

SOLD TO THE FINE

gumroad@garyvictoranderson.com



Playing With Chaos

Programming Fractals and Strange Attractors
in JavaScript



Keith
Peters

Contents

Chapter 1: Introduction	7
Fractals, Strange Attractors and Chaos Theory	7
Why JavaScript?	9
The Site and Code	11
Chaos Utils	12
Project Setup	14
Math	18
Summary	18
Chapter 2: What Is a Fractal	19
The Basics	19
The Sierpinski Gasket	20
The Sierpinski Code	26
The Koch Curve	34
The Koch Code	37
The Koch Snowflake	40
Paradoxes	44
Summary	45
Chapter 3: Symmetry and Regularity	46
Introduction	46
Simple Shape Fractals	47
Changing the Angles	54
Breaking Symmetry	56
Changing the Shape	64
Using Color	69
Getting Irregular	71
Tree Fractal	75
Breaking Symmetry	81
Altering Trunk Length	84

Breaking Regularity	89
Pythagorean Fractal	94
Breaking Symmetry	106
Breaking Regularity	108
Using Color	109
Summary	111
Chapter 4: Fractal Dimensions	112
Coastlines	112
The Strategy	113
The Code	116
Color	122
Islands	124
Taking Measurements	129
Fractal Dimensions	133
The Richardson Effect	133
Measuring Fractals	134
Landscapes	135
Summary	137
Chapter 5: Order from Chaos	139
The Chaos Game	139
The Strategy	140
The Code	144
Variations	151
Random Initial Points	156
Color	159
Iterated Function Systems	162
The Strategy	162
Transformation Matrices	163
Barnsley Fern	167

Variations	174
Colors	175
Other IFS Fractals	177
Random IFS Fractals	181
Summary	185
Chapter 6: Diffusion-Limited Aggregation	186
Background	186
Brownian Motion	187
The Strategy	188
The Code	189
DLA in Action	197
Variations	201
Summary	207
Chapter 7: The Mandelbrot Set	208
Introduction	208
Background	209
Complex Numbers	211
Graphing Complex Numbers	212
The Mandelbrot Formula	216
Mandelbrot Code	220
Zooming In	227
More Iterations!	234
Coloring Mandelbrots	236
Hacking	248
Summary	252
Chapter 8: Julia Sets and Fatou Dusts	253
Background	254
Theory	255
Definitions	256
Connectedness	257

The Code	261
Interactive Julia Rendering	272
Mandelbrot and Julia: The Connectedness Connection	281
Summary	290
Chapter 9: Chaos Theory and Strange Attractors	291
Chaos Theory	291
Population Models	293
The Butterfly Effect	303
Logistic Map and Bifurcation Diagram	304
Strange Attractors	313
The Lorenz Attractor	314
Lorenz 3D	321
Other Attractors	325
Summary	331
Chapter 10: L-Systems	332
Background	332
Intro to L-Systems	333
Coding the L-System	334
Coding the Graphics Interpreter	338
Advanced Commands: Push and Pop	347
Further Explorations	357
Summary	357
Chapter 11: Cellular Automata	358
Background	358
Wolfram Rules	359
More Robust Rules	363
More Neighbors	372
2D Cellular Automata	377
Classifying Cellular Automata	377
Coding 2D Cellular Automata	379

Summary	387
Chapter 12: Fractals in the Real World, and Resources	388
Fractals in the Real World	388
Uses for Fractals	393
Resources	395
Books	396
Websites	397
Summary	398

Chapter 1: Introduction

Fractals, Strange Attractors and Chaos Theory

Fractals, strange attractors and chaos theory are subjects that did not exist a few decades ago, but are now forming the basis for new breakthroughs in science, physics, image processing, medicine, cryptography, finance and other fields. These subjects are serious business, but this is not a serious book.

This book, as its name implies, is about PLAYING with fractals, strange attractors and chaos theory. By playing, I mean learning to write computer programs that will generate fractals and other related forms. Sure, I could simply write and release a few programs that you could play with and use to make pretty shapes. However, by teaching you the principles behind the shapes and the code that creates them, you will gain a far greater understanding of and appreciation for the subject. You will also be able to experiment with these concepts on a level that you could probably never achieve if you were just randomly tweaking some sliders in someone else's application.

There are already plenty of programs out there – both commercial and open source – that allow you to experiment with fractals. But few give you any understanding of what is going on behind the resulting pixels that appear on your screen. There are also more than a few books about the

science and math behind fractals. However, I've found that the vast majority of these books tackle the subject in a rather dry and academic way. To the casual reader who just wants to explore the subject and learn how to create some cool fractals, most of these books will seem far too heavy and impenetrable.

In Playing with Chaos, I have tried to make the subject approachable and fun. In doing so, I've tried to borrow from the spirit of two subjects: recreational mathematics and algorithmic/generative art.

Recreational mathematics describes an interest in mathematics as a purely enjoyable activity. This may seem like an odd concept, especially if you've ever struggled through a math class in school, but there are tons of popular games and puzzles out there based on math or logic. The Sudoku craze of the last few years indicates that there are more than a few recreational mathematicians in the world. There is something very satisfying about getting to a certain point in one of these games or puzzles and seeing the rest of it just fall into place logically. Personally, I get that same feeling when I successfully translate some mathematical formula into code and see the results on screen.

Algorithmic art and generative art are closely related concepts that use mathematical formulas or other algorithms to create images, sculpture, music or other artforms. Algorithmic and generative artists almost always utilize some kind of computer program to generate their art – and in fact, they most often write these programs themselves. Running a program that someone else created and calling the result your “art” is considered a stretch by most artists. But some applications, such as Context Free Art and Structure Synth, blur the lines between “program” and “programming language.”

By combining these two subjects, you get the challenge of figuring out a complex mathematical problem – with the payoff of a striking image or animation.

Note that I am not a scientist or professional mathematician. Therefore, my technical explanations may lack some of the depth that might be found in more serious tomes. Though I will try my best to avoid any incorrect explanations, it is entirely possible that a few errors will slip through. If you realize that the aim of this book is to help you understand and program your own fractals, and have fun doing so, I don't think you'll run into any insurmountable problems.

Why JavaScript?

Some may question why I chose JavaScript as the language to program the examples in the book. The answer is that there are multiple reasons.

First, JavaScript is free, open, cross-platform, doesn't require any special programming environment, and (with HTML5 running in modern browsers) is powerful enough to do most of the computation and rendering that will be needed. Chances are that the computer you are currently using has everything you need to create and run almost all of the examples in this book. All that is required is a text editor and a modern browser – any of the latest versions of Internet Explorer, Google Chrome, Firefox, Safari, or Opera.

Of course, there are more powerful and performant languages, and in a couple of cases, I had to leave out some examples I would have loved to include that were not possible to get running decently in JavaScript. Still, I think there is more than enough material in this book to keep you busy and interested for quite a while.

Second, notwithstanding a few oddities, JavaScript is a relatively easy language to pick up and become productive in very quickly. While I am assuming you have some basic programming experience, you won't need to have an extensive background in JavaScript itself to get the examples up and running. Chances are, no matter what programming language you might be familiar with, you will be able to get up to speed with the code in this book rather quickly, even if you've never touched JavaScript before.

However, if you are unfamiliar with JavaScript programming for the Web, it may be helpful to brush up on the HTML canvas object. In a nutshell, a canvas defines a rectangular area within the browser in which you can perform graphic operations such as drawing lines, circles, rectangles, etc. Actually, all the graphic operations are performed on a "context" object, which you get by calling `getContext("2d")` on the canvas itself. (On browsers that support WebGL, you can get a 3d context by passing in a different string, but that's beyond the scope of this book.) Once you get that far, the methods of the context that you will be using are mostly straightforward. If there's anything you don't fully understand about them, the Mozilla Developer Network has a useful reference here: <https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D>.

Finally, the intent of this book is not to supply you with a bunch of programs that you blindly copy and run. The idea is that you will understand what is happening in the code and why, and use the example programs as a starting point for your own explorations. As such, this can also include converting the code to the language and platform of your choice. JavaScript's syntax is simple enough that the examples should be readily translatable into most other currently popular languages.

In this spirit, I have avoided the use of any specialized libraries or frameworks, such as jQuery or others that would require a learning curve of their own, or make it more difficult to port your programs to another language. Each example will consist of a very simple HTML file with a bit of CSS and a couple of externally loaded JavaScript source files.

The Site and Code

The code for all of the examples shown in this book is available for you to download here: <http://playingwithchaos.net>. You can either type the code in as you go through this book, or simply download the files and run them. I've always found that typing the code in manually is far more effective in terms of learning.

Note that formatting code for insertion in a book is difficult under the best circumstances. In this case, this book can be read on a variety of electronic devices and with any user-specified font size. Thus, achieving perfect, readable code formatting within the book itself is virtually impossible. I've done my best to wrap long lines and indent the code to make it as readable as possible. But depending on your device and font size, the code displayed in this book might be pretty ugly. If so, you may find it easier to have the original source files open and follow along with them.

On that site, you'll also find higher resolution versions of all the images in this book, in color where the original images were rendered with color. In addition, you may find other information, errata or updates about the book or the subjects contained in it, as well as contact information to ask questions, submit errors or make suggestions.

Chaos Utils

Because the vast majority of the examples will require a good bit of the same setup code and often require similar functions over and over again, I've created a file that takes care of much of this boilerplate functionality. This file should not in any way be considered a "library" or "framework" but more of a collection of a few useful utility functions that you can use rather than writing them anew every time you need them. These are well-documented within the file itself, but you'll be using them often enough that most of the functions should become second nature after a couple of tries.

To prove it's nothing all that fancy, here it is in full:

```
var chaos = (function() {  
  
    return {  
  
        /**/  
         * Initializes chaos by finding the canvas on the  
         * page and resizing it to the  
         * full size of the browser.  
        */  
        init: function() {  
            this.canvas = document.getElementById("canvas");  
            this.context = this.canvas.getContext("2d");  
            this.setSize(window.innerWidth,  
                        window.innerHeight);  
        },  
  
        setSize: function(width, height) {  
            this.width = this.canvas.width = width;  
            this.height = this.canvas.height = height;  
        },  
  
        /**/  
         * Clears the canvas by filling it with the color  
         * specified, or erasing all  
         * pixels if no color is specified.  
        */  
        clear: function(color) {  
            if(color) {
```

```

        this.context.fillStyle = color;
        this.context.fillRect(0, 0,
                             this.width,
                             this.height);
    }
    else {
        this.context.clearRect(0, 0,
                             this.width,
                             this.height);
    }
},
/***
 * Pops up a bitmap image representation of the
 * canvas in a new window.
 */
popImage: function() {
    var win = window.open("", "Canvas Image"),
        src = this.canvas.toDataURL("image/png");

    win.document.write("<img src='" + src
        + "' width='" + this.width
        + "' height='" + this.height + "'/>");
}
};

}());

```

This creates a global variable called `chaos`. The value of this variable is defined with an immediately invoked function expression (IIFE). As its name implies, this function is immediately executed, and returns an object that is assigned to the `chaos` variable. This object has a few functions on it, and will be assigned various properties as well when these functions are executed.

The most important function is the `init` function. This should be called after the HTML file has loaded. This function will locate the canvas on the page, get the 2D drawing context, and set the size of the canvas to match the full size of the browser client area. It will then assign `canvas`, `context`, `width` and `height` to the `chaos` object so these can be used anytime you need them from your main project file.

There is also a `setSize` function. This is useful if you don't want the canvas to be the size of the screen. You can use this right after calling `init` to make any size canvas you want, up to several thousand pixels in size, which would be suitable for high-resolution prints.

A `clear` function clears all content from the canvas. If called with no argument, it will clear the canvas to a transparent color. If a valid color string is passed in, such as `"#FFCC99"`, `"rgb(1, 0, 0)"` or `"rgba(0, 0, 0, .5)"`, that color will be used to fill the canvas instead.

Because you'll be creating scores of interesting images as you move through the chapters of this book, it would be nice if there was a way to save them. You could just take a screen capture of your browser, but that's not very elegant. You can't save the image in a canvas directly, but you can grab the image data from it and assign it to the source of a new HTML image element. This is included as one of the useful functions in the `chaos` utils. Just

call `chaos.popImage()` any time to open a new window containing a bitmap image representation of your current canvas. You may have to disable the pop-up blocker in your browser for this to work. Once the image is showing, you can then usually right click on it and save it to your hard drive.

Project Setup

All of the examples in this book will follow the exact same setup, so let's go over it and create a sample project so everyone is on the same page.

First is the HTML file that pulls all the pieces together:

```

<!DOCTYPE html>
<html>
<head>
    <title>Sample Project</title>

    <style type="text/css">
        html, body {
            margin: 0px;
        }
        canvas {
            display: block;
        }
    </style>

    <script type="text/javascript" src="chaos.js">
    </script>
    <script type="text/javascript" src="sample.js">
    </script>

```

</head>

<body>

<div>

<canvas id="canvas"/>

</div>

</body>

</html>

It starts out with the HTML5 doctype declaration, then the HTML tag containing the head and body. The head section contains the project title and a brief bit of in-line CSS that prepares the canvas to be shown at the full size of the browser. Then it links to two scripts: the chaos utils file and the script file containing the code of the example itself. The body simply has a div that contains a canvas element with an id of “canvas”. You’ve already seen how the chaos utils `init` method finds that element and resizes it.

Now, let’s create a very simple `sample.js` file to see how that will fit into the whole setup:

```

window.onload = function() {
    init();

    function init() {
        chaos.init();

```

```

document.body.addEventListener("keyup",
    function(event) {
        switch(event.keyCode) {
            case 32: // space
                draw();
                break;

            case 80: // p
                chaos.popImage();
                break;

            default:
                break;
        }
    });
}

function draw() {
    var x = Math.random() * (chaos.width - 100),
        y = Math.random() * (chaos.height - 100),
        w = 20 + Math.random() * 100,
        h = 20 + Math.random() * 100,
        r = Math.floor(Math.random() * 256),
        g = Math.floor(Math.random() * 256),
        b = Math.floor(Math.random() * 256);
    chaos.context.fillStyle =
        "rgb(" + r + "," + g + "," + b + ")";
    chaos.context.fillRect(x, y, w, h);
}
}

```

The whole thing is contained within a function that is assigned to `window.onload` so the function is executed only after the whole HTML page has loaded and the canvas is ready to use. Using `window.onload` is not necessarily the most robust method of triggering code for a more complex application or web page, but won't cause any problems here and I wanted to keep things simple so I could focus on the code that creates the fractals.

Inside the `onload` function, other functions and variables can be defined. These will make up the core of each example. In this sample file, there is an `init` function defined, and this is called immediately. This is the pattern you'll see in all of

the examples in this book. The `init` function then calls `chaos.init()`, which grabs references to the canvas and its 2D context, resizes the canvas and stores its new width and height. It then sets up a listener for the `keyup` event. Within the handler function, it checks if the space bar has been pressed and calls another function, `draw`. It also checks for a press of the “p” key, in which case it calls `popImage`. You’ll be seeing very similar code to this throughout the examples.

The `draw` function gets a random x, y position based on `chaos.width` and `chaos.height`, and a random size and color to create a rectangle with. It then sets the `fillStyle` and calls `fillRect`, both on `chaos.context`. The result is in Figure 1.1.



Figure 1.1. Random boxes.

Math

As with most programming – and graphics programming in particular – there is some math involved in this book. I think Chapter 8 is the heaviest with all the use of complex numbers. Beyond that, many of the examples use basic trigonometry – mainly the sine and cosine functions. If you’re not up to speed on those, it might be a good idea to brush up on the subject. A great resource is the Khan Academy site. The first couple of videos in their basic trigonometry section should set you up nicely.<https://www.khanacademy.org/math/trigonometry/basic-trigonometry>

Summary

Well, that’s about it for this brief introduction. If you’ve mostly understood what’s going on here, you should be all set for the rest of the book, at least in terms of the code setup, and be able to concentrate on the concepts behind creating the fractals. Now, let’s dive into coding up some fractals!

Chapter 2: What Is a Fractal

The Basics

A good question to get out of the way early is, of course, “What is a fractal?”

I like to begin my answer with the statement that a fractal is a shape with certain special characteristics. Technically, you could say that a fractal is a mathematical or logical pattern with certain special characteristics, which, when plotted out graphically, will give you a shape with special characteristics. In this same way, you could describe a circle as a shape, or as the set of all points a certain distance away from a center point, which, when graphed, will give you a round shape. It's a bit of a semantic difference, but it does open up the idea of representing fractals in other ways, such as with sounds or music.

But, pushing all that to the back of the mind, let's go forward with the idea that a fractal is a shape. Unlike many shapes such as squares, circles or even polygons, which can all be defined by relatively simple rules, a fractal is a complex shape made up of other shapes. That's its first special characteristic.

The next interesting thing about fractals is that if you take each one of these sub-shapes from which a fractal is made, you will see that they are all copies of the shape as a whole.

This is known as self-similarity and is perhaps the most obvious characteristic you will come across in your investigation of fractals.

But, if each sub-shape is a copy of the whole shape, then each sub-shape must also be composed of sub-sub-shapes, which are also copies of the whole. In addition, those sub-sub-shapes must be made up of sub-sub-sub-shapes, again copies of the whole. This self-similarity is theoretically infinite. That's the next fascinating fact about fractals.

So you have complexity, self-similarity and infinity. These are fascinating shapes indeed!

The Sierpinski Gasket

It would be helpful to see an example. Let's start with the triangle in Figure 2.1.

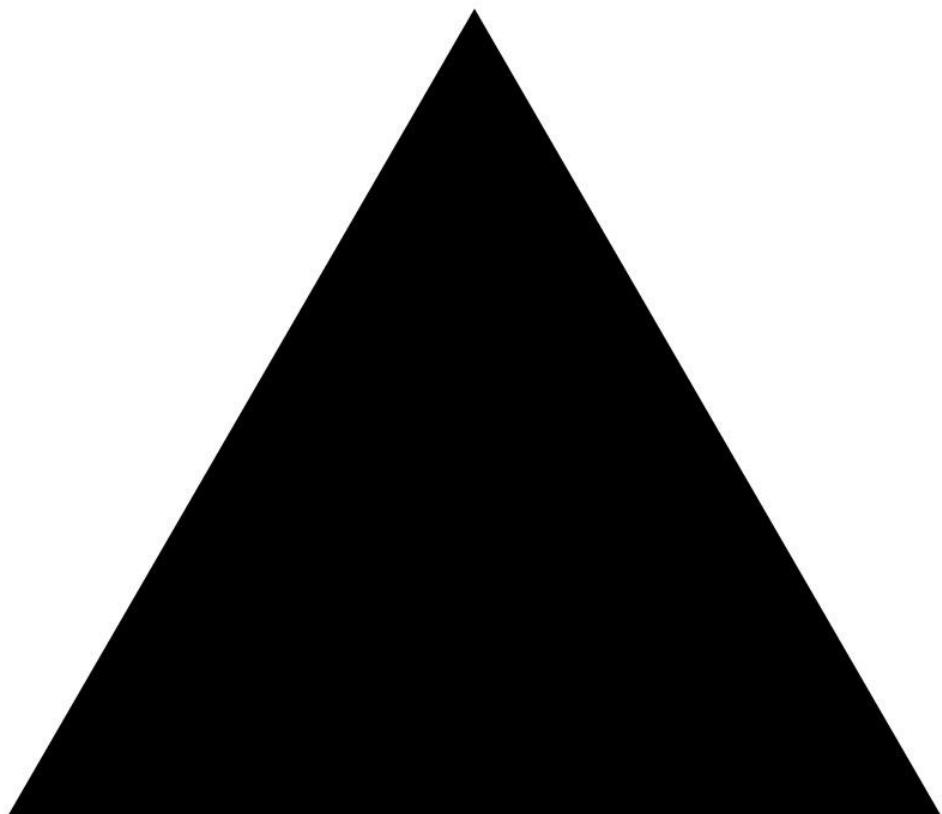


Figure 2.1. A triangle.

Now, let's transform this simple triangle into a triangle that is made of other triangles as in Figure 2.2.

