Nature of Code

Patrick Dwyer
Fall 2005
Week 7 - October 18th

Fractals and Recursion

Benoit Mandelbrot coined the term fractal in 1975 to describe self-similar shapes achieved via the process of recursion. Most of the stuff we encounter in our physical world can described by idealized geometrical forms -- a postcard has a rectangular shape, a ping-pong ball is spherical, etc. However, many types of naturally occurring structures cannot be described by such simple means (snowflakes, trees, coastlines, mountains, etc.) Fractals allow us to describe and simulate these types of self-similar shapes (by "self-similar" we mean no matter how "zoomed out" or "zoomed in" we are, the shape ultimately appears the same.)

We know that a function can call another function. We do this all the time. But we can take this a step further and create functions that call themselves. Functions that call themselves are called "recursive" and are appropriate for solving different types of problems. This occurs often in mathematical calculations; the most common example of this is computing a "factorial." The factorial of any number n, written `n!`, is defined as:

```
n! = 1 * 2 * 3 * . . .* n

Where
0! = 1
```

We could write a normal (non-recursive) function to do this in processing:

```
int factorial(int n) {
      int result = 1;
      for (int i = 0; i < n; i++) {
            result = result * (i + 1);
      }
      return result;
}
```

While this would generate the correct answer, we can also define a factorial in a recursive manner:

```
n! = n * (n -1)!

Where
0! = 1
```

Here we've defined the result in terms of a recursion, or a self reference. The factorial of N is N times the factorial of N - 1. We can translate this to a function that accomplishes the same thing:

```
int factorial(int n) {
      if (n == 0) {
```

```
            return 1;
      } else {
            return n * factorial(n - 1);
      }
}
```

We can apply the same principle of recursion to our visual environment. For instance, the following method:

```
void drawCircle(int x, int y, float radius) {
      ellipse(x, y, radius, radius);
      if (radius > 2) {
            drawCircle(x, y, radius * 0.75);
      }
}
```

What does this method do? It draws an ellipse based upon the parameters passed to the method, and then recursively calls itself with new parameters, based upon the values of the current method call. This example is somewhat trivial, as we could quite easily do the same thing with iteration (a `for` loop, for instance), but there are situations where more complex behavior, such as a method calling itself multiple times, is more easily addressed with recursion. Consider the render method taken from the RecursiveTree example for this week (comments removed for brevity):

```
    public void render(float l_) {

        l_ = l_ * 0.66f;

        if (l_ > minLength) {

            pushMatrix();
            rotate(theta);
            line(0, 0, 0, -l_);
            translate(0, -l_);
            render(l_);
            popMatrix();

            pushMatrix();
            rotate(-theta);
            line(0, 0, 0, -l_);
            translate(0, -l_);
            render(l_);
            popMatrix();

        }
    }
```

This method uses the provided parameters to draw two branches, calling itself at the end of each branch. This kind of recursion would be much more complicated to mimic with a non-recursive solution. It is important to note here:

- All recursive functions must have an exit condition. This is similar to iteration; all of our `for` or `while` loops must have a boolean test that determines when the loop is finished. Without an exit condition recursive functions will quickly crash processing (and possibly the computer).
- When using recursion for drawing, the processing methods `pushMatrix` and `popMatrix` are extremely useful. When we call `pushMatrix` we save our current position and rotation so we can come back to it later. Once we call `pushMatrix` in our example, we proceed to recursively call our function. When the recursion finishes, we call `popMatrix` to return to our previous position. This lets us bypass a lot of tedious math for figuring out our drawing position every step of the way.

While recursion works well for some problems, others we can address using iteration and ArrayList. Consider the following steps:

- Create an empty ArrayList
- Place a single object in the ArrayList
- For each object in the ArrayList, remove the object and add two new objects to the ArrayList based upon the removed object.
- Repeat the previous step as many times as desired

This series of steps can be used to create many types of recursive shapes, such as the Koch Fractal example for this week.

L-Systems

Using the technique of recursion, we can create highly complex systems with a few simple rules. One of the best illustrations of this comes in the form of L-Systems. Similar in result to a recursive tree structure, L-Systems combine the complexity of recursion and the step by step process of iteration to generate “Tree” or “Plant” like visualizations.

To create an L-System, we first need to create our own programming language. This sounds quite a bit harder than it really is; our programming language will only define five operations:

- Draw a straight line
- Turn Left
- Turn Right
- Save our position
- Retrieve our position

These five operations match the basic operations we need to create an L-System. Each L-System is defined by an *axiom* and a *rule*. We can think of the axiom as a starting position, and the rule as the directions for drawing the system. A basic L-System might be defined with the following:

```
Axiom: F
Rule: FF-[-F+F]
```

The symbols may seem strange at first, but we can pair each of them with the five operations in our min-programming language:

- F =>Draw a straight line
- + => Turn Left
- - => Turn Right
- [=> Save our position
-] => Retrieve our position

So our basic L-System we defined has an Axiom that equates; “Draw a straight line”, while the rule equates; “Draw a straight line twice, turn right, save our position, turn right and draw a straight line, turn left and draw a straight line, and retrieve our position.” Following this rule won’t result in a very interesting drawing, just a long straight line with two shorter lines branching from the tip. The complexity of an L-System derives from recursive application of the Rule to the Axiom.

