Processing: a programming handbook for visual designers and artists

Casey Reas
Ben Fry

The MIT Press
Cambridge, Massachusetts
London, England

MIT Press books may be purchased at special quantity discounts for business or sales promotional use. For information, please email special_sales@mitpress.mit.edu or write to Special Sales Department, The MIT Press, 55 Hayward Street, Cambridge, MA 02142.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Reas, Casey.
Processing : a programming handbook for visual designers and artists / Casey Reas & Ben Fry ; foreword by John Maeda.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-262-18262-1 (hardcover : alk. paper)
1. Computer programming. 2. Computer graphics—Computer programs. 3. Digital art—Computer programs. 4. Art—Data processing. 5. Art and technology. I. Fry, Ben. II. Title.

QA76.6.R4138 2007
005.1—dc22

2006034768

10 9 8 7 6 5 4 3 2 1

Simulate 1: Biology

This unit discusses the concept of software simulation and the topics of cellular automata and autonomous agents.

Simulation is used within physics, economics, the social sciences, and other fields to gain insight into the complicated systems that comprise our world. Software simulations employ the most powerful computers to model aspects of the world such as weather and traffic patterns. A tremendous amount of intellectual energy in the field of computer graphics has been dedicated to accurate simulation of lighting, textures, and the movement of physical materials such as cloth and hair. An entire genre of computer games exists that simulate city planning, military campaigns, and even everyday life. Computers constitute a powerful medium for simulating the processes of the world, and increasing computer speeds offer more sophisticated possibilities.

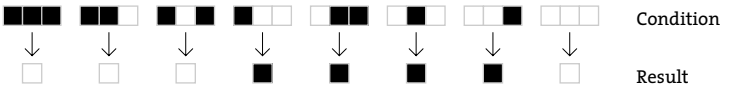
Within the arts, new technologies have often been used to represent and simulate nature. In ancient Greece, pneumatics animated sculptures. In the eighteenth century, precise gears provided the technical infrastructure for lifelike sculptures such as Vaucanson's Duck, which could "open its wings and flap them, while making a perfectly natural noise as if it were about to fly away."[1] In our contemporary world, computers and precision motors enable dancing robots and realistic children's toys that speak and move. One of the most fascinating simulations in recent art history is Wim Delvoye's *Cloaca* machine, which chemically and physically simulates the human digestive system.

Cellular automata

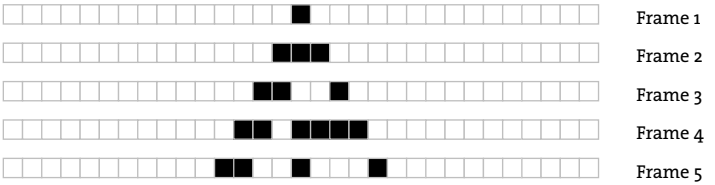
A cellular automaton (CA) is a self-operating system comprised of a grid of cells and rules stating how each cell behaves in relation to its neighbor. CAs were first considered by John von Neumann in the 1940s; they became well known in the 1970s after the publication of John Conway's Game of Life CA in a *Scientific American* article by Martin Gardner. CAs are intriguing because of their apparent simplicity in relation to the unexpected results they produce.

Steven Wolfram made important innovations in CA research in the early 1980s. Wolfram's one-dimensional CAs, each consisting of a single line of cells with rules, determine the value of each cell at each frame. The value of each cell is determined by its own value and those of its two neighbors. For example, if the current cell is white and its neighbors are black, it may also become black in the next frame. A set of rules determines when cells change their values. Since there are three cells with only two possible values (black or white), there are eight possible rules. In the diagram below, the three rectangles on the top are the three neighboring cells. The cell in the top middle is the current cell being evaluated, and those on the left and right are its neighbors.

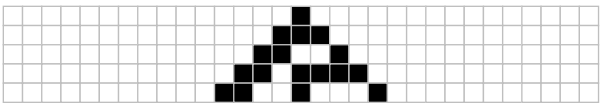
Depending on the current cell's value and that of its neighbors, the cell beneath the current cell is changed to black or white. The rules give rise to a number of possible variations and produce diverse results. One potential set of rules follows:



The CA starts with an initial state and is then updated to the next frame based on its rules. Each cell in the row is evaluated in relation to its two adjacent cells. Visual patterns begin to emerge from the minimal configuration of all white cells with one black cell in the middle:



The new one-dimensional image at each frame refers only to the previous frame. Because each frame is one-dimensional, it can be combined with the others to create a two-dimensional image, revealing the history of each frame of the CA within a single image that can be read from top to bottom:



The rules for this one-dimensional CA can be encoded as an array of 0s and 1s. Using the image at the top of this page as reference, if the configuration in the top row stays the same, the resulting bottom row can be defined as an array of 8 values, each a 1 or 0. This configuration can be coded as 0, 0, 0, 1, 1, 1, 1, 0 where 0 is white and 1 is black. Each of the other 256 possible configurations can be coded as a sequence of 8 numbers. Changing these numbers in the following example of a one-dimensional CA creates different images. The images on page 464 present some of the possible rules and their results.