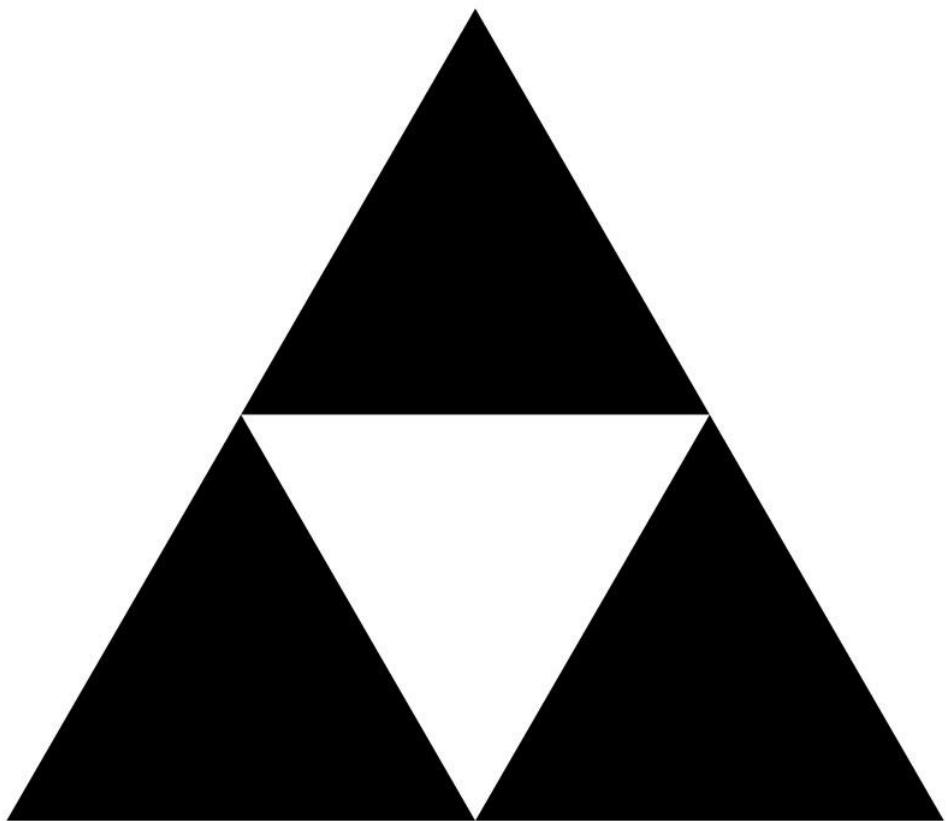


Figure 2.2. A triangle made of triangles.

So you have three smaller triangles, which together make up a larger triangle. Each is roughly a copy of the whole shape. Now, let's iterate this again, and you have Figure 2.3, with each smaller triangle composed of yet smaller ones.

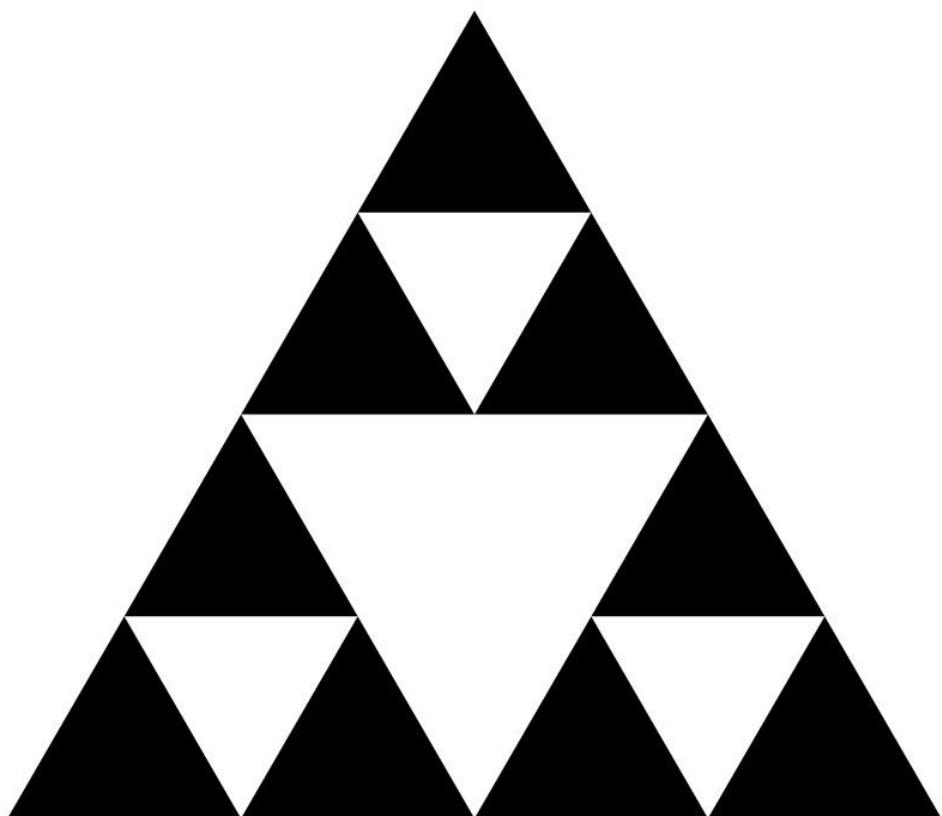


Figure 2.3. Another iteration.

And again in Figure 2.4.

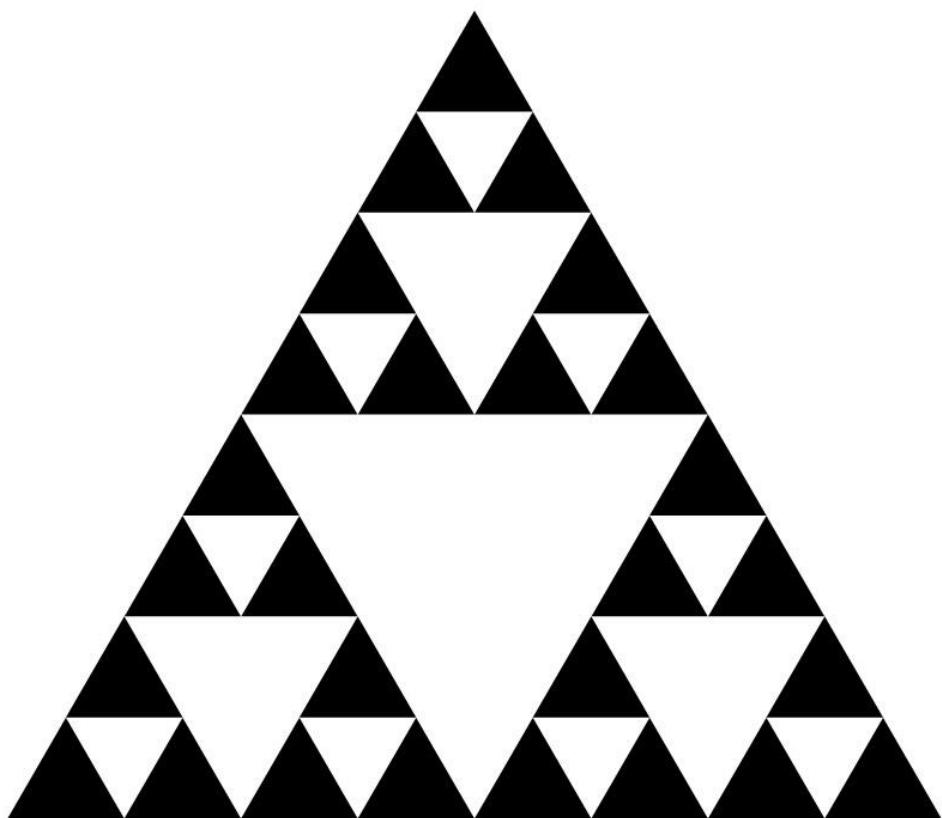


Figure 2.4. Yet more triangles.

If you continue this infinitely, you have in Figure 2.5 the fractal known as the Sierpinski gasket, named after the Polish mathematician Waclaw Sierpiński, who first brought this figure to light.

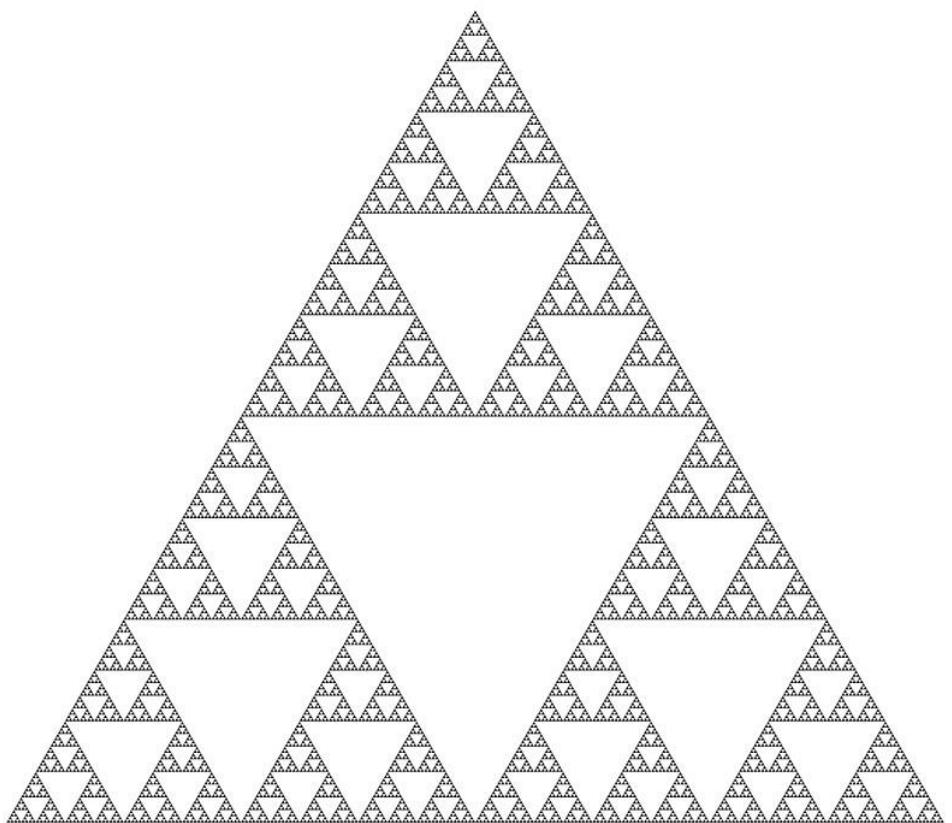


Figure 2.5. The Sierpinski gasket.

Earlier, I said that fractals are THEORETICALLY infinite. This theoretical distinction is important. When you render out fractal shapes to a computer screen or print them on paper, you very quickly run into the limitation of resolution. After just a few iterations, you are trying to draw a shape that is smaller than a single pixel or blob of ink. At that point,

it doesn't make sense to iterate anymore. You're not going to see any difference.

Of course, if you are working on a computer, you can programmatically zoom in. If you consider that the large triangle in the sample image is one unit on each side, then it is zoomed in already to about 600x, making it 600 pixels on each side. If you zoomed in to 1200x, then each shape would be twice as large. Thus, you could zoom in twice as far. You could continue to zoom in this way, making the image so large that you could only see a small portion of it on the screen at any one time. In this way, you could see far more detail at higher iterations. But at some point, you would still hit the limit of how far you could zoom. The numbers would just get too large to hold in a single memory value. This is not a theoretical problem at all. You will definitely reach that limit while zooming into some of the fractals you create in this book.

Later, you'll see that fractal structures exist in abundance in the real world. These also eventually lose resolution, at the very least when they get down to the cellular or atomic level.

The Sierpinski Code

Let's look at the code that creates the Sierpinski gasket images. The HTML file will be exactly the same as the sample app you created in Chapter 1, other than the fact that the title is changed and it loads `sierpinski.js` instead of `sample.js`. So all I need to show you is that new `sierpinski.js` file. I'll just dump it out here and then go through it all, piece by piece.

```
window.onload = function() {  
    var size,
```

```

maxDepth = 0;

init();

function init() {

    chaos.init();

    size = chaos.height * 0.5;

    draw();

    document.body.addEventListener("keyup",
        function(event) {
            console.log(event.keyCode);
            switch(event.keyCode) {
                case 32: // space
                    maxDepth += 1;
                    draw();
                    break;

                case 80: // p
                    chaos.popImage();
                    break;

                default:
                    break;
            }
        });
}

function draw() {
    chaos.clear();
    chaos.context.save();
    chaos.context.translate(chaos.width * 0.5,
                           chaos.height * 0.6);
    chaos.context.scale(size, size);
    drawTriangle(maxDepth);
    chaos.context.restore();
}

function drawTriangle(depth) {
    var angle = -Math.PI / 2;
    if(depth === 0) {
        chaos.context.beginPath();

```

```

// move to top point of triangle
chaos.context.moveTo(Math.cos(angle),
                     Math.sin(angle));
angle += Math.PI * 2 / 3;

// draw line to lower right point
chaos.context.lineTo(Math.cos(angle),
                     Math.sin(angle));

// draw line to final point
angle += Math.PI * 2 / 3;
chaos.context.lineTo(Math.cos(angle),
                     Math.sin(angle));

// fill will close the shape
chaos.context.fill();
}

else {
    // draw the top triangle
    chaos.context.save();
    chaos.context.translate(Math.cos(angle) * 0.5,
                           Math.sin(angle) * 0.5);
    chaos.context.scale(0.5, 0.5);
    drawTriangle(depth - 1);
    chaos.context.restore();

    // draw the lower right triangle
    angle += Math.PI * 2 / 3;
    chaos.context.save();
    chaos.context.translate(Math.cos(angle) * 0.5,
                           Math.sin(angle) * 0.5);
    chaos.context.scale(0.5, 0.5);
    drawTriangle(depth - 1);
    chaos.context.restore();

    // draw the lower left triangle
    angle += Math.PI * 2 / 3;
    chaos.context.save();
    chaos.context.translate(Math.cos(angle) * 0.5,
                           Math.sin(angle) * 0.5);
    chaos.context.scale(0.5, 0.5);
    drawTriangle(depth - 1);
    chaos.context.restore();
}
}

```

This code starts by calling the `init` function, which calls `chaos.init`, exactly what the sample file in Chapter 1 did. It then sets the `size` variable to 0.5 (the height of the canvas). This is calculated to make the triangle fill a good part of the canvas. It then calls the `draw` function and sets up some key listeners, which you'll see shortly. First, let's look through the `draw` function:

```
function draw() {  
    chaos.clear();  
    chaos.context.save();  
    chaos.context.translate(chaos.width * 0.5,  
                           chaos.height * 0.6);  
    chaos.context.scale(size, size);  
    drawTriangle(maxDepth);  
    chaos.context.restore();  
}
```

The `draw` function clears the canvas and calls `chaos.context.save()`. You'll be creating the image using 2D context transformations, translating the drawing context and scaling it before drawing each triangle, rather than drawing them in different places. Calling `save` saves the current state of the context, so you can restore it after drawing.

The next statement translates, or moves, the drawing context so that the 0, 0 point will now be centered horizontally and a bit more than halfway down the page vertically. This will be the center of the fractal you will draw.

Next, `drawTriangle` is called, passing in `maxDepth`, which is 0 for now.

Finally, `restore` is called on the context, resetting the context to its original transformation state.

Let's look at the first part of `drawTriangle` next:

```
function drawTriangle(depth) {  
    var angle = -Math.PI / 2;  
    if(depth === 0) {
```

```

chaos.context.beginPath();

// move to top point of triangle
chaos.context.moveTo(Math.cos(angle),
                     Math.sin(angle));
angle += Math.PI * 2 / 3;

// draw line to lower right point
chaos.context.lineTo(Math.cos(angle),
                     Math.sin(angle));

// draw line to final point
angle += Math.PI * 2 / 3;
chaos.context.lineTo(Math.cos(angle),
                     Math.sin(angle));

// fill will close the shape
chaos.context.fill();
}

```

The `depth` value passed in is 0 the first time around, so this is the part of the function that will be executed. All that's being done here is using the canvas drawing API to draw a triangle on the canvas. This code block starts by calling `beginPath`, and then it calculates the three points of the triangle using sine and cosine with the correct angles. It uses these to do a `moveTo` and two `lineTo`s and finishes up by calling `fill` to fill the triangle with the default solid black color. By itself, this would draw a triangle just larger than a single pixel. But because the context was scaled up based on the `size` variable, when you execute the code, you'll see a large black triangle almost filling the canvas.

To see what happens next, let's look at those key handlers:

```

document.body.addEventListener("keyup",
  function(event) {
    switch(event.keyCode) {
      case 32: // space
        maxDepth += 1;
        draw();
        break;

      case 80: // p
        chaos.popImage();
    }
  }
)

```

```

        break;

    default:
        break;
    }
});
```

The important one is key code 32, the space bar. This increments that `maxDepth` variable and calls `draw` again. As before, this will clear the context, translate the context to center screen, scale it up and call `drawTriangle` once again, this time passing in the new `maxDepth` value of 1.

This time around, the `if` statement in `drawTriangle` will be false, and it will fall into the second part of the function:

```

else {
    // draw the top triangle
    chaos.context.save();
    chaos.context.translate(Math.cos(angle) * 0.5,
                           Math.sin(angle) * 0.5);
    chaos.context.scale(0.5, 0.5);
    drawTriangle(depth - 1);
    chaos.context.restore();

    // draw the lower right triangle
    angle += Math.PI * 2 / 3;
    chaos.context.save();
    chaos.context.translate(Math.cos(angle) * 0.5,
                           Math.sin(angle) * 0.5);
    chaos.context.scale(0.5, 0.5);
    drawTriangle(depth - 1);
    chaos.context.restore();

    // draw the lower left triangle
    angle += Math.PI * 2 / 3;
    chaos.context.save();
    chaos.context.translate(Math.cos(angle) * 0.5,
                           Math.sin(angle) * 0.5);
    chaos.context.scale(0.5, 0.5);
    drawTriangle(depth - 1);
    chaos.context.restore();
}
```

Here, you see three chunks of code that are almost identical. This would be a great opportunity to optimize with a loop, but I kept it unrolled for clarity.

Each chunk starts out with the `angle` variable defined at the beginning of the function, `-Math.PI / 2` in radians, or -90 degrees. It uses this angle and the same trig used for drawing to perform a translation of 0.5 (times the current scale, of course). This puts the 0, 0 point right in the center of where the top triangle will be drawn. It then scales the context by 0.5 and calls `drawTriangle`, passing in `depth - 1`.

A method calling itself is known as recursion, something you'll see often in fractal code. This time, `depth` will be 0, so a triangle will be drawn. But due to the second level of transforms just made, it will be smaller and towards the top of the screen.

Finally, `restore` is called. Because `save` was called at the beginning of the block, the context is back to where it started at the beginning of the function. This code is repeated twice more with different angles, drawing the bottom two triangles. The result is in Figure 2.2.

When the space bar is pressed again, the cycle is run through again, but now `maxDepth` will be 2, so the function will run recursively twice before actually drawing anything. This will make nine quarter-sized triangles, spread out in the same formation as in Figure 2.3.

Play around with this example code. Change some values and see what happens. Throw in some rotation maybe. A call to `Math.random()` instead of constant values. Some different colors based on depth. Experimenting like that will help you understand the code, and you might come up with something unique.

Figures 2.6 and 2.7 are a couple that I came up with. Figure 2.6 was created by changing some of the angles to break symmetry, while Figure 2.7 (one of my favorites) randomizes the placement of each new triangle. You'll find the code for these images in the book's downloadable code.

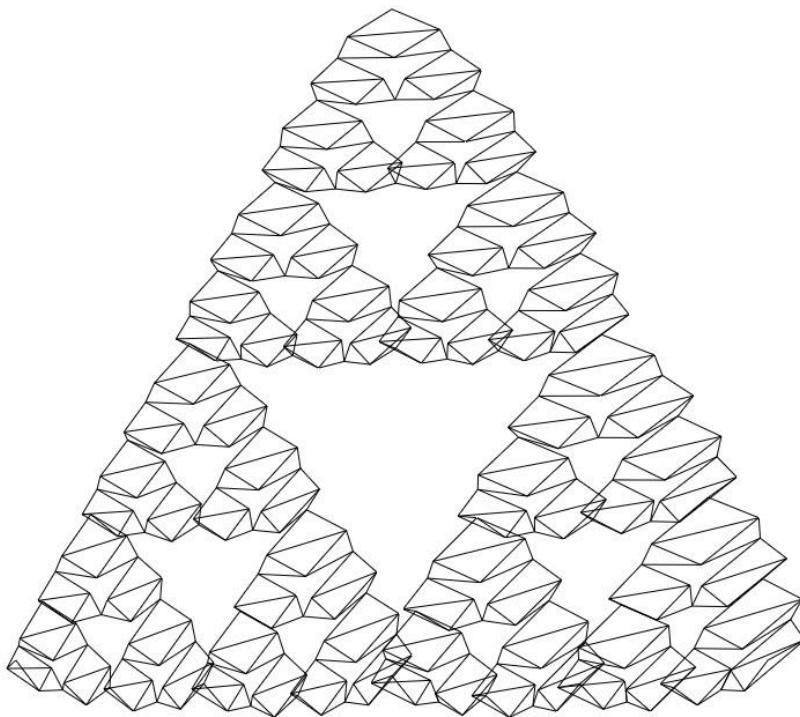


Figure 2.6. Non-symmetrical Sierpinski.