We’ll define the current state of our L-System as our `production`, and initially we’ll set that equal to our Axiom (our starting condition). For each generation (application of the rule) we will simply replace every occurrence of the letter F with the rule, so our first two generations look like:

```
1: F
2: FF-[-F+F]
```

Seems simple enough so far, but take a look at the third generation:

```
3: FF-[-F+F] FF-[-F+F]-[-FF-[-F+F]+ FF-[-F+F]]
```

We can see that our production (current state of the L-System) quickly gets longer and more complicated. When we draw each different generation we find that what started as a simple line, or a short line with two branches, quickly becomes a highly complex series of branching and dividing lines. Thankfully drawing our L-System is easier than creating it, we just need to step through our production one character at a time, and follow the procedure we outlined in our programming language:

```
public void render() {
    // start at the bottom center of the screen
    translate(width / 2, height);

    for (int i = 0; i < production.length(); i++) {
        char step = production.charAt(i);

        if (step == 'F') {
            // draw a line
            line(0, 0, 0, -drawLength);
```

```
            // move to the end of the line
            translate(0, -drawLength);
        } else if (step == '+') {
            // rotate
            rotate(theta);
        } else if (step == '-') {
            // rotate
            rotate(-theta);
        } else if (step == '[') {
            // save our position
            pushMatrix();
        } else if (step == ']') {
            // reset our position
            popMatrix();
        }
    }
}
```

L-Systems are a complex subject, dealing with recursion, Finite State Machines (our little programming language), and iteration, but they can yield amazing results.

Mandelbrot & Julia Sets

Building on the work we did with 2D Graphing, we can explore the strange territory of the Mandelbrot and Julia sets. These sets (also known as Fractals) are based on equations describing a series of complex numbers, where each number tends towards 0 or infinity as it is iterated over with a self-describing function. Thankfully we don't need to fully understand the somewhat odd math behind these fractals to start drawing them.

Where our previous examples with 2D graphing used relatively simple equations to draw each point on the screen, these fractals use much more complex equations based upon recursion:

$$Z_{n+1} = Z^2_n + C$$

In our code we unroll this recursion into a controlled series of iterations over the fractal function. Depending on how many iterations we use, a different image will be generated. If you want to work with these fractals, explore the fractal example for this week.

Cellular Automata

"A cellular automaton is a collection of "colored" cells on a grid of specified shape that evolves through a number of discrete time steps according to a set of rules based on the states of neighboring cells. The rules are then applied iteratively for as many time steps as desired. von Neumann was one of the first people to consider such a model, and incorporated a cellular model into his "universal constructor." Cellular automata were studied in the early 1950s as a possible model for biological systems (Wolfram 2002, p. 48). Comprehensive studies of cellular automata have been performed by S. Wolfram starting in the 1980s, and Wolfram's fundamental research in the field culminated in the publication of his book A New Kind of Science (Wolfram 2002) in which Wolfram presents a gigantic collection of results concerning automata, among which are a number of groundbreaking new discoveries."

Eric W. Weisstein. "Cellular Automaton." From MathWorld--A Wolfram Web Resource.
http://mathworld.wolfram.com/CellularAutomaton.html

In all of the examples we've looked at to date, our objects generally have existed in only one "state". They may move around with advanced behaviors/physics, but ultimately they have stayed the same type of object. More interesting results can be achieved by having entities that change/evolve over time. We examine cellular automata as our first example of a system of multiple objects with varying states. In the two examples below, each cell has one of two states (0 or 1, dead or alive, white or black, etc.). Researching further into CA systems, you might achieve varying results with multi-state systems as well as by applying the idea of "states" from CAs to systems of moving objects.

1D Automata

- The system exists as a row of cells (stored as an array of integers)
- Each cell (i.e. element in the array) is either "on" or "off", 0 or 1
- Each cell has two neighbors, one to the left and one to the right
- Each cell's state at time T+1 is determined by its own state and the state of its neighbors at time T. There are 8 possible combinations of a cell and its neighbors. A ruleset states what the new state will be based any given combination of a cell and it's neighbors:

```
[Left Cell, Current Cell, Right Cell] => New Cell State
```

For example, we might use the following ruleset:

```
[0, 0, 0] => 0
[0, 0, 1] => 1
[0, 1, 0] => 0
[0, 1, 1] => 1
[1, 0, 0] => 1
[1, 0, 1] => 0
[1, 1, 0] => 1
[1, 1, 1] => 0
```

You can explore these in the 1D Automata example for this week.

