ch/cellab/manual/chap5.html` (June 23, 2009, 23:01).

In spite of the rather lofty aspirations of CA originators von Neumann and Ulam, the basic concepts behind CA are quite simple:

Create a finite set of cells in a grid-like configuration, where each cell has a set of states (most commonly two, for "on" or "off") that is controlled by its surrounding cells (its neighborhood) from the previous generation.

The simplest type of CA is one-dimensional (1D), meaning that a single cell's state is controlled only by neighboring cells along one axis. The classic 1D CA uses three contiguous cells, shown in Figure 7-1.

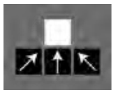

Figure 7-1. Cell with controlling contiguous cells

Since there are three cells—defined as the neighborhood—each with an on or off state, there is a total of eight possible configurations (2^3), shown in Figure 7-2. In running a simulation, you apply very simple rules that change the pixel values (turn them off or on) based on the neighborhood configuration. Figure 7-3 shows the previous figure updated with some simple rules. Please note these rules are arbitrary and can thus be changed, as I'll demonstrate later in the chapter.

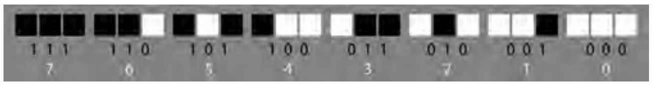

Figure 7-2. Three cell (on/off) configurations

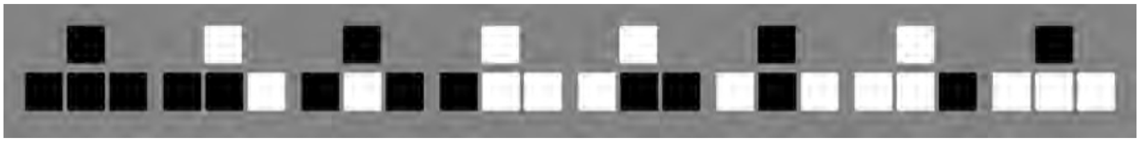

Figure 7-3. CA visual rules table

Programming a simple 1D CA is fairly straightforward. As discussed earlier, each pixel's state will be determined by three contiguous pixels (itself and the pixels to its left and right). These new calculated pixel values will, in a sense, represent pixel values in the next generation. Thus, in a programming implementation it is helpful to use two separate arrays (named `bits` and `pixels` in the upcoming example) representing the preset and future pixels. The initial simple 1D Cellular Automata program is listed next.

```
/**
 * Simple 1D Cellular Automata
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 *
 * Good CS article
 * http://www.generation5.org/content/2003/caIntro.asp
 */

// array for bit values
int[] bits;
/* CA rules, 4th val is new bit state
(0=off, 1 = 0n), based on each rule */
int[][] rules = {
    {0,0,0,0},
    {1,0,0,1},
    {0,1,0,1},
    {0,0,1,1},
    {1,1,0,1},
    {0,1,1,1},
    {1,0,1,1},
    {1,1,1,0}
};
```

```
void setup(){
  size(400, 400);
  // map color values between 0-1.0
  colorMode(RGB, 1.0);
  // access pixels array of sketch window
  loadPixels();
  // instantiate bits array to size of sketch
  bits = new int[width*height];
  // initialize starting bit state
  initNeighborhood();
}


// create initial state
void initNeighborhood(){
  // turn bottom middle bit on
    bits[width*(height-1) + width/2] = 1;
}

// update bits based on CA rules
void createGeneration(){
  for(int i=0; i<height-1; i++){
    for(int j=0; j<width; j++){


       // 1st and last columns use each other as neighbors in calculation
          int firstCol = (j==0) ? width-1 : j-1;
          int endCol = (j>0 && j<width-1) ? j+1 : 0;

      // check rules
      for(int k=0; k<rules.length; k++){
        if (bits[width*(i+1)+firstCol] == rules[k][0] &&
          bits[width*(i+1)+j] == rules[k][1] &&
          bits[width*(i+1)+endCol] == rules[k][2]){
              bits[width*i+j] = rules[k][3];
          }
       }
     }
  }
}

// paint screen pixels based on stored values in bits
void setCells(){
  for (int i=0; i<bits.length; i++){
    // casts int to color data type for pixel value
    pixels[i] = color(bits[i]);
  }
    // call whenever changing pixels array
  updatePixels();
}

void draw(){
  // calculates CA
  createGeneration();
  // copies bit values to PImage pixels[]
  setCells();
}
```

In this first CA example, I tried to create a very simple implementation, with the trade-off for the simplicity being a lack of parameterization to easily customize the program. But don't worry, in the CA examples to follow, I'll provide lots of opportunity for customization (with, of course, the requisite increased complexity).

In the **`simple 1D CA`** example, I used `int` arrays (`bits[]` and `rules[][]`) as the main data structures. You'll see shortly why this provided for both an efficient and simple solution. The basic program execution proceeds with the creation of a single on/off state, followed by a rules analysis and then the creation of the next generation. In this implementation the initial state is simply turning the bottom center pixel to "on" (painted white), while all the other pixels are initialized to "off" (painted black). The main execution happens repeatedly within `draw()`, which allows the CA to proceed across the entire screen, until a steady state is reached. Later in the chapter, we'll look at some two-dimensional CA examples that actually never reach this type of (static) steady-state and continually show the genesis of later generations. If you haven't yet, try running the example. A screen-shot of the final steady-state of the CA is shown in Figure 7-4.

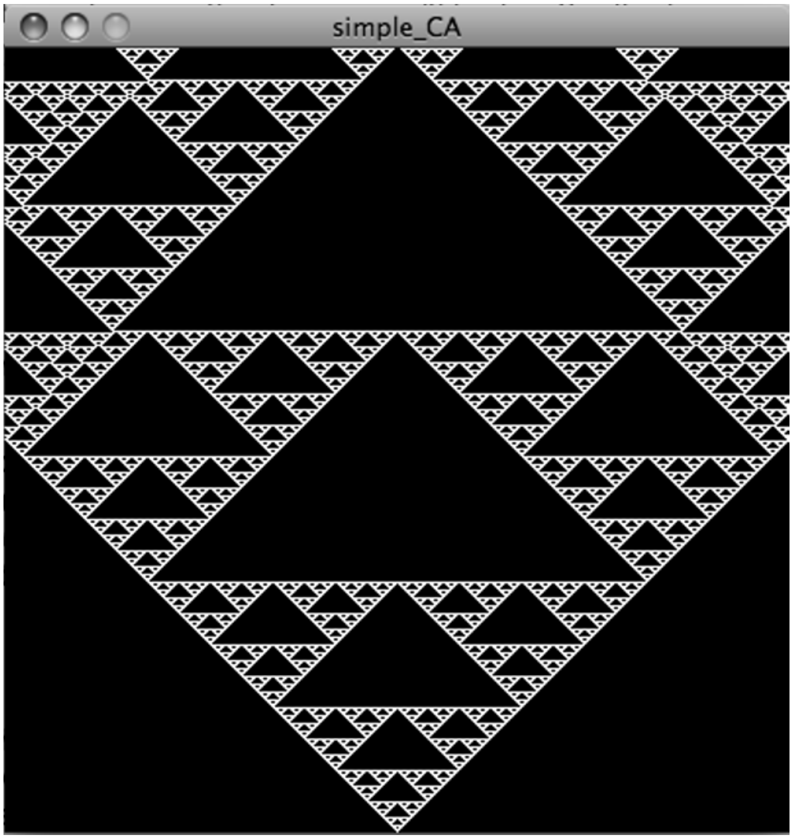

Figure 7-4. Simple_1D_CA screen-shot

If you haven't seen CA before, perhaps the order (and beauty) of the output surprised you. You'll see shortly that this example demonstrates just the very tip of the iceberg of what's possible with CA.

Returning to the example code, I want to clarify why I chose to use `int` arrays. Processing structures its `color` data type as a packed 32-bit integer, in the format aaaaaaaa rrrrrrrr gggggggg bbbbbbbb, with 8-bits for alpha, red, green, and blue respectively. Because of this relationship between the `int` and `color` types in Processing, it's possible to cast a plain old integer into a specific pixel value, as I do in the `setCells()` function, with the line

```
pixels[i] = color(bits[i]);
```

Casting again is the converting of one data type into another. There are specific rules about type casting, and not all types can be converted to one another. I cover type casting in Processing in Chapter 2. The **`simple_1D_CA`** example will only use an off or on state for each pixel, so I utilized a 0 and 1 to record these two states respectively, which obviously fit well within the range of the `int` type.

It might seem more efficient to try to utilize a smaller data structure than a 32-bit integer to record 1 bit of information, such as Processing's `char` type, which is 16 bits; or `byte`, which is only 8 bits; or best of all perhaps Processing's `Boolean` type, which presumably would be a 1-bit data structure, only needing to account for `true` or `false`. Because Processing (really Java) has an internal memory management scheme in conjunction with a virtual machine, it turns out these seemingly smaller data types internally utilize more memory than assumed, and in some cases would also require additional casting to work as a valid `color` type.

Returning to the **`simple_1D_CA`** example, the `rules[][]` array (shown again next)

```
int[][] rules = {
    {0,0,0,0},
    {1,0,0,1},
    {0,1,0,1},
    {0,0,1,1},
    {1,1,0,1},
    {0,1,1,1},
    {1,0,1,1},
    {1,1,1,0}
};
```

functions as a look-up table of the CA rules. `rules[][]` is a 2D array, or an array of arrays, with an overall length of eight, with each internal array having a length of four. The first three values in the internal arrays account for the neighborhood states I discussed earlier, and the fourth value has the rule for that state. For example, when one is looking at the first internal array, `{0,0,0,0}`, if the neighborhood is all off (all 0's), then the pixel being evaluated will be turned/remain off. In the second array `{1,0,0,1}`, if the pixel on the left is on and the next two pixels to its right are off, then the pixel being evaluated will be turned/remain on. The reason I wrote "turned/remain" is because the evaluation will occur on the pixel row directly below the pixel affected, so the affected pixel could either be on or off. Please note also the rule (the fourth value in each array) is hard-coded in this initial example, but in later examples in the chapter, you'll be able to pass arguments to create variations to the rules, and thus output.

The main work in the example is handled by the `createGeneration()` function, listed again next:

```
void createGeneration(){
  for(int i=0; i<height-1; i++){
    for(int j=0; j<width; j++){

       // 1st and last columns use each other as neighbors in calculation
          int firstCol = (j==0) ? width-1 : j-1;
          int endCol = (j>0 && j<width-1) ? j+1 : 0;
```

```
      // check rules
      for(int k=0; k<rules.length; k++){
        if (bits[width*(i+1)+firstCol] == rules[k][0] &&
          bits[width*(i+1)+j] == rules[k][1] &&
          bits[width*(i+1)+endCol] == rules[k][2]){
          bits[width*i+j] = rules[k][3];
        }
      }
    }
  }
}
```

Using nested `for` loops, the function moves through the `bits[]` array, where each value in the array represents a specific pixel in the sketch window. The actual pixels are stored in another array aptly named "`pixels`."

> *Please remember that to access the sketch window's global `pixels` array, you need to first call Processing's `loadPixels()` function, which I did up in the `setup()` function.*

The actual CA rules evaluation occurs within the conditional block, within the nested `for` loops. In regard to program flow, `createGeneration()` is called from within `draw()`, which executes at Processing's default frame rate (60 FPS). Each draw cycle, the nested `for` loops process the entire `bits` array checking for matches against the rules. As I mentioned earlier the analysis occurs on the row directly beneath the actual bit (ultimately pixel) affected. You can think of the rows as representing different generations (present and future respectively).

> *Nesting three `for` loops is bit confusing at first glance (alright, even on later glances), and in truth is not terribly efficient, performance-wise speaking. You could conceivably use at least one less loop and treat the `bits` and `pixels` arrays as single-dimensional arrays (which of course they really are). However, I personally find it easier to think about (and process) arrays that represent 2D data (a table structure) using a procedure that accounts for rows and columns, which is what the extra loop provides.*

With regard to the `rules` table, within the nested conditional block when all three statements evaluate to true, then the rule (the fourth value in the same nested array) is applied. Since the rules look-up table accounts for all possible neighborhood configurations, every pixel's state is accounted for this way. A final point about this function refers to the two rather ugly lines

```
int firstCol = (j==0) ? width-1 : j-1;
int endCol = (j>0 && j<width-1) ? j+1 : 0;
```

If you're not familiar with this syntax, it uses the ternary operator `?:` (The same one exists in ActionScript.) I have to admit to not really being a big fan of it, but in this case it seemed to keep the function from getting too pudgy. The ternary operator allows you to do terse `if/else` expressions, but, some would say, with decreased readability—until (I guess) you get really used to it. If it isn't obvious, the ternary operation is `(Boolean condition) ? (stuff to do if true) : (stuff to do if false)`. The reason I included these two expressions in the first place was to account for the first and last pixel in each column. Since the CA neighborhood in this example includes three contiguous pixels (used to evaluate every pixel) there will be an edge problem (a missing third pixel) on the first and last pixel in each row. To account for this I wrap the window pixels, by using the pixels on the opposite edge of the sketch window as the third pixel. In other words, when a right column edge pixel is evaluated, the pixel to its left and the first pixel on the left side of the screen are used for the rules evaluation. (If you're wondering, I didn't invent this idea, but saw it implemented in numerous other CA implementations.)

Again, this initial sketch was intended to give you a down-and-dirty look at CA; more interesting variations are coming. However, if you simply can't wait for the next example, you can create some variation in this example by altering the initial starting state in the `initNeighborhood()` function. For example, Figure 7-5 shows a screen-shot created using the following starting position: `bits[width*(height-1) + width/8] = 1`. You can also easily adjust the rules by changing which conditions result in on or off pixels—just be prepared for some funky results.

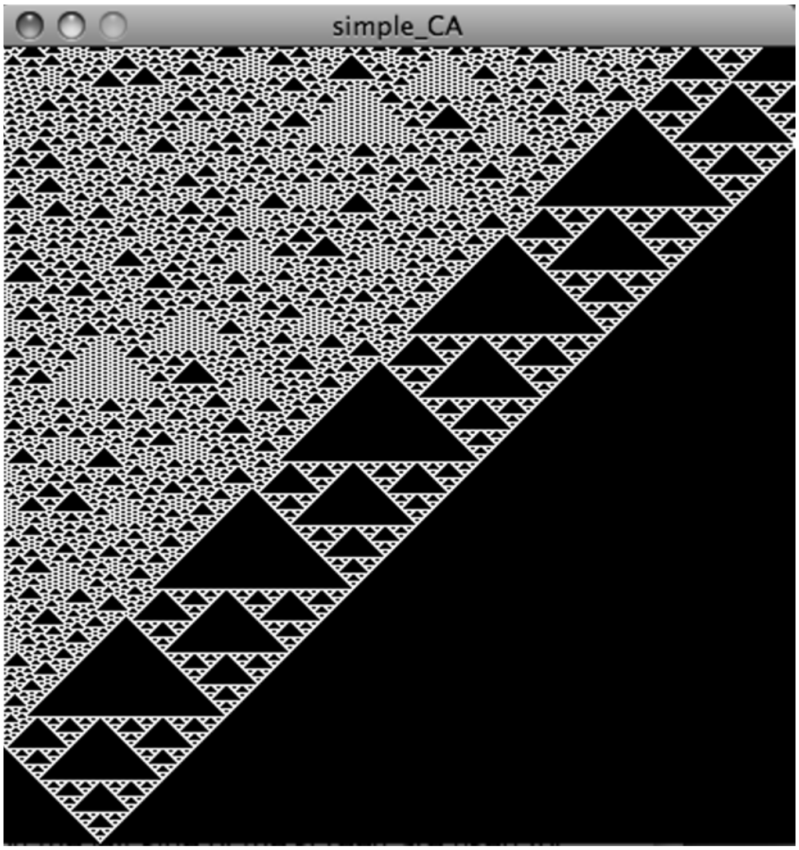

Figure 7-5. Simple_1D_CA screen-shot based on altered initial state

In considering a more robust and creative application of CA, it will help to create a well-structured program, so I'll utilize an OOP approach. (I hope you feel refreshed after the brief procedural respite.) The next example will function as both a framework for extended 1D and 2D CA development, as well as a small showcase of their creative potential.

1D CA Framework

There were a number of issues I tried to address in this example. I wanted users to be able to

- customize the CA, including altering color
- zoom-in to see the CA at different pixel resolutions
- use the CA as part of a larger image
- create their own CA subclasses

To follow along with the example, create a new sketch named whatever you like, and then add a new tab to the sketch named "Shape." Here's the `Shape` class code.

```
/**
 * Cellular Automata
 * Shape class - convenience class
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

class Shape {

  // instance properties
  PVector loc = new PVector();
  float w;
  float h;



  // default constructor
  Shape(){
  }
  Shape(float w, float h){
    this.w = w;
    this.h = h;
  }

  // constructor
  Shape(float x, float y, float w, float h){
    loc.x = x;
    loc.y = y;
    this.w = w;
    this.h = h;
  }

  // setters
  void setLoc(float x, float y){
    loc.x = x;
    loc.y = y;
  }

  void setLoc(PVector loc){
    this.loc = loc;
  }

  void setSize(float w, float h){
    this.w = w;
    this.h = h;
  }

}
```

This Shape class is very straightforward with concepts I've covered throughout the book. It will serve as a base class, encapsulating a location and size, which other classes will extend. Next, create a tab named "Cell," including the following code:

```
/**
 * Cellular Automata
 * Cell class
 * - encapsulates drawing to pixel buffer
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

class Cell extends Shape{

  color c;
  // reference to CA obj
  CA ca;

  Cell(float x, float y, float w, float h, CA ca){
    super(x, y, w, h);
    this.ca = ca;
  }

  void setColor(color c){
    this.c = c;
  }

  // draw to pixels buffer
  void create(){
    float origin = int(loc.y) * ca.w + int(loc.x);
    for (int i=0; i<w; i++){
      for (int j=0; j<h; j++){
          // - pretty nasty pixls[index] expression
        ca.p.pixels[int(min(origin + j*ca.w + i, ca.w*ca.h))] = c;
      }
    }
  }
}
```

The Cell class, which you'll notice extended Shape, will function primarily as a utility class that allows pixels to be grouped into a larger block or cell. The class is also straightforward, with the exception of its create() method, which converts the **cell** construct, from a higher-level 2D component, to specific index values in a pixel buffer. I'll return to this method later in the chapter.