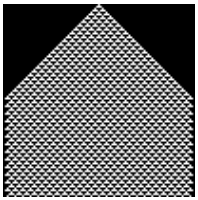
49-01

```
int[] rules = { 0, 0, 0, 1, 1, 1, 1, 0 };
int gen = 1;  // Generation
color on = color(255);
color off = color(0);

void setup() {
  size(101, 101);
  frameRate(8);  // Slow down to 8 frames each second
```

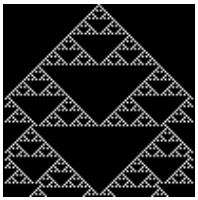
49-01 cont.

```
  background(0);
  set(width/2, 0, on);  // Set the top middle pixel to white
}

void draw() {
  // For each pixel, determine new state by examining current
  // state and neighbor states and ignore edges that have only
  // one neighbor
  for (int i = 1; i < width-1; i++) {
    int left = get(i-1, gen-1);         // Left neighbor
    int me = get(i, gen-1);             // Current pixel
    int right = get(i+1, gen-1);        // Right neighbor
    if (rules(left, me, right) == 1) {
      set(i, gen, on);
    }
  }
  gen++;                 // Increment the generation by 1
  if (gen > height-1) {  // If reached the bottom of the screen,
    noLoop();            // stop the program
  }
}

// Implement the rules
int rules(color a, color b, color c) {
  if ((a == on ) && (b == on ) && (c == on )) { return rules[0]; }
  if ((a == on ) && (b == on ) && (c == off)) { return rules[1]; }
  if ((a == on ) && (b == off) && (c == on )) { return rules[2]; }
  if ((a == on ) && (b == off) && (c == off)) { return rules[3]; }
  if ((a == off) && (b == on ) && (c == on )) { return rules[4]; }
  if ((a == off) && (b == on ) && (c == off)) { return rules[5]; }
  if ((a == off) && (b == off) && (c == on )) { return rules[6]; }
  if ((a == off) && (b == off) && (c == off)) { return rules[7]; }
  return 0;
}
```

John Conway's Game of Life predates Wolfram's discoveries by more than a decade. Gardner's article in *Scientific American* describes Conway's invention as "a fantastic solitaire pastime he calls 'life.' Because of its analogies with the rise, fall and alternations of a society of living organisms, it belongs to a growing class of what are called 'simulation games'—games that resemble real-life processes."[2] Life was not originally run with a computer, but early programmers rapidly became fascinated with it, and working with the Game of Life in software enabled new insight into the patterns that emerge as it runs.

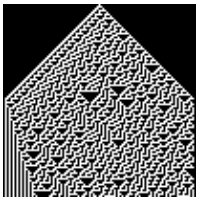


0,0,1,1,0,1,1,0

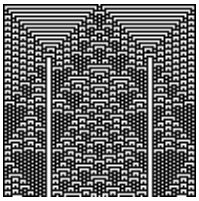


0,1,0,1,1,0,1,0

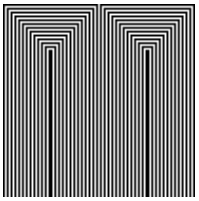
1,0,1,1,0,1,1,0



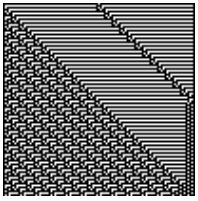
0,0,0,1,1,1,1,0



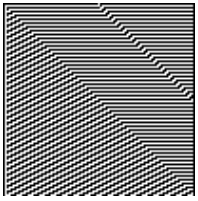
0,1,0,0,1,0,0,1



0,1,0,0,1,1,0,1

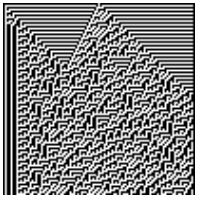


0,0,1,0,1,0,0,1



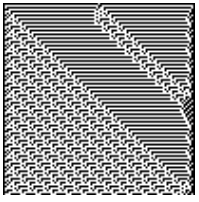
0,0,1,0,1,1,1,1

0,0,1,1,1,0,1,1

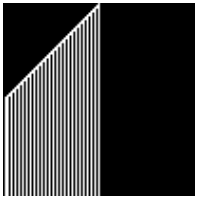


0,0,1,0,1,1,0,1

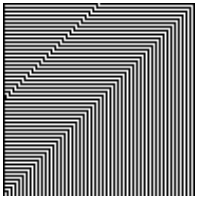
1,0,1,0,0,1,0,1



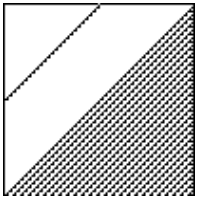
0,1,1,0,1,0,1,1



0,1,0,0,1,1,1,0



0,1,0,1,0,1,0,1



1,0,0,1,1,0,1,1

Wolfram's one-dimensional cellular automata

Use the numbers below each image as the data for the `rules[]` array in code 49-01 to watch each pattern generate.

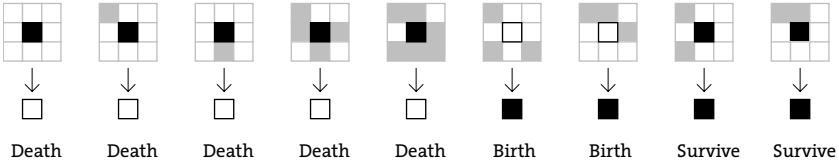
The Game of Life is a two-dimensional CA in which the rules for determining the value of each cell are defined by the neighboring cells in two dimensions. Each cell has eight neighboring cells, each of which can be named in relation to the directional orientation of the cell—north, northeast, east, southeast, south, southwest, west, northwest:

NW	N	NE
W		E
SW	S	SE

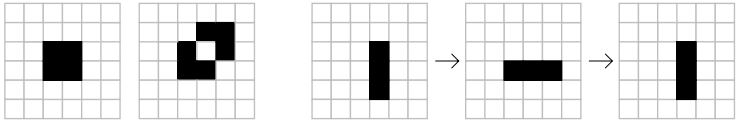
The rules for turning a cell on (alive) and off (dead) relate to the number of neighboring cells:

1. Death from isolation: Each live cell with less than two live neighbors dies in the next generation
2. Death from overpopulation: Each cell with four or more live neighbors dies in the next generation
3. Birth: Each dead cell with exactly three live neighbors comes to life in the next generation
4. Survival: Each live cell with two live neighbors survives in the next generation

Applying these rules to a cell reveals how different configurations translate into survival, death, and birth. In the image below, the cell being currently evaluated is in the center and is black if alive; neighbor cells are gray if alive and white if empty:

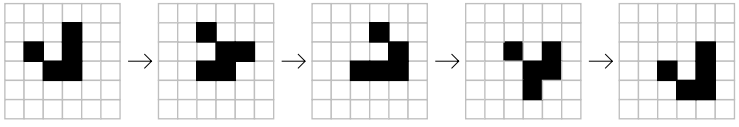


Different spatial configurations of cells create repeating patterns with each new generation. Some shapes are stable (do not change at each frame), some repeat, and some move across the screen:



Stable Configurations

Periodic Configuration



Moving object (this configuration moves one unit right and down over three generations)

Shapes called gliders are arrangements of cells that move across the grid, go through frames of physical distortion, and then arrive back at the same shape shifted by one grid unit. Repeating this pattern propels them across the grid.

The current state for the Game of Life is stored in a two-dimensional array of integers. A second grid hosts the next generation. At the end of each frame, the newly created generation becomes the old generation, and the process repeats. Cells are marked alive with the number 1 and and dead with 0. This makes it simple to count the number of neighbors by adding the values of neighboring cells. The `neighbors()` function looks at neighbors and counts the values of the adjacent cells. These numbers are used to set white or black pixels within `draw()`.

49-02

```
int[][] grid, futureGrid;

void setup() {
  size(540, 100);
  frameRate(8);
  grid = new int[width][height];
  futureGrid = new int[width][height];
  float density = 0.3 * width * height;
  for (int i = 0; i < density; i++) {
    grid[int(random(width))][int(random(height))] = 1;
  }
  background(0);
}

void draw() {
  for (int x = 1; x < width-1; x++) {
    for (int y = 1; y < height-1; y++) {
      // Check the number of neighbors (adjacent cells)
      int nb = neighbors(x, y);
      if ((grid[x][y] == 1) && (nb <  2)) {
        futureGrid[x][y] = 0;  // Isolation death
        set(x, y, color(0));
      } else if ((grid[x][y] == 1) && (nb >  3)) {
        futureGrid[x][y] = 0;  // Overpopulation death
        set(x, y, color(0));
      } else if ((grid[x][y] == 0) && (nb == 3)) {
        futureGrid[x][y] = 1;  // Birth
        set(x, y, color(255));
      } else {
        futureGrid[x][y] = grid[x][y];  // Survive
      }
    }
  }
```

```
  // Swap current and future grids
  int[][] temp = grid;
  grid = futureGrid;
  futureGrid = temp;
}

// Count the number of adjacent cells 'on'
int neighbors(int x, int y) {
  return grid[x][y-1] +    // North
         grid[x+1][y-1] +  // Northeast
         grid[x+1][y] +    // East
         grid[x+1][y+1] +  // Souteast
         grid[x][y+1] +    // South
         grid[x-1][y+1] +  // Southwest
         grid[x-1][y] +    // West
         grid[x-1][y-1];   // Northwest
}
```

49-02 cont.

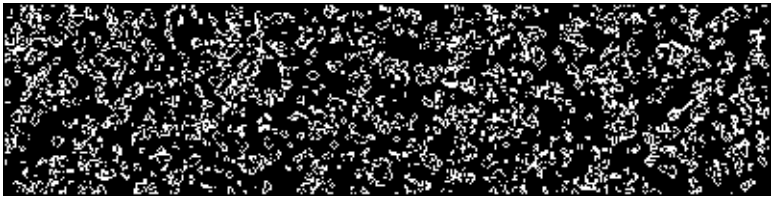
Changing the `neighbors()` function in code 49-02 to utilize the modulo operator (`%`) makes it possible for the cells to wrap from one side of the screen to the other. The `for` structures inside `draw()` also need to change to loop from `0` to `width` and `0` to `height`.

```
int neighbors(int x, int y) {
  int north = (y + height-1) % height;
  int south = (y + 1) % height;
  int east = (x + 1) % width;
  int west = (x + width-1) % width;
  return grid[x][north] +     // North
         grid[east][north] +  // Northeast
         grid[east][y] +      // East
         grid[east][south] +  // Southeast
         grid[x][south] +     // South
         grid[west][south] +  // Southwest
         grid[west][y] +      // West
         grid[west][north];   // Northwest
}
```

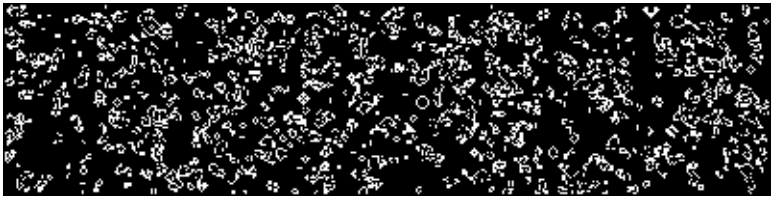
49-03

Research into cellular automata did not stop with Conway and Wolfram. Others have developed continuous CAs that are not limited to on/off states. Probabilistic CAs, for example, can partially or totally determine their rules through probabilities rather than absolutes. CAs possess the ability to simulate lifelike phenomena in spite of their basic format. For example, the patterns created with a one-dimensional CA can mimic patterns found in the shells of organisms such as cone snails. Other CAs produce images similar to those created by biochemical reactions.