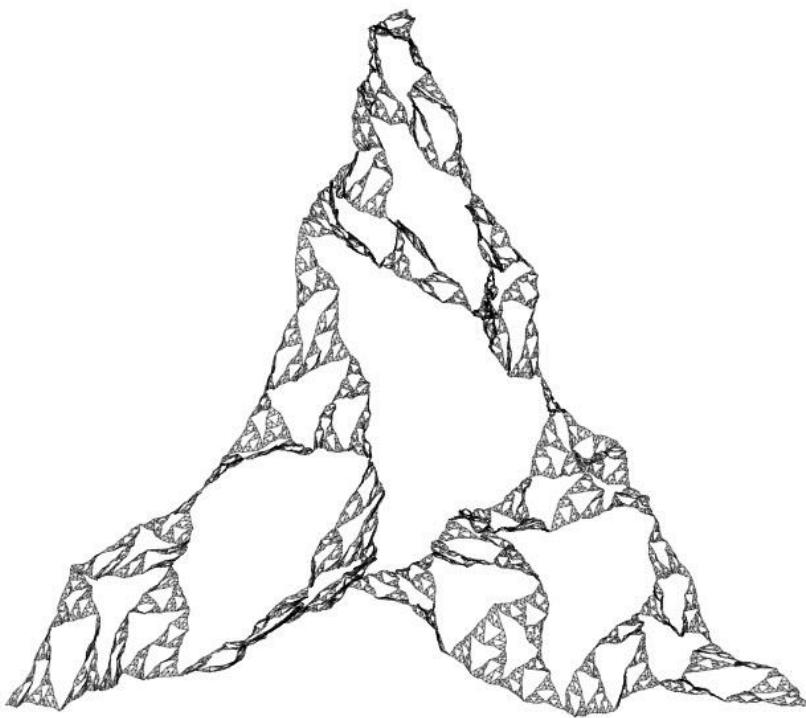


Figure 2.7. Random Sierpinski.

The Koch Curve

Let's look at another famous fractal, the Koch curve, named for the Swedish mathematician Helge von Koch, who first described it. The curve starts out as the sequence of four straight line segments, as seen in Figure 2.8.

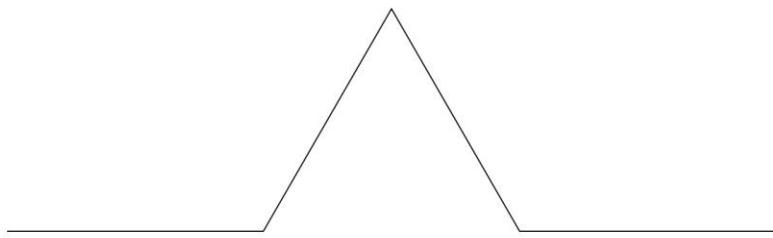


Figure 2.8. The Koch curve.

Each segment is the same length, and the horizontal space in the center is also the same length, making the shape in the middle an equilateral triangle missing its bottom.

The Koch curve is a type of fractal known as a shape replacement fractal. Essentially, you'll be replacing each segment of the shape with a smaller copy of the whole shape itself. This is the same thing that occurred in the Sierpinski gasket, but it's a bit more obvious here. After the first iteration, you'll see Figure 2.9.

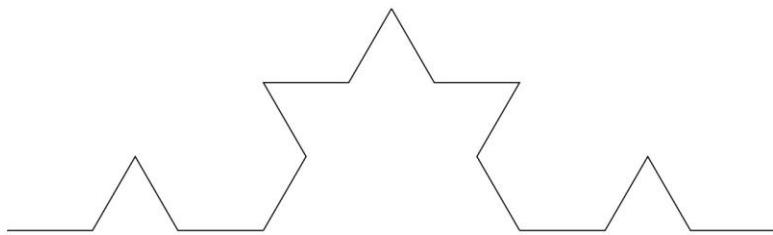


Figure 2.9. The Koch curve, iteration 1.

You now have a shape with 16 smaller line segments. Let's iterate again. The result is in Figure 2.10.

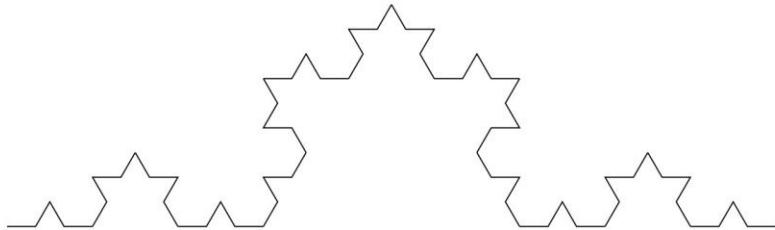


Figure 2.10. The Koch curve, iteration 2.

And again in Figure 2.11.

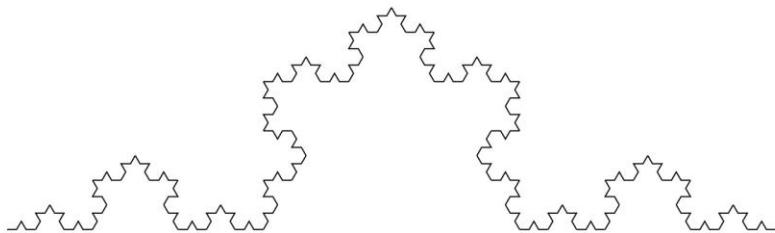


Figure 2.11. The Koch curve, iteration 3.

Continued to infinity, this gives you the Koch curve, Figure 2.12.

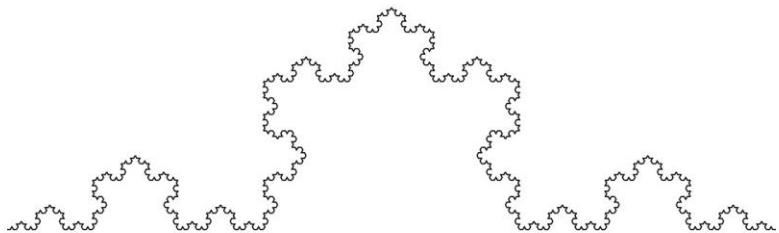


Figure 2.12. The Koch curve in full.

Of course, like the Sierpinski gasket, you can only go a few iterations before you're drawing lines at a sub-pixel level and you stop seeing a difference.

The Koch Code

Here again, I'll throw all the code at you and then look at it piece by piece.

```
window.onload = function() {
    var maxDepth = 0;

    init();

    function init() {

        chaos.init();

        draw();

        document.body.addEventListener("keyup",
            function(event) {
                console.log(event.keyCode);
                switch(event.keyCode) {
                    case 32: // space
                        maxDepth += 1;
                        draw();
                }
            }
        );
    }
}
```

```

        break;

    case 80: // p
        chaos.popImage();
        break;

    default:
        break;
    }
});

}

function draw() {
    var p0 = {
        x: chaos.width * 0.1,
        y: chaos.height * 0.75
    }

    var p1 = {
        x: chaos.width * 0.9,
        y: chaos.height * 0.75
    }

    chaos.clear();
    chaos.context.lineWidth = 2;

    koch(p0, p1, maxDepth);
}

function koch(p0, p1, depth) {
    var dx = p1.x - p0.x,
        dy = p1.y - p0.y,
        // the length of the main segment:
        dist = Math.sqrt(dx * dx + dy * dy),
        // the length of each sub-segment:
        unit = dist / 3,
        // the angle of the main segment:
        angle = Math.atan2(dy, dx),
        pa, pb, pc;

    // calculate the three intermediate points:
    pa = {
        x: p0.x + Math.cos(angle) * unit,
        y: p0.y + Math.sin(angle) * unit
    };
    pb = {

```

```
        x: pa.x + Math.cos(angle - Math.PI / 3) * unit,  
        y: pa.y + Math.sin(angle - Math.PI / 3) * unit  
    };  
    pc = {  
        x: p0.x + Math.cos(angle) * unit * 2,  
        y: p0.y + Math.sin(angle) * unit * 2  
    };  
  
    if(depth === 0) {  
        chaos.context.beginPath();  
        chaos.context.moveTo(p0.x, p0.y);  
        chaos.context.lineTo(pa.x, pa.y);  
        chaos.context.lineTo(pb.x, pb.y);  
        chaos.context.lineTo(pc.x, pc.y);  
        chaos.context.lineTo(p1.x, p1.y);  
        chaos.context.stroke();  
    }  
    else {  
        koch(p0, pa, depth -1);  
        koch(pa, pb, depth -1);  
        koch(pb, pc, depth -1);  
        koch(pc, p1, depth -1);  
    }  
}  
}
```

As before, the HTML file only changes the name and includes the `koch.js` file.

You can see that this looks similar in structure to the Sierpinski example. The same strategy is being used: iterate the `koch` function until `depth` reaches zero, then do the actual drawing.

The `koch` function is also passed two points, `p0` and `p1`. These are the start and end of the current line segment being iterated. The initial two points are calculated to be near the left and right edge of the screen and about 3/4 of the way down. From there `koch` calculates three intermediate points, `pa`, `pb` and `pc`, as you can see in Figure 2.13. These, with the original two, make up the four new segments to replace that single line segment.

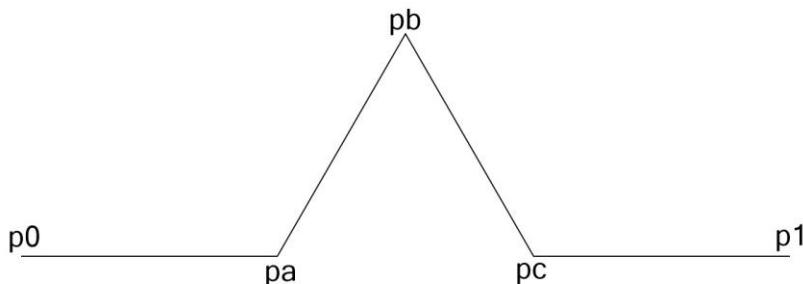


Figure 2.13. Calculating three intermediate points.

If `depth` is zero, the function just draws the four lines and it's done. If not, it calls `koch` recursively, once for each of the new segments, reducing the depth value it passes in.

As with the Sierpinski experiment, play around with this one and see what you can do with it. Change the angles or lengths, randomize some parameters or maybe even create more than three intermediate points to make a different shape altogether.

The Koch Snowflake

There's another neat thing you can do with a trio of Koch curves. If you arrange them in a triangle, you get the shape you see in Figure 2.14.

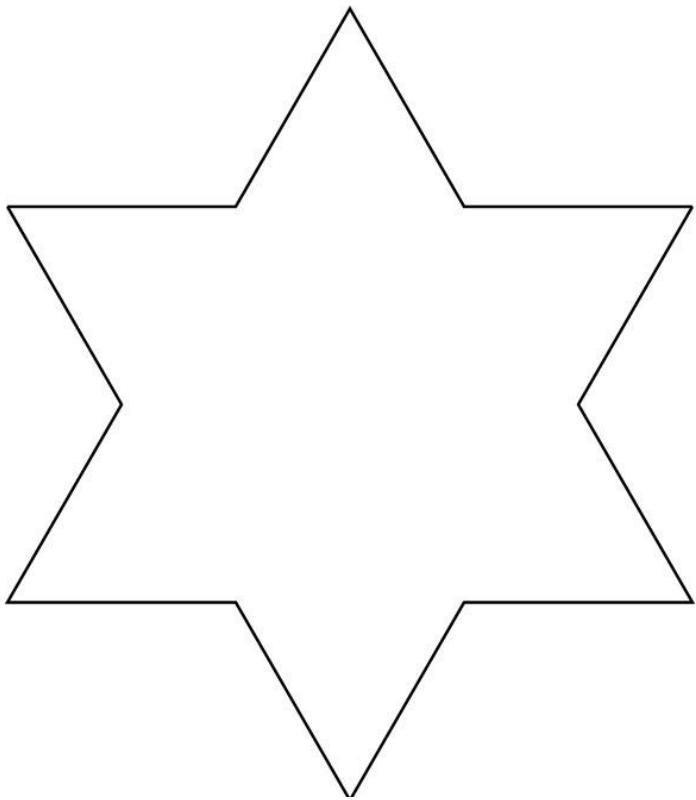


Figure 2.14. The Koch Snowflake.

This is known as the Koch snowflake. It doesn't look like much of a snowflake just yet, but iterate it once and you get Figure 2.15.

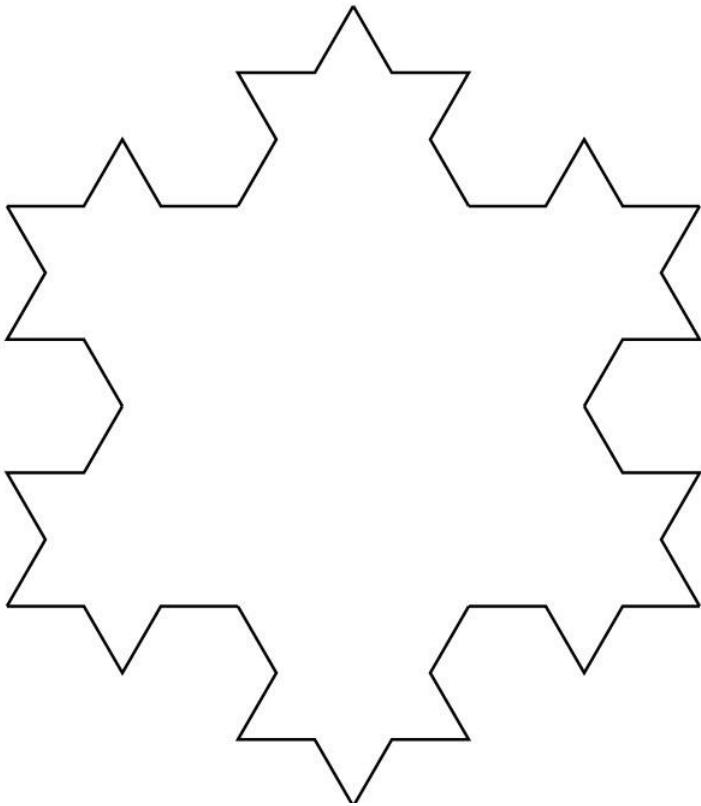


Figure 2.15. The Koch snowflake, iteration 1.

And a few more times gives you the snowflake in all its glory, Figure 2.16.

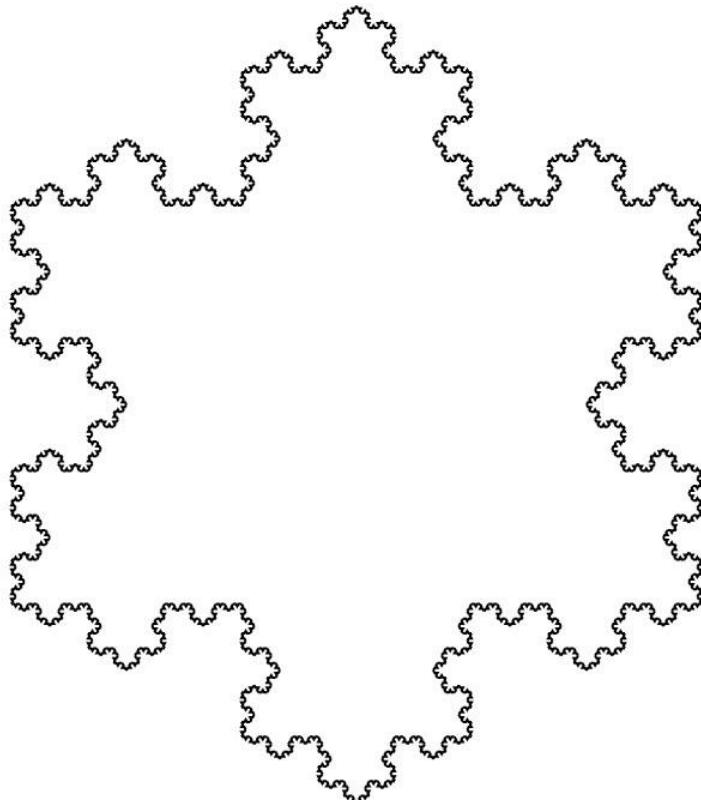


Figure 2.16. The Koch snowflake in full.

The only part of the code that changed here is in the `draw` function:

```
function draw() {  
    var p0 = {  
        x: chaos.width * 0.32,  
        y: chaos.height * 0.28  
    }  
  
    var p1 = {  
        x: chaos.width * 0.68,  
        y: chaos.height * 0.28  
    }  
  
    var p2 = {  
        x: p1.x + Math.cos(Math.PI * 2 / 3)  
            * (p1.x - p0.x),  
        y: p1.y + Math.sin(Math.PI * 2 / 3)
```

```
        * (p1.x - p0.x)  
    }  
  
    chaos.clear();  
    chaos.context.lineWidth = 2;  
  
    koch(p0, p1, maxDepth);  
    koch(p1, p2, maxDepth);  
    koch(p2, p0, maxDepth);  
}
```

Whereas you originally made only two points and called the `koch` function once in `draw`, here you are making three points forming a triangle and calling `koch` three times with each possible pair of points.

Paradoxes

Earlier I mentioned that fractals are theoretically infinite. But, when you start considering their infinite nature, some odd things happen. Let's take the length of the Koch curve for example. Taking the original four-segment curve you began with, and assigning a unit length of 1 to each segment, it's easy to see that the length of the curve as a whole is 4.

Now, when you iterate each segment, you replace it with four smaller segments, each of which is $1/3$ the length of the original. Thus, each segment is $4/3$ the size of the original. And, after a single iteration, the length of the curve as a whole is also $4/3$ its original size, or about 5.333. It has grown by $1/3$. Iterating again, it grows by another $1/3$, making it 7.111. It continues to grow this way with each iteration: 9.48, 12.64, 16.85, etc. After 20 iterations, the length of the curve is just over 946.

If you were to continue the iteration infinitely, the length of the line would increase to infinity, yet it would still fit nicely on your computer screen.

If that's not odd enough, consider the Koch snowflake. The three curves become the perimeter of the snowflake shape, and realize that the area of that shape does not increase infinitely. If you were to draw a square around it, that square would have a finite area and the snowflake would always remain within the square. So you have the seeming paradox of an infinitely long curve defining a shape with a finite area.

Then take the good old Sierpinski gasket. Each time you iterate, you're making three new triangles for each old one. But you're also subtracting 1/4 of the total area of the shape. Continued infinitely, you have an infinite number of triangles taking up zero space! These things keep me up at night.

Summary

You now have a start on understanding and creating some basic fractals. In the next chapter, you'll be creating several new types of fractals and exploring the concepts of symmetry and regularity. Altering these two factors can bring a lot of variety to your fractal shapes and create shapes that begin to resemble what you see in nature.

Chapter 3: Symmetry and Regularity

Introduction

In this chapter, you'll learn about the concepts of symmetry and regularity, and how altering these factors in different ways can lead to much more interesting and natural-looking fractal shapes.

Remember from Chapter 2 that one of the characteristics of a fractal is self-similarity, that each of the parts making up a fractal shape is a copy of the shape as a whole. In the examples you have built so far, this was exactly the case. The Sierpinski gasket replaces each triangle with three smaller triangles in the same configuration. The Koch curve and snowflake both replace each line segment with a copy of the whole curve. In other words, they are very regular. Every copy is exact.

All of these shapes are also symmetrical. They all have left-right symmetry; you can flip them horizontally and have the same picture. The Sierpinski gasket and Koch snowflake also have radial symmetry; you can rotate the images 120 degrees and they would look the same.

But symmetry and regularity are not mandatory characteristics of a fractal. In fact, if you experimented with the code in Chapter 2, you may have already come up with

some irregular, non-symmetrical shapes on your own. Let's explore the concept more deeply.

Simple Shape Fractals

The first fractal in this chapter is one I came up with on my own years ago. Not that it's all that original or complex. In fact, it's just a loose version of the iterated function systems you're going to see in Chapter 5. But I'm getting ahead of myself here.

You'll see a resemblance to the Sierpinski gasket code here, but the setup is a bit more flexible and allows for more variation. This gives you the ability to create not only Sierpinski-like shapes, but other unique shapes as well.

Here's the strategy:

1. Choose a shape. Draw a large instance of that shape in the middle of the screen.
2. Choose three angles, a distance and a scaling factor.
3. Rotate to the first angle. Move out on the rotated axis to the chosen distance. Scale down and draw another instance of the shape at that position.
4. Repeat step 3 for the other two angles.
5. Repeat steps 3 and 4 for the three new shapes that you just created, using the same angles and distances but scaled down.
6. Iterate steps 3 and 4 again for all the new shapes you created in step 5.

7. Continue iterating until you reach a chosen maximum depth. Usually five to eight iterations is plenty.

Choosing a circle for a shape, 0, 120 and 240 for angles, and sensible values for distance and scaling, you get Figure 3.1 after a single iteration.