Next, create a new tab named "**CA**," which will be the base class for **1D** (and some **2D)** CA. Add the following code to the CA tab:

```
/**
 * Cellular Automata
 * CA class (base class)
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

abstract class CA extends Shape{
  // instance properties, including default values
  int cellScale = 3;
  int rows = 10, cols = 10;
  float rowSpan, colSpan;
  Cell[][] cells;
  PImage p;
  color[] pixls;
  color[] nextPixls;
  int[] state;

  // default start colors
  color onC = 0xff000000;
  color offC = 0xffffffff;

  // default constructor
  CA(){
    super(200.0, 200.0);
    initCA();
  }
```

```
// constructor
CA(float w, float h, int cellScale){
  super(w, h);
  this.cellScale = cellScale;
  rows = ceil(h/cellScale);
  cols = ceil(w/cellScale);
  initCA();
}

// initialize
void initCA(){
  pixls = new color[rows*cols];
  nextPixls = new color[rows*cols];
  // record current pixel on/off state as integer array
  state = new int[pixls.length];
  colSpan = w/cols;
  rowSpan = h/rows;

  cells = new Cell[rows][cols];
  for (int i=0; i<rows; i++){
    for (int j=0; j<cols; j++){
      // instantiate cells
      cells[i][j] = new Cell(colSpan*j, rowSpan*i, colSpan, rowSpan, this);
    }
  }
  p = createImage(int(w), int(h), RGB);
}

// set starting state (single pixel)
void setInitState(int id){
  resetState();
  // turn initial pixel on
  pixls[id] = onC;
  recordState();
  paintInitState();
}

// set starting state (array of pixels)
void setInitState(int[] ids){
  resetState();
  for (int i=0; i<ids.length; i++){
    pixls[ids[i]] = onC;
  }
  recordState();
  paintInitState();
}

// set starting state (single pixel using 2D coord)
void setInitState(int row, int col){
  resetState();
  pixls[row*(cols-1) + (col-1)] = onC;
  recordState();
  paintInitState();
}

// record pixel state in integer array (1 = on, 0 = off)
void recordState(){
  for (int i=0; i<pixls.length; i++){
    if (pixls[i] == onC){
      state[i] = 1;
    }
    else {
      state[i] = 0;
    }
  }
}

// update pixels based on state integer array
```

```
void updateState(){
  for (int i=0; i<state.length; i++){
    if (state[i] == 1){
      pixls[i] = onC;
    }
    else {
      pixls[i] = offC;
    }
  }
}

// ensure starting pixel state is rendered
void paintInitState(){
  arrayCopy(pixls, nextPixls);
  paint();
}

// reset all pixels to off
void resetState(){
  for (int i=0; i<pixls.length; i++){
    pixls[i] = offC;
  }
}

// paint "dem perty" cells
void paint(){
  p.loadPixels();
  for (int i=0; i<rows; i++){
    for (int j=0; j<cols; j++){
      cells[i][j].setColor(nextPixls[cols*i + j]);
      cells[i][j].create();
    }
  }
  p.updatePixels();
```

```
    image(p, -w/2+loc.x, -h/2+loc.y);
    arrayCopy(nextPixls, pixls);
  }

  // must be implemented in subclass (or subclass will be abstract)
  abstract void init();
  abstract void createGeneration();

}
```

CA is an abstract class, meaning that it can't be instantiated directly. Abstract classes can include properties and methods like in a standard class, as well as abstract method stubs, or unimplemented methods such as

```
abstract void init();
abstract void createGeneration();
```

These methods are declared with the abstract **keyword** and do not include a method block (are unimplemented).

Abstract methods must be implemented in any class that extends the abstract class, or the subclass automatically becomes abstract as well.

One of the main benefits of an abstract class is an enforced common interface. For example, it is safe to assume that any (non-abstract) class that extends CA will include implemented init() and createGeneration() methods. Each CA subclass can implement these methods to suit its own needs. In other words, the method interfaces will be common (between CA subclasses) but the implementation of the methods will not. Abstract classes help enforce good black-box design, where the interface to the box, not what happens inside the box, is what's known (accessible).

Even though abstract classes can't be directly instantiated, they can still contain constructors that are invoked by subclass constructors. This provides the same benefit of a standard superclass in regard to being able to efficiently initialize an object though chained constructors. The CA subclasses we'll create will use the CA constructor for this purpose.

Next we'll look at the instance property arrays (cells, pixls, nextPixls, state) in CA declared at the top of the class,

- cells references the higher level component constructs I mentioned earlier. The number and size of the Cell objects will be based on the overall size of the CA object and the cellScale property; larger scale values will create fewer but larger cells.
- pixls and nextPixls will directly reference the CA object's pixel buffer—present and next generation respectively.

- state will help with bookkeeping of a sort, keeping track of the pixel on/off state, without having to worry about the specific pixel color values (since the pixels will not only be black and white as with the previous example).

The initCA() method initializes the arrays and instantiates the Cell objects. Notice in the instantiation call I pass a reference to the CA object, as the last argument

```
cells[i][j] = new Cell(colSpan*j, rowSpan*i, colSpan, rowSpan, this);
```

Finally, with line p = createImage(int(w), int(h), RGB); I create an off-screen image that contains a pixel buffer (pixels array). Unlike the simple_1D_CA example, I will not write directly to the sketch window pixels array. Instead, I'll write to the pixel array of the off-screen image; then when I want to render the CA to the screen, I'll draw the off-screen image using image(p, x, y). Also, remember that Processing includes both a pixels array global variable and a PImage pixels array instance property (this was one of the reasons I chose to name my color array pixls in the example.)

In the simple_1D_CA example, the starting state was limited to a single pixel. Here, we can also use an array of pixels. The overloaded setInitState() methods provide a public interface for initiating the neighborhood state. There are also a number of component utility methods, including recordState(), updateState(), paintInitState(), and resetState(). These will allow us to both run the CA in real time, as well as to step through each generation, using, for example, a mouse event.

Finally the paint() method, included again next, coordinates the drawing of the pixels to the screen. This is a bit more involved than perhaps at initial glance, so I'll walk through the process.

```
// paint "dem perty" cells
  void paint(){
    p.loadPixels();
    for (int i=0; i<rows; i++){
      for (int j=0; j<cols; j++){
        cells[i][j].setColor(nextPixls[cols*i + j]);
        cells[i][j].create();
      }
    }
    p.updatePixels();
    image(p, -w/2+loc.x, -h/2+loc.y);
    arrayCopy(nextPixls, pixls);
  }
```

The first step is to "safely" load the p.pixels array using the p.loadPixels() call. Honestly, I find this step a bit of a clunky implementation and inconsistent with standard OOP. And in truth, it may even be possible to access the PImage pixels array without this call. However, it is strongly advised in the language reference that this call always be used to ensure the array is properly created/loaded. Here's what the reference has to say about it:

> *"Certain renderers may or may not seem to require loadPixels() or updatePixels(). However, the rule is that any time you want to manipulate the pixels[] array, you must first call loadPixels(), and after changes have been made, call updatePixels(). Even if the renderer may not seem to use this function in the current Processing release, this will always be subject to change."*

Next in paint(), within the for loops, the cells' colors are updated based on the nextPixls array, and then the cells.create() method is invoked. To see again what happens in the create() method, click on the **Cell** tab. I've copied the method again next:

```
// From Cell.pde
// draw to pixels buffer
  void create(){
    float origin = int(loc.y) * ca.w + int(loc.x);
    for (int i=0; i<w; i++){
      for (int j=0; j<h; j++){
          // - pretty nasty pixls[index] expression
        ca.p.pixels[int(min(origin + j*ca.w + i, ca.w*ca.h))] = c;
      }
    }
  }
```

The create() method draws a block of pixels, based on the width and height specified for the Cell object. This block is drawn directly into the p.pixels array, which, you'll remember, was instantiated back in CA; this was the reason I needed to pass a reference to CA when I instantiated the Cell objects. Drawing the block of pixels in the right place in the p.pixels array was tricky, especially since the pixels arrays in Processing are one-dimensional. The full-length expression I needed for targeting each pixel in the correct order based on the nested for loops was

```
ca.p.pixels[int(min(int(loc.y) * ca.w + int(loc.x) + j*ca.w + i, ca.w*ca.h))] = c;
```

That is one scary-looking line of code that should make your head hurt; it really did mine while I was trying to figure it out. The int() casting and min() calls are needed to ensure rounding errors do not allow the index value to go out of (array length) bounds. In the create() method, you'll notice, I broke the expression into two lines to make it a bit more comprehensible.

Returning to the paint() method in CA, after p.pixels is written to, I call p.updatePixels() to ensure the pixels array is properly updated, and then I draw the image to the sketch window with image(p, -w/2+loc.x,

-h/2+loc.y). The last step, arrayCopy(nextPixls, pixls) copies the nextPixls array values to pixls, updating the pixel state (previous generation) to current generation values. arrayCopy() is an efficient Processing function for copying the contents (or part of the contents) of one array into another. You can read more about the function at http://processing.org/reference/arrayCopy_.html.

To use CA, we need to create a subclass that extends it. The first I'll show will build upon what we looked at in the simple_1D_CA example, with added parameterization; then we'll look at an interesting variation on the 1D CA.

Create a new tab named "**CA_1D**" and copy the following code into it:

```
/**
 * Cellular Automata
 * CA_1D class
 * neighborhood:   | ? |
 *               * | * | *
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

class CA_1D extends CA{
  // instance properties

  // CA rules
  boolean[] rules = new boolean[8];
  color[][] table = new color[8][4];

  // default constructor
  CA_1D(){
    super();
    init();
  }

  // constructor
  CA_1D(int w, int h, int cellScale){
    super(w, h, cellScale);
    init();
```

```
}

// REQUIRED implementation – initializes stuff
void init(){
  // initialize 1D rules
  initRules();
  // build rules table
  buildTable();
  //set default pixel starting state - bottom center pixel set to on
  int middleBottomCell = (rows-1)*(cols) + (cols)/2;
  setInitState(middleBottomCell);
  // record pixel on/off state in integer state table
}

// initialize rules
void initRules(){
  rules[0] = false;
  rules[1] = true;
  rules[2] = true;
  rules[3] = true;
  rules[4] = true;
  rules[5] = true;
  rules[6] = true;
  rules[7] = false;
}

// build rules table
void buildTable() {
  table[0][0] = offC;
  table[0][1] = offC;
  table[0][2] = offC;
  table[0][3] = rules[0] ? onC : offC;
  table[1][0] = offC;
  table[1][1] = offC;
  table[1][2] = onC;
```

```
    table[1][3] = rules[1] ? onC : offC;
    table[2][0] = offC;
    table[2][1] = onC;
    table[2][2] = offC;
    table[2][3] = rules[2] ? onC : offC;
    table[3][0] = offC;
    table[3][1] = onC;
    table[3][2] = onC;
    table[3][3] = rules[3] ? onC : offC;
    table[4][0] = onC;
    table[4][1] = offC;
    table[4][2] = offC;
    table[4][3] = rules[4] ? onC : offC;
    table[5][0] = onC;
    table[5][1] = offC;
    table[5][2] = onC;
    table[5][3] = rules[5] ? onC : offC;
    table[6][0] = onC;
    table[6][1] = onC;
    table[6][2] = offC;
    table[6][3] = rules[6] ? onC : offC;
    table[7][0] = onC;
    table[7][1] = onC;
    table[7][2] = onC;
    table[7][3] = rules[7] ? onC : offC;
  }

  // REQUIRED implementation
  void createGeneration(){
    for (int i=0; i<rows-1; i++){
      for (int j=0; j<cols; j++){
        for (int k=0; k<rules.length; k++){
          // 1st and last columns use each other as neighbors in calculation
          int firstCol = (j==0) ? cols-1 : j-1;
          int endCol = (j>0 && j<cols-1) ? j+1 : 0;
          // rules determined by binary table: 0 = offCol, 1 = onC.
          // [111][110][101][100][011][010][001][000]
          if (pixls[cols*(i+1) + firstCol] == table[k][0] &&
            pixls[cols*(i+1) + j] == table[k][1] &&
            pixls[cols*(i+1) + endCol] ==  table[k][2]){
              nextPixls[(cols)*i + j] = table[k][3];
          }

        }
      }
    }
    // paint pixels on screen
    paint();
  }

  // update rules - requires 8 boolean values
  void setRules(boolean[] rules) {
    this.rules = rules;
    buildTable();
  }

  void setOnColor(color onC){
    this.onC = onC;
    buildTable();
    updateState();
    paintInitState();
  }

  void setOffColor(color offC){
    this.offC = offC;
    buildTable();
    updateState();
    paintInitState();
  }
}
```

Although this example will follow rules similar to those of the `simple_1D_CA` example, the implementation will be different. Rather than using `int` arrays to store 0's and 1's only, I used `color[]` arrays, to refer to actual pixel values; I did this to allow for display of a full range of color.

The `CA_1D` constructors, as discussed earlier, call the `CA` constructors for initialization. In addition, the subclass has its own initialization routine, `init()`, which, you'll remember, must be implemented since the method was declared `abstract` in `CA`. The `init()` implementation includes the call `setInitState(middleBottomCell)`, for setting a default starting on/off state (the first generation).

Reading through the rest of the class, I split the `rules` table into two methods, `initRules()` and `buildTable()`, to allow users to be able to set custom rules, which I'll demonstrate shortly. The `createGeneration()` method, like `init()`, was required to be implemented since it was also declared `abstract` in `CA`. Its implementation is quite similar to the same named function in the `simple_1D_CA` example. However, now the conditional block is comparing actual pixel color values, instead of just 0's and 1's. This method also includes a call to `paint()`, defined in `CA`. The rest of the class code consists of setter methods, which I'll assume are self-explanatory. Next, we'll generate some sample CA, using the new classes.

In the main tab, which should still be blank if you've been following along, add the following code and run the sketch:

```
/**
 * Cellular Automata Main Tab - 01
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

CA_1D ca;

void setup(){
  size(600, 600);
  background(255);
  ca = new CA_1D(600, 600, 1);
}

void draw(){
  translate(ca.w/2, ca.h/2);
  ca.createGeneration();
}
```

If the sketch ran successfully you should see `output`, shown in Figure 7-6, similar to the `simple_1D_CA` example, only with the black and white colors reversed.

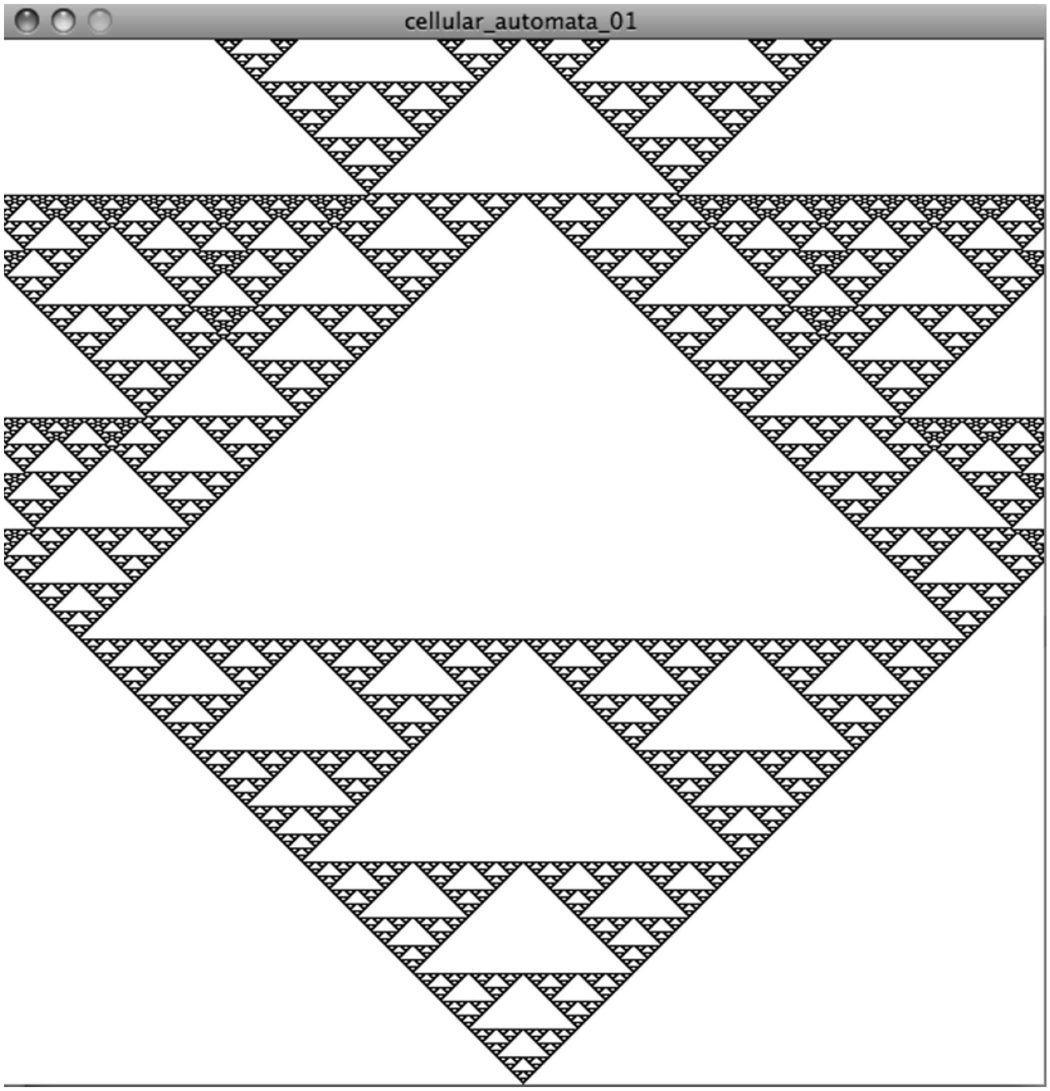

Figure 7-6. 1D Cellular Automata screen-shot, stage 1

Obviously if all we wanted to do was this, the `simple-1D-CA` implementation would have sufficed. In the next step we'll change the colors of the cells as well as the scale. Here's the updated main tab code, with the changes in **bold**.

```
/**
 * Cellular Automata Main Tab - 02
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

CA_1D ca;
color onC = 0xff22ee33;
color offC = 0xff772299;

void setup(){
  size(600, 600);
  background(255);
  ca = new CA_1D(600, 600, 5);
  ca.setOnColor(onC);
  ca.setOffColor(offC);
}

void draw(){
  translate(ca.w/2, ca.h/2);
  ca.createGeneration();
}
```

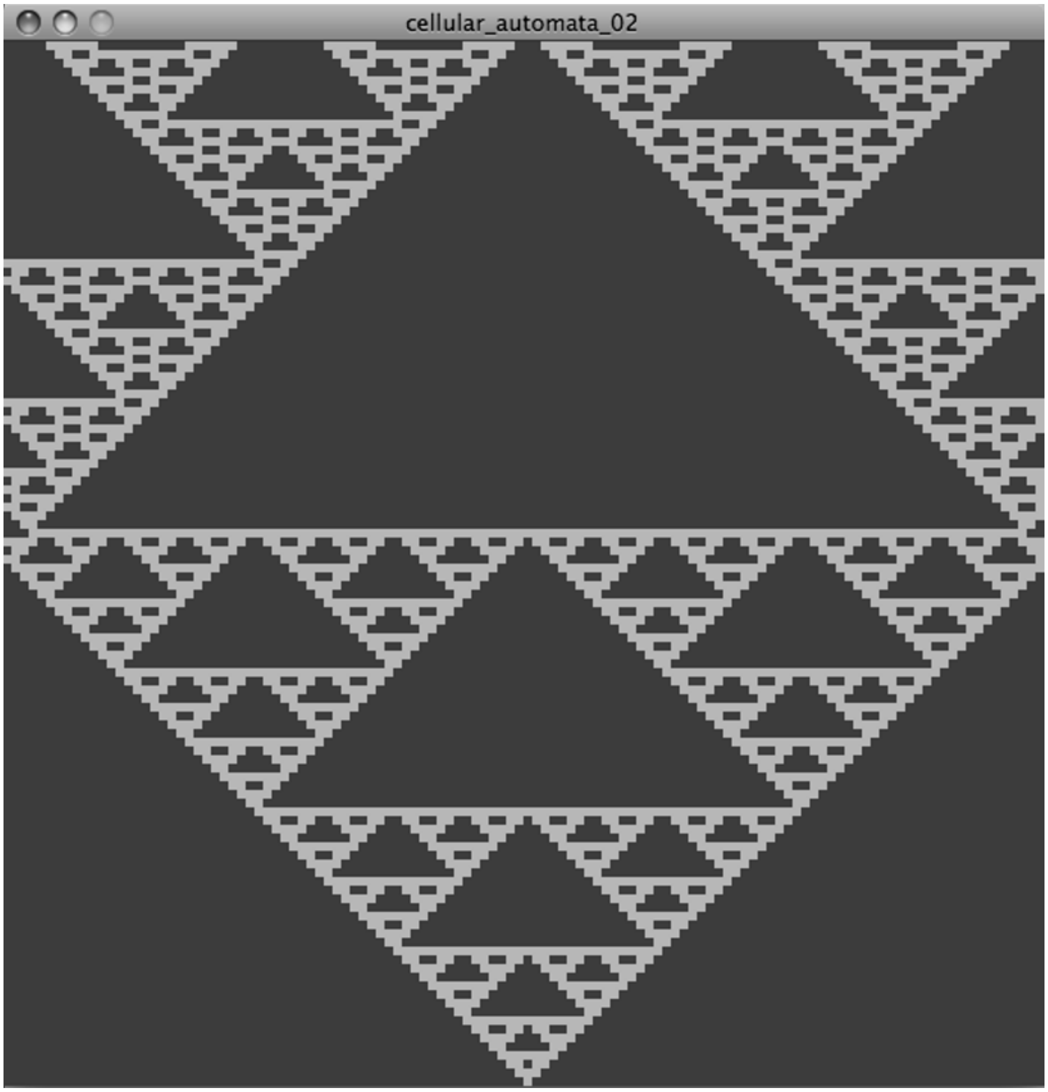

Figure 7-7. 1D Cellular Automata screen-shot, stage 2

Notice in the purple and green output how the larger scale factor turns each pixel into a block, creating, in a sense, a magnified bitmap of the image. In this implementation, save for memory limitations, there is no maximum limit for the scale factor.