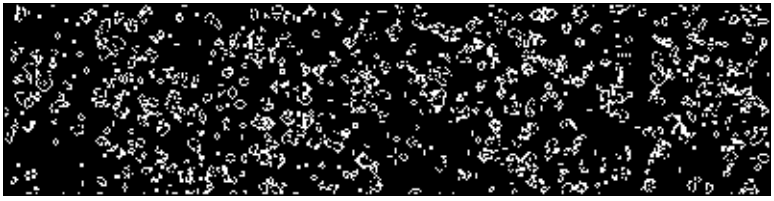
frameCount = 1



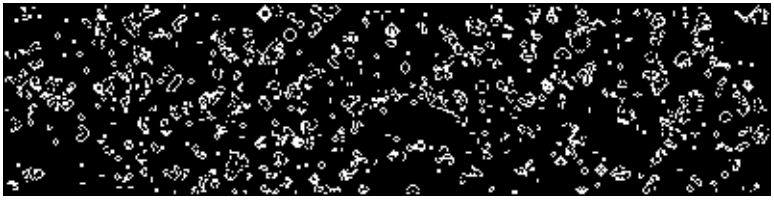
frameCount = 10



frameCount = 20



frameCount = 30



frameCount = 40

Conway's Game of Life

Using a few simple rules defined in code 49-02, the color relations between adjacent pixels create a dynamic ecosystem.

Autonomous agents

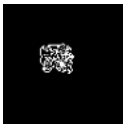
An autonomous agent is a system that senses and acts on its environment according to its own agenda. People, spiders, and plants are all autonomous agents. Each agent uses input from the environment as a basis for its actions. Each pursues it own goals, either consciously or through reflex. In his book *The Computational Beauty of Nature*, Gary William Flake defines an autonomous agent as "a unit that interacts with its environment (which probably consists of other agents) but acts independently from all other agents in that it does not take commands from some seen or unseen leader."[3] Agents aren't a part of a coordinated global plan, but structure does emerge from their interactions with other agents and the environment. The seemingly coordinated behavior of an ant colony and the order within a school of fish illustrate structured behavior emerging from the collective actions of individual agents.

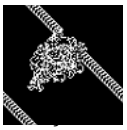
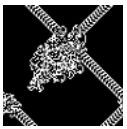
Like the examples of cellular automata presented above, autonomous agents can also exist in a grid world. Chris Langton's ant is a fascinating example. The ant can face only one of four directions: north, south, east, or west. Like cellular automata, the ant moves one frame at a time, behaving according to the following rules:

1. Move one frame forward
2. If on a white pixel, change the pixel to black and turn 90 degrees right
3. If on a black pixel, change the pixel to white and turn 90 degrees left

As the ant moves through the environment, it returns to the same pixel many times and each visit reverses the color of the pixel. Therefore, the future position of the ant is determined by its past movements. The remarkable thing about this ant is that with any starting orientation (north, south, east, or west), a sequence of actions that produce a straight path always emerges. From the seemingly chaotic mess upon which the ant embarks, an ordered path always develops. This program is not intended as a simulation of a real insect. It's an example of a software agent with extremely simple rules behaving in an entirely unexpected but ultimately predictable and structured manner. The instruction to eventually construct a straight path is never given, but it emerges through the rules of the ant in relation to its environment. The environment contains the memory of the ant's previous frames, which the ant uses to determine its next move.

In this example program, the ant's world wraps around from each edge of the screen to the opposite edge. Wrapping around to the other side of the screen, the ant is disrupted by its previous path. The order eventually emerges and the ant begins a new periodic sequence producing linear movement. In the code, directions are expressed as numbers. South is 0, east is 1, north is 2, and west is 3. At each frame, the ant moves one pixel forward based on its current orientation and then checks the color of the pixel at its location. It turns right by subtracting 1 and turns left by adding 1. Run the code to see the sequence change through time.



49-04

```
int SOUTH = 0;          // Direction numbers with names
int EAST = 1;           // so that the code self-documents
int NORTH = 2;
int WEST = 3;
int direction = NORTH;  // Current direction of the ant
int x, y;               // Ant's current position

color ON = color(255);  // Color for an 'on' pixel
color OFF = color(0);   // Color for an 'off' pixel

void setup() {
  size(100, 100);
  x = width/2;
  y = height/2;
  background(0);


void draw() {
  if (direction == SOUTH) {
    y++;
    if (y == height) {
      y = 0;
    }
  } else if (direction == EAST) {
    x++;
    if (x == width) {
      x = 0;
    }
  } else if (direction == NORTH) {
    if (y == 0) {
      y = height-1;
    } else {
      y--;
    }
  } else if (direction == WEST) {
    if (x == 0) {
      x = width-1;
    } else {
      x--;
    }
  }

  if (get(x, y) == ON) {
    set(x, y, OFF);
```

49-04
cont.

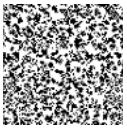
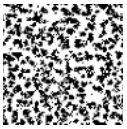
```
      if (direction == SOUTH) {
        direction = WEST;
      } else {
        direction--;
      }
    } else {
      set(x, y, ON);
      if (direction == WEST) {
        direction = SOUTH;
      } else {
        direction++;  // Rotate direction
      }
    }
  }
```

Mitchel Resnick's termite is another example that demonstrates order emerging from extremely simple rules. Like Langton's ant, this termite is not intended as a simulation of a real organism, but it exhibits remarkable behavior. The termite exists on a grid where a white unit represents open space and black represents a wood chip. The termite wanders through the space, and when it finds a wood chip it picks it up and wanders until it runs into another wood chip. Finding a wood chip causes it to drop its current load, turn around, and continue to wander. Over time, ordered piles emerge as a result of its effort.