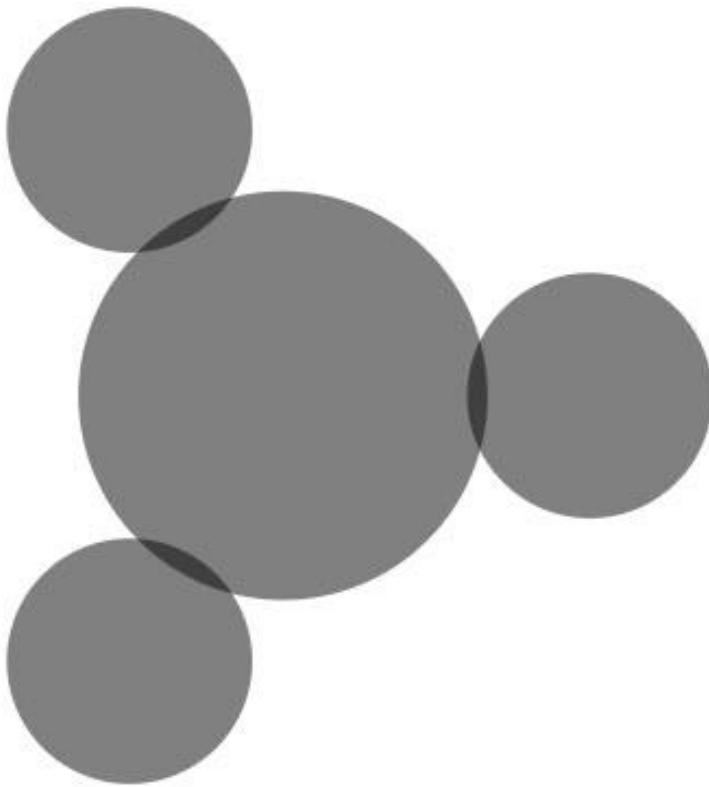


Figure 3.1. A single iteration.

As you can see, the large center circle was drawn, and then the context was rotated 0 degrees, translated a certain distance (to the right) and scaled down. Then, another circle was drawn at that point. The context was then rotated to 120 and 240 degrees and circles were drawn there as well.

After a second iteration, you get Figure 3.2.

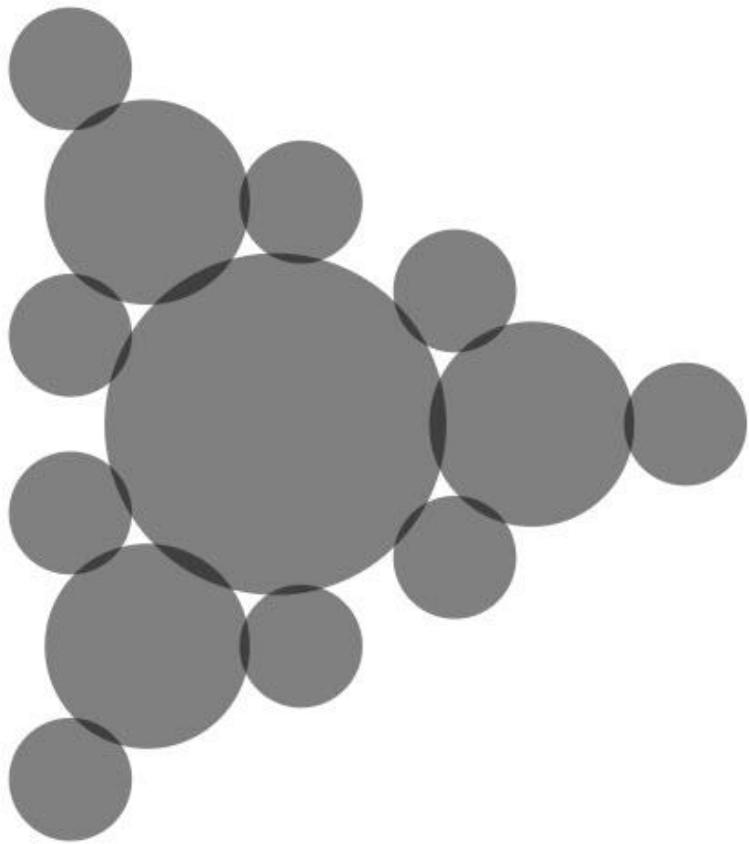


Figure 3.2. Two iterations.

You can now see that three circles have been drawn around each smaller circle. After a third iteration, you'll see Figure 3.3.

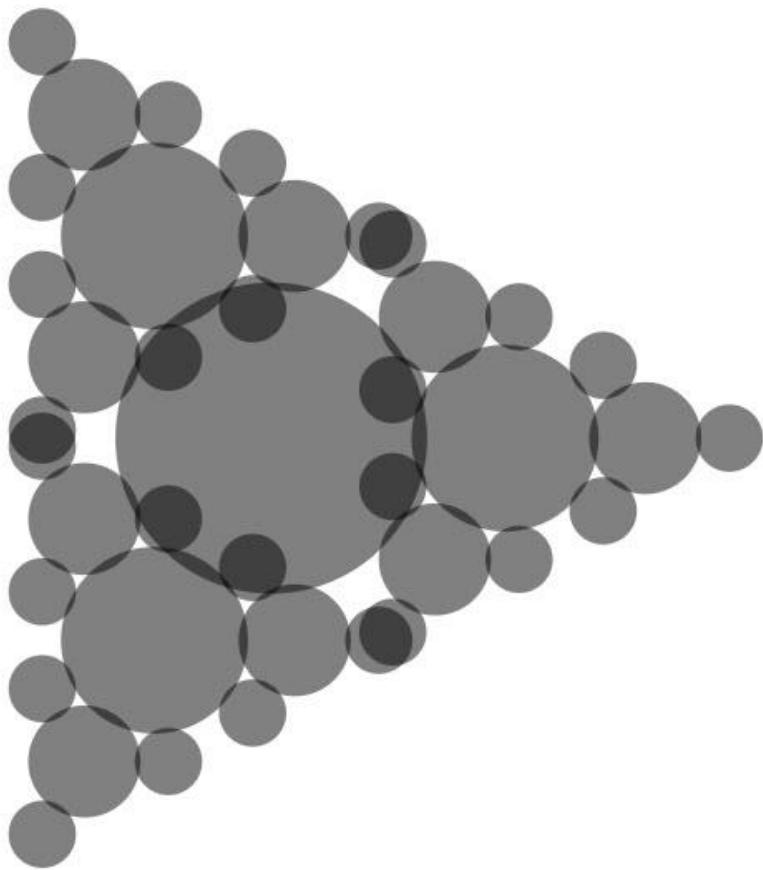


Figure 3.3. Three iterations.

Finally, after several more iterations, you get Figure 3.4.

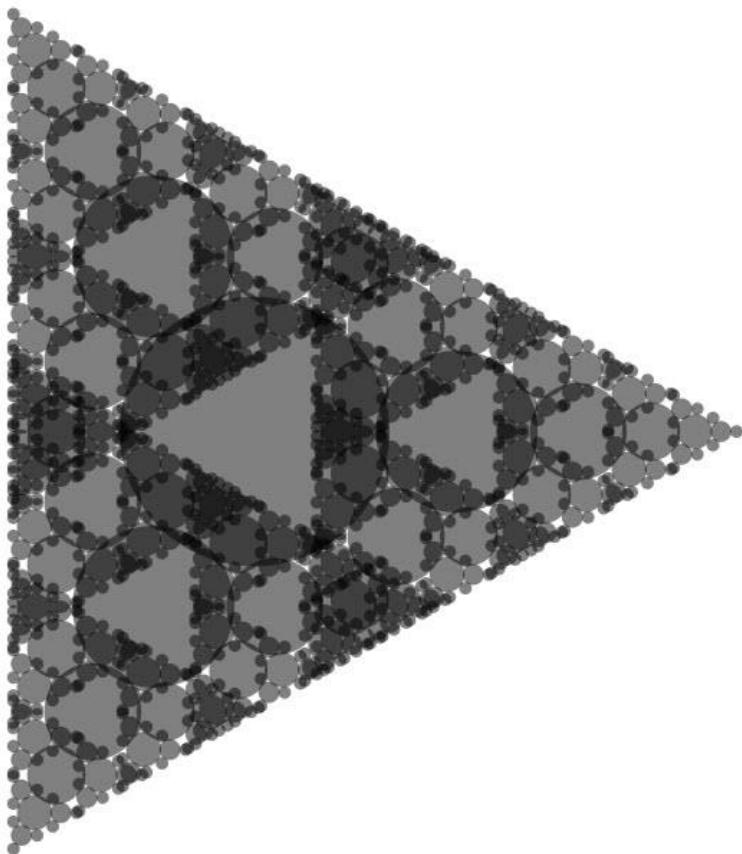


Figure 3.4. Almost Sierpinski.

Oddly enough, with circles alone, the resulting fractal has formed a triangle with lots of inner triangles, looking suspiciously like a Sierpinski gasket.

Now let's look at the code that created this to see how it can be altered to make something more interesting:

```
window.onload = function() {
    var maxDepth = 0,
        numShapes = 3,
        angles = [
            0,                      // 0 degrees
            Math.PI * 2 / 3,        // 120 degrees
            Math.PI * 4 / 3         // 240 degrees
        ],
        ...
```

```
size = 0,
dist = 0,
scaleFactor = .6;

init();

function init() {

    chaos.init();

    size = chaos.height / 8;
    dist = size * 1.5;

    draw();

    document.body.addEventListener("keyup",
        function(event) {
            console.log(event.keyCode);
            switch(event.keyCode) {
                case 32: // space
                    maxDepth += 1;
                    draw();
                    break;

                case 80: // p
                    chaos.popImage();
                    break;

                default:
                    break;
            }
        });
}

function draw() {
    chaos.clear();
    chaos.context.save();
    chaos.context.translate(chaos.width * 0.5,
                           chaos.height * 0.5);
    drawShape();
    iterate(maxDepth);
    chaos.context.restore();
}

function iterate(depth) {
    for(var i = 0; i < numShapes; i += 1) {
```

```

        chaos.context.save();
        chaos.context.rotate(angles[i]);
        chaos.context.translate(dist, 0);
        chaos.context.scale(scaleFactor,
                            scaleFactor);
        drawShape();
        if(depth > 0) {
            iterate(depth - 1);
        }
        chaos.context.restore();
    }
}

function drawShape() {
    chaos.context.fillStyle = "rgba(0, 0, 0, .5)";
    chaos.context.beginPath();
    chaos.context.arc(0, 0, size,
                      0, Math.PI * 2, false);
    chaos.context.fill();
}
}

```

This code is in the file `ssf1.js`, which is loaded by `ssf1.html`. I hope that by now the structure of these programs is becoming familiar to you. You'll be seeing more examples with the same format.

First, the code defines variables for the number of shapes, angles, size, distance and scale factor. The size and distance are dependent on the size of the canvas, so they need to be assigned after `chaos.init` is called.

The `draw` function is nearly identical to the other examples you've seen. It translates the context to the center of the screen, calls `drawShape` and then `iterate` with the current `maxDepth`.

The `drawShape` function currently just draws a 50% alpha circle at position 0, 0 with the given size. Of course, where that circle actually ends up being drawn and how big it will be are dependent on the various transformations you'll be making.

The `iterate` function runs a `for` loop `numShapes` times and does the needed transformations, rotating to the current angle, translating the required distance and then scaling down by the `scaleFactor`. It then calls `drawShape` to draw a shape at the current position.

If `depth` is greater than zero, `iterate` calls itself recursively, reducing the depth by one. Doing so will result in three more scaled down shapes being drawn around the current one.

Now, let's start messing with it.

Changing the Angles

One of the simplest things you can do is change the angles being used. Right now, the shape is rather formal and rigid due to the original angles chosen. But, add a small amount to each, like so, and you'll get a very different picture:

```
angles = [
  .3,                      // 0 degrees
  .3 + Math.PI * 2 / 3,    // 120 degrees
  .3 + Math.PI * 4 / 3    // 240 degrees
],
```

This gives you Figure 3.5.

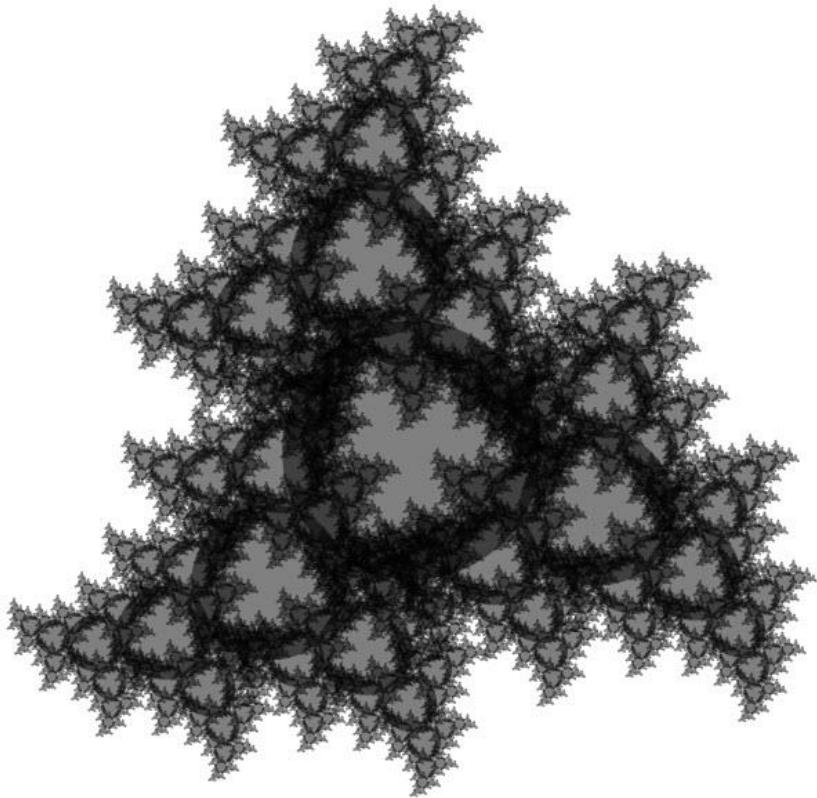


Figure 3.5. Definitely not a Sierpinski!

Following up on that, you can try other offset values, or try all kinds of offsets this way:

```
offset = Math.random() * Math.PI * 2,  
angles = [  
    offset,                      // 0 degrees  
    offset + Math.PI * 2 / 3,    // 120 degrees  
    offset + Math.PI * 4 / 3    // 240 degrees  
,
```

This last code snippet generates a random offset each time the page is refreshed. Now, you'll get either an almost-

Sierpinski, something like in Figure 3.5, or at the far end of the spectrum, something like you see in Figure 3.6.

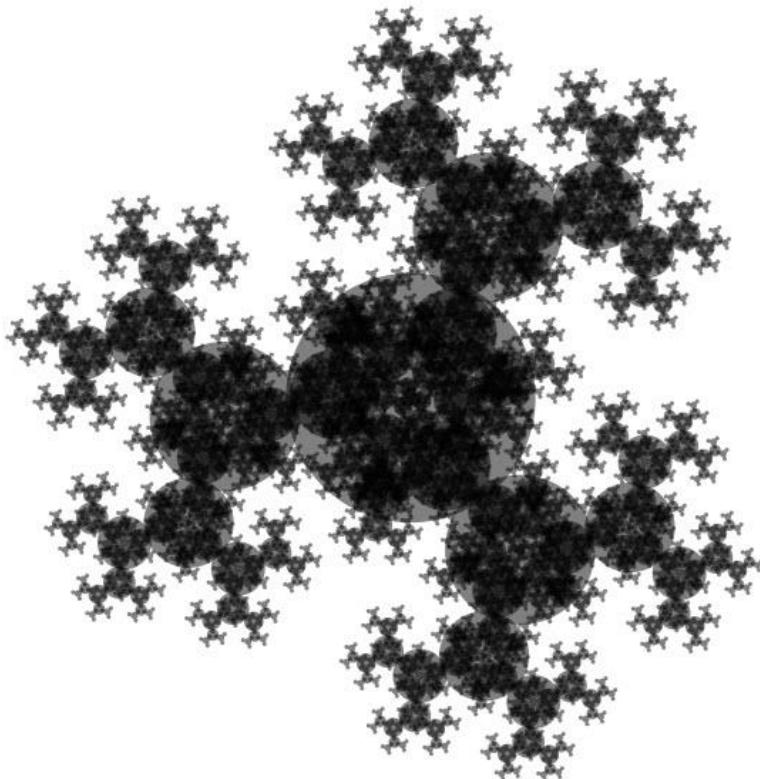


Figure 3.6. The “opposite” of a Sierpinski?

Breaking Symmetry

While it's getting a little bit more interesting, you'll see before long that variety is lacking. Part of the reason for this is that all of these images still have radial symmetry. If you take any one of them and rotate it 120 degrees, you'll see

the exact same image. Naturally, that's because the angles we've set up are all multiples of 120 degrees, plus a constant offset.

Well, that's easy enough to change! Just randomize the angles and you're off to the races with variety:

```
angles = [  
    Math.random() * Math.PI * 2,  
    Math.random() * Math.PI * 2,  
    Math.random() * Math.PI * 2  
,
```

You can find this piece of code already made for you in the file `ssf2.js` along with `ssf2.html`.

Run this a number of times, and iterate it, and you'll create all kinds of interesting images like the ones you see in Figures 3.7, 3.8 and 3.9.

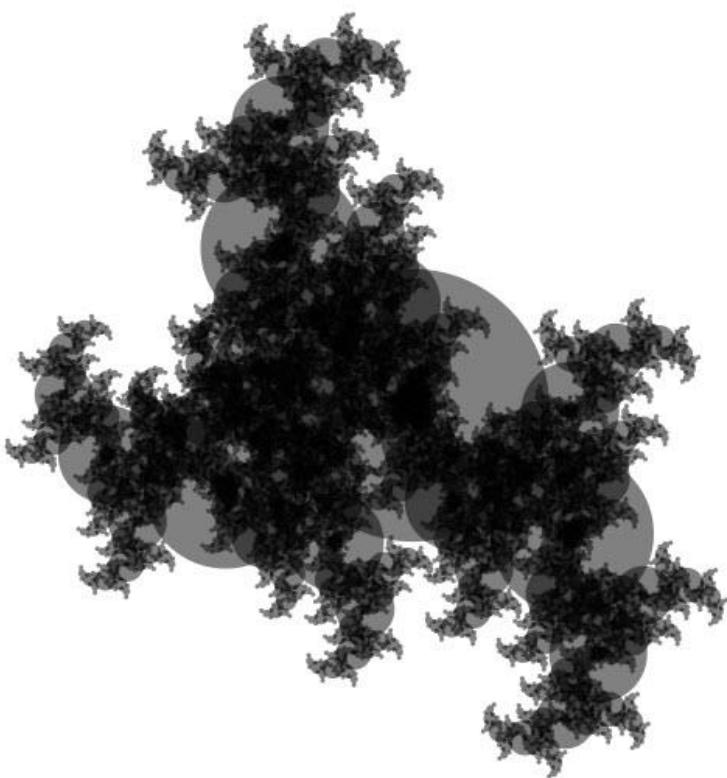


Figure 3.7.

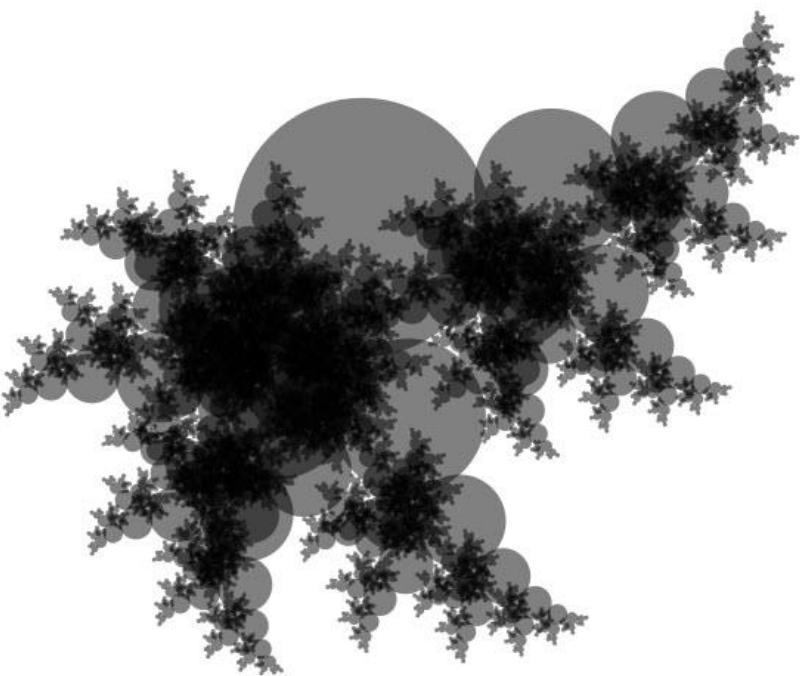


Figure 3.8.