As we briefly looked at earlier, you can change the initial on/off start state to influence the final output. In this implementation you can specify a single cell as we did earlier, or an array of cells, which I'll demonstrate next, shown in Figure 7-8; again the new code is **bold**.

```
/**
 * Cellular Automata Main Tab - 03
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

CA_1D ca;
color onC = 0xff221166;
color offC = 0xffffff00;

void setup(){
  size(600, 600);
  background(255);
  ca = new CA_1D(600, 600, 12);
  ca.setOnColor(onC);
  ca.setOffColor(offC);

  //add multiple starting states
  int seedCount = 50;
  int[] cells = new int[seedCount];
  for (int i=0; i<seedCount; i++){
    cells[i] = int((ca.rows-1) * (ca.cols) + random(ca.cols));
  }
  ca.setInitState(cells);
}

void draw(){
  translate(ca.w/2, ca.h/2);
  ca.createGeneration();
}
```

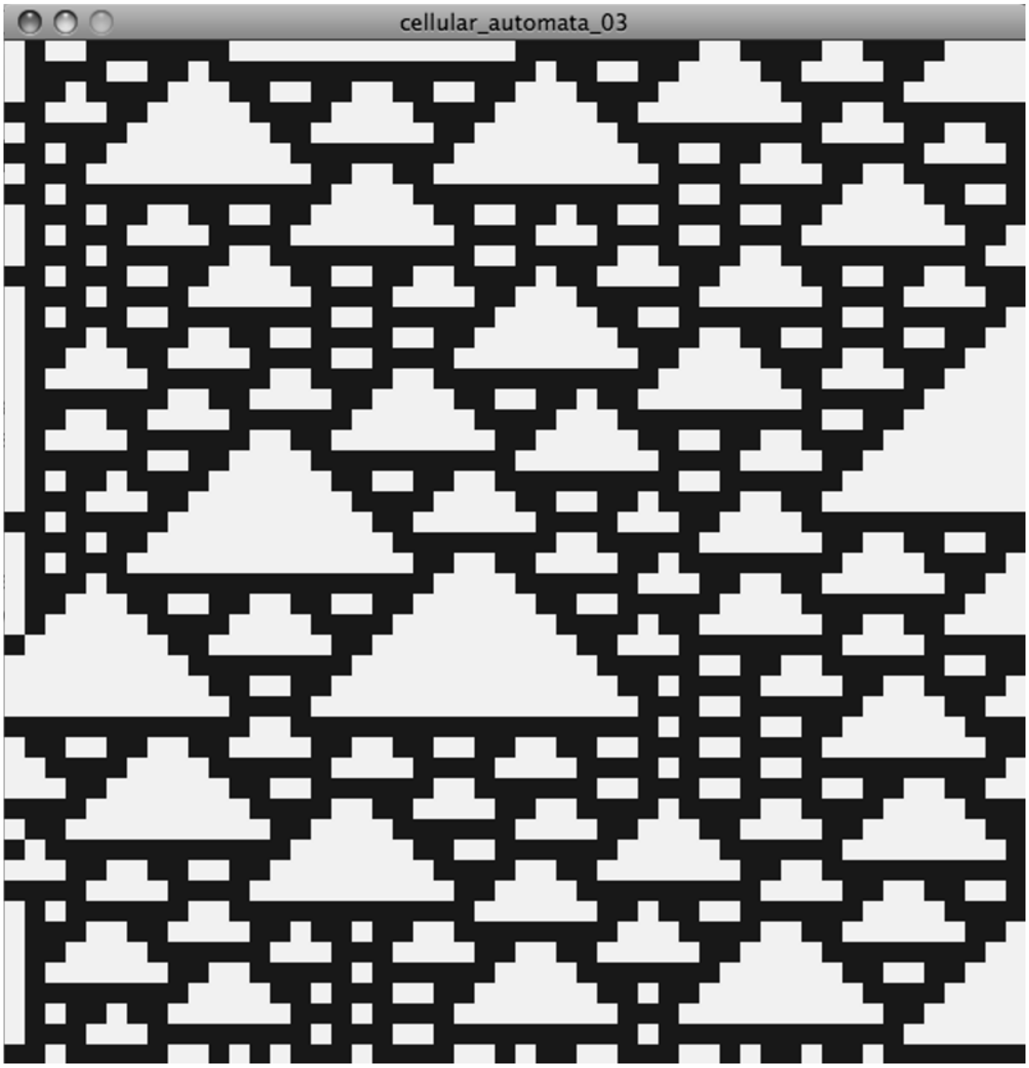

Figure 7-8. 1D Cellular Automata screen-shot, stage 3

The next variation, shown in Figures 7-9 and 7-10, demonstrates how changing the rules affects the output. The example creates a table of 16 CA, each with a different rule set. Replace your main tab code with the following to run the example:

```
/**
 * Cellular Automata Main Tab - 04
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

CA_1D[] cas = new CA_1D[16];
boolean[] rules = new boolean[8];
color onC = 0xff000000;
color offC = 0xff111111;

void setup(){
  size(800, 800);
  background(255);
  for (int i=0; i<cas.length; i++){
    cas[i] = new CA_1D(200, 200, 2);
    // calculate random rules
   for (int j=0; j<8; j++){
      rules[j] = boolean(round(random(1)));
    }
    cas[i].setRules(rules);
  }
}

void draw(){
  translate(cas[0].w/2, cas[0].h/2);
  int step = cas.length/4;
  for (int i=0; i<step; i++){
    for (int j=0; j<step; j++){
      pushMatrix();
      translate(cas[step*i + j].w*i, cas[step*i + j].h*j);
      cas[step*i + j].createGeneration();
      popMatrix();
    }
  }
}
```

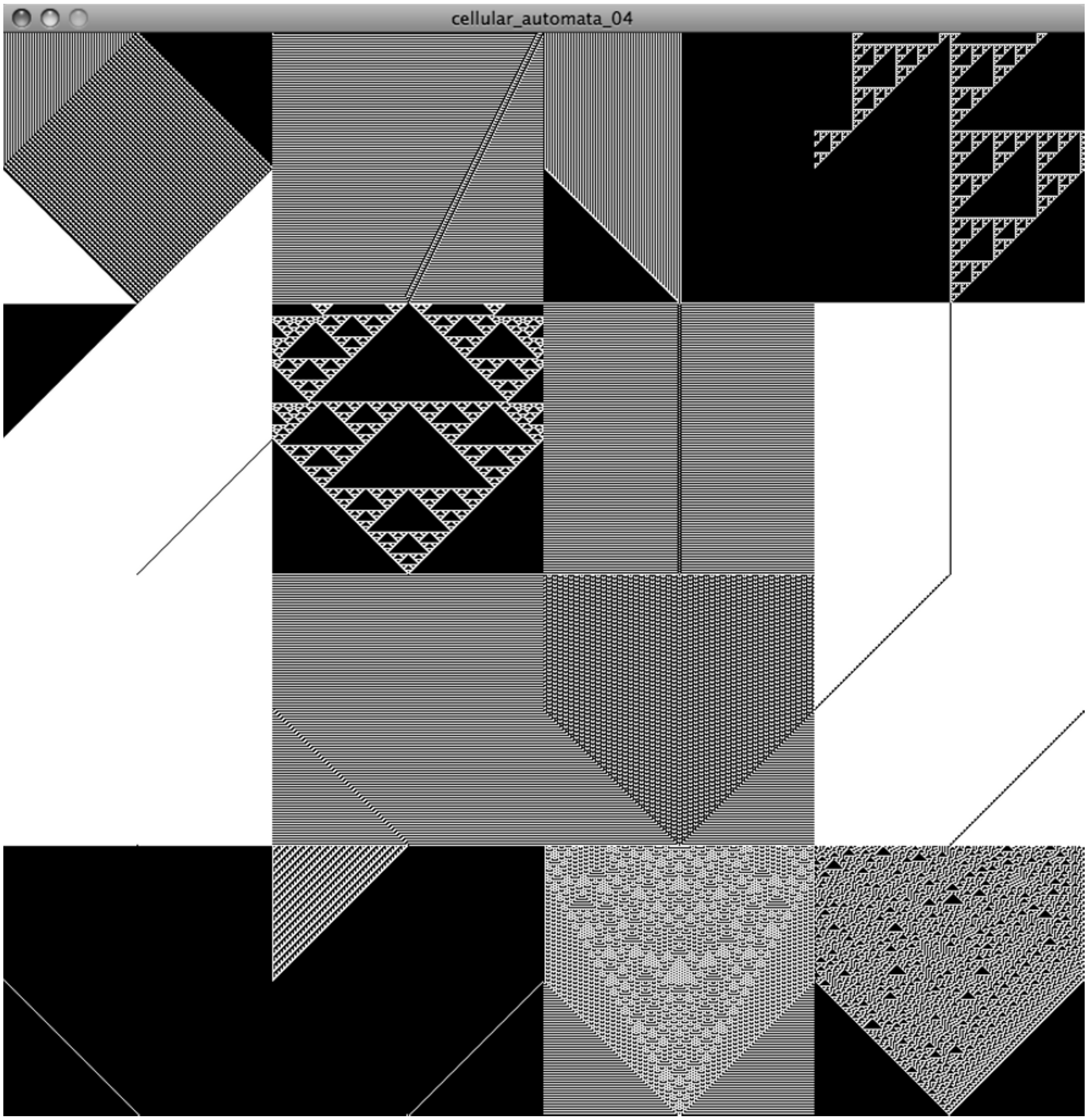

Figure 7-9. 1D Cellular Automata screen-shot, stage 4, screenshot 1

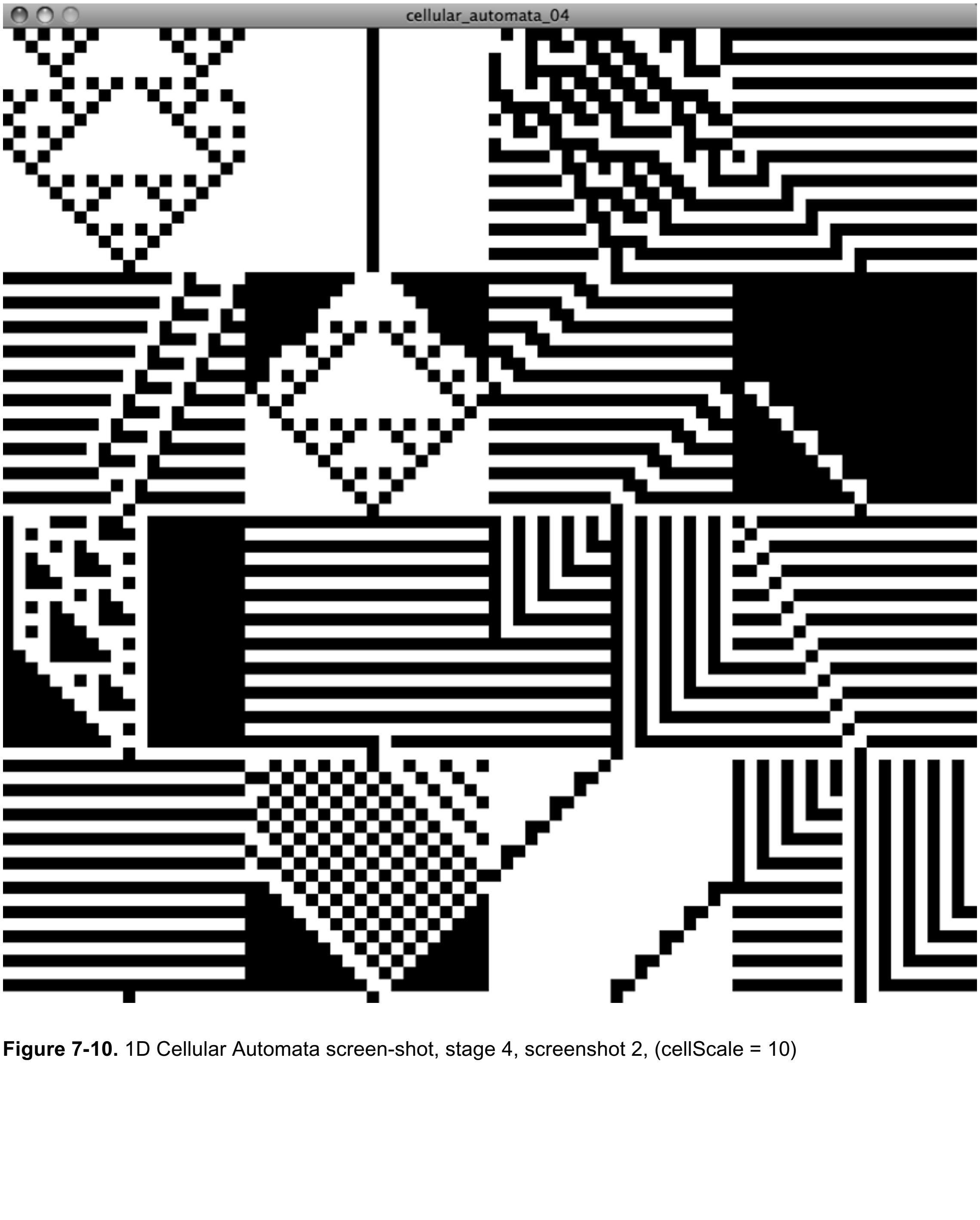

Figure 7-10. 1D Cellular Automata screen-shot, stage 4, screenshot 2, (cellScale = 10)

You may want to run this example a number of times to see the range of potential output. You can also be more selective in specifying the rules—not making them all random. I randomized the rules using the expression `rules[j] = boolean(round(random(1)))`.

> *Although the `boolean` type in Processing evaluates to the constants `true` or `false`, it is possible to cast a 1 and 0 to these respective constants; this also works the other way around (e.g., `int(true)` evaluates to 1).*

Finally, I include one more variation that puts all the aspects discussed thus far together and adds random rotation as well (see Figures 7-11 and 7-12). The new code is in **bold**.

```
/**
 * Cellular Automata Main Tab - 05
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

CA_1D[] cas = new CA_1D[16];
boolean[] rules = new boolean[8];
// for random rotation
float[] rots = new float[cas.length];

void setup(){
  size(800, 800);
  background(255);
  for (int i=0; i<cas.length; i++){
    cas[i] = new CA_1D(200, 200, round(random(1, 20)));
    // calculate random rules
    for (int j=0; j<8; j++){
      rules[j] = boolean(round(random(1)));
    }
    cas[i].setRules(rules);
    // calculate random color
    cas[i].setOnColor(color(random(255), random(255), random(255)));
    cas[i].setOffColor(color(random(255), random(255), random(255)));
    // calculate random rotation
    rots[i] = HALF_PI*round(random(1, 3));
  }
}

void draw(){
  translate(cas[0].w/2, cas[0].h/2);
  int step = cas.length/4;
  for (int i=0; i<step; i++){
    for (int j=0; j<step; j++){
      pushMatrix();
      translate(cas[step*i + j].w*i, cas[step*i + j].h*j);
      rotate(rots[i]);
      cas[step*i + j].createGeneration();
      popMatrix();
    }
  }
}
```