In the code that creates the termite and its environment, the `angles[]` array contains the possible directions in which the termite can move. At each frame the termite moves one space on the grid. The angles specify which neighboring pixel it can move into:

NW −1,−1	N 0,−1	NE 1,−1
W −1,0		E 1,0
SW −1,1	S 0,1	SE 1,1

When space in front of the termite is open, it moves to the next space in the current direction or the next space in an adjacent direction. For example, if the current direction is south, it will move to the next space in the south, southeast, or southwest direction. If the current direction is northeast, it will move to the next space in the northeast, east, or north direction. When the termite does not have space in front and it's carrying a wood chip, it will reverse its direction and move one space in the new direction. When it does not have a space in front and it's not carrying a wood chip, it moves into the space occupied with the wood chip and picks it up.

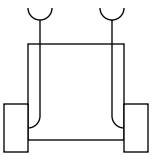


```
int[][] angles = {{ 0, 1 }, { 1, 1 }, { 1, 0 }, { 1,-1 },    49-05
                  { 0,-1 }, {-1,-1 }, {-1, 0 }, {-1, 1 }};

int numAngles = angles.length;
int x, y, nx, ny;
int dir = 0;
color black = color(0);
color white = color(255);

void setup() {
  size(100, 100);
  background(255);
  x = width/2;
  nx = x;
  y = height/2;
  ny = y;
  float woodDensity = width * height * 0.5;
  for (int i = 0; i < woodDensity; i++) {
    int rx = int(random(width));
    int ry = int(random(height));
    set(rx, ry, black);
  }
}

void draw() {
  int rand = int(abs(random(-1, 2)));
  dir = (dir + rand + numAngles) % numAngles;
  nx = (nx + angles[dir][0] + width) % width;
  ny = (ny + angles[dir][1] + height) % height;

  if ((get(x,y) == black) && (get(nx,ny) == white)) {
    // Move the chip one space
    set(x, y, white);
    set(nx, ny, black);
    x = nx;
    y = ny;
  } else if ((get(x,y) == black) && (get(nx,ny) == black)) {
    // Move in the opposite direction
    dir = (dir + (numAngles/2)) % numAngles;
    x = (x + angles[dir][0] + width) % width;
    y = (y + angles[dir][1] + height) % height;
  } else {
    // Not carrying
    x = nx;
```

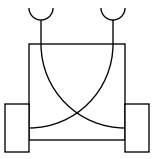
```
      y = ny;
    }
    nx = x;  // Save the current position
    ny = y;
  }
```

Other simulations of autonomous agents have been created without restrictive grids. These agents are allowed to move freely through their environment. Because they use floating-point numbers for position, they have more potential variations in location and orientation than the gridded CAs. Two of the best-known autonomous agents are Valentino Braitenberg's Vehicles and Craig Reynolds's Boids.

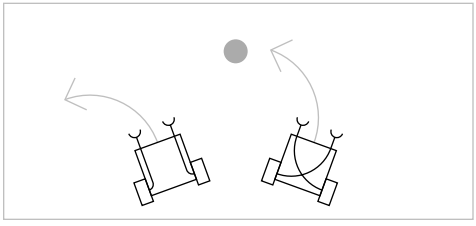
The neuroanatomist Valentino Braitenberg published *Vehicles: Experiments in Synthetic Psychology* in 1984. In this small, delightful book he presents conceptual schematics for fourteen unique synthetic creatures he calls Vehicles. Vehicle 1 has one sensor and one motor that are connected so that a strong stimulus will make the motor turn quickly and a weak stimulus will make the motor turn slowly. If the sensor registers nothing, the vehicle will not move. Vehicle 2 has two sensors and two motors. If they are correlated the same way as in Vehicle 1 they create Vehicle 2a and if they are crossed they create Vehicle 2b. If the sensor is attracted to light, for example, and there is a light in the room, Vehicle 2a will turn away from the light and Vehicle 2b will approach the light. Braitenberg characterizes these machines as correspondingly cowardly and aggressive to feature the anthropomorphic qualities we assign to moving objects:



Vehicle 2a



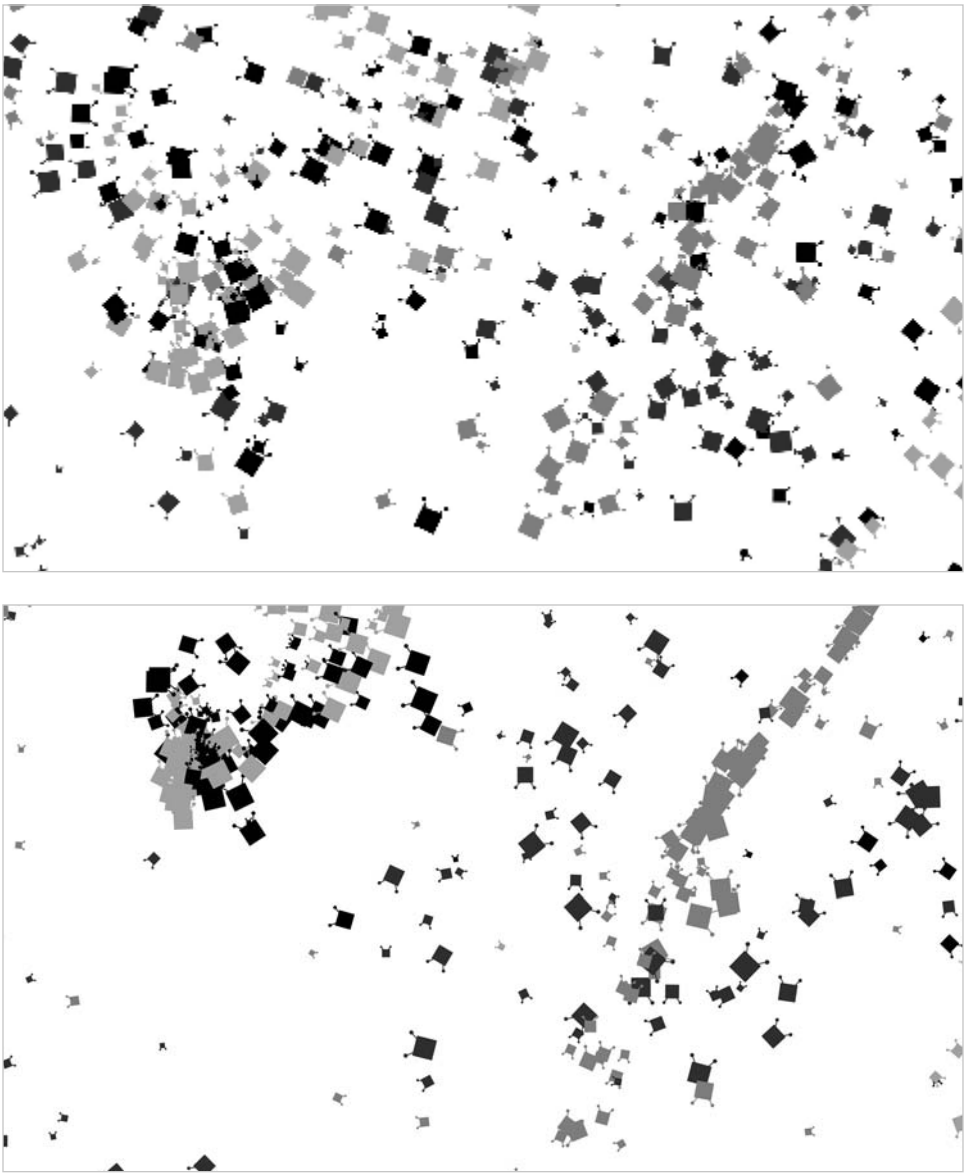
Vehicle 2b



Vehicle 2a and 2b movement in relation to a stimulus

Vehicle 3a and 3b are identical to Vehicle 2a and 2b but the correlation between the sensor and the motor is reversed—a weak sensor stimulus will cause the motor to turn quickly and a strong sensor stimulus causes the motors to stop. Vehicle 3a moves toward the light and stops when it gets too close, and 3b approaches the light but turns and leaves when it gets too close. If more than one stimulus is placed in the environment, these simple configurations can yield intricate paths of movement as they negotiate their attention between the competing stimuli.

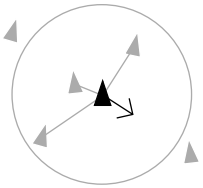
In 1986, Craig Reynolds developed the Boids software model to simulate coordinated animal motion like that of flocks of birds and schools of fish. To refute the common conception that these groups of creatures navigate by following a leader, Reynolds presented three simple behaviors that simulated a realistic flocking behavior without



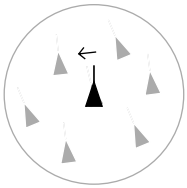
Braitenberg's Vehicles

Five hundred vehicles move through the environment. Each gray value represents a different category of vehicles. The vehicles in each category share the same behavior (follow the same rules), so over time they form groups.

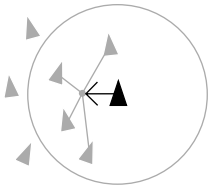
the need for a hierarchy. These behaviors define how each creature behaves in relation to its neighbors:



Separation:
Steer to avoid crowding local flockmates



Alignment:
Steer toward the average heading of local flockmates



Cohesion:
Steer to move toward the average position of local flockmates

The flocking rules provide an evocative example of emergence, the phenomenon of a global behavior originating from the interactions of simple, local behaviors. The flocking behavior of the group is not overtly programmed, but emerges through the interaction of each software unit based on the simple rules. The Pond example on page 497 is an implementation of Boids.