2D Automata

In the above 1D case, a cell's state at time T+1 is computed as a function of its own state and its neighbors' state at time T. The same is true for the 2D case, however, instead of a cell having only 2 neighbors, in two dimensions it will have 8 neighbors. In 1970, John Conway, building on the work on John von Neumann, developed the "Game of Life". The name refers to the fact that each cell is either alive or dead (in our code, again, represented as 0's and 1's.). The rules are as follows:

- Loneliness: If a cell is alive and has less than 2 live neighbors, it dies.
- Overpopulation: If a cell is alive and has more than 3 live neighbors, it dies.
- Reproduction: If a cell is dead, and it has exactly 3 live neighbors, it comes to life
- Statis: In all other cases, it stays as is.

Assume we have a cell represented as an integer "cell", with a total number of alive neighbors represented as an integer "neighbors". We can write code for these rules as follows:

```
if ( (cell == 1) && (neighbors <  2) ) {
      cell = 0;
} else if ( (cell == 1) && (neighbors >  3) ) {
      cell = 0;
} else if ( (cell == 0) && (neighbors == 3) ) {
      cell = 1;
} else {
      cell = cell;
}
```

For this to really work in code, however, we use a two-dimensional array to store information related to all the cells in the system.

2D Arrays

We know that an array keeps track of multiple pieces of information in a specific, linear order. However, the data associated with certain systems (a digital image, a board game, a "cellular automata") lives in two dimensions. To visualize this data, we need a multi-dimensional structure and we can do this by expanding the idea of an array beyond one dimension.

A 2 dimensional array is really nothing more than an array of arrays (a 3 dimensional array is an array of arrays of arrays.). In other words, if a 1 dimensional array looks like:

```
int[] myArray = {0,1,2,3};
```

a two-dimensional array looks like this:

```
int[][] myArray = {{0,1,2,3},{3,2,1,0},{3,5,6,1},{3,8,3,4}};
```

For our purposes, we want to think of the 2D array as a matrix:

```
int[][] myArray = {  {0, 1, 2, 3},
                     {3, 2, 1, 0},
                     {3, 5, 6, 1},
                     {3, 8, 3, 4}  };
```

To walk through every element of a one-dimensional array, we use a for loop:

```
int[] myArray = new int[10];
for (int i = 0; i < myArray.length; i++) {
      myArray[i] = 0;
}
```

For an N-dimensional array, in order to reference every element we must use N-nested loops:

```
int COLS = 10;
```

```
int ROWS = 10;
int[][] myArray = new int[COLS][ROWS];

for (int i = 0; i < COLS; i++) {
      for (int j = 0; j < ROWS; j++) {
            myArray[i][j] = 0;
      }
}
```

This should be familiar from our work with 2D graphing.

In the case of a 2D cellular automata, such as the "Game of Life", we need two 2D arrays. One stores the all the states at time T, and the other stores the states at time T+1. After each cycle of computing the next generation, the new system (T+1) becomes the old system (T), and we compute yet another new system (T+1).

The key to using two 2-dimensional arrays is storing the previous generation of the automata in one array, while storing the new state of the system in the other array. So we check every element in the array, count how many neighbors are alive, and apply the rules of life (Note, that this is a simplified version of the code used in the 2D Automata example for this week):

```
for (int x = 1; x < COLS-1;x++) {
      for (int y = 1; y < ROWS-1;y++) {

            int nb = 0;
            if (old_board[x-1][y-1] == 1)  { nb++; }    //top left
            if (old_board[x  ][y-1] == 1)  { nb++; }    //top
            if (old_board[x+1][y-1] == 1)  { nb++; }    //top right
            if (old_board[x-1][y  ] == 1)  { nb++; }    //left
            if (old_board[x+1][y  ] == 1)  { nb++; }    //right
            if (old_board[x-1][y+1] == 1)  { nb++; }    //bottom left
            if (old_board[x  ][y+1] == 1)  { nb++; }    //bottom
            if (old_board[x+1][y+1] == 1)  { nb++; }    //bottom right

            // The Rules
            if ( (old_board[x][y] == 1) && (nb <  2) ) {
                  new_board[x][y] = 0;
            } else if ( (old_board[x][y] == 1) && (nb >  3) ) {
                  new_board[x][y] = 0;
            } else if ( (old_board[x][y] == 0) && (nb == 3) ) {
                  new_board[x][y] = 1;
            } else {
                  new_board[x][y] = old_board[x][y];
            }
      }
}
```

To make the system work we swap old and new after each cycle, so that new becomes old, and we can make a "new" generation:

```
int[][] tmp = old_board;
old_board = new_board;
new_board = tmp;
```

Assignment

- Develop your own recursive system to generate complex, fractal-like shapes. What are the parameters of your system? Can you make the recursive drawing an object with instance variables associated with those parameters?
- Read through Chapter 6 of “The Computational Beauty of Nature” on L-Systems. Can you develop code that goes beyond our basic Finite State Machine?
- Redo the Game of Life example with object oriented programming. Create a class for each individual cell as well as one for the whole system itself.
- What types of systems can you model with Cellular Automata? Consider allowing cells to have more than 2 states and develop your own rules for changing states.
- Examine the Predator/Prey System described on p. 191 of Computational Beauty of Nature. Can you use the principle of cellular automata to model and visualize it?
- Consider the state of a cell to be its color. What types of image processing filters can you create using the principles of Cellular Automata?