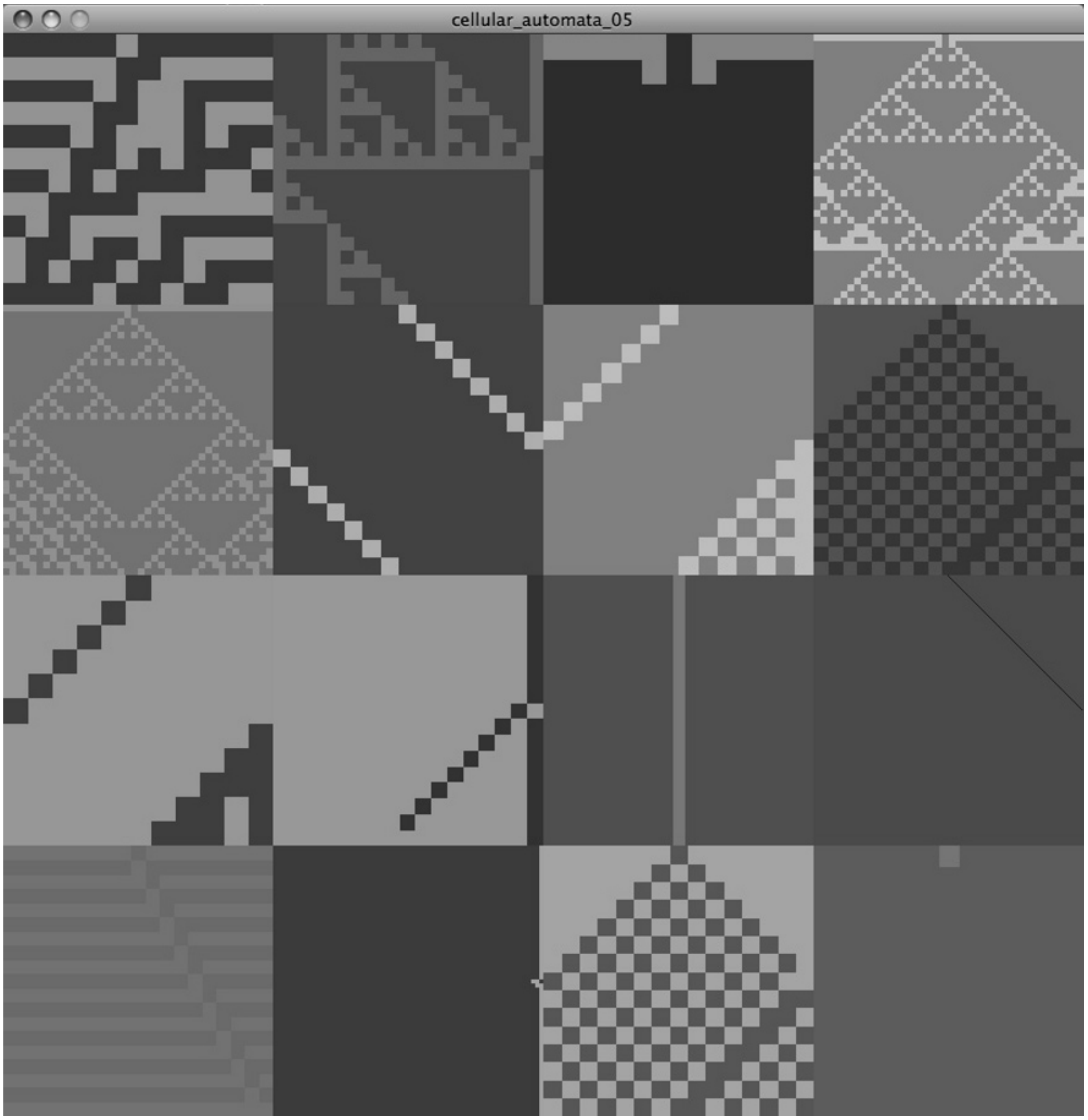

Figure 7-11. 1D Cellular Automata screen-shot, stage 5, screenshot 1

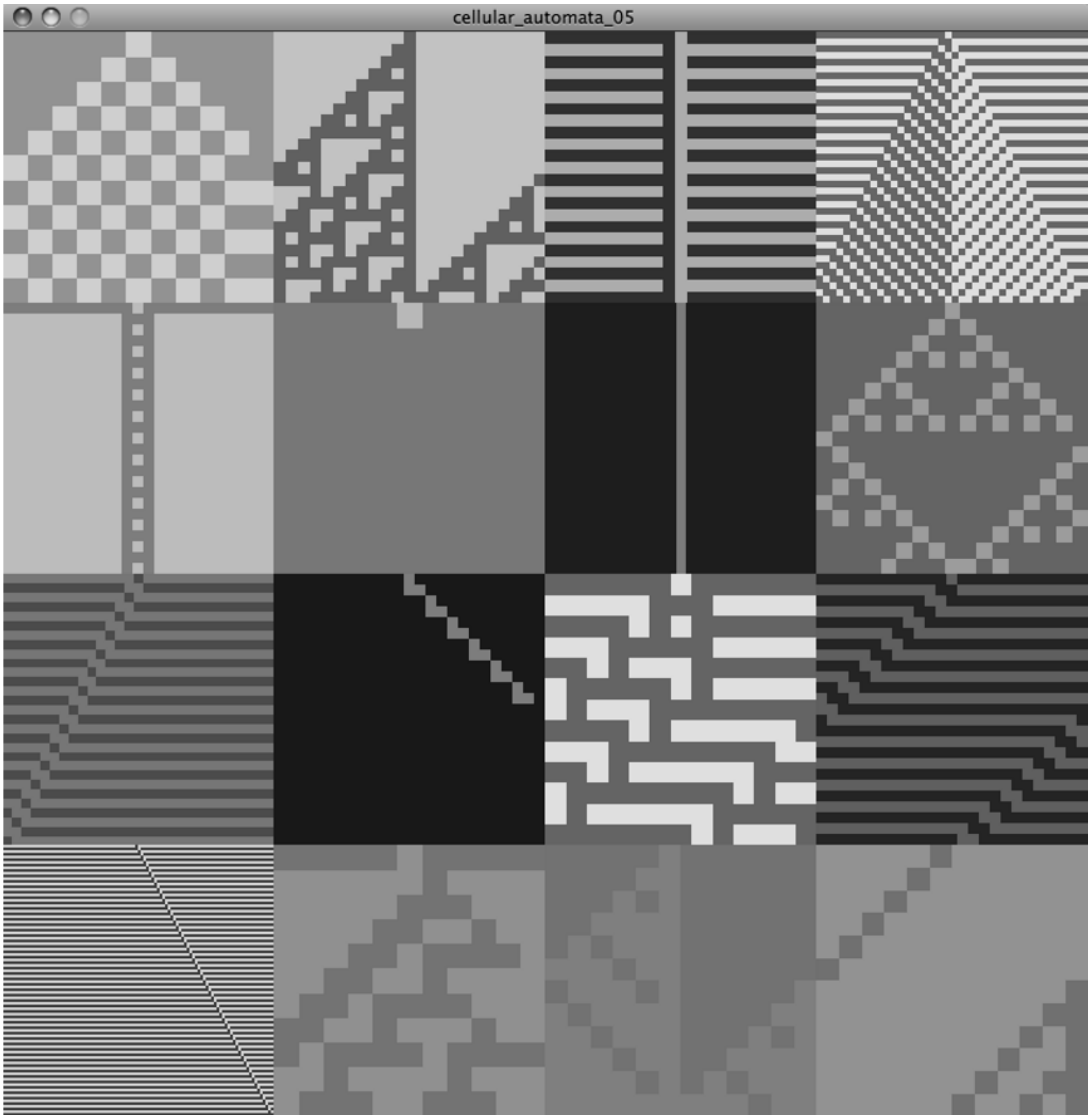

Figure 7-12. 1D Cellular Automata screen-shot, stage 5, screenshot 2

Using very simple rules, the `CA_1D` class allowed you to create somewhat complex patterns based on discrete on/off cell states. You saw by changing the rules you could create some interesting variations. It would, of course, also be possible to change the evaluation neighborhood and rules structure further to create other variations. However, one of the limiting factors in this approach is the use of discrete cell states, either being on or off. In the next example, I'll extend our discussion of one-dimensional CA using a continuous (versus discrete) method of cell genesis.

(Almost) Continuous CA

Rather than flipping cells on or off, the continuous CA will average the actual color values of cells in its neighborhood to determine the cell's color in the next generation. However, averaging alone is not enough to create something very interesting. Figure 7-13 illustrates a CA output using just the averaging process.

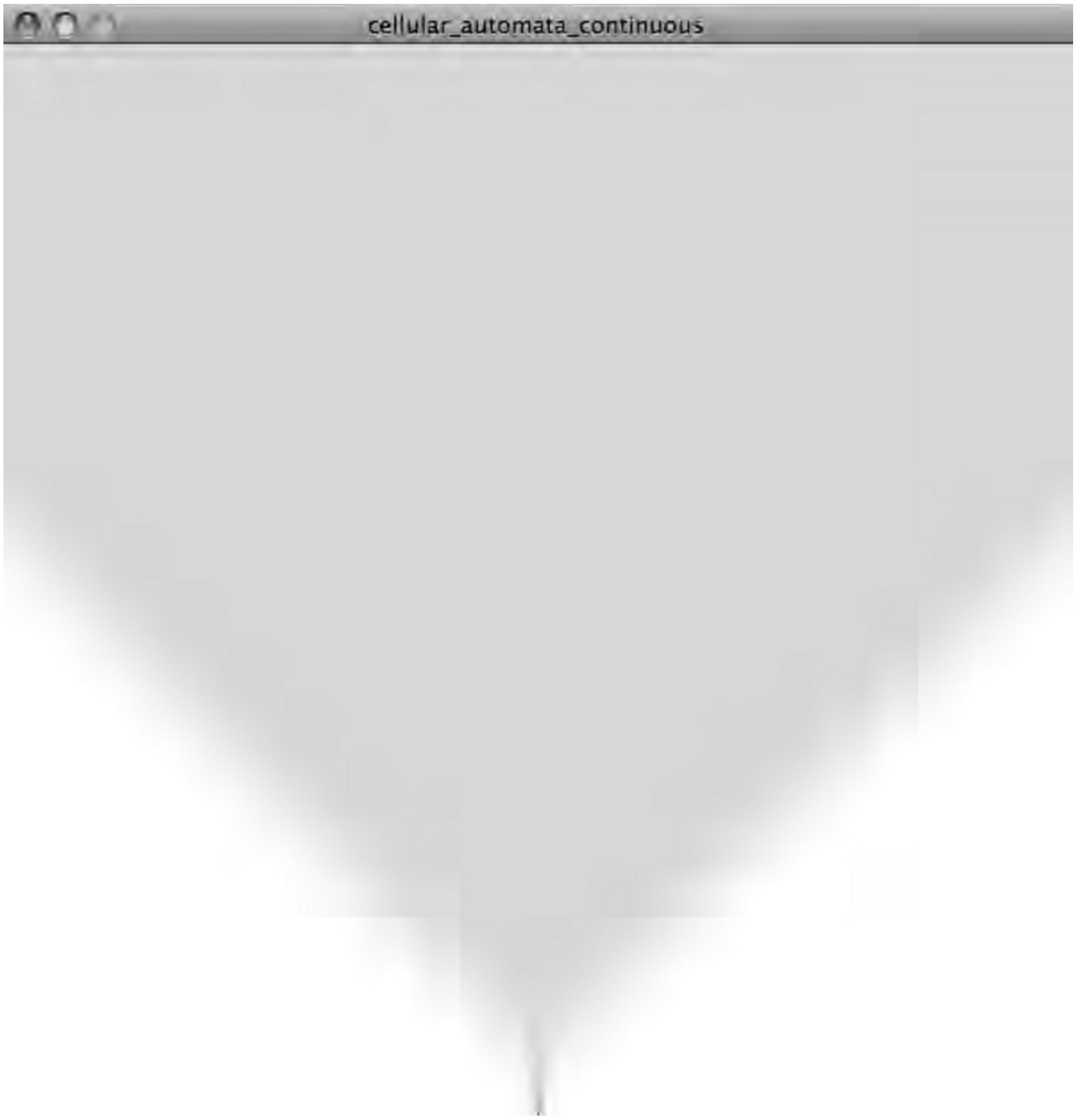

Figure 7-13. 1D Continuous Cellular Automata screenshot

To get more interesting results, we need to introduce some chaos into the process. This will involve a two-stage process. First, I'll introduce a constant value that will be added to each averaged color component (r, g, b). Second, I'll introduce a threshold, or maximum, that I'll use as a constraint for each component value. Here's an example of the process using pseudo code with cells *C1*, *C2*, and *C3* representing the neighborhood and *CN* as the new cell.

```
// pseudo code
c = constant
t = threshold
CN.r = (C1.r + C2.r + C3.r)/3 + c
CN.g = (C1.g + C2.g + C3.g)/3 + c
CN.b = (C1.b + C2.b + C3.b)/3 + c
if  (CN.r > t)  then  CN.r -= t
if  (CN.g > t)  then  CN.g -= t
if  (CN.b > t)  then  CN.b -= t
Create next generation using CN
```

In truth, my Continuous CA implementation will not be technically continuous. This has not only to do with my pathological inability to follow directions (even my own), but also the results I'll generate, which will be essentially indistinguishable from a "real" continuous CA. The term continuous perhaps evokes real numbers for you. (Yes, you probably are a geek if this is true.) In a continuous system, pretty much every value can be represented using the range 0.0 to 1.0. Most continuous CA examples that I've seen do in fact use this range. However, my system will be a little simpler and based on integers in the range of 0–255. Next is a Continuous CA class.

Using the existing sketch, add a new tab named "**CA_1DC**" and add the following class code to it:

```
/**
 * Cellular Automata
 * CA_1DC class
 * neighborhood:   | ? |
 *               * | * | *
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

class CA_1DC extends CA{
```

```
//instance properties
float[] consts = {
  23, 23, 23  };
float[] thresholds = {
  255, 255, 255  };

// default constructor
CA_1DC(){
  super();
  init();
}

// constructor
CA_1DC(int w, int h, int cellScale){
  super(w, h, cellScale);
  init();
}

void init(){
  int middleBottomCell = (rows-1)*cols + cols/2;
  setInitState(middleBottomCell, onC);
}

// set starting state (single pixel)
void setInitState(int id, color c){
  resetState();
  pixls[id] = c;
  paintInitState();
}

// set starting state (array of pixels)
void setInitState(int[] ids, color[] c){
  // reset();
  resetState();
  for (int i=0; i<ids.length; i++){
    pixls[ids[i]] = c[i];
  }
  paintInitState();
}

// set starting state
void setInitState(int row, int col, color c){
  // reset pixels
  resetState();
  pixls[row*(cols-1) + (col-1)] = c;
  paintInitState();
}

/* rules:
 1. average 3 neighboring colors, e.g. (c[j-1] + c[j] + c[j+1])/3
 2. add a constant, e.g. c + const
 3. if color components > 255 subtract 255 */
void createGeneration(){
  for (int i=0; i<rows-1; i++){
    for (int j=0; j<cols; j++){
      // use 1st colum as j+1, for end pixel in each column
      int firstCol = (j==0) ? cols-1 : j-1;
      int endCol = (j>0 && j<cols-1) ? j+1 : 0;
      int row = cols*(i+1);
      float r =  ((pixls[row + firstCol] >> 16 & 0xFF) + (pixls[row + j] >> ➥
        16 & 0xFF) + (pixls[row + endCol] >> 16 & 0xFF))/3 + consts[0];
      float g =  ((pixls[row + firstCol] >> 8 & 0xFF) + (pixls[row + j] >>  ➥
        8 & 0xFF) + (pixls[row + endCol] >> 8 & 0xFF))/3 + consts[1];
      float b =  ((pixls[row + firstCol] & 0xFF) + (pixls[row + j] & 0xFF) +  ↩
        (pixls[row + endCol] & 0xFF))/3 + consts[2];
      if (r>thresholds[0]){
        r-=thresholds[0];
      }

      if (g>thresholds[1]){
        g-=thresholds[1];
      }

      if (b>thresholds[2]){
        b-=thresholds[2];
      }
       nextPixls[(cols)*i + j] = int(r) << 16 | int(g) << 8 | int(b);
    }
  }
  // paint pixels on screen
  paint();
}

// pass custom rules
void setconsts(float[] consts) {
  this.consts = consts;
}

void setThresholds(float[] thresholds) {
  this.thresholds = thresholds;
}
}
```

Since the class extends `CA` it follows a structure very similar to `CA_1D`. Again, one of the nice things about a consistent framework is that it allows you to almost intuit how to work with a class. Thus, I'll assume you can make your way through most of this source code on you own. Where I think I can offer some clarification is in the `createGeneration()` method. You'll remember as a `CA` subclass, both `init()` and `createGeneration()` need to be implemented. Next is the snippet of code in the function again that handles the main CA calculation.

```
float r =  ((pixls[row + firstCol] >> 16 & 0xFF) + (pixls[row + j] >> ↩
    16 & 0xFF) + (pixls[row + endCol] >> 16 & 0xFF))/3 + consts[0];
float g =  ((pixls[row + firstCol] >> 8 & 0xFF) + (pixls[row + j] >>  ↩
    8 & 0xFF) + (pixls[row + endCol] >> 8 & 0xFF))/3 + consts[1];
float b =  ((pixls[row + firstCol] & 0xFF) + (pixls[row + j] & 0xFF) +  ↩
    (pixls[row + endCol] & 0xFF))/3 + consts[2];

   if (r>thresholds[0]){
r-=thresholds[0];
}

   if (g>thresholds[1]){
  g-=thresholds[1];
}

if (b>thresholds[2]){
    b-=thresholds[2];
}
nextPixls[(cols)*i + j] = int(r) << 16 | int(g) << 8 | int(b);
```

I chose to use bitwise operators to work with the components, as they are substantially faster than using Processing's `red()`, `green()`, `blue()`, and `color()` functions; to see this for yourself try substituting Processing's color component functions in the expressions. For example, the red expression would look like this:

```
float r =  (red(pixls[row + firstCol]) + red(pixls[row + j]) + red(pixls[row + endCol]))/3 +
consts[0];
```

I covered bitwise operations in Chapter 2 and more extensively in *Processing Creative Coding and Computational Art*, *Appendix B*.

To allow for more variation in the CA I created three constants, as well as three threshold values. Again, the constants are simply added to each averaged color component, and then the new component value is reduced to the amount greater than the threshold (or if the value is less than the threshold it remains unchanged). Finally, the color is put back together. To try out the new `CA_1DC` class, enter the following code in the main tab:

```
/**
 * Continuous Cellular Automata Main Tab - 01
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

// global variables
CA_1DC cca;
void setup(){
  size(600, 600);
  cca = new CA_1DC(600, 600, 1);
}

void draw(){
  translate(cca.w/2, cca.h/2);
  cca.createGeneration();
}
```

This first example, shown in Figure 7-14, is grayscale as the three constants are all the same values, and the `onC` and `offC` colors are black and white by default. Notice, though, the interesting pattern that's generated.