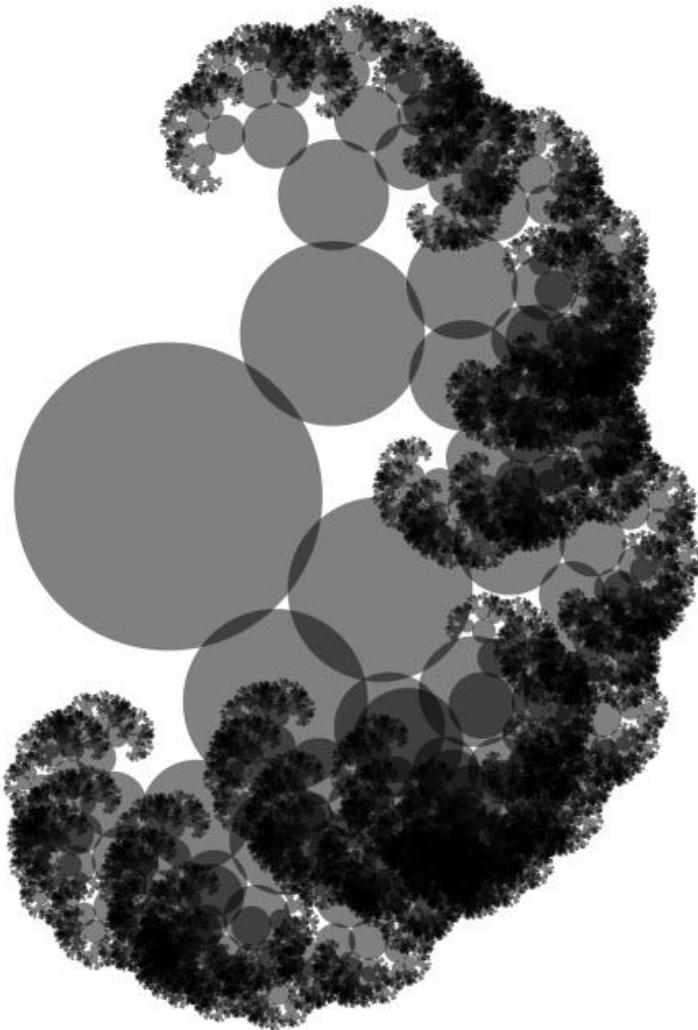


Figure 3.9.

Now that symmetry is broken, you can get a much larger variety of images. Why not break it even more? Right now, you're using the same value for `dist` in every case. You can

easily change that. In the `init` function, change the assignment of `dist` to look like this:

```
size = chaos.height / 10;
dist = [
    size * Math.random() * 3 + 1,
    size * Math.random() * 3 + 1,
    size * Math.random() * 3 + 1
];
```

This creates an array of three distance values, just like the array of angle values. Note that I also changed the `size` variable to be a bit smaller, as the drawings were tending to go offscreen with the larger distances.

Now, change the `iterate` function so it uses a single value from this array, like so:

```
function iterate(depth) {
    for(var i = 0; i < numShapes; i += 1) {
        chaos.context.save();
        chaos.context.rotate(angles[i]);
        chaos.context.translate(dist[i], 0);
        chaos.context.scale(scaleFactor,
            scaleFactor);
        drawShape();
        if(depth > 0) {
            iterate(depth - 1);
        }
        chaos.context.restore();
    }
}
```

These latest changes can be found in the files `ssf3.js` and `ssf3.html`. Using multiple distance values increases the variety of possible images even more. Figures 3.10, 3.11 and 3.12 show just a few of the possible results.

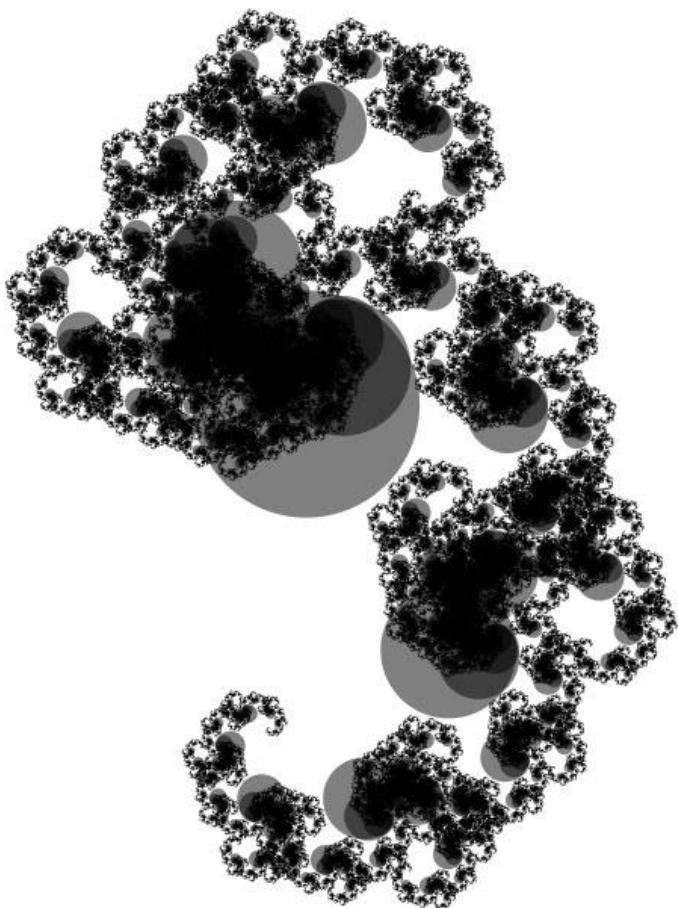


Figure 3.10.

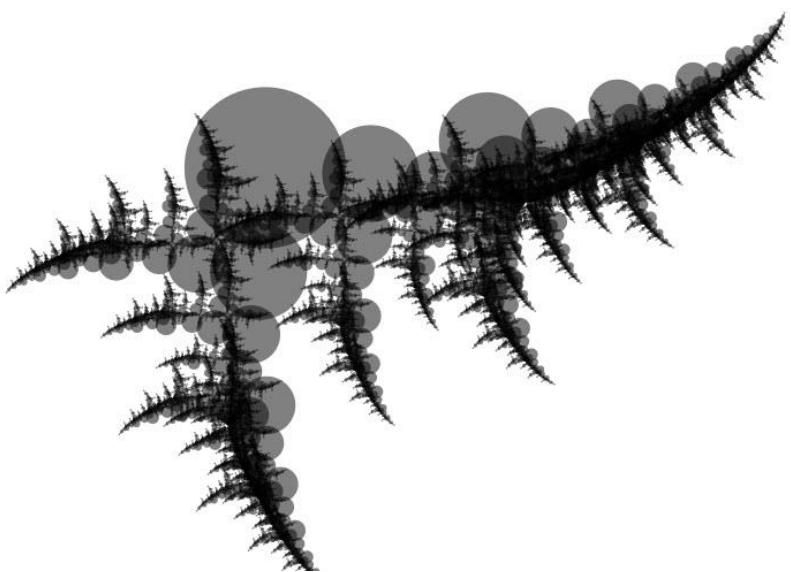


Figure 3.11.

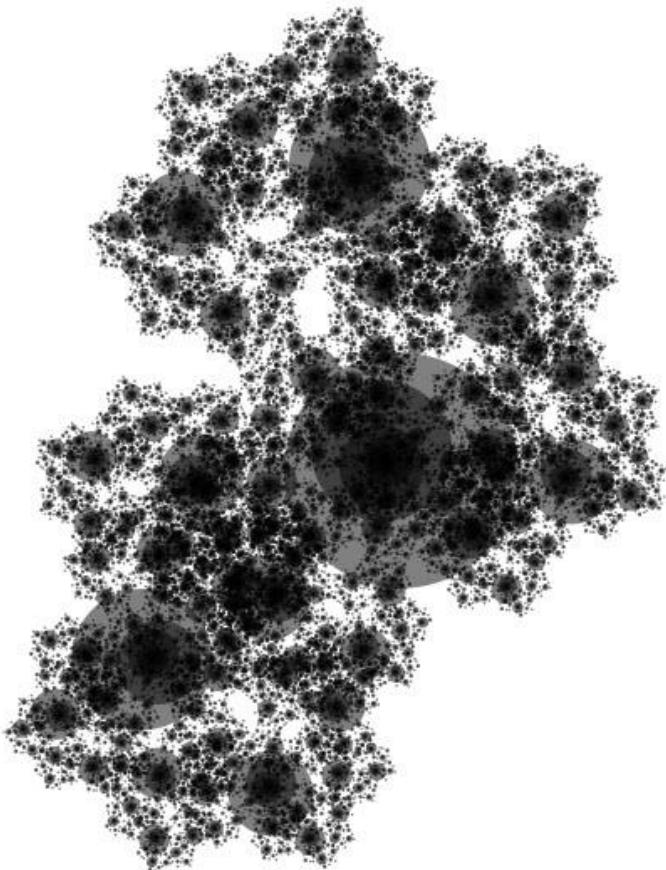


Figure 3.12.

Changing the Shape

This is only the start of how you can alter this program to create different images. So far, you've only used a circle as a shape. This is defined in the `drawShape` function. Well, it's

easy enough to change that. Whatever you draw in that function will be the shape that is drawn for every single iteration, just scaled down, rotated and repositioned. Here are some other options in Figures 3.13, 3.14 and 3.15.

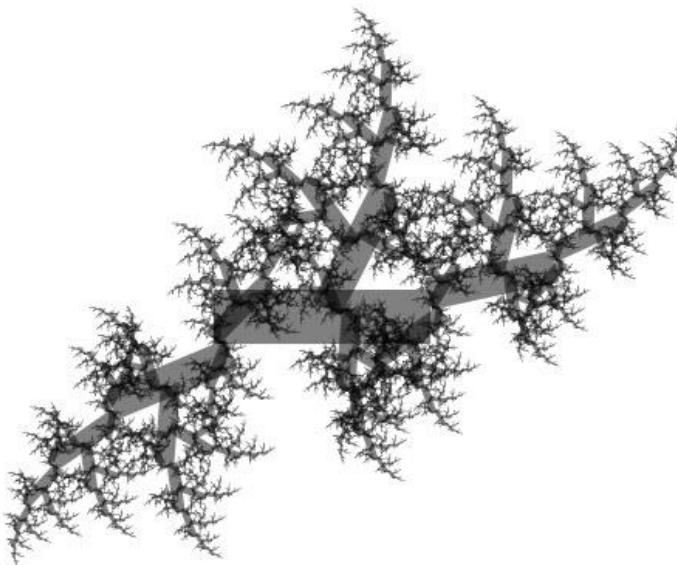


Figure 3.13. The shape is a rectangle.

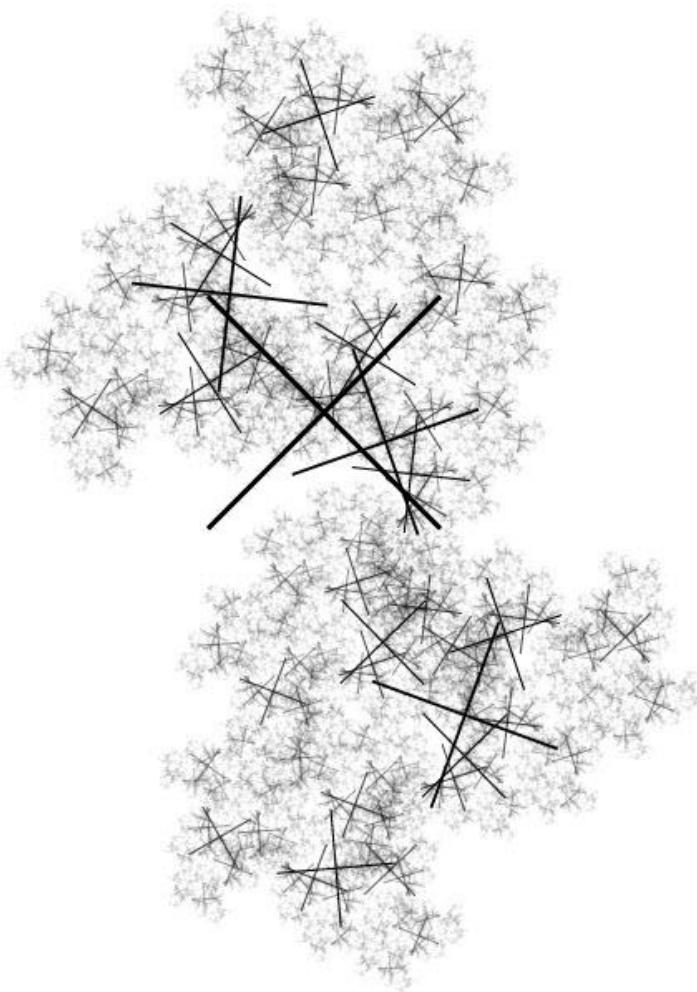


Figure 3.14. An “x” made by drawing two crossed lines.

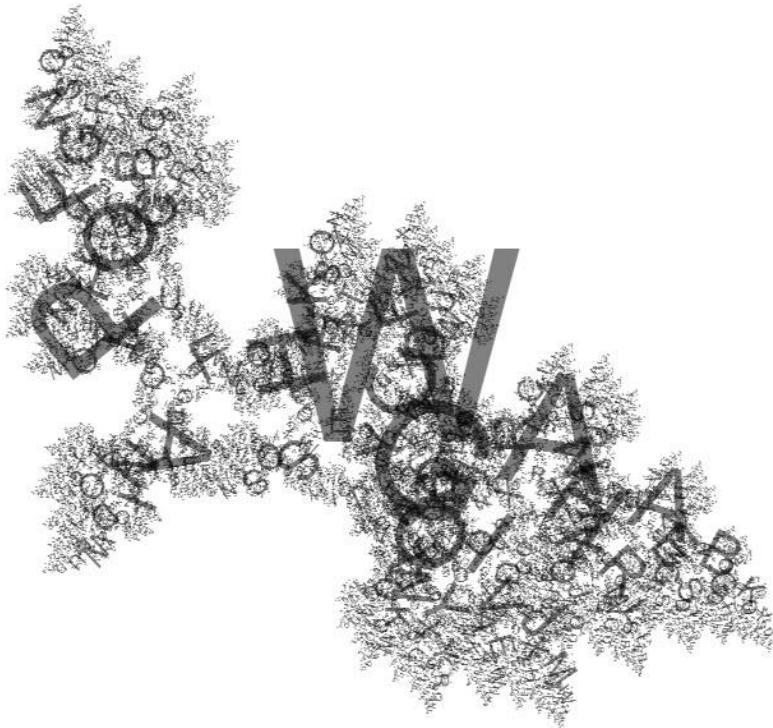


Figure 3.15. Random letters are used as the shape.

Finally, Figure 3.16 uses the word “fractal” to make a fractal!

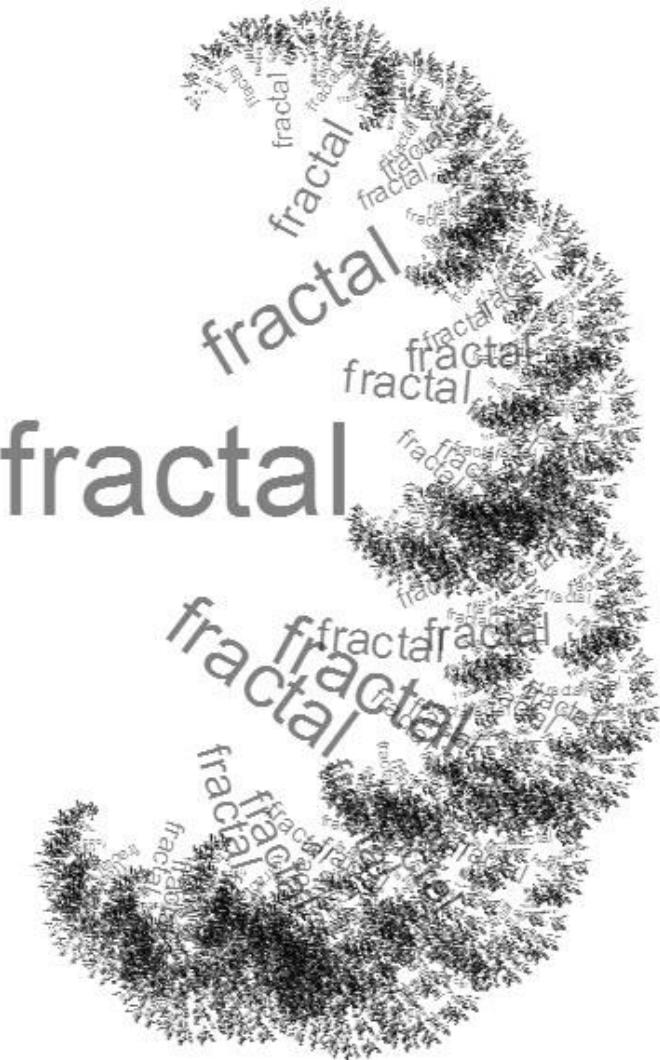


Figure 3.16. Fractal fractal!

The code for all of those examples is also available in the `drawShape` function of `ssf3.js`. Just uncomment the section you want to try out. And, of course, try creating your own variations.

Using Color

Another thing you can start changing up is color. Gray was nice at first, but it's getting old now. Of course, you can easily change the color by changing the values passed into `chaos.context.fillStyle` in the `drawShape` function. This will cause all the shapes to be drawn with a new color. But you can do better than that. How about a different color for each depth level?

To do this, you'll need an array of colors for each depth. Set that up at the beginning of the code:

```
colors = [
  "#CC0000",
  "#CC6600",
  "#CCCC00",
  "#66CC00",
  "#00CC00",
  "#00CC66",
  "#00CCCC",
  "#0066CC",
  "#0000CC"
];
```

Then, the `drawShape` function will need to know the current depth at which it's drawing. So you'll need to pass that as an argument when you call `drawShape` from within the `iterate` function:

```
function iterate(depth) {
  for(var i = 0; i < numShapes; i += 1) {
    chaos.context.save();
    chaos.context.rotate(angles[i]);
    chaos.context.translate(dist[i], 0);
    chaos.context.scale(scaleFactor, scaleFactor);
    drawShape(depth);
    if(depth > 0) {
      iterate(depth - 1);
    }
    chaos.context.restore();
  }
}
```

Finally, `drawShape` will take that `depth` value and use it to pull a color out of the `colors` array:

```
function drawShape(depth) {  
    chaos.context.fillStyle = colors[maxDepth - depth];  
    chaos.context.beginPath();  
    chaos.context.arc(0, 0, size, 0, Math.PI * 2, false);  
    chaos.context.fill();  
}
```

I found that using `maxDepth - depth` worked better than `depth` alone here, especially when iterating through multiple times. It keeps the colors from shifting on each iteration. If that's not clear, try using `depth` alone for `fillStyle` and you'll see what I mean.

These latest changes are in `ssf4.js` and `ssf4.html` and give you images like in Figure 3.17. Of course, if you are reading this on a device that does not support color, you're just going to have to run the code yourself to see the color.

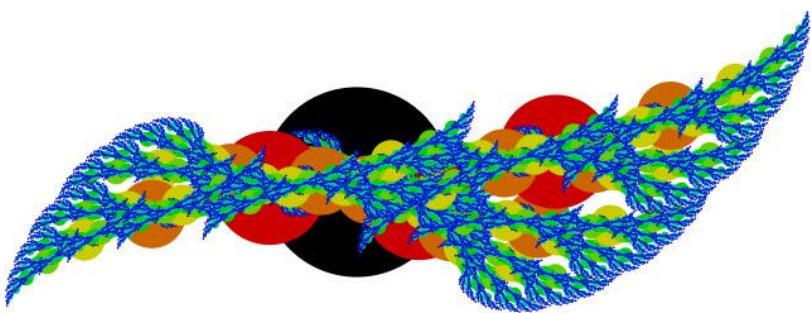


Figure 3.17. Multi-colored.

Getting Irregular

The title of this chapter is “Symmetry and Regularity”. I began by having you create regular, symmetrical fractals. Then, I had you break symmetry and see how that opened up many new avenues for variety. Next up, you’re going to break regularity and get a look at what that does to a fractal.

To recap, regularity refers to the characteristic of self-similarity within a fractal. Up until now, the fractal images you’ve created have all been regular. In other words, each component of the image was an exact, smaller copy of the whole. Technically, I should say that it would be an exact copy if the fractal could be iterated infinitely. Obviously, because you are dealing with physical pixels on a screen, each smaller copy has significantly less resolution than the whole. But in a practical sense, it’s safe to say that they are exact copies.

But realize that the term is “self-similarity” not “self-identity”. There’s no requirement that the parts have to be exact copies. Let’s mix them up a bit and see what happens.