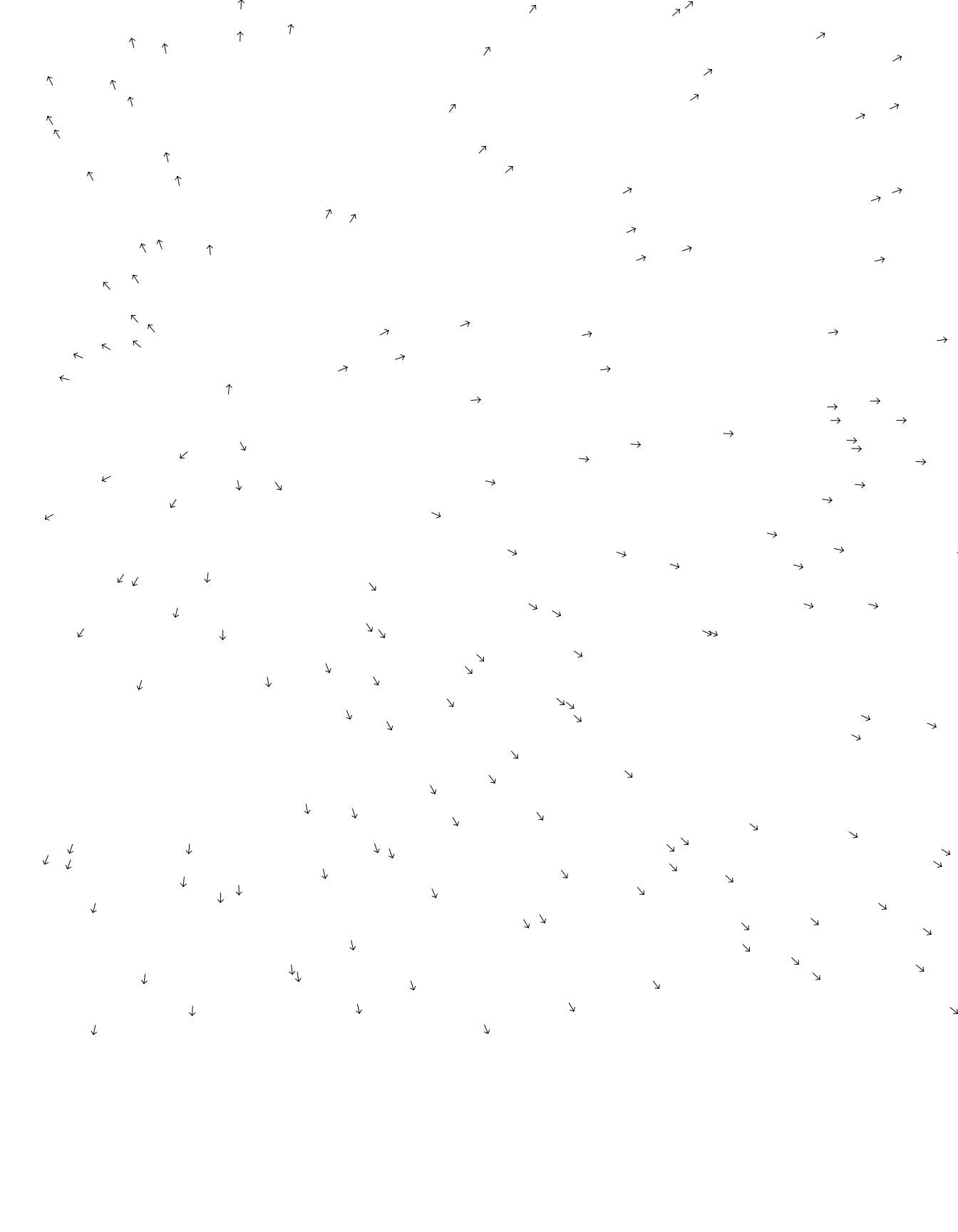
The autonomous agent simulations presented here were at the cutting edge of research over twenty years ago. They have since become some of the most popular examples for presenting the ideas of agency and emergence. The ideas from research in autonomous agents has been extended into many disciplines within art and science including sculpture, game design, and robotics.

Exercises

1. *Increase the size of the grid for Wolfram's one-dimensional CA. There are 256 possible rule sets for this one program and only 13 are presented in this unit. Try some of the other options. Which do you find the most interesting? Can the diverse results be put into categories?*
2. *Increase the size of the grid for Conway's Game of Life. Can you find other stable, periodic, or moving configurations?*
3. *Extend the termite code to have more than one termite moving chips of wood.*

Notes

1. Alfred Chapuis and Edmond Droz, *Automata: A Historical and Technological Study*, translated by Alec Reid (Editions du Griffon, 1958).
2. Martin Gardner, "Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'Life,'" *Scientific American* 223 (October 1970), pp. 120 - 123.
3. Gary William Flake, *The Computational Beauty of Nature* (MIT Press, 1998), p. 261.



Synthesis 4: Structure and Interface

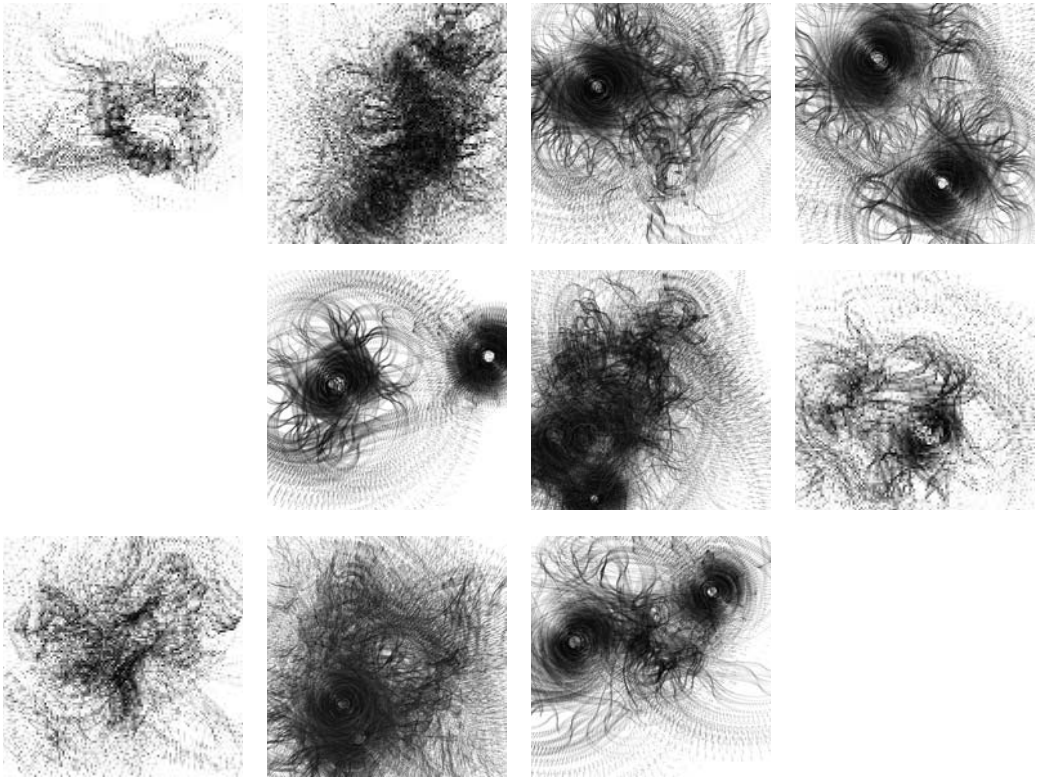
This unit presents examples that synthesize concepts from Structure 4 to Simulate 2.