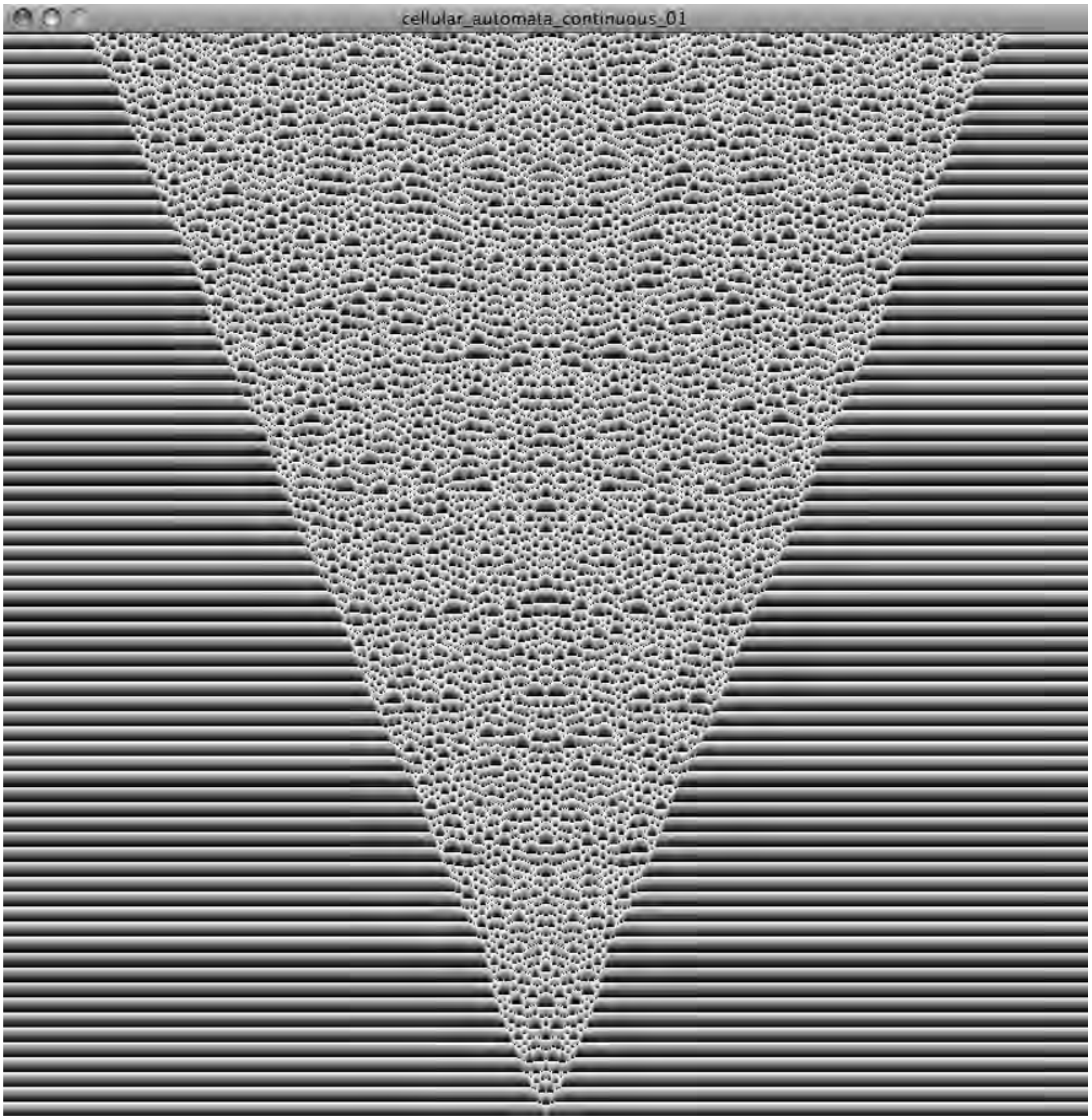

Figure 7-14. 1D Continuous Cellular Automata screen-shot, stage 1

In the next example, shown in Figures 7-15, 7-16, and 7-17, I'll create a table of CA varying the constants and thresholds in each CA, but keeping the three values in each array the same. Here's the code:

```
/**
 * Continuous Cellular Automata Main Tab - 02
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

// global variables
int rows = 4, cols = 4;
int rowSpan, colSpan;
int cellScale = 1;
float threshMin = 128, threshMax = 255;
float constMin = 2, constMax = 127;
// for random seed placement
int seedCount = 2;

// declare arrays
CA_1DC[] cacs;
int[] seeds;
color[] clrs;

void setup(){
  size(800, 800);
  initialize();
}

void initialize(){
  this.rows = rows;
  this.cols = cols;
  rowSpan = height/rows;
  colSpan = width/cols;

  cacs = new CA_1DC[rows*cols];
  seeds = new int[seedCount];
  clrs = new color[seedCount];
  for (int i=0; i<cacs.length; i++){
    cacs[i] = new CA_1DC(colSpan, rowSpan, cellScale);

    for (int j=0; j<seedCount; j++){
      seeds[j] = int((cacs[i].rows-1)*(cacs[i].cols) + int(random(cacs[i].cols)));
      clrs[j] = color(random(255), random(255), random(255));
    }
    cacs[i].setInitState(seeds, clrs);

    float t = random(threshMin, threshMax);
    cacs[i].setThresholds(new float[] { t, t, t });
    float c = random(constMin, constMax);
    cacs[i].setconsts(new float[] { c, c, c });
  }
}

void draw(){
  for (int i=0; i<rows; i++){
    for (int j=0; j<cols; j++){
      pushMatrix();
      // simplify stuff
      int index = cols*i + j;
      float x = cacs[index].w*j;
      float y = cacs[index].h*i;
      float w = cacs[index].w;
      float h = cacs[index].h;
      // move top left corner to 0,0
      translate(w/2, h/2);
      // move into position in table
      translate(x, y);
      // do CA magic
      cacs[index].createGeneration();
      popMatrix();
    }
  }
}
```

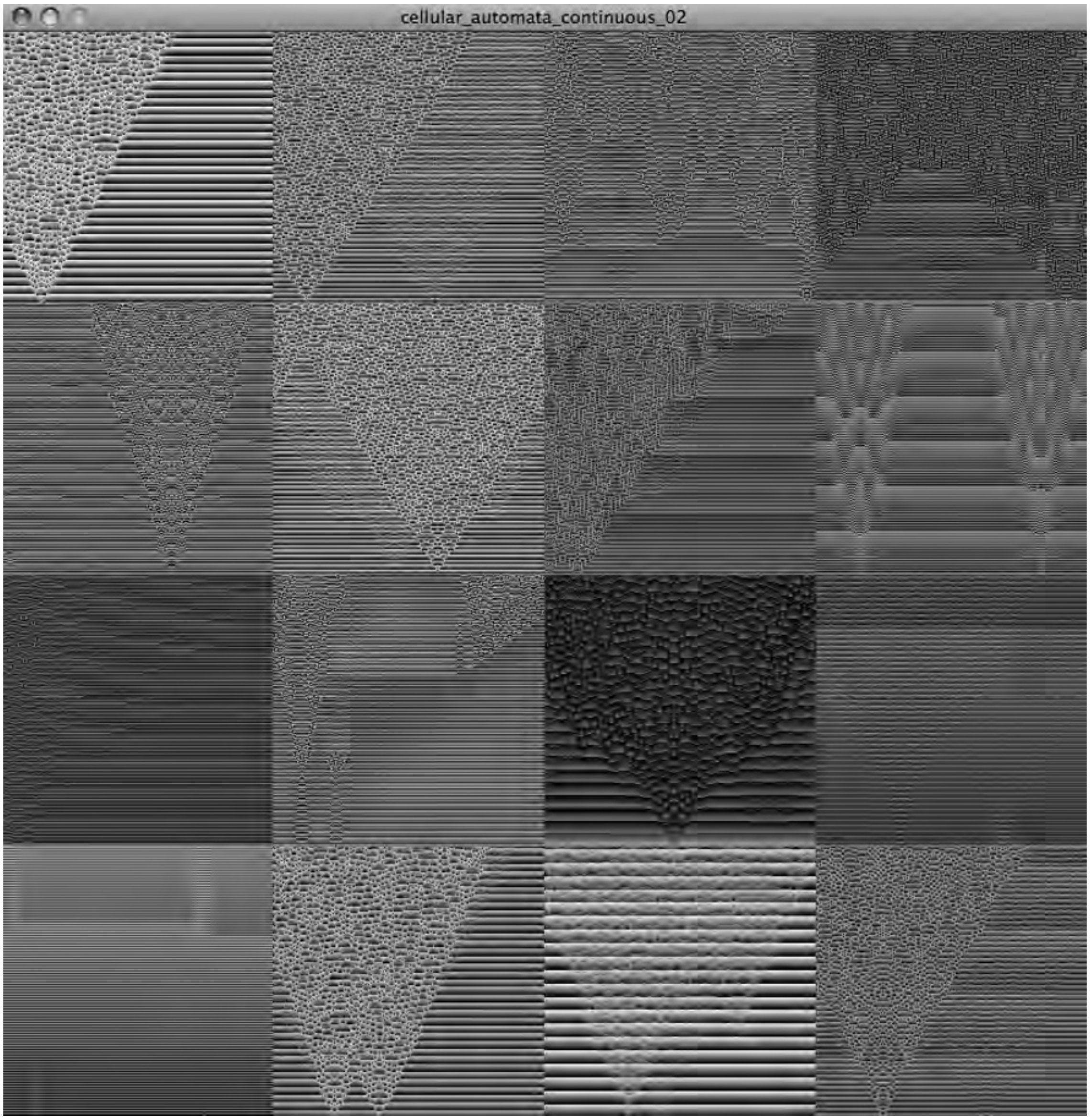

Figure 7-15. 1D Continuous Cellular Automata screen-shot, stage 2

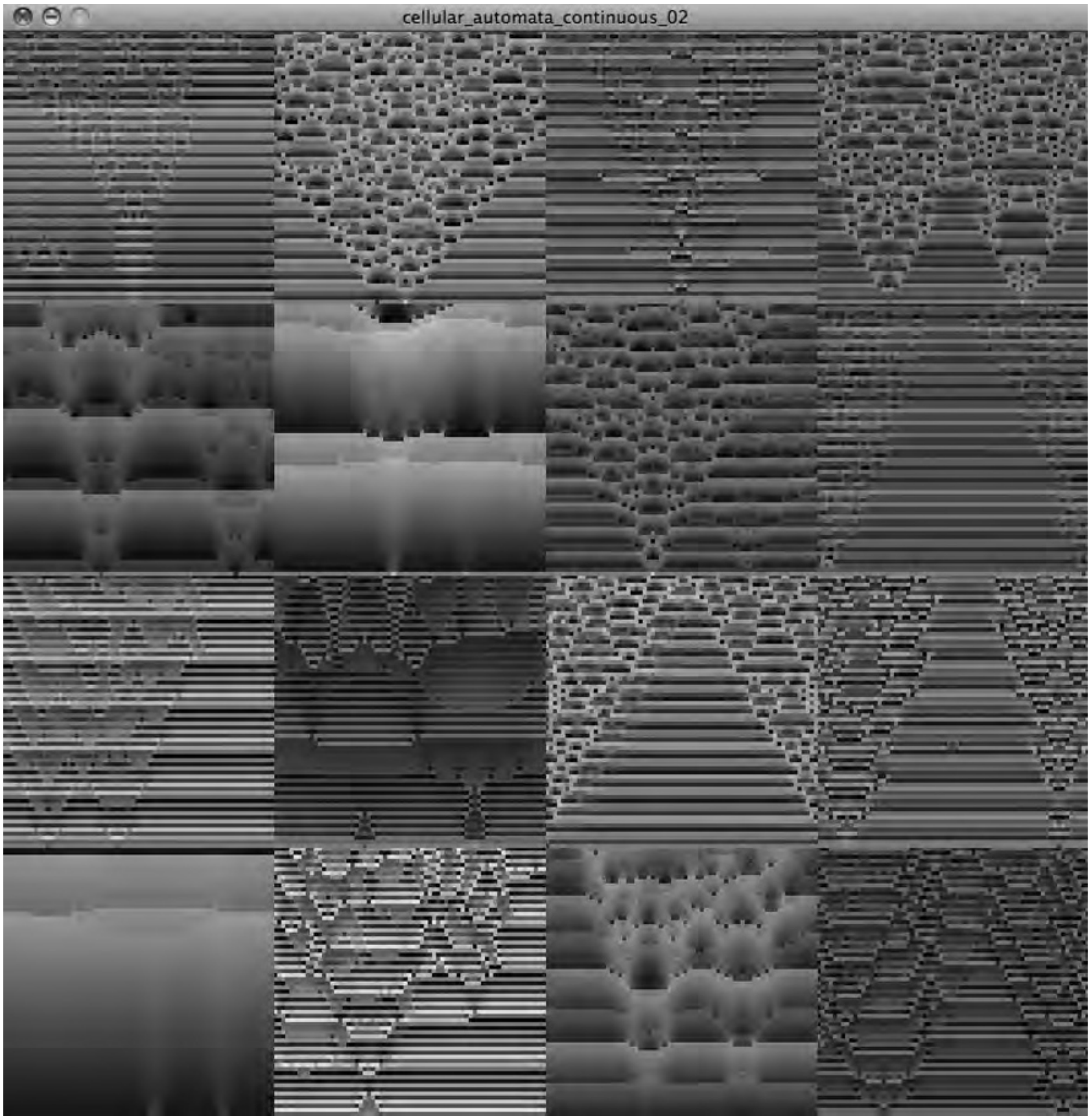

Figure 7-16. 1D Continuous Cellular Automata screen-shot, stage 2 (cellScale = 3)

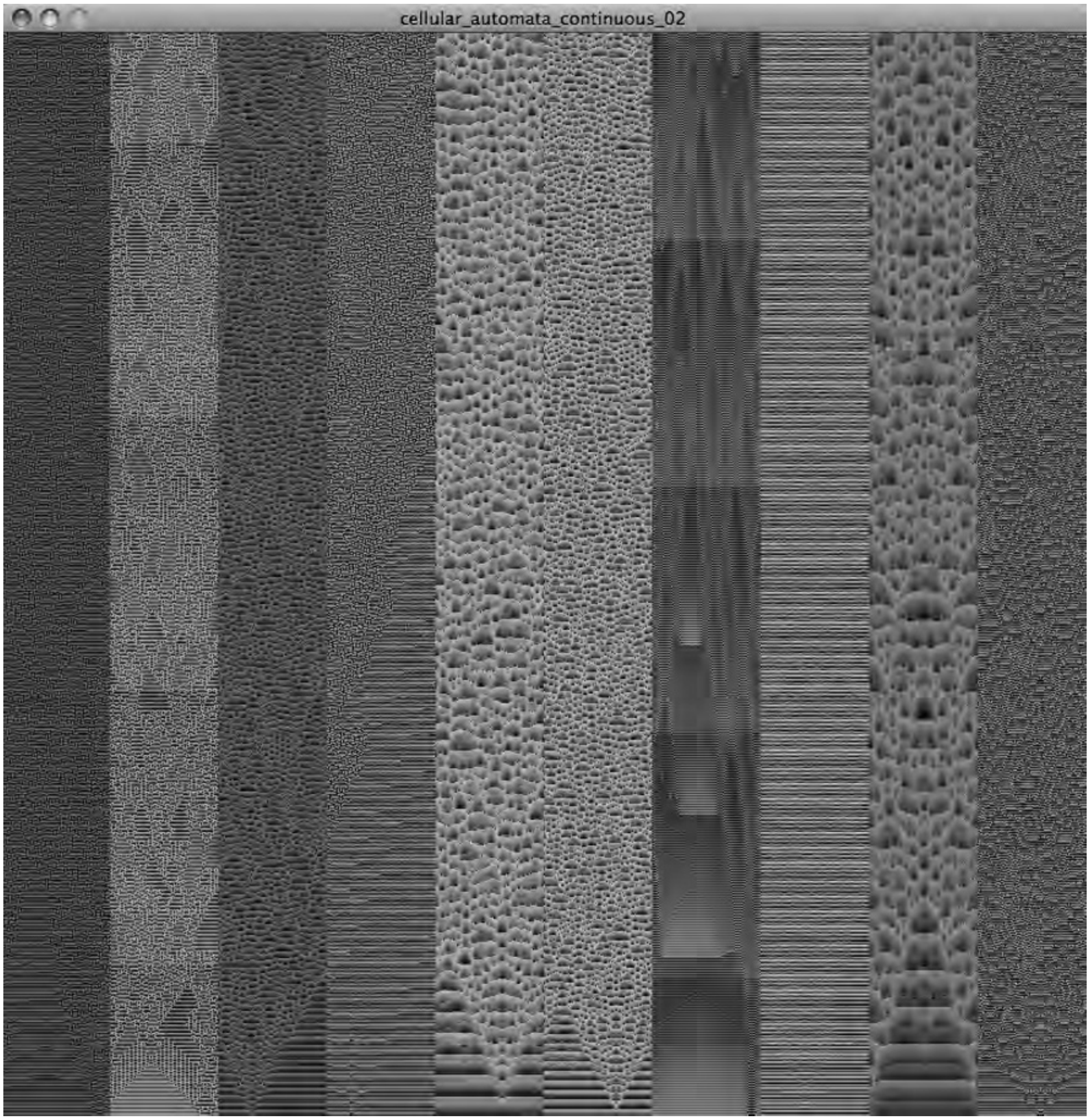

Figure 7-17. 1D Continuous Cellular Automata screen-shot, stage 2 (1 row, 10 columns)

There is nothing really new in this code. I suggest messing around with the values a bit to see what's possible. I'm sure you'll agree that the continuous approach yields much more interesting images than the straight 1D discrete CA discussed earlier. You might also try creating your own CA rules, maybe using trig functions in the continuous calculations, for example; there is a lot of untapped aesthetic potential here. Before I move on to 2D CA, I want to include one more interactive continuous CA example.

One of the benefits of being able to see a table of smaller CA is the increased rate of exploration. However, the small images make it difficult to fully see all the detail (or perhaps output an image for a tee-shirt). It would be nice if you could enlarge any image without sacrificing detail and/or resolution. In the last continuous CA example, I'll code interactivity to allow us to do this. I'll also add some additional functionality to create even more variation, shown in Figures 7-18, 7-19, 7-20, and 7-21. Additions/changes to the existing code are **bold**.

```
/**
 * Continuous Cellular Automata Main Tab - 03
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

// global variables
int rows = 8, cols = 8;
int rowSpan, colSpan;
int cellScale = 1;
float threshMin = 128, threshMax = 255;
float constMin = 2, constMax = 127;
// for random seed placement
int seedCount = 1;

// declare arrays
CA_1DC[] cacs;
float[][] thresholds;
float[][] consts;
int[][] seeds;
color[][] clrs;

// for interactivity
int overID = 0;
boolean iSFirstClick = true;
```

```
void setup(){
  size(800, 800);
  initialize();
}

void initialize(){
  this.rows = rows;
  this.cols = cols;
  rowSpan = height/rows;
  colSpan = width/cols;

  cacs = new CA_1DC[rows*cols];
  thresholds = new float[cacs.length][3];
  consts = new float[cacs.length][3];
  seeds = new int[cacs.length][seedCount];
  clrs = new color[cacs.length][seedCount];

  for (int i=0; i<cacs.length; i++){
    cacs[i] = new CA_1DC(colSpan, rowSpan, cellScale);

    for (int j=0; j<seedCount; j++){
      seeds[i][j] = int((cacs[i].rows-1)*(cacs[i].cols) + int(random(cacs[i].cols)));
      clrs[i][j] = color(random(255), random(255), random(255));
    }
    cacs[i].setInitState(seeds[i], clrs[i]);

    for (int j=0; j<thresholds[0].length; j++){
      thresholds[i][j] = random(threshMin, threshMax);
      consts[i][j] = random(constMin, constMax);
    }
    cacs[i].setThresholds(thresholds[i]);
    cacs[i].setconsts(consts[i]);
  }
}

// draw selected CA full screen with original values
void calcCA(int i){
  // factor to scale the initial pixel state
  float widthFctr = width/cacs[i].w;
  // get original rows and cols value before updated
  float oldRows = cacs[i].rows;
  float oldCols = cacs[i].cols;

  // new output will fill the sketch window
  rows = cols = 1;
  int scl = cacs[0].cellScale;
  // reinitialize cacs
  cacs = new CA_1DC[1];
  cacs[0] = new CA_1DC(width, height, scl);
  // updates initial seeds, if originally set
  for (int j=0; j<seedCount; j++){
    if (seeds[i][j] !=0){
      seeds[i][j] = int((cacs[0].rows-1)*cacs[0].cols + (seeds[i][j]- ↩
        (oldRows-1)*oldCols)*widthFctr);
    } else {
      // if default centered seed was used
      seeds[i][j] = (cacs[0].rows-1)*cacs[0].cols + cacs[0].cols/2;
    }
  }
  // set with original values
  cacs[0].setInitState(seeds[i], clrs[i]);
  cacs[0].setThresholds(thresholds[i]);
  cacs[0].setconsts(consts[i]);
}
```

```
void draw(){
  for (int i=0; i<rows; i++){
    for (int j=0; j<cols; j++){
      pushMatrix();
      // simplify stuff
      int index = cols*i + j;
      float x = cacs[index].w*j;
      float y = cacs[index].h*i;
      float w = cacs[index].w;
      float h = cacs[index].h;
      // move top left corner to 0,0
      translate(w/2, h/2);
      // move into position in table
      translate(x, y);
      // do CA magic
      cacs[index].createGeneration();
      popMatrix();

      // check which cell mouse is over
      if (mouseX > x && mouseX < x + w &&
        mouseY > y && mouseY < y + h){
        overID = index;
      }
    }
  }
}

// if the first time clicking on sketch, select CA to enlarge
void mouseClicked(){
  if (iSFirstClick){
    calcCA(overID);
    iSFirstClick = false;
  }
}
```