The reason the copies are the same in each case is because you are using the same variables for each iteration. Try changing a couple of them randomly. Here’s what I did in the file `ssf5.js` in the `iterate` function:

```
function iterate(depth) {  
    for(var i = 0; i < numShapes; i += 1) {  
        chaos.context.save();  
        chaos.context.rotate(angles[i] +  
            Math.random() - .5);  
        chaos.context.translate(dist[i] *  
            (Math.random() * .5 + 1), 0);  
        chaos.context.scale(scaleFactor,  
            scaleFactor);  
        drawShape(depth);  
        if(depth > 0) {  
            iterate(depth - 1);  
        }  
    }  
}
```

```
        }
        chaos.context.restore();
    }
}
```

This code adds a random value from -.5 to .5 to the current angle and multiplies the current distance by a value from .5 to 1.5. Because these random values are calculated on the fly, you'll see the shapes jumping around as you go through each iteration. But I find the results quite striking. Here are some I came up with in Figures 3.18, 3.19, 3.20 and 3.21:

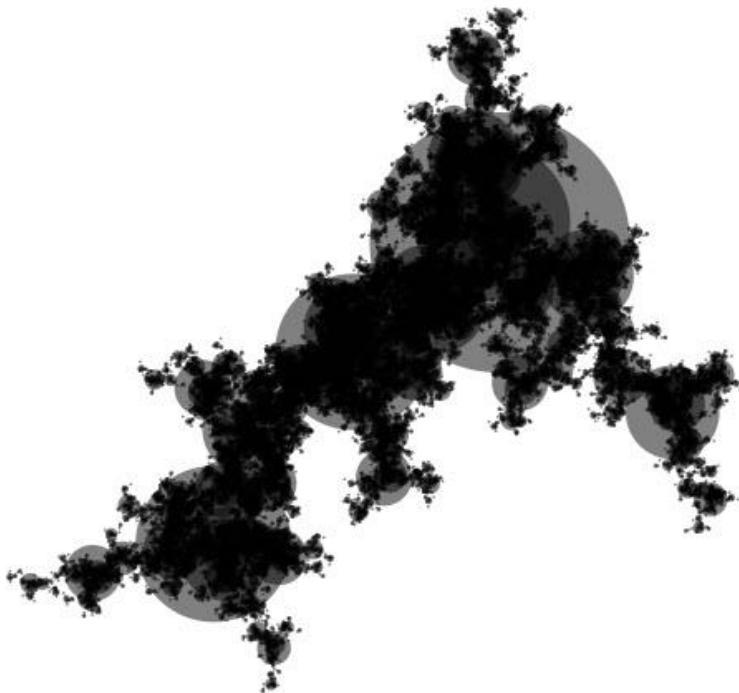


Figure 3.18.

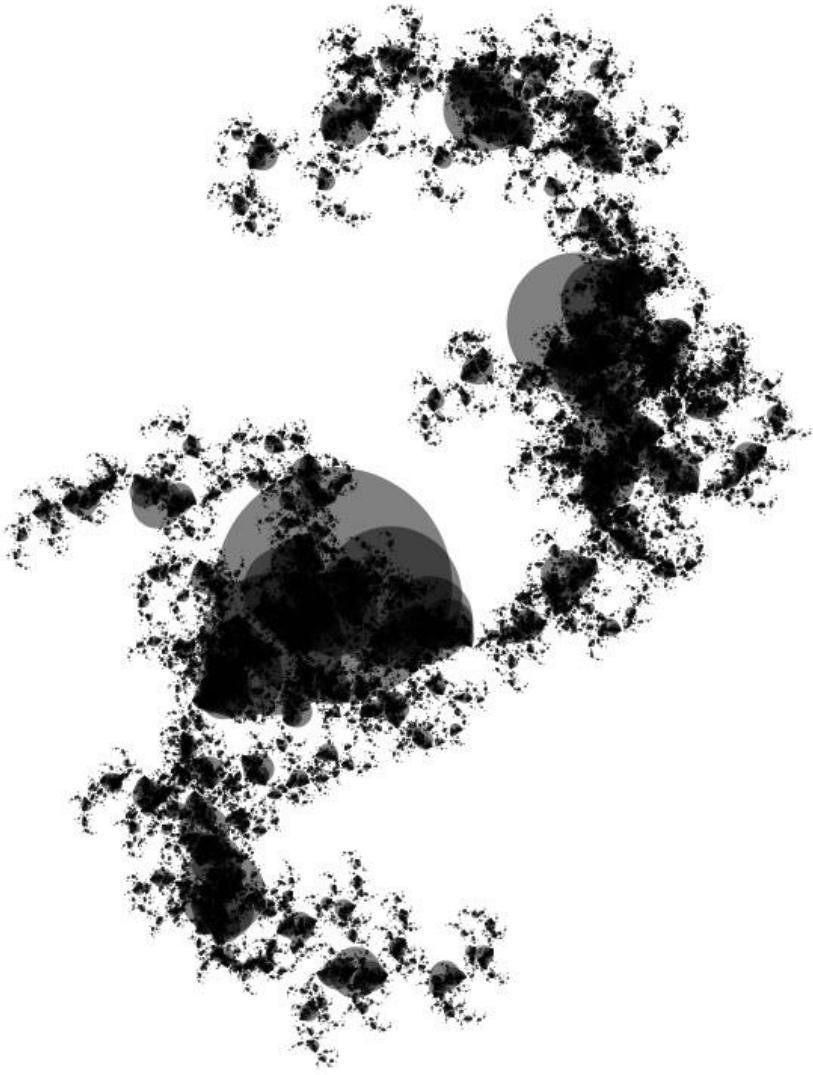


Figure 3.19.



Figure 3.20.

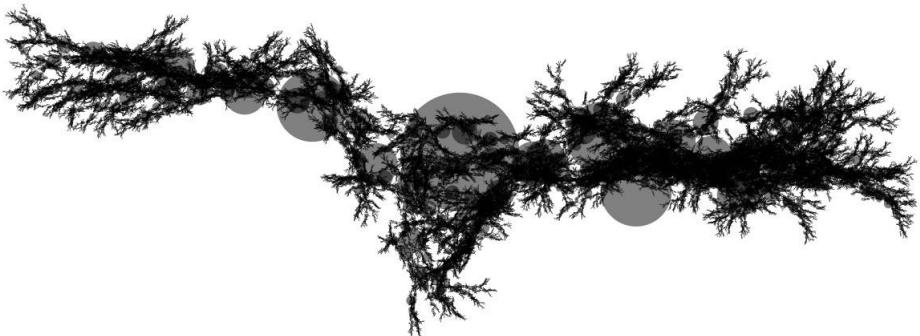


Figure 3.21.

The differences can be subtle, but I think these last pieces have a more organic and natural feel to them, like blobs of ink or piles of twigs – random, but with an underlying order as well. These are some of my favorite types of pieces.

Okay, I feel like I've beaten this one to death. But, in fact, there are still all kinds of ways you can build on this simple example. I didn't even touch

the `numShapes` or `scaleFactor` variables. Try creating a fractal that makes four or five shapes each iteration, but be careful, you can get into some astronomical numbers quickly. Or try messing around with the scale factor. I'm sure you'll come up with ideas I would never have imagined. I can't wait to see them!

Tree Fractal

On to the next project! For this one, I'm going back to a classic fractal shape: the fractal tree. Real trees are themselves obvious fractal shapes. The trunk splits into two or more branches. Those branches branch off to a number of smaller branches, which themselves branch off to still

smaller branches. This continues right down to the twig level. Of course, a real tree is neither perfectly symmetrical nor perfectly regular. So there will be some work to do to get it looking realistic.

The strategy is to draw the tree's trunk first. This will go about halfway up the full height of the tree. From there, the tree will split into two separate branches angled to the left and right. When iterating, each branch can then be replaced by a copy of the full tree: a trunk and two branches.

As usual, here's the full code, which you can find
in `tree1.js` and `tree1.html`:

```
window.onload = function() {
    var maxDepth = 0,
        angles = [
            -Math.PI / 4,
            Math.PI / 4
        ],
        baseSize = 0;

    init();

    function init() {

        chaos.init();

        baseSize = chaos.height * .8;
        draw();

        document.body.addEventListener("keyup",
            function(event) {
                console.log(event.keyCode);
                switch(event.keyCode) {
                    case 32: // space
                        maxDepth += 1;
                        draw();
                        break;

                    case 80: // p
                        chaos.popImage();
                        break;

                    default:
                }
            }
        );
    }
}
```

```

        break;
    }
});

}

function draw() {
    chaos.clear();
    chaos.context.save();
    chaos.context.translate(chaos.width * 0.5,
                           chaos.height * 0.9);
    drawTree(maxDepth, baseSize, 0);
    chaos.context.restore();
}

function drawTree(depth, size, angle) {
    // draw trunk
    chaos.context.save();
    chaos.context.rotate(angle);
    chaos.context.beginPath();
    chaos.context.moveTo(0, 0);
    chaos.context.lineTo(0, -size / 2);
    chaos.context.stroke();
    chaos.context.translate(0, -size / 2);

    if(depth === 0) {
        // we're done. draw branches.
        drawBranch(size / 2, angles[0]);
        drawBranch(size / 2, angles[1]);
    }
    else {
        // more iteration to be done.
        // draw two mini trees instead of branches.
        drawTree(depth - 1, size / 2, angles[0]);
        drawTree(depth - 1, size / 2, angles[1]);
    }
    chaos.context.restore();
}

function drawBranch(size, angle) {
    chaos.context.save();
    chaos.context.rotate(angle);
    chaos.context.beginPath();
    chaos.context.moveTo(0, 0);
    chaos.context.lineTo(0, -size);
    chaos.context.stroke();
    chaos.context.restore();
}

```

```
}
```

The first big chunk of code through the `init` function should look familiar. There are a few new variables, but largely it's doing the same thing as all of the previous examples.

Then, the `draw` function comes in and translates the context to center screen horizontally and near the bottom vertically. This is where the tree will be "planted." The function then calls `drawTree` with a few default arguments.

The `drawTree` function is what will be iterated to create the fractal. Like the iterated functions in previous examples, it gets a `depth` value to know when to stop iterating. It also gets `size` and `angle` values passed in. These are exactly what you might guess: `size` is how large to draw this tree and `angle` is how much to rotate it. The first time through, `size` is equal to `baseSize`, which is calculated in the `init` function as 80% of the canvas height. The `angle` is zero. So the first tree iteration is full size, straight up and down.

The first thing `drawTree` does is draw the trunk from `0, 0` to `0, -size / 2`. Remember that the context has been translated, so `0, 0` is now the base of the tree, near the bottom center of the screen. The trunk is thus drawn as half the full height of the tree. Next, the context is translated `-size / 2` on the y-axis. This puts the origin at the very top of the trunk, ready to draw the branches.

If `depth` is zero at this point, the function goes right ahead and draws two branches using the `drawBranch` function. They are drawn at half the current size, and at -45 degrees and $+45$ degrees (`-Math.PI / 4` and `Math.PI / 4` radians). This gives you the first iteration of a tree, which you can see in Figure 3.22.

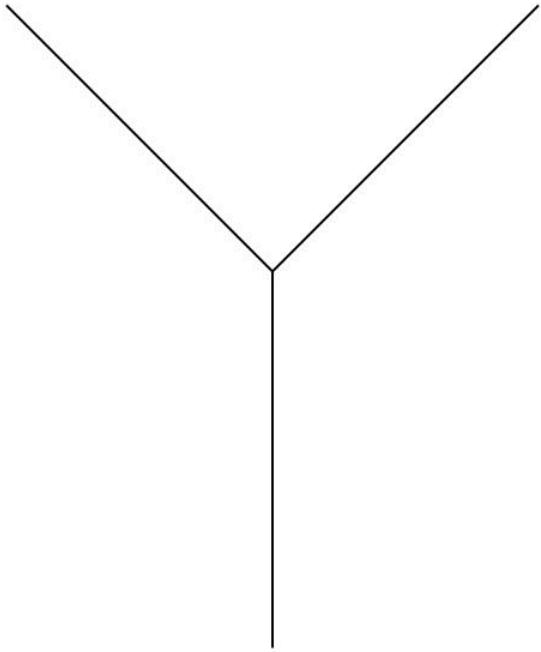


Figure 3.22. The simplest of trees.

However, if `depth` is more than zero, `drawTree` is called recursively. The `depth` is reduced by one, `size` is divided by two and the new trees are drawn at the same angles at which the branches were drawn: -45 and +45. This gives you Figure 3.23.

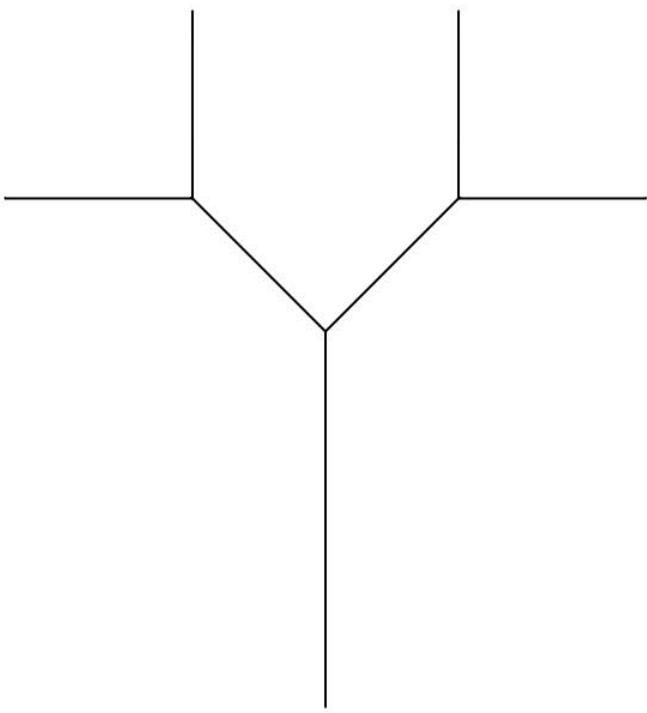


Figure 3.23. Branches with branches.

This is repeated for each iteration, until you get something like Figure 3.24.

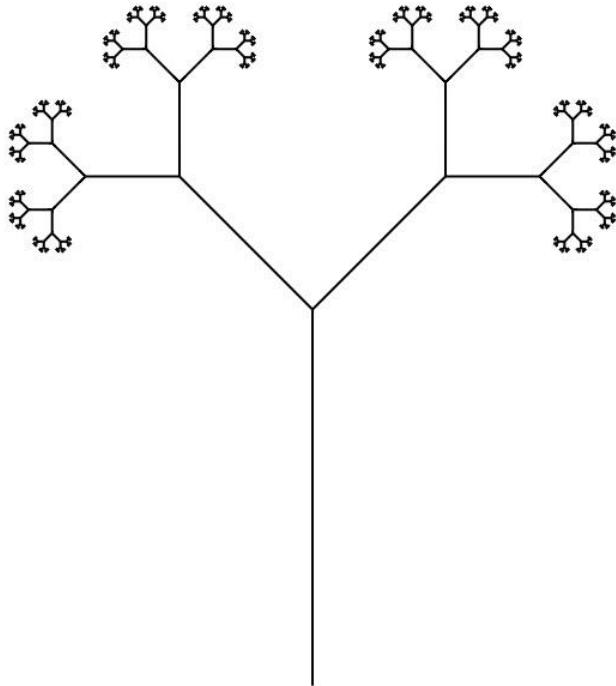


Figure 3.24. Fully iterated fractal tree.

Congratulations, you've made a tree! Now, let's make it look more like a real tree.

Breaking Symmetry

Breaking symmetry in the tree is all too simple. Just change the angles. In the file `tree2.js`, I've just changed the line that defines the angles to make them random values between 0 degrees and +/- 90 degrees:

```
angles = [  
    -Math.PI / 2 * Math.random(),  
    Math.PI / 2 * Math.random()  
,
```

Now, each time you run the program, the tree will use different angles for the left and right branches. But, because these are defined just a single time at the top of the file, these values are used for every branch in every iteration. This gives you quite a variety of trees, such as the ones in Figures 3.25, 3.26 and 3.27.

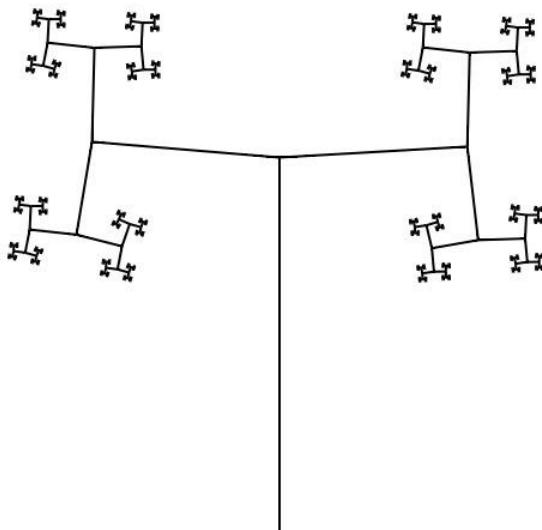


Figure 3.25.

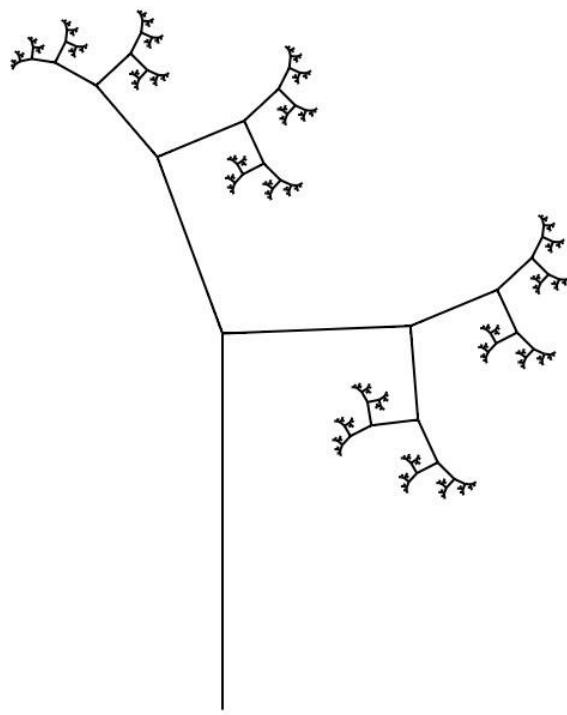


Figure 3.26.

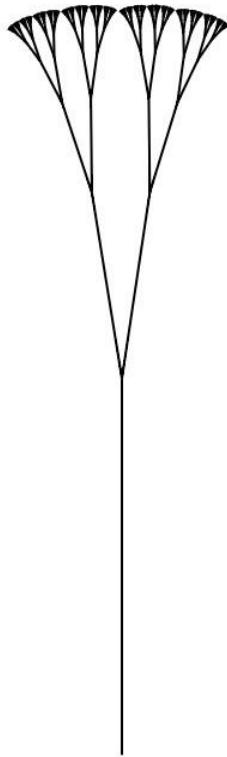


Figure 3.27.

Altering Trunk Length

At this point, the trunk is hardcoded at half the height of the full tree, with the branches taking up the other half. You can change that by adding a `scaleFactor` variable and making a few other tweaks to remove the hard-coded `size / 2` lines.

Let's start by making `scaleFactor` .6, right up at the top of the file with the other variables. This will result in a shorter trunk and longer branches.

The rest of the changes go in the `drawTree` function:

```
function drawTree(depth, size, angle) {  
    // draw trunk  
    chaos.context.save();  
    chaos.context.rotate(angle);  
    chaos.context.beginPath();  
    chaos.context.moveTo(0, 0);  
    chaos.context.lineTo(0,  
        -size * (1 - scaleFactor));  
    chaos.context.stroke();  
    chaos.context.translate(0,  
        -size * (1 - scaleFactor));  
  
    if(depth === 0) {  
        // we're done. draw branches.  
        drawBranch(size * scaleFactor, angles[0]);  
        drawBranch(size * scaleFactor, angles[1]);  
    }  
    else {  
        // more iteration to be done.  
        // draw two mini trees instead of branches.  
        drawTree(depth - 1,  
            size * scaleFactor,  
            angles[0]);  
        drawTree(depth - 1,  
            size * scaleFactor,  
            angles[1]);  
    }  
    chaos.context.restore();  
}
```

This new code uses `-size * (1 - scaleFactor)` instead of `-size / 2` to draw the trunk. And it uses `size * scaleFactor` instead of `size / 2` to draw the branches, or subtrees. All of these changes can be found in the file `tree3.js`. Immediately, this gives you a more interesting tree shape, as you can see in Figure 3.28.