The previous units introduced object-oriented programming, saving and importing files, creating graphical user interfaces, and simulating biology and physics. This unit focuses on integrating these concepts with an emphasis on object-oriented thinking. As mentioned, object-oriented programming is an alternative way to think about structuring programs. Knowing when to use object-oriented programming and how to structure the objects is an ability that develops with time and experience. The programs in this unit apply object-oriented thinking to previously introduced topics.

It's now possible to integrate elements of software including variables, control structures, arrays, and objects in tandem with visual elements, motion, and response to create exciting and inventive software. Because of space restrictions and our desire not to overwhelm the reader, this book omits discussion of many programming concepts, but the topics presented provide a solid foundation for diverse exploration.

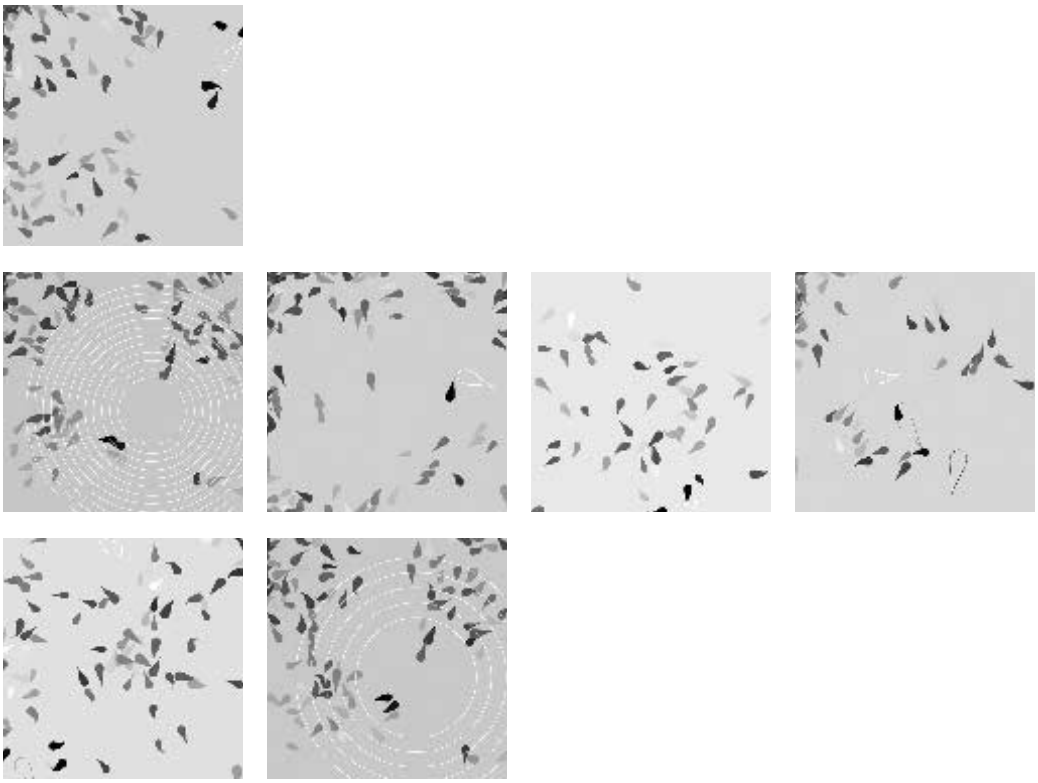
The programs presented in this unit are the most challenging in the book, but they include only ideas and code that have been previously introduced. They're challenging in their composition, but all of the components are built from the concepts discussed in this text. These programs include a game, drawing software, generative form, and simulations.

The four programs presented here were written by different programmers. Unlike most of the other examples in the book, which have been written in a similar style, each of these programs reflects the personal programming style of its author. Learning how to read programs written by other people is an important skill.
The software featured in this unit is longer than the brief examples that fill this book. It's not practical to print it on these pages, but the code is included in the Processing code download at www.processing.org/learning.



WithoutTitle. The images on this page were created with a sophisticated drawing program that combines elements of code from Motion 2 (p. 291), Structure 5 (p. 453), and Drawing 2 (p. 413). A dense thicket of lines circulates around the position of the cursor; moving the position of the cursor affects the epicenter and how the lines expand and contract.

Program written by Lia (http://lia.sil.at)



Pond. These images were generated from an implementation of Craig Reynolds's Boids rules, explained in Simulate 1 (p. 461). As each fish follows the rules, groups are formed and disperse. Clicking the mouse sends a wave through the environment and lures the creatures to the center. Each creature is an instance of the `Fish` class. The direction and speed of each fish is determined by the rules. The undulating tail is drawn with Bézier curves and moves from side to side in relation to the current direction and speed.

Program written by William Ngan (www.metaphorical.net)

Swingtree. This software simulates a tree swaying in the wind. Move the mouse left and right to change the direction and move it up and down to change the size. The connections between each branch are set by data stored in a text file. When the program starts, the file is read and parsed. The values are used to create instances of the `Branch` and `Segment` classes.

Program written by Andreas Schlegel (www.sojamo.de) at ART+COM (www.artcom.de)