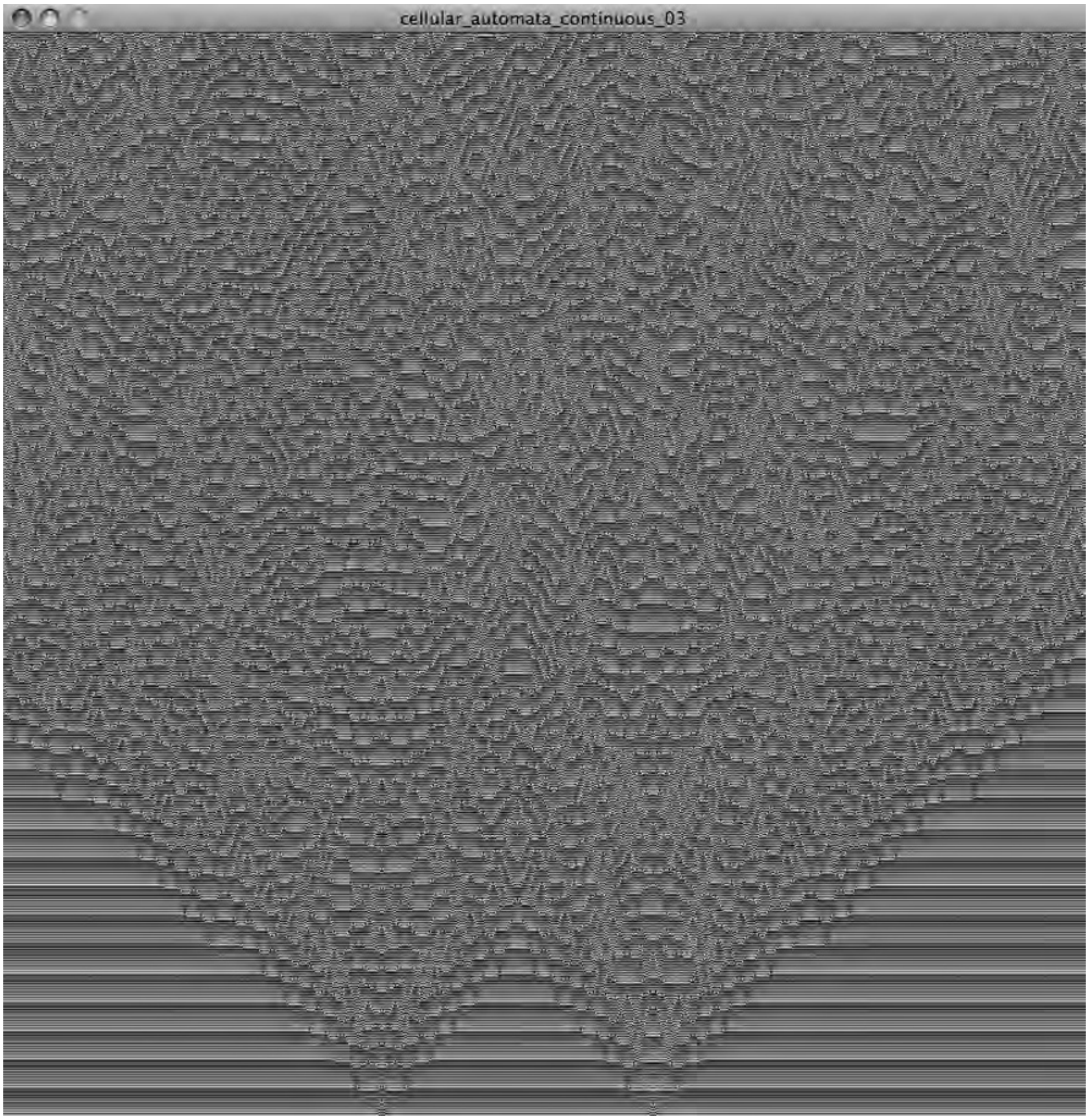

Figure 7-18. 1D Continuous Cellular Automata screen-shot, stage 3

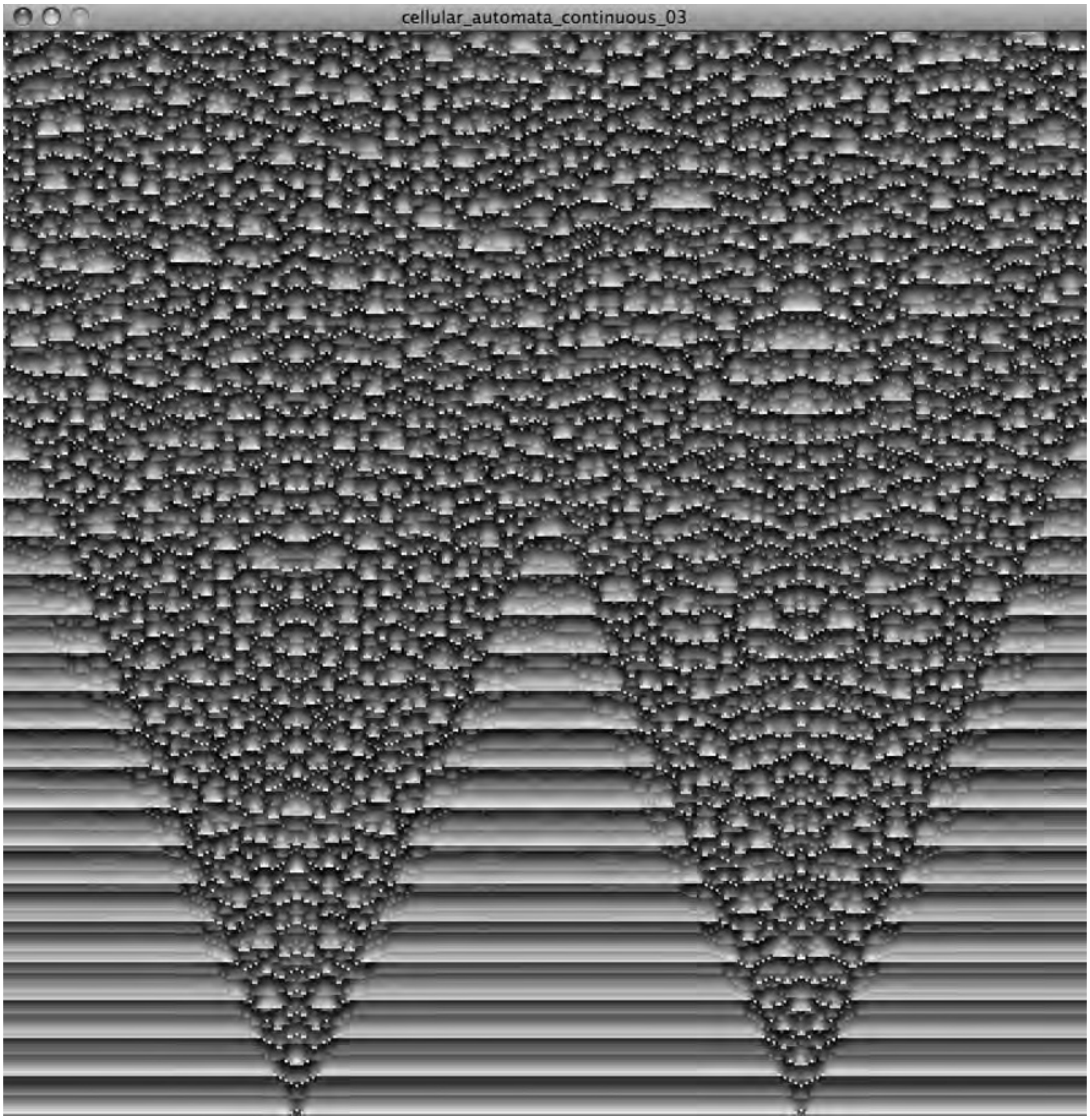

Figure 7-19. 1D Continuous Cellular Automata screen-shot, stage 3

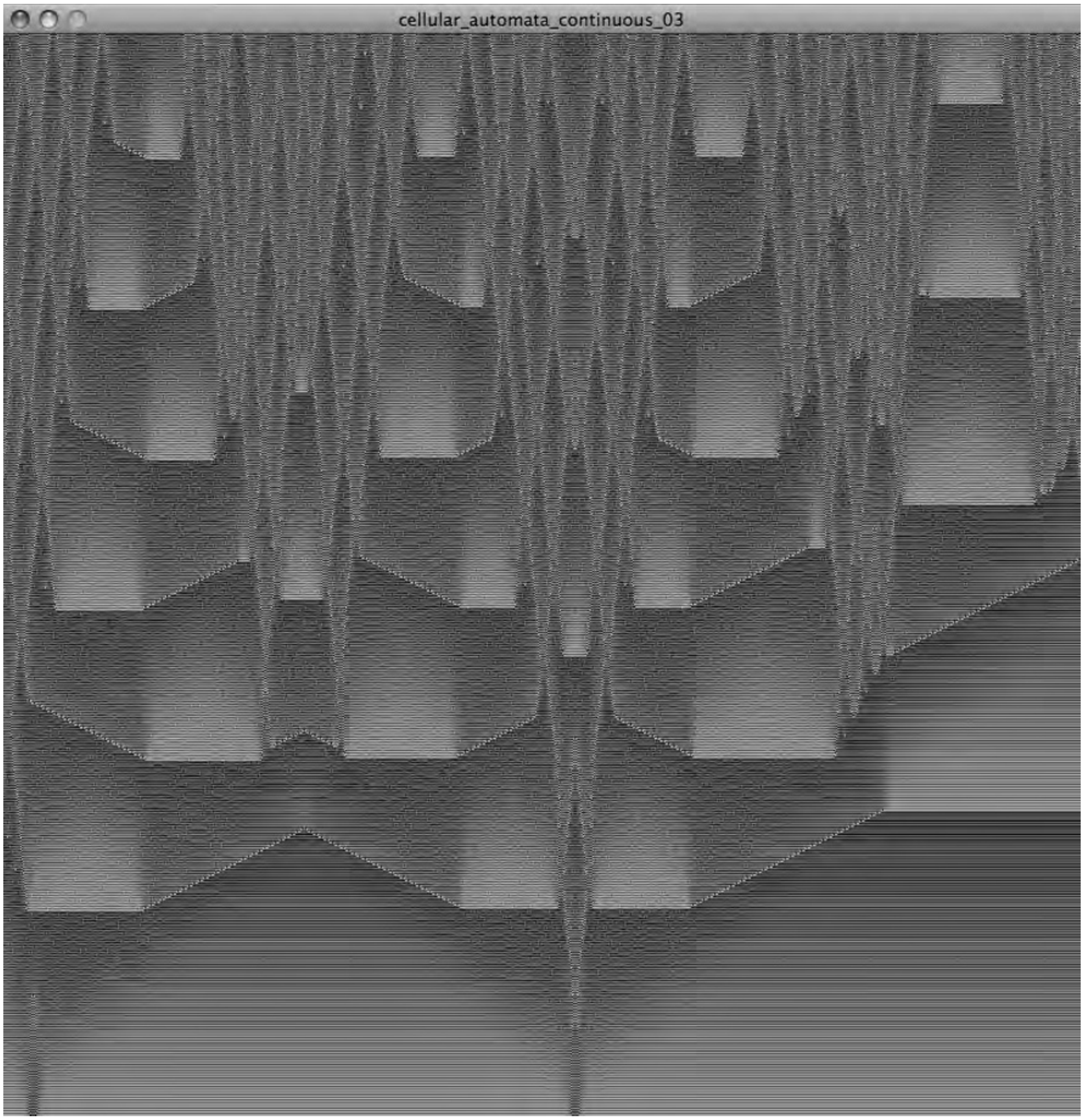

Figure 7-20. 1D Continuous Cellular Automata screen-shot, stage 3

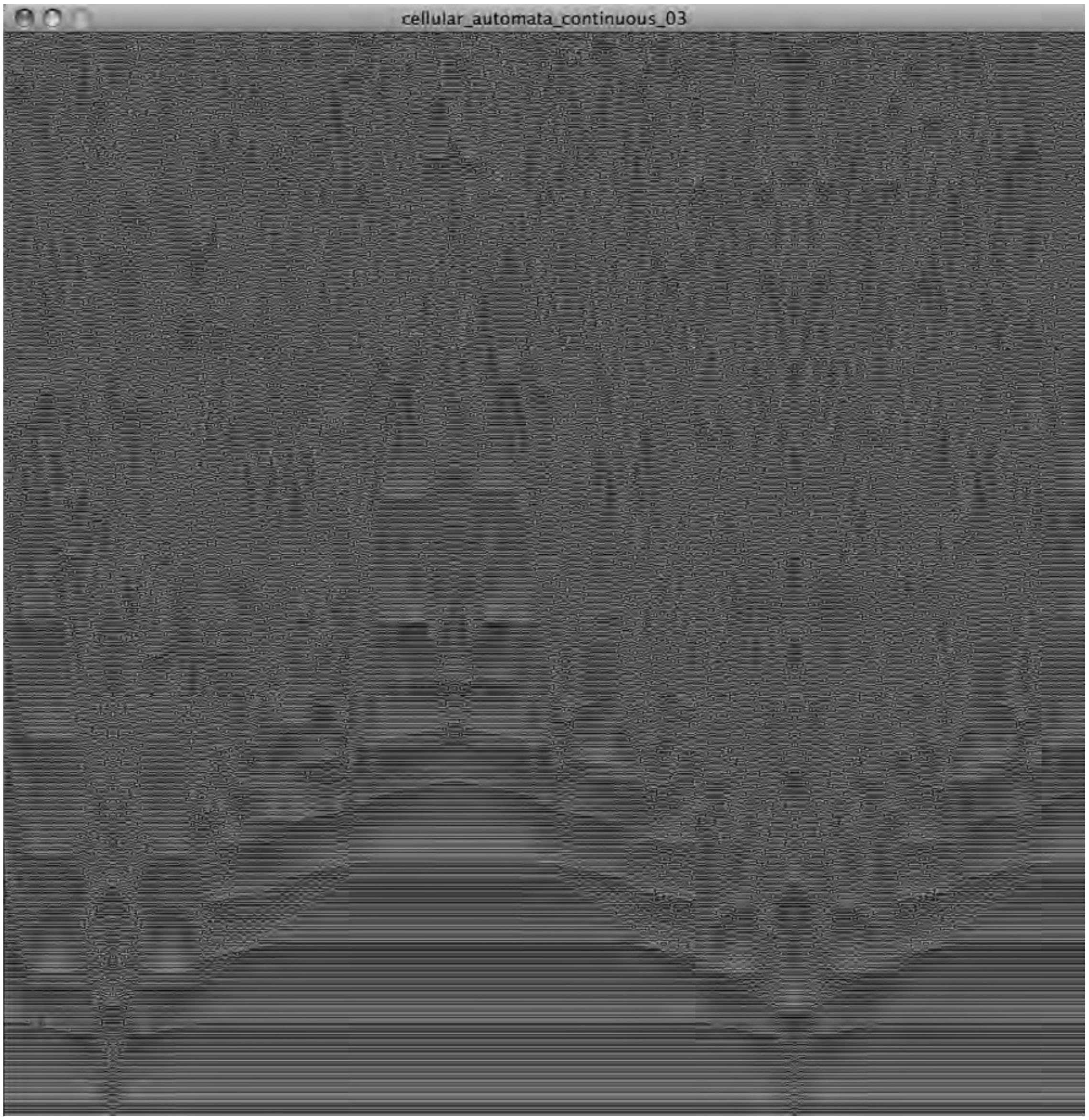

Figure 7-21. 1D Continuous Cellular Automata screen-shot, stage 3

Looking at the code in the example, you'll notice I added a bunch of 2D arrays. The main coding challenges I had to deal with were retaining all the critical data for each thumbnail (i.e., colors, constants, and threshold values) and coming up with an event detection method that would know which thumbnail had been clicked on.

I found it easiest to create the new calcCA() method to isolate the redrawing of the large image from the initial drawing of the thumbnails. The calcCA() method is called from within the mouseClicked() function, which is one of Processing's built-in event functions. Each time calcCA(overID) is called an index value is passed to the function representing the thumbnail clicked on. Within draw() notice the home-brewed detection block:

```
if (mouseX > x && mouseX < x + w &&
  mouseY > y && mouseY < y + h){
  overID = index;
}
```

Admittedly this approach is neither elegant nor efficient, as the detection check happens every draw cycle, and I needed to create the overID global variable. However, a more OOP'ish solution would have required doing a bit of tinkering with the base CA class, which, at this point in the chapter, wasn't going to happen. Of course, adding detection to the CA class and rewiring this example (even adding interactivity to all the previous examples) would be an excellent thing to try on your own. One other point that might be confusing is the expression I used to scale the seeds in the original thumbnail to the larger image, the line

```
seeds[i][j] = int((cacs[0].rows-1)*cacs[0].cols + (seeds[i][j]-↩
  (oldRows-1)*oldCols)*widthFctr);
```

This is a pretty ornery-looking line of code. Since the seeds are all on the bottom row of cells (which you'll remember are stored in a 1D array), it was simplest to only deal with the last row in calculating the scaling. Thus, I simply added the scaled index positions to the rest of the array. To get a clearer sense of why I handled it this way, try scaling a table and seeing how specific index values shift within the table; it's messy!

One-dimensional CA offered a glimpse into how simple rules can lead to remarkable complexity. By adding an additional dimension and generating two-dimensional CA, we can blow open the doors of this fascinating potential. That said, 2D CA is a pretty large area of research (and this has already been a long chapter), so I'll just introduce the topic here and provide examples that both reveal interesting aspects of this research area and also create the beginnings of a 2D CA framework for studying them further.

2D CA

Going from 1D to 2D CA is not very difficult. However, as I mentioned earlier, it opens up lots of new possibilities; from a coding standpoint it's simply a matter of incorporating a second axis in the rules analysis. You can also create 3D CA by adding a third axis, which I won't be covering here, but you can learn more about here: http://risais.home.comcast.net/~risais/3dca/3dca.htm (November, 15, 2009 14:08). The examples to follow will be based on the most famous 2D CA, *Game of Life ("Life")*, developed by John Conway in 1970.

Conway developed Life based on the earlier work of John von Neumann, one of the originators of CA mentioned at the beginning of the chapter. What is so interesting about Conway's Life CA is the range of output it's capable of producing, In fact, Life "theoretically" has the capacity to function as a computer, or more precisely the capacity to calculate any algorithm. You can read more theoretical information about Life at `http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life` (August, 6, 2009 11:38 am).

In spite of Life's computational power, it is pretty simple to code, which is perhaps the most fascinating aspect of this whole area of research—from simple steps can emerge incredible complexity. As previously mentioned, implementing Life will involve two axes (*x* and *y*), and each cell's neighborhood will be defined by its eight surrounding cells (see Figure 7-22).