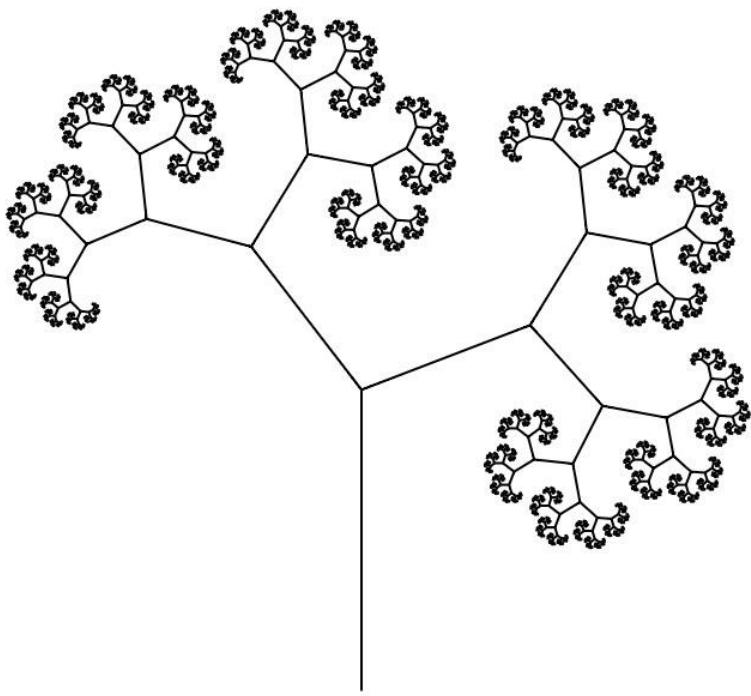


Figure 3.28. Shorter trunk, longer branches.

Finally, you can generate all kinds of trunk sizes by giving `scaleFactor` a random value:

```
scaleFactor = .2 + Math.random() * .6;
```

This gives you a value from .2 up to .8. Now, you'll get wild trees like in Figure 3.29 ...

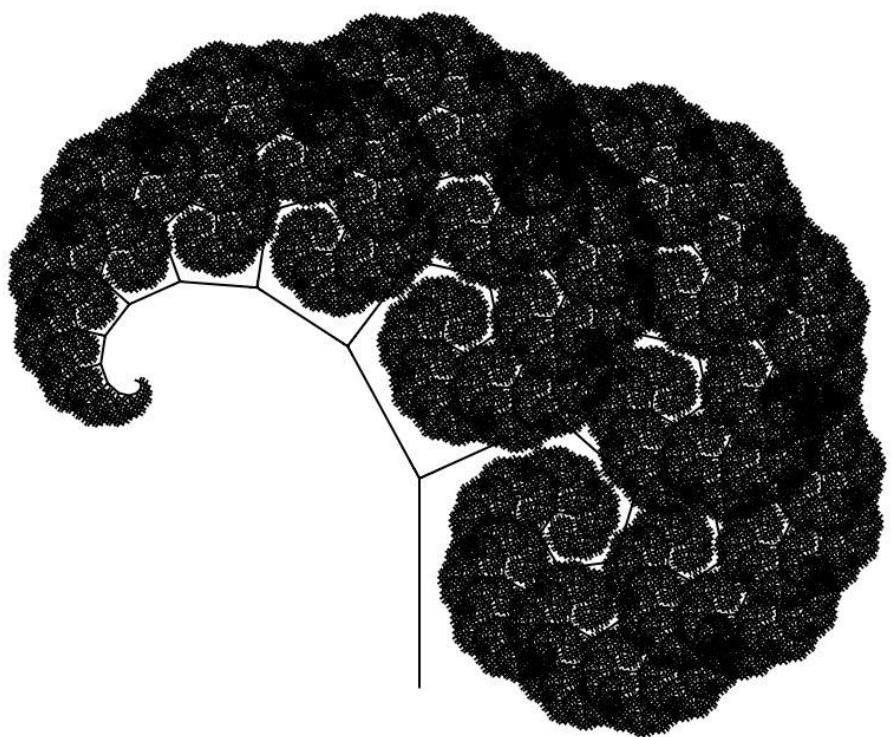


Figure 3.29. Short trunk, crazy branches.

... or rather boring ones like in Figure 3.30.

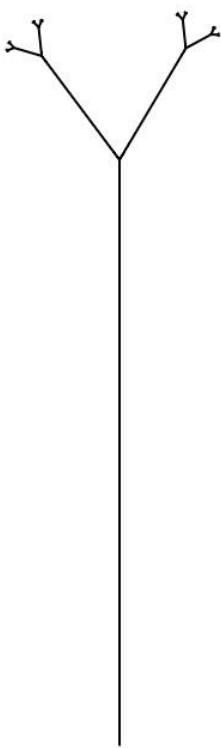


Figure 3.30. Long trunk, yawn.

And, occasionally, you'll see something quite striking, such as in Figure 3.31.

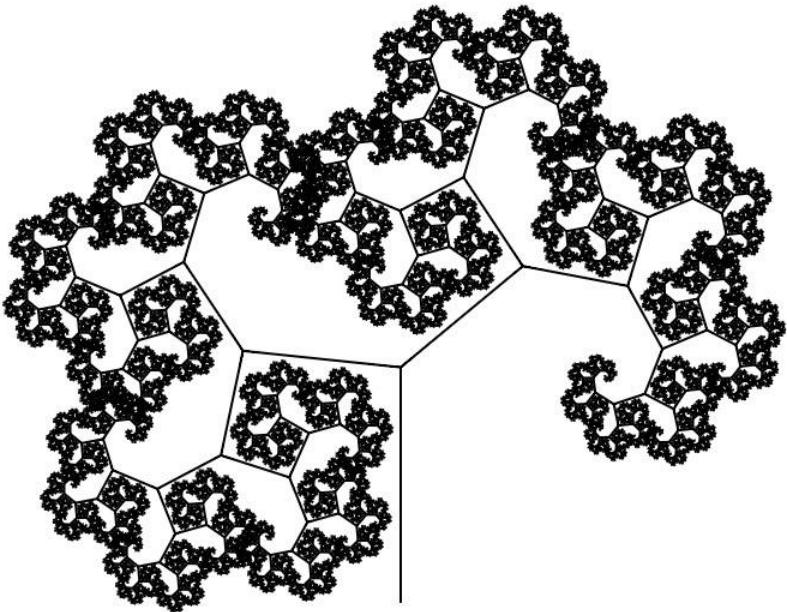


Figure 3.31, Just right!

However, this code is still not creating very realistic trees. Of course, you're not going to get photorealistic trees by drawing a bunch of straight lines, but I think the next section will start to create some more natural-looking, tree-like forms.

Breaking Regularity

The main problem with the trees at this point is that each iteration is the same: it uses the same angles and same scaling, so you don't get any of the variety you would see in nature. Well, that's easy enough to handle. Just randomize those factors each time you use them.

What I chose to do in `tree4.js` was just re-randomize the `angles` and `scaleFactor` variables at the top of the `drawTree` function, like so:

```
angles = [  
  -Math.PI * .4 * Math.random(),  
  Math.PI * .4 * Math.random()  
];  
scaleFactor = .55 + Math.random() * .25;
```

Those values are a little different than they were in the earlier examples when the values were static. I came up with these numbers by trial and error and seeing what looked good. I'm sure you can find something that looks even better. But, with this code, you can create a huge variety of tree forms. Some of them will look like psycho alien trees, of course. But now and then you'll wind up with some natural-looking figures like those in Figures 3.32, 3.33 and 3.34. If not trees exactly, they at least look like some pressed flowers or dried-up weeds.

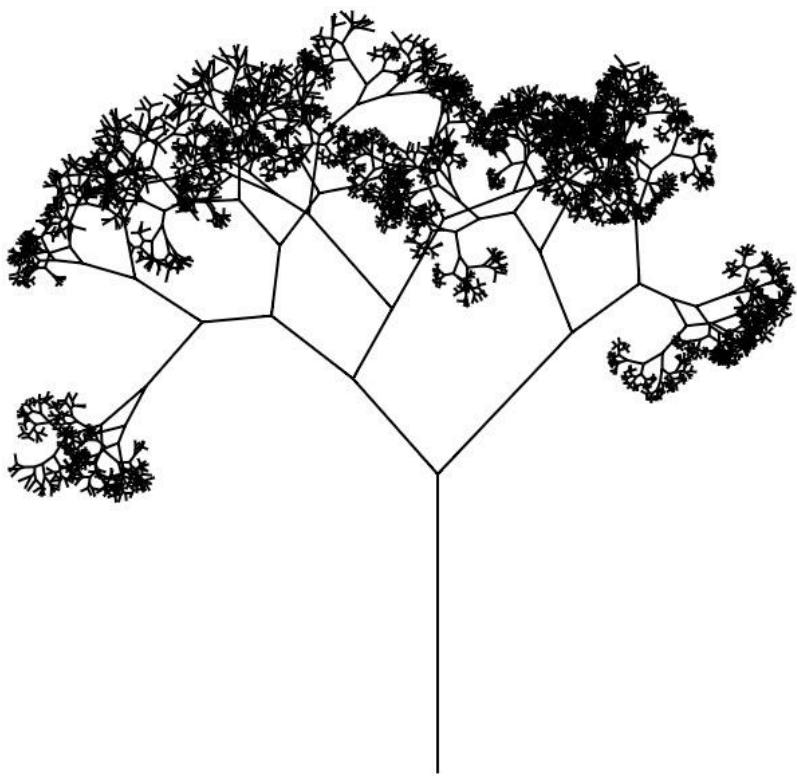


Figure 3.32.

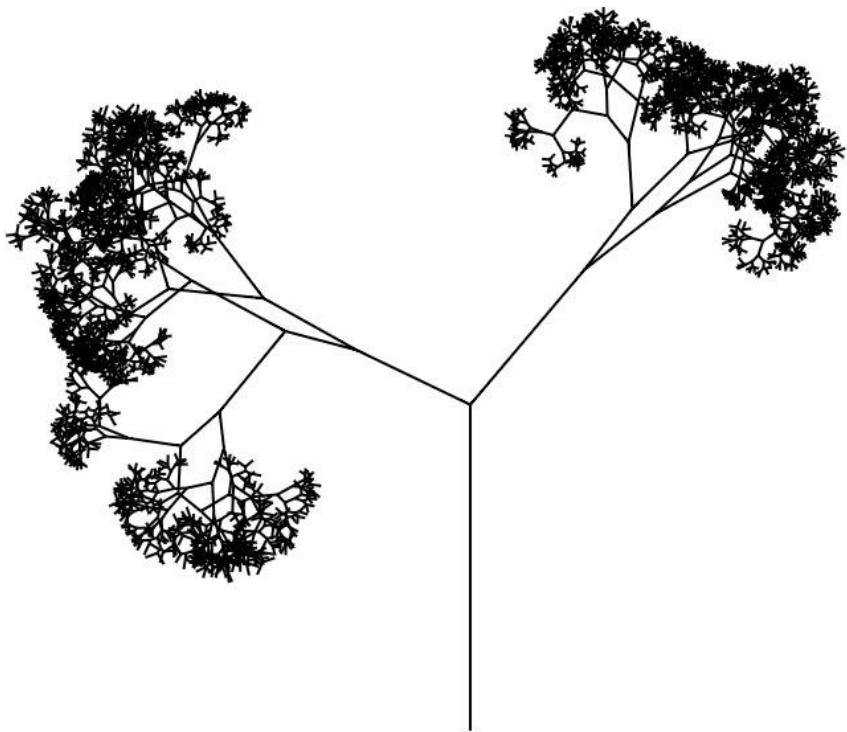


Figure 3.33.

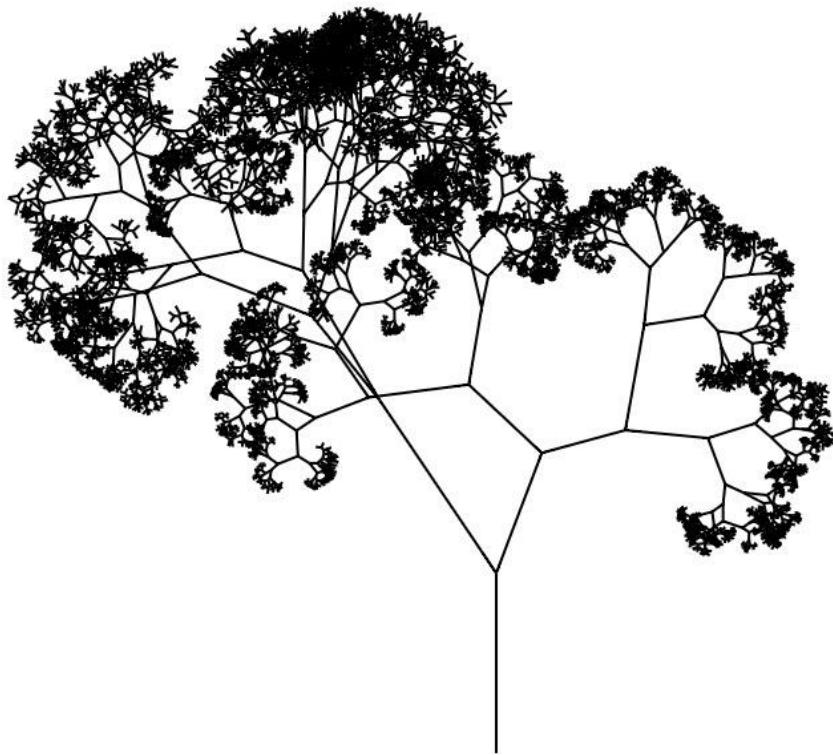


Figure 3.34.

Okay, I'll leave you here to explore this one further on your own. There is still a lot to experiment with here – colors, line widths, more than two branches, etc. When you get bored, come back and start in on the next section.

Pythagorean Fractal

For the last fractal in this chapter, I'm going to pull out the Pythagorean theorem. You might remember this from high school. "A squared plus B squared equals C squared." To clarify, let's turn to a picture, Figure 3.35.

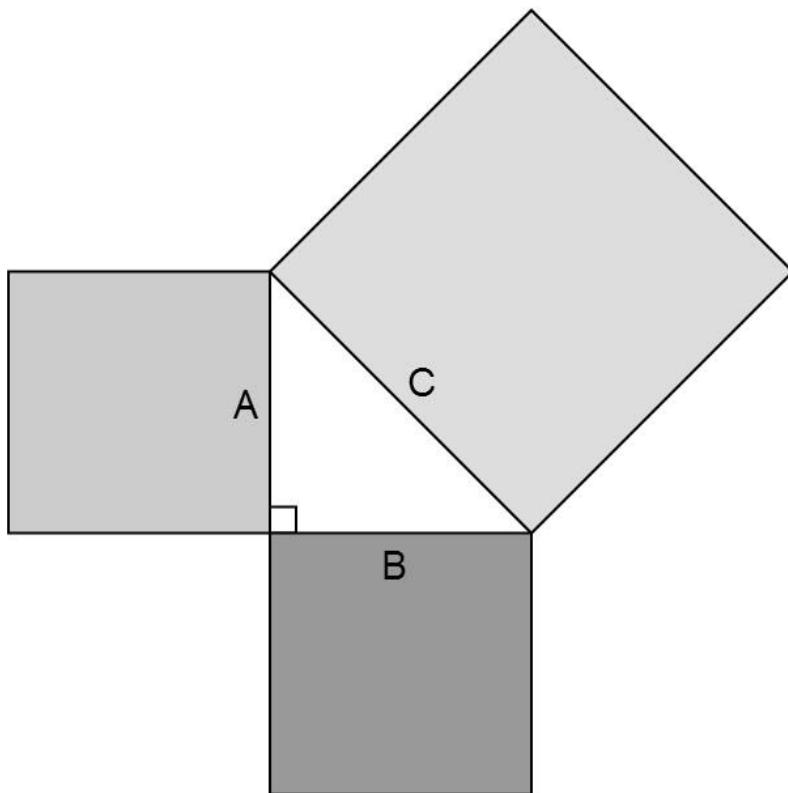


Figure 3.35. The Pythagorean Theorem, visualized.

There you see a right triangle with sides A, B and C. If you square each side, the areas of square A and square B will

add up to the area of square C. Now, you're probably asking, "What does this have to do with fractals?"

Well, first of all, let's turn the whole thing on its side, as in Figure 3.36.

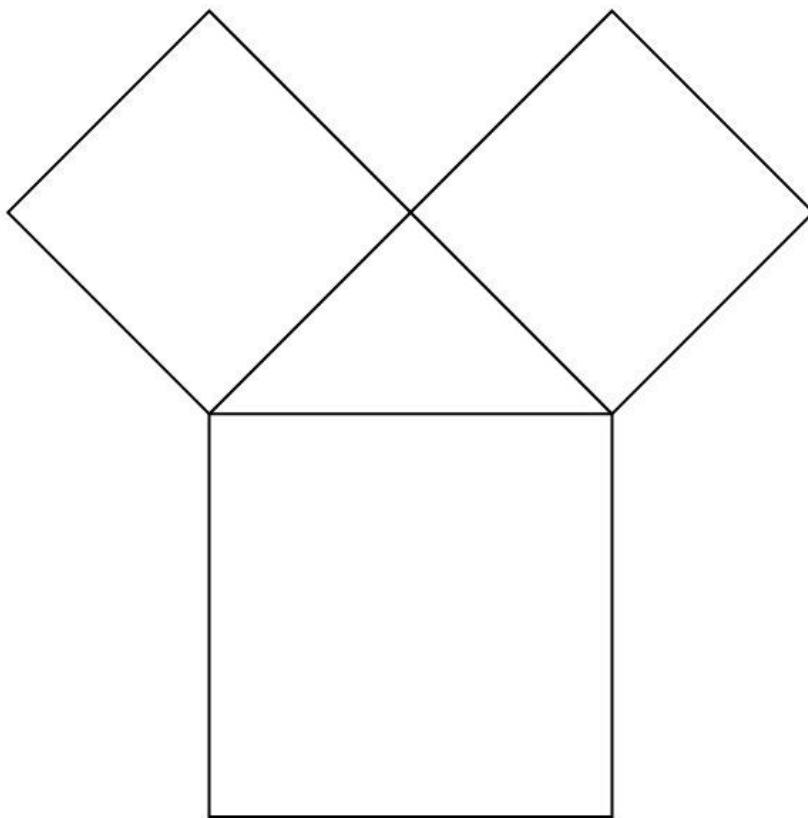


Figure 3.36.

Now, with a little imagination, you might be able to see a similarity between this figure and Figure 3.22, the first basic tree fractal. The large square is the trunk, and the two smaller ones are branches. Then, you can treat each

smaller square as the trunk of a new sub-tree and get what you see in Figure 3.37.

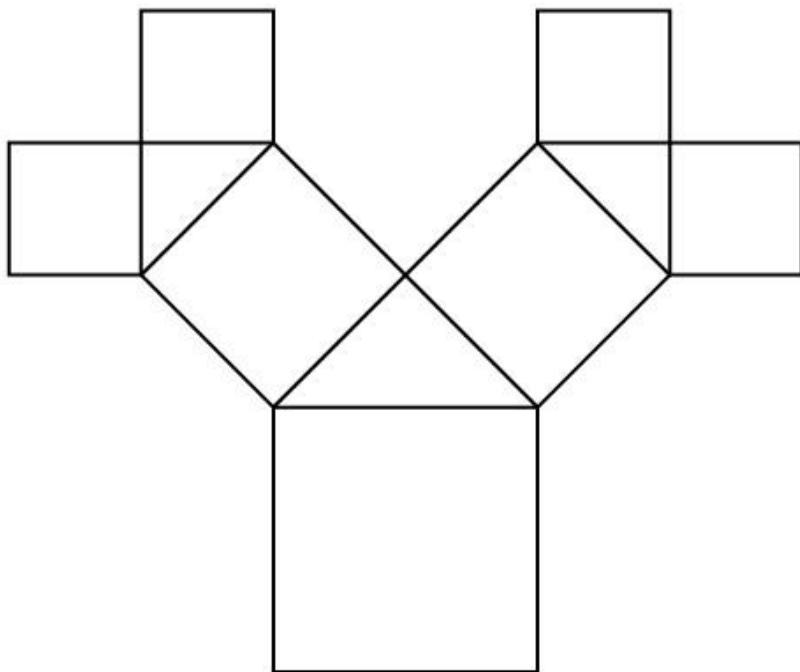


Figure 3.37.

And, just to be clear, although the sides of the triangle I've used so far are equal, they can be different, such as in Figure 3.38. As long as one angle is 90 degrees, the Pythagorean theorem holds true.