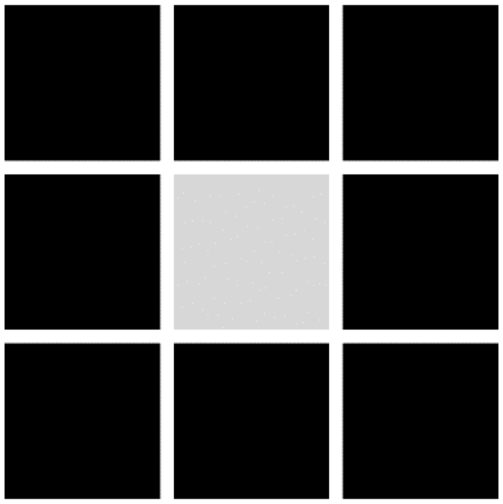

Figure 7-22. 2D CA Game of Life neighborhood

The rules of Life are as follows (taken directly from the Wikipedia page previously referenced, `http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`):

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with more than three live neighbors dies, as if by overcrowding.
3. Any live cell with two or three live neighbors lives on to the next generation.
4. Any dead cell with exactly three live neighbors becomes a live cell.

The basic coding structure we'll follow will be very similar to what we've done throughout the chapter. In fact, we'll use our current CA framework. To get started, create a new tab in the existing sketch used for the continuous CA example, or you can use any sketch that includes the `Shape`, `Cell`, and `CA` classes. Name the new tab "**`CA_2D`**" and enter the following code into the tab:

```
/**
 * Cellular Automata
 * CA_2D class
 * neighborhood: * | * | *
 *               * | ? | *
 *               * | * | *
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

class CA_2D extends CA{
  // instance properties

  // default constructor
  CA_2D(){
    super();
    init();
  }

  // constructor
  CA_2D(int w, int h, int cellScale){
    super(w, h, cellScale);
    init();
  }

  // REQUIRED implementation - initializes stuff
  void init(){
    // set default starting state
    /* R-pentomino pattern
    **
   **
    *
    */
    int[] initState = {
                          ((rows)/2-1)*(cols) + (cols-1)/2+1,
                          ((rows)/2-1)*(cols) + (cols-1)/2,
                          (rows)/2*(cols) + (cols-1)/2,
                          (rows)/2*(cols) + (cols-1)/2-1,
                          ((rows)/2+1)*(cols) + (cols-1)/2
                        };
    setInitState(initState);
  }

  // set starting state (array of pixels)
  void setInitState(int[] ids){
    resetState();
    for (int i=0; i<ids.length; i++){
      pixls[ids[i]] = onC;
    }
    paintInitState();
  }

// set starting state (single pixel)
  void setInitState(int row, int col){
    resetState();
    pixls[row*(cols-1) + (col-1)] = onC;
    paintInitState();
  }

  // REQUIRED implementation
  void createGeneration(){
    for (int i=0; i<rows; i++){
      for (int j=0; j<cols; j++){
        // 1st and last columns use each other as neighbors in calculation
        int firstCol = (j==0) ? cols-1 : j-1;
        int endCol = (j>0 && j<cols-1) ? j+1 : 0;
        // 1st and last rows use each other as neighbors in calculation
        int firstRow = (i==0) ? rows-1 : i-1;
        int endRow = (i>0 && i<rows-1) ? i+1 : 0;

        int sum = 0;
        if (pixls[cols*(firstRow) + firstCol] == onC){
          sum+=1;
        }
        if (pixls[cols*(firstRow) + j] == onC){
          sum+=1;
        }
        if (pixls[cols*(firstRow) + endCol] == onC){
          sum+=1;

        }
        if (pixls[cols*i + endCol] == onC){
          sum+=1;

        }
        if (pixls[cols*(endRow) + endCol] == onC){
          sum+=1;

        }
        if (pixls[cols*(endRow) + j] == onC){
          sum+=1;

        }
        if (pixls[cols*(endRow) + firstCol] == onC){
          sum+=1;
        }
        if (pixls[cols*i + firstCol] == onC){
          sum+=1;
        }

        if (pixls[cols*i + j] == onC){
          if (sum < 2 || sum > 3){
            nextPixls[cols*i + j] = offC;
          }
          // if sum is 2 or 3
          else {
            nextPixls[cols*i + j] = onC;
          }
        }
        // if pixel is offC
        else {
          if (sum == 3){
            nextPixls[cols*i + j] = onC;
          }
        }
      }
    }
    // paint pixels on screen
    paint();
  }

  void setOnColor(color onC){
    this.onC = onC;
  }

  void setOffColor(color offC){
    this.offC = offC;
  }

}
```

The class is very similar to the `CA_1D` class, although the `init()` and `createGeneration()` methods are implemented differently. The rules analysis in `createGeneration()` is a bit lengthier than in the 1D examples, but it should still be pretty self-explanatory as it follows the Life rules enumerated earlier. Before discussing the `init()` method, let's try out the new code. In the main tab enter the following and then run the sketch:

```
/**
 * Cellular Automata 2D _ main tab - 01
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

CA_2D ca2;

void setup(){
  size(600, 600);
  background(255);
  ca2 = new CA_2D(600, 600, 1);
}

void draw(){
  translate(ca2.w/2, ca2.h/2);
  ca2.createGeneration();
}
```

If the code ran okay, you should have seen a bunch of white pixels growing and moving about the sketch window. Figure 7-23 shows the sketch after 1000 frames. To be more specific, albeit cryptic, the sketch screenshot shows the R-pentomino pattern after 1000 generations following Life's rules.

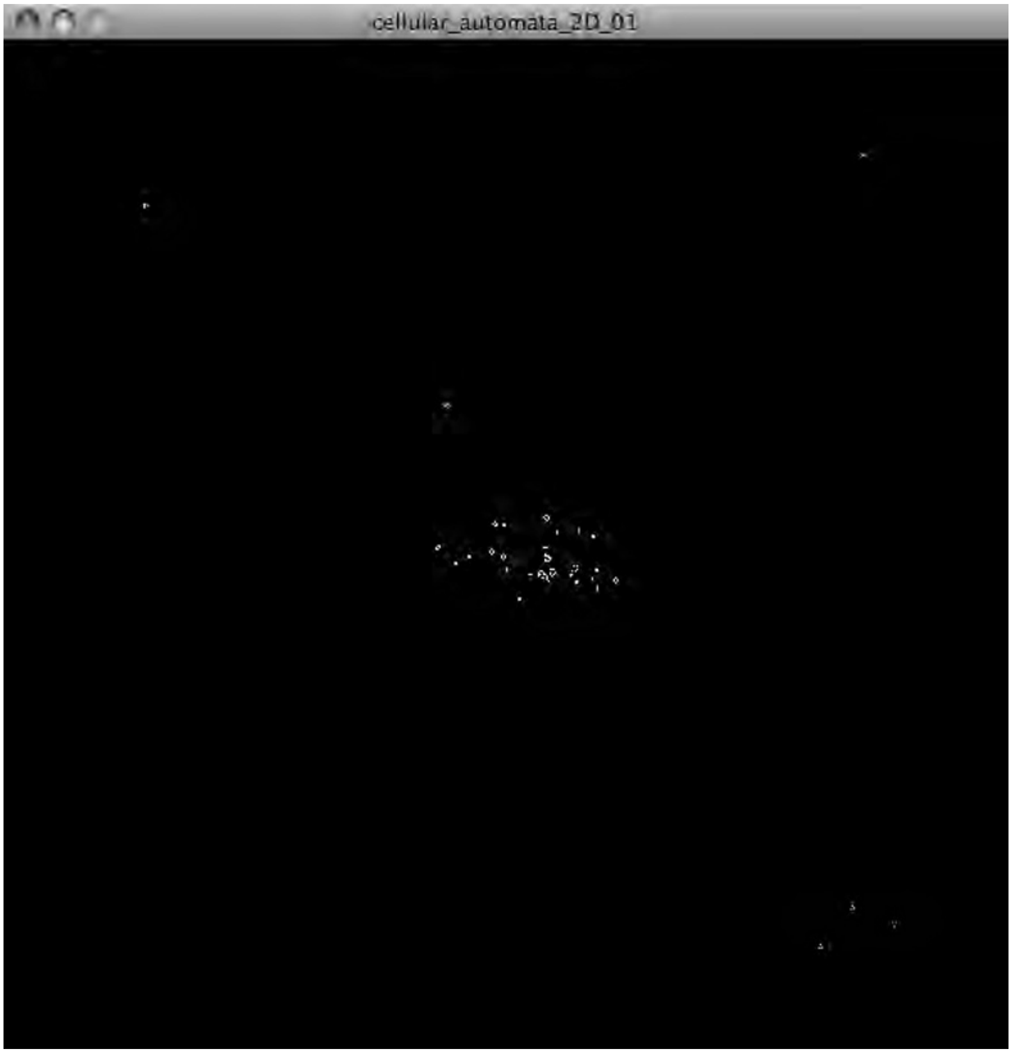

Figure 7-23. 2D CA Game of Life R-pentomino pattern after 1000 frames

Returning to the `CA_2D` code, here's the `init()` method again.

```
// REQUIRED implementation - initializes stuff
  void init(){
    // set default starting state
    /* R-pentomino pattern
    **
   **
```

```
    *
    */
    int[] initState = {
                          ((rows)/2-1)*(cols) + (cols-1)/2+1,
                          ((rows)/2-1)*(cols) + (cols-1)/2,
                          (rows)/2*(cols) + (cols-1)/2,
                          (rows)/2*(cols) + (cols-1)/2-1,
                          ((rows)/2+1)*(cols) + (cols-1)/2
                        };
    setInitState(initState);
  }
```

This method creates a default starting on/off pixels state, which we've also done in earlier examples. However, rather than beginning with a single pixel or random array of pixels, we're beginning with a very specific pixel pattern, in this case one named "R-pentomino" (also sometimes referred to as F-pentomino). To better see this pattern, we'll modify our example sketch, increasing the scale of the pixels and also adding an interactive element. Replace the code in the main tab with the following:

```
/**
 * Cellular Automata 2D _ main tab - 02
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

CA_2D ca2;

void setup(){
  size(600, 600);
  background(255);
  ca2 = new CA_2D(600, 600, 20);
  translate(ca2.w/2, ca2.h/2);
  ca2.paint();
}

void draw(){
}
```

```
void mousePressed(){
  translate(ca2.w/2, ca2.h/2);
  ca2.createGeneration();
}
```

When you run the sketch, click anywhere within the sketch window to iteratively move through the sketch one iteration at a time. Figures 7-24, 7-25, and 7-26 show the sketch at iterations 0, 30, and 100 respectively.

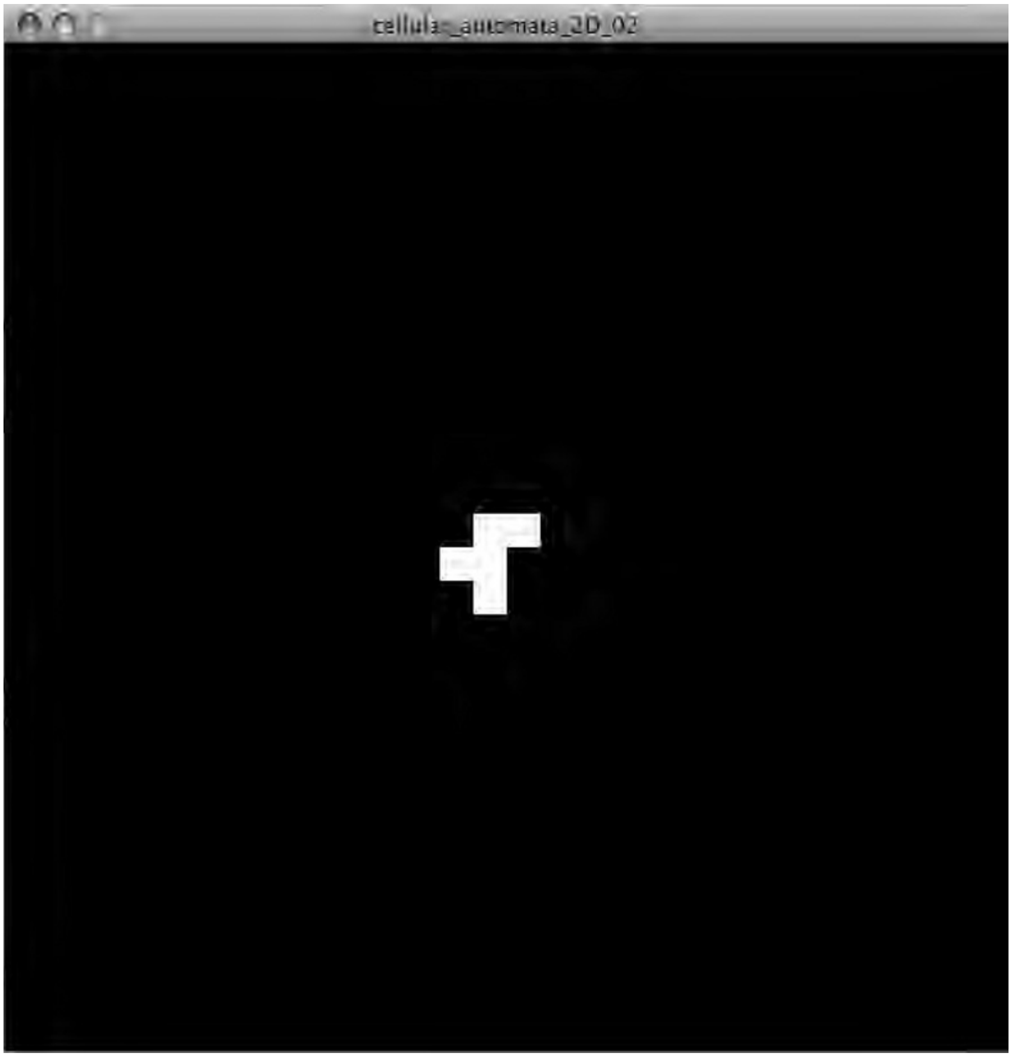

Figure 7-24. 2D CA Game of Life R-pentomino pattern at start, cellScale = 20

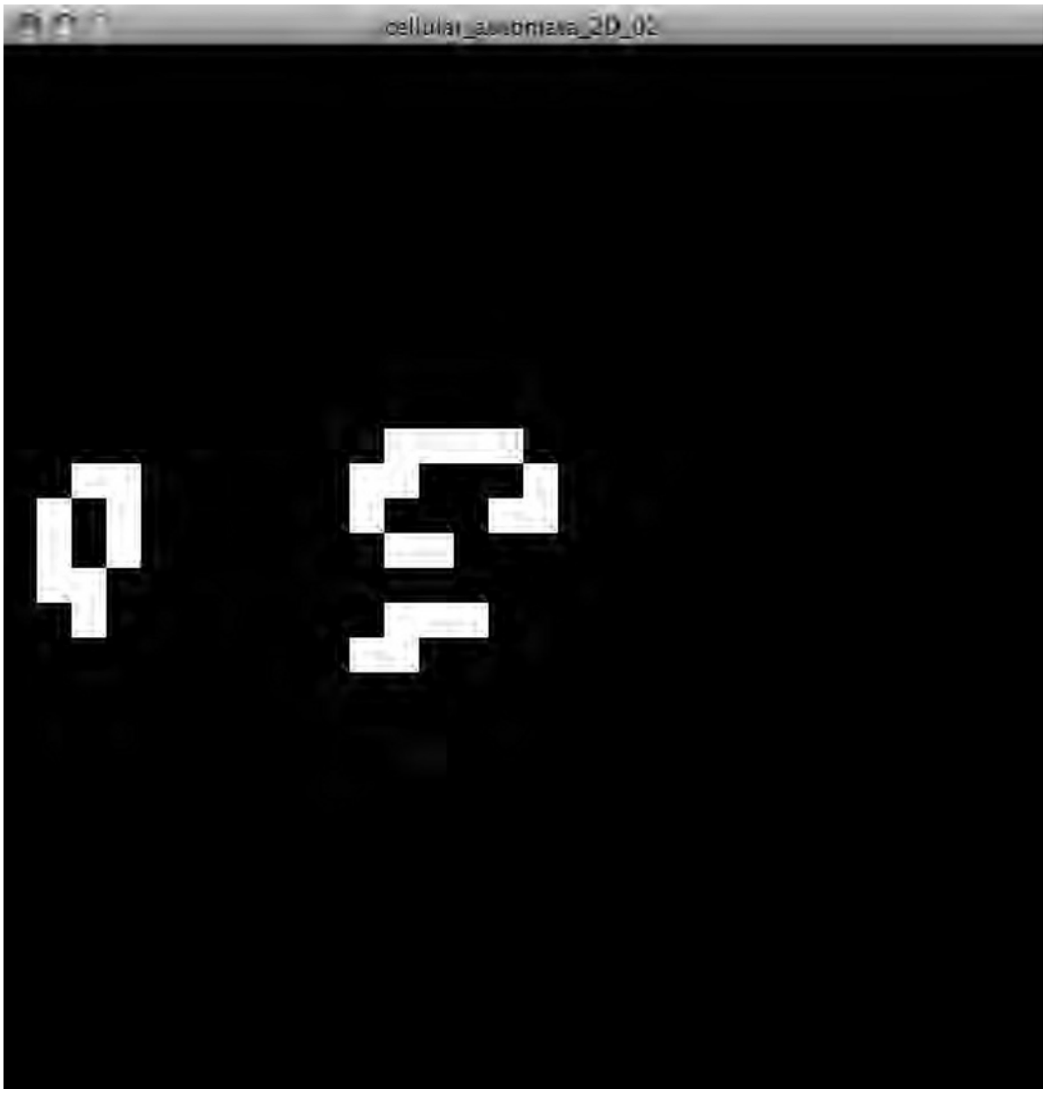

Figure 7-25. 2D CA Game of Life R-pentomino pattern after 30 frames, cellScale = 20

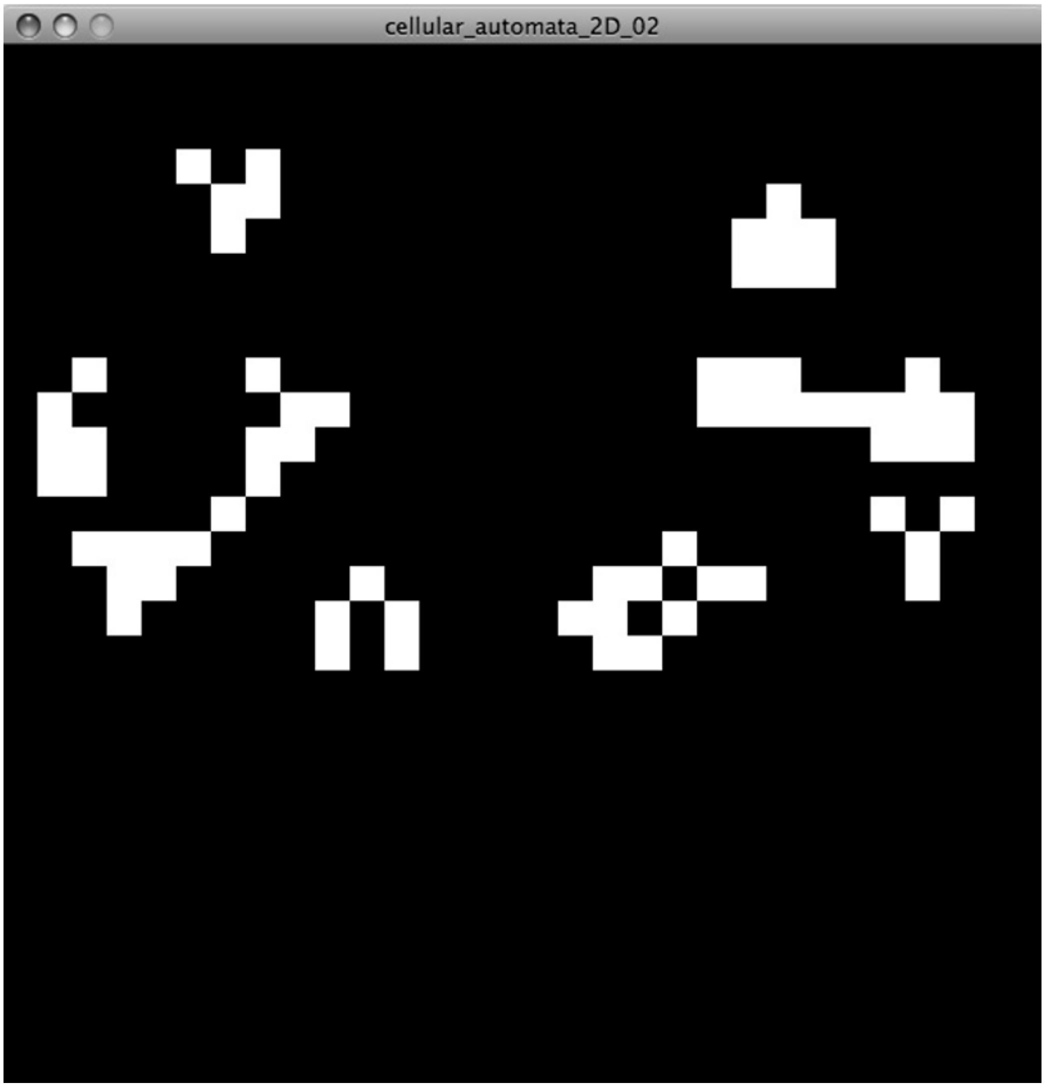

Figure 7-26. 2D CA Game of Life R-pentomino pattern after 100 frames, cellScale = 20

A pentomino is simply a shape composed of five symmetrical squares that are all connected orthogonally. You can read more about them at `http://en.wikipedia.org/wiki/Pentomino` (August, 6, 2009 14:00). Based on Life's rules, the R-pentomino pattern creates some unexpected results, which was indeed what Conway discovered when he first tried inputting the pattern, by *hand* mind you! It turns out that the R-pentomino pattern doesn't reach a stable state until a little over 1100 iterations (certainly a lot of work to try to do by hand). It also turns out that many of the sub-patterns created during these 1000 iterations of R-pentomino reveal many of the other classic patterns that Life produces, including ones formally classified as still-lifes, gliders, oscillators, guns, and puffers among others. Here is a nice link that discusses some of these interesting patterns: `http://www.math.com/students/wonders/life/life.html` (August, 6, 2009 14:12).

Although it would be interesting to try to create new patterns, there is already a treasure trove of them, with many creative hybrid patterns that combine multiple sub-patterns; some of these can be very complex. For example, Figure 7-27 shows a pattern called “c/3 long spaceships,” by Hartmut Holzwart and David Bell, which is composed of about 40,000 characters.

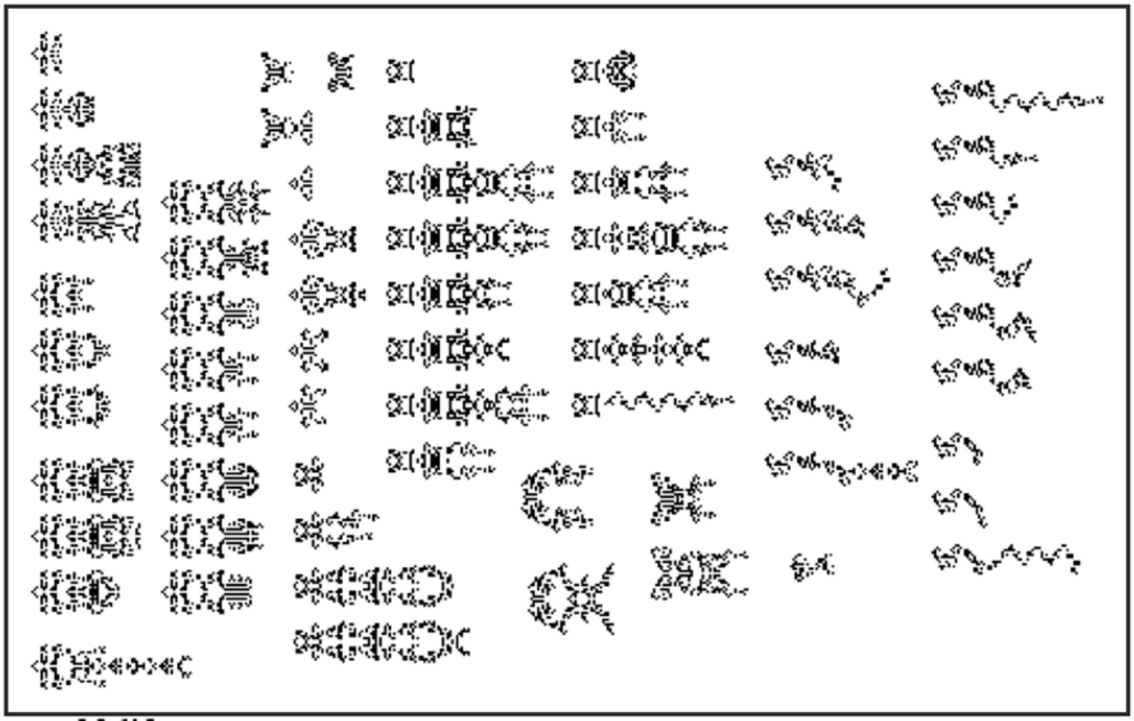

Figure 7-27. “c/3 long spaceships” pattern, by Hartmut Holzwart and David Bell

You probably wouldn’t want to try to code the *c/3 long spaceships* pattern by hand; loading it (like you would an image) would obviously be a much better solution. Fortunately, this loading problem has been solved—*well, sort of*. There have been file formats created for storing Life patterns (see `http://psoup.math.wisc.edu/mcell/ca_files_formats.html#Life%201.05`, August 6, 2009 14:56) enabling people to load and distribute them. A common Life pattern format is “Life 1.05,” which uses the `.lif` suffix (you’ll also see .life suffixes). It’s a very simple ASCII format that lists on/off pixels as a series of *’s and .’s respectively; in addition, each block of characters is preceded by a point location, specifying where on a Cartesian coordinate system to draw the block of pixels. For example, to draw the R-pentomino pattern at coordinate 100, 100, the `.lif` file would look like this. (Please note the #D is for file descriptions/comments. Some files also include a #N or #R for specifying rules, which we’ll ignore.)

```
#Life 1.05
#D R-pentomino
#D Adapted by Ira Greenberg
#D The Essential Guide to Processing for Flash Developers
#D Friends of ED, 2009
#P 100 100
.**
**.
.*.
```

I mentioned earlier that the Life pattern format *sort of* solved the problem; the other part is being able to parse the `.lif` file. As you might suspect, a Google search did not turn up a Processing `.lif` parser, so I decided to write one. My parser takes a `URL` address argument (as a link to a `.lif` file), which can be local, in the sketch's data directory, or on the web. In my example, the `URL` will be on the web, within a freely accessible Life patterns catalog.

> *If you're running an example online (as an applet), and the `.lif` file is on the web, it needs to reside on the same server as your applet or the applet must be **signed**; this is for security reasons. To learn how to sign an applet, check out this Processing hack I wrote a while back: `http://processing.org/hacks/hacks:signapplet` (August 7, 2009 15:05).*

I've coded the parser as a Processing class that will work within our CA framework. In the existing CA sketch (that includes the `CA_2D` class) create a new tab named "**`LIF_PARSER`**." I'll state in advance that this class is pretty dense, BUT it will be our last example, so you'll be able to rest your brain shortly. Also, in the next chapter we'll look at Processing's XML implementation, so this final example will also be a good data loading primer. Add the following to the **`LIF_PARSER`** tab:

```
/**
 * Cellular Automata
 * LIF_Parser class
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */
class LIF_Parser{
  // stores symbols (. *)
  String[] symbs = {};
  // path to .lif file
```

```
String url;
// stores all lines in .lif file
String[] lines;
// stores number of lines of symbols within each coord group
int[] indices;
// utility counter to increment coords[][] array
int coordsCounter = 0;
// stores origin in a sense of each symbol group
int[][] coords;
// bits buffer for pattern
int[] bits;
// bits array size (w, h)
int w, h;


// constructor
LIF_Parser(String url){
  this.url = url;
  // load .lif file
  lines = loadStrings(url);
  // get numbers of lines within each coord group
  indices = getIndices();
  // instantiate coords array values of where to draw each part of pattern
  coords = new int[indices.length][2];
  // isolate coords and reformat as int[][]
  parseCoords();
  //shift coords to remove negative values and isolate symbols
  shiftCoords();
  // create bits array based on pattern
  calcBits();
}

/* parses initial line strings, creating
 int[][] of coord data and isolates symbols*/
```

```
void parseCoords(){
  for (int i=0; i<lines.length; i++){
    String tempStr = "";
    // detect coords
    if (lines[i].charAt(0) == '#' && lines[i].charAt(1) == 'P'){
      // collect coord locs
      for (int j = 2; j<lines[i].length(); j++){
        tempStr += lines[i].charAt(j);
      }
      String tempStr2 = "";
      for (int j=0; j<tempStr.length(); j++){
        if (j>0 && tempStr.charAt(j) == ' '){
          tempStr2 += ',';
        }
        else if (tempStr.charAt(j) != ' '){
          tempStr2 += tempStr.charAt(j);
        }
      }
      coords[coordsCounter][0] = int(split(tempStr2, ','))[0];
      coords[coordsCounter][1] = int(split(tempStr2, ','))[1];
      coordsCounter ++;
    }
    else {
      // collect symbols
      if (lines[i].charAt(0) != '#'){
        symbs = append(symbs, lines[i]);
      }
    }
  }
}


/* add offset to x and y coords, based on lowest
 values, to avoid negative values */
```

```
void shiftCoords(){
  int xMin = 0, yMin = 0;
  // get lowest values
  for (int i=0; i<coords.length; i++){
    if (coords[i][0] < xMin){
      xMin = coords[i][0];
    }
    if (coords[i][1] < yMin){
      yMin = coords[i][1];
    }
  }
  // shift all coords
  for (int i=0; i<coords.length; i++){
    coords[i][0] += abs(xMin);
    coords[i][1] += abs(yMin);
  }
}

/* structure of data
 * stores number of symbols within each group
 * delimited by #P coordX coordY  in .lif file */
int[] getIndices(){
  int j = 0;
  int[] indices = {};

  for (int i=0; i<lines.length; i++){
    if(lines[i].charAt(0) != '#'){
      j++;
    }
    else {
      if (j!= 0){
        indices = append(indices, j);
      }
      j = 0;
    }
  }
  // get last group
  indices = append(indices, j);
  return indices;
}

// calculate bits
void calcBits(){
  // counter
  int ctr = 0;
  for (int i=0; i<indices.length; i++){
    for (int j=0; j<indices[i]; j++){
      // calculate max horizontal dimension
      if (coords[i][0] + symbs[ctr].length() > w){
        w = coords[i][0] + symbs[ctr].length();
      }
      // calculate max vertical dimension
      if (coords[i][1] + indices[i] > h){
        h = coords[i][1] + indices[i];
      }
      ctr++;
    }
  }
  // instantiate bits array
  bits = new int[w*h];
  // reset counter
  ctr = 0;
  //fill bits array
  for (int i=0; i<indices.length; i++){
    for (int j=0; j<indices[i]; j++){
      for (int k=0; k<symbs[ctr].length(); k++){
        if (symbs[ctr].charAt(k) == '.'){
          bits[w*(coords[i][1] + j) + (coords[i][0]+k)] = 0;
        }
        else if (symbs[ctr].charAt(k) == '*'){
          bits[w*(coords[i][1] + j) + (coords[i][0]+k)] = 1;
        }
      }
      ctr++;
    }
  }
}


}
```

Rather than break down all this code in detail, I'll discuss the class in a top-level way, which I think will more quickly help demystify it; really it's not that complicated (it just looks that way). Here's the basic algorithm:

1. Load the URL using Processing's `loadStrings()` function. `loadStrings()` brings in an external file as a `String` array, delimited by line breaks.
2. Calculate and store the number of lines of symbols within each coordinate group. It's possible that the file will contain only one coordinate group.
3. Isolate and store the coordinate values where to place each symbol group.
4. Shift the coordinate values so they are all positive.
5. Calculate and store an array of bits based on the symbols '*' = 1 or '.' = 0.

I strongly suggest going through the class to see how I coded each part of the algorithm. If you come across a Processing function you haven't seen before, be sure to highlight it and hit `command+shift+f` (Mac) or `control+shift+f` (Win), to read about it in the Processing reference.

We're almost ready to test out the new parser. First, though, we need to add one more method to the `CA_2D` class. At the bottom of the class, add the following method (be sure to put it above the final closing curly brace of the class):

```
// put pattern array into pixls as initial on/off state
  void setPattern(LIF_Parser lp){
    resetState();
    float deltaW = (cols - lp.w)/2.0;
    float deltaH = (rows - lp.h)/2.0;
    int ctr = 0;
    for (int i=0; i<rows; i++){
      for (int j=0; j<cols; j++){
        if (i >= deltaH && i < lp.h+deltaH &&
          j >= deltaW && j < lp.w+deltaW){
          if (lp.bits[ctr] == 0){
            pixls[int(i*cols+j)] = offC;
          }
          else if (lp.bits[ctr] == 1){
            pixls[int(i*cols+j)] = onC;
          }
          // pixls[int(i*cols+j)] = lp.pixls[ctr];
          ctr++;
        }

      }
    }
    paintInitState();
  }
```

This method enables the `CA_2D` object to accept a `.lif` pattern and embed the pattern in the `pixls` array. As with most things relating to pixels in Processing, the only challenging part was accounting for the two-dimensional structure of the pattern in the one-dimensional arrays. I used the local variables `deltaW` and `deltaH` to help center the pattern `bits` in the `pixls` array.

The last step is running the new parser. Replace what's in the main tab with the following to give it a test drive:

```
/**
 * Cellular Automata 2D Parser - main tab
 * By Ira Greenberg <br />
 * The Essential Guide to Processing for Flash Developers,
 * Friends of ED, 2009
 */

LIF_Parser lp;
String url = "http://www.radicaleye.com/lifepage/patterns/aqua50.lif";
CA_2D ca2;
```

```
void setup(){
  size(800, 600);
  background(255);
  ca2 = new CA_2D(width, height, 1);
  ca2.setOnColor(0xffff9900);
  ca2.setOffColor(0xff112233);
  lp = new LIF_Parser(url);
  ca2.setPattern(lp);
}

void draw(){
  translate(ca2.w/2, ca2.h/2);
  ca2.createGeneration();
}
```

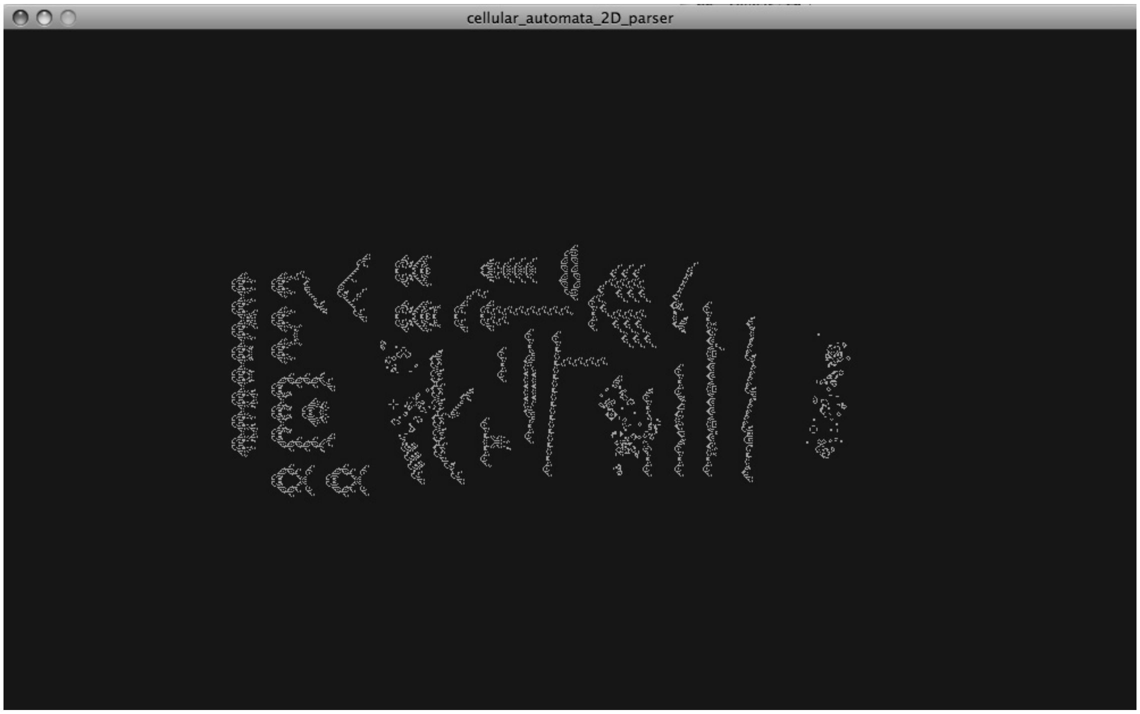

Figure 7-28. “p2 c/2 spaceships” pattern, by Hartmut Holzwart and Dean Hickerson

If it worked you should have seen something that looked like Figure 7-28. I recommend trying a bunch more patterns, which you can find around the web. There is a good pattern catalog at `http://radicaleye.com/lifepage/#browse` (August 7, 2009, 17:41). Just replace the quoted `String` address part of the line, `String url = "http://www.radicaleye.com/lifepage/patterns/aqua50.lif";`, with the new address. One last thing I suggest you also try is building a table structure of all the different CA discussed this chapter. Since they all work with the CA framework, you should be able to run them all simultaneously; then send me an email of what you get at `processing@iragreenberg.com`.

Summary

This chapter introduced the exciting concepts of emergence and complexity showcasing cellular automata. Building a CA framework, we looked at 1D, Continuous and 2D implementations, including an interactive example that allowed us to select CA thumbnails for enlargement (for our burgeoning tee-shirt business). CA reveal how simple rules can lead to very unexpected emergent complexity. This idea has much broader implications than for simply making cool images (not to knock cool images), but relates to how large complex systems, across many disciplines, emerge, grow, transform, and even perish. CA is just one computational approach for simulating and “playing” with complexity. Processing is a great environment for exploring this area because of its robustness, and ease, in handling pixel operations. Next chapter we’ll build upon some of the concepts we looked at during this chapter, as well as earlier in the book, as we explore creative data visualization in Processing.