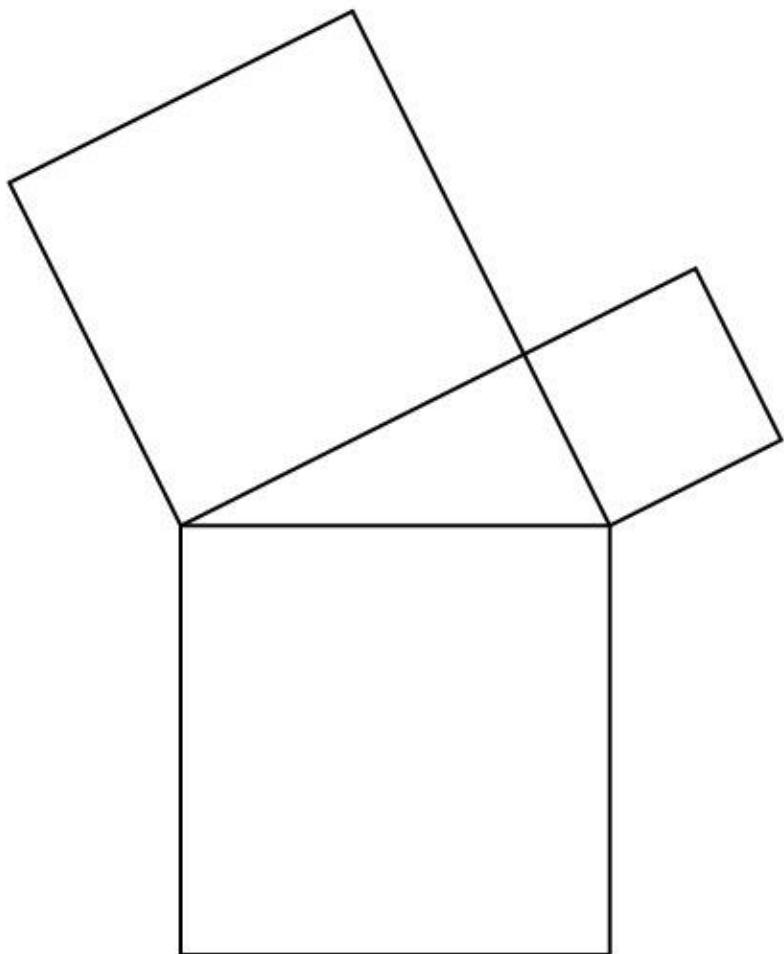


Figure 3.38.

This opens the door to breaking symmetry, which, as you've seen, is a good thing. But I'm jumping ahead of myself here,

and it's time to get coding, right? Here's all the code in `pytree1.js`, which is included in `pytree1.html`:

```
window.onload = function() {
    var maxDepth = 0,
        angle = Math.PI / 4,
        baseSize = 0;

    init();

    function init() {
        chaos.init();

        baseSize = chaos.height * .2;
        draw();

        document.body.addEventListener("keyup",
            function(event) {
                console.log(event.keyCode);
                switch(event.keyCode) {
                    case 32: // space
                        maxDepth += 1;
                        draw();
                        break;

                    case 80: // p
                        chaos.popImage();
                        break;

                    default:
                        break;
                }
            });
    }

    function draw() {
        chaos.clear();
        chaos.context.lineWidth = 2;
        chaos.context.save();
        chaos.context.translate(chaos.width * 0.5,
            chaos.height * 0.9);

        // move a bit to the left
        // to ensure the tree is centered
        chaos.context.translate(-baseSize / 2, 0);
        drawPyTree(maxDepth, baseSize, 0);
        chaos.context.restore();
    }
}
```

```

}

function drawPyTree(depth, size) {
    chaos.context.save();

    // draw trunk
    drawSquare(size);

    // calculate sizes of two branches
    var branch0Size = size * Math.cos(angle);
    var branch1Size = size * Math.sin(angle);

    // move to top left of big square.
    // rotate and draw branch 0
    chaos.context.translate(0, -size);
    chaos.context.rotate(-angle);
    if(depth === 0) {
        drawSquare(branch0Size);
    }
    else {
        drawPyTree(depth - 1, branch0Size);
    }

    // move to bottom right of branch 0
    // rotate 90 and draw branch 1
    chaos.context.translate(branch0Size, 0);
    chaos.context.rotate(Math.PI / 2);
    if(depth === 0) {
        drawSquare(branch1Size);
    }
    else {
        drawPyTree(depth - 1, branch1Size);
    }
    chaos.context.restore();
}

function drawSquare(size) {
    chaos.context.beginPath();
    chaos.context.rect(0, 0, size, -size);
    chaos.context.fill();
}
}

```

Everything up through the `draw` function should be familiar to you if you've worked through any of the other examples in

this chapter. The `drawPyTree` gets more complex than the other examples, though, so let's take it a step at a time:

```
chaos.context.save();  
  
// draw trunk  
drawSquare(size);  
  
// calculate sizes of two branches  
var branch0Size = size * Math.cos(angle);  
var branch1Size = size * Math.sin(angle);
```

First, the function saves the current context and draws the large square that will be the trunk of the tree. It then uses basic trigonometry to calculate the sizes of the two squares that will make up the branches. Note that only one angle is defined in the code. Because the angles of a triangle always add up to 180 degrees, and here one angle is 90 degrees, defining one of the remaining angles automatically determines the last one. At this point, `angle` is set to `Math.PI / 4` radians, or 45 degrees, meaning the other angle will also be 45 degrees. In any case, the code only needs that one angle to do everything that it does.

The way the `drawSquare` function is defined, it will always draw its square from the lower left corner up and to the right. So, in order to draw the first, left-hand branch, you'll need to translate and rotate the context into position. That's the next chunk of code in `drawPyTree`:

```
// move to top left of big square.  
// rotate and draw branch 0  
chaos.context.translate(0, -size);  
chaos.context.rotate(-angle);  
if(depth === 0) {  
    drawSquare(branch0Size);  
}  
else {  
    drawPyTree(depth - 1, branch0Size);  
}
```

Since the context is currently centered on the lower-left corner of the big square, it needs to be translated 0 on the x-axis and `-size` on the y-axis to put it on the top-left corner.

Then, the context is rotated by `-angle` to correctly orient the next branch. Now, the branch is ready to draw. Similar to the previous fractals, if `depth` is zero, just draw a square of the right size. Otherwise recursively call `drawPyTree` with a decremented depth and the new size.

Assuming this is the first iteration, the picture now looks like Figure 3.39.

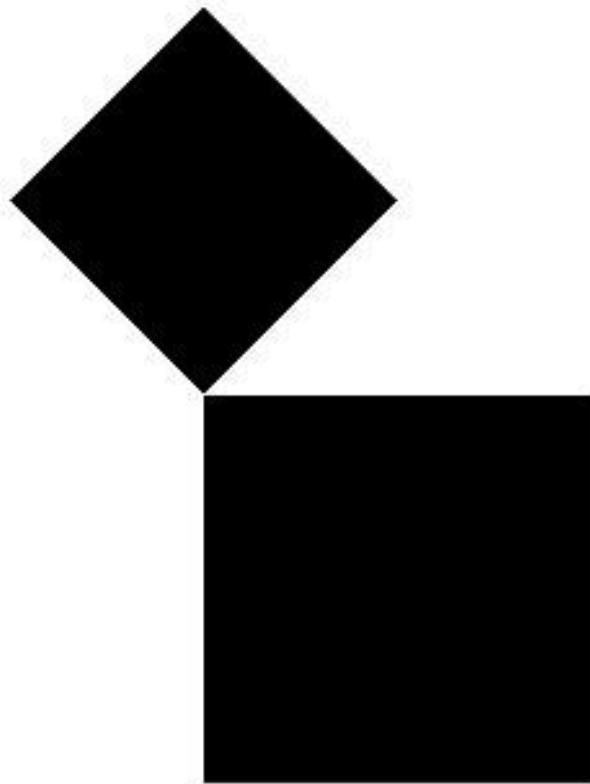


Figure 3.39. Half a Pythagorean tree.

Now, you need to draw the next branch. Remember that the context is centered on the lower-left corner of the small square, and it is rotated to the orientation of that square. The next square needs to be drawn so that it connects the two squares and closes that inner rectangle.

If you translate along the x-axis a distance of `branch0Size`, the context will now be centered on the lower-right corner of that small square. Then, if you rotate 90 degrees, the context will be perfectly oriented to draw that final square. That's the next block of code in `drawPyTree`:

```
// move to bottom right of branch 0
// rotate 90 and draw branch 1
chaos.context.translate(branch0Size, 0);
chaos.context.rotate(Math.PI / 2);
if(depth === 0) {
    drawSquare(branch1Size);
}
else {
    drawPyTree(depth - 1, branch1Size);
}
chaos.context.restore();
```

Again, based on `depth`, either draw a square or iterate a new tree. Restore the context and you're done. Figure 3.40 shows the result.

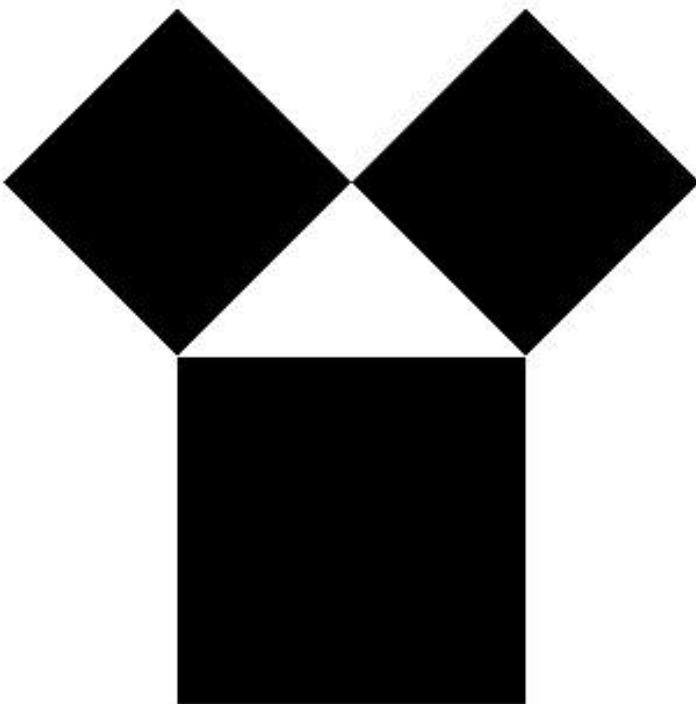


Figure 3.40. Pythagorean tree, single iteration.

Iterating a couple more times gives you Figure 3.41.

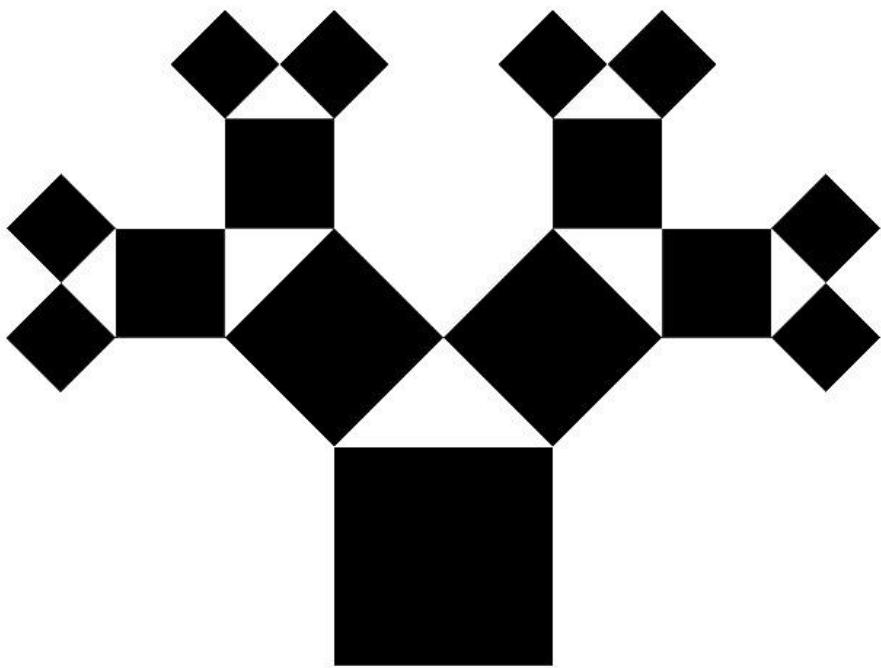


Figure 3.41. Pythagorean tree, more iterations.

And a bunch more iterations give you Figure 3.42.

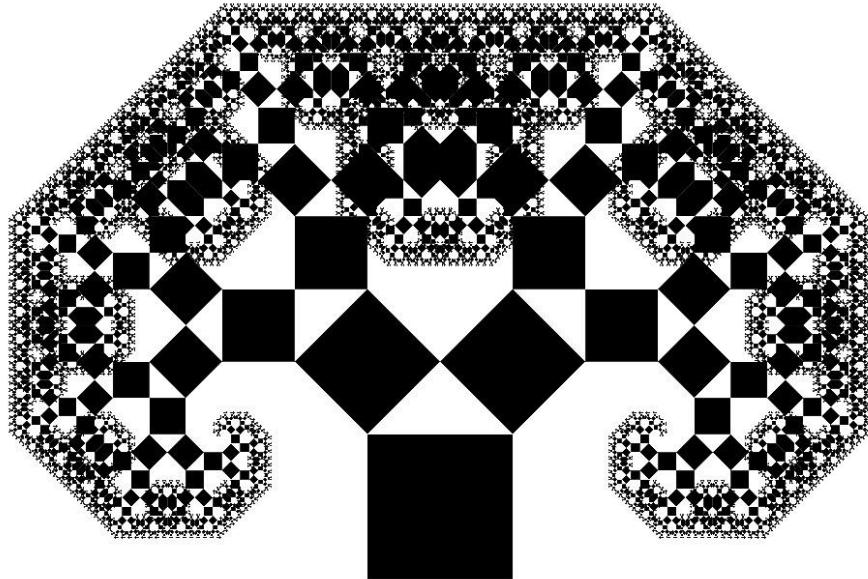


Figure 3.42. Pythagorean tree in full bloom.

Breaking Symmetry

You knew this was coming, right? In this case, you can break symmetry by changing a single line. Just change your `angle` definition to:

```
angle = Math.random() * Math.PI / 2,
```

You'll find this change in the file `pytree2.js`. Now, the first angle will be anywhere from 0 to 90 degrees, changing each time you run the file. The other angle of the triangle will fall in line. If `angle` winds up close to 45 degrees, you'll get a relatively symmetrical tree like the one you just created. If it's very close to 0 or 90 degrees, you'll get a tall,

curving beanstalk that goes quickly off the screen like in Figure 3.43.

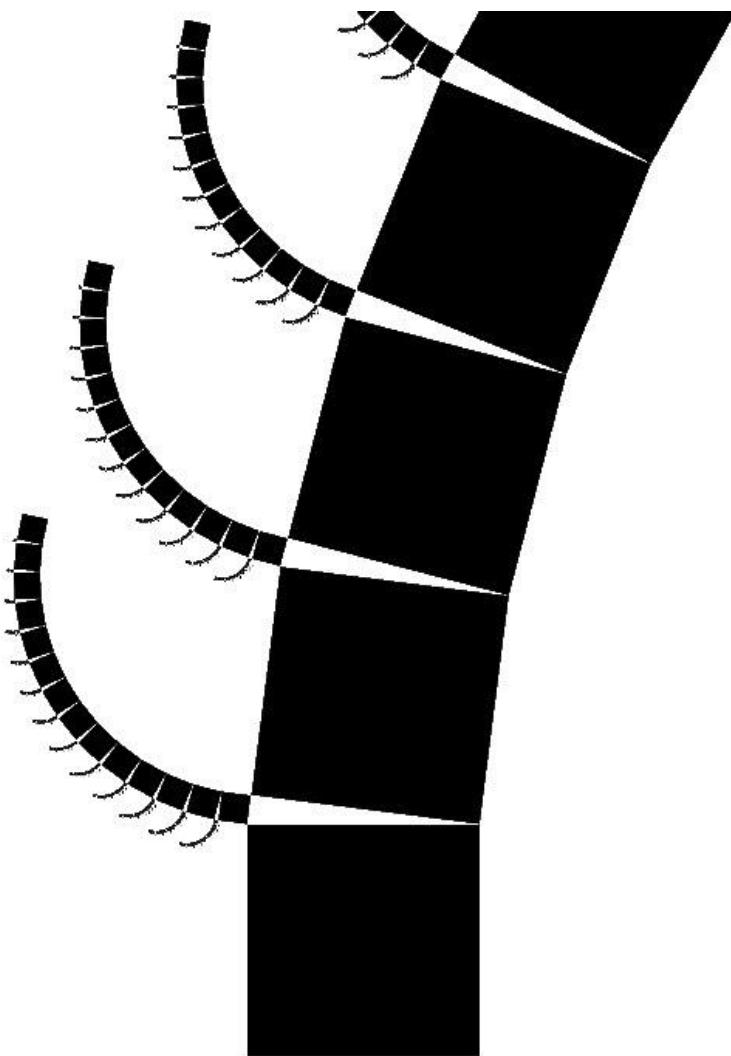


Figure 3.43. Pythagorean beanstalk.

But, when you hit the sweet spot in between these two, you wind up with a cool-looking tree like in Figure 3.44.

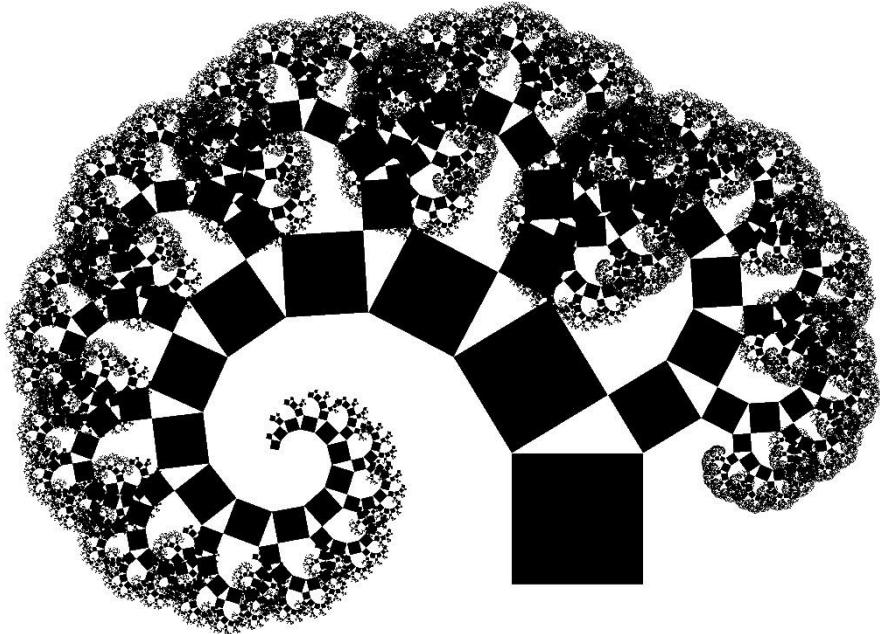


Figure 3.44. Pythagorean perfection.

Breaking Regularity

You were way ahead of me on this one too, right? Again, breaking regularity is easy here. Just add this line to the start of the `drawPyTree` function:

```
angle = Math.random() * Math.PI / 2;
```

You can find this in the file `pytree2.js`. This gives you a random angle every iteration, and creates crazy, wandering trees like in Figure 3.45. Personally, I like this one a lot.

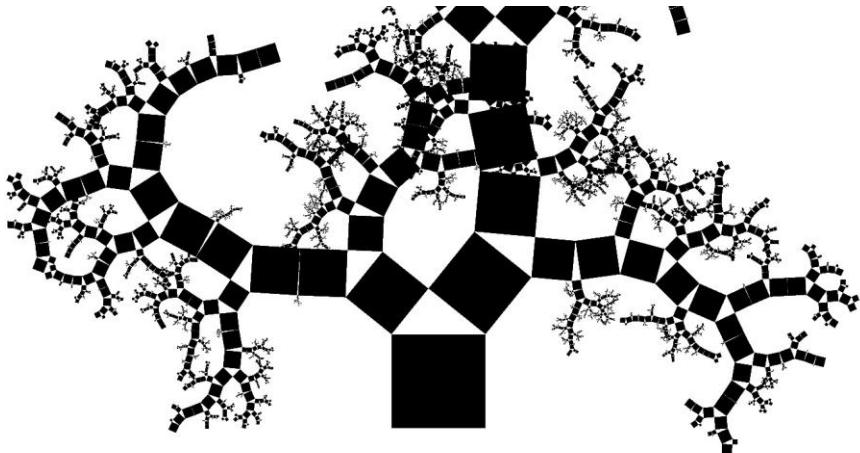


Figure 3.45. Pythagorean irregularity.

Using Color

Black and white trees are nice, but I usually think of trees as having brownish trunks and green leaves. So I created a `setColor` function in `pytree3.js` that takes `depth` as a parameter. This function interpolates between a brown and green color and sets the context's fill style to the specified color:

```
function setColor(depth) {  
    var r0 = 104,  
        g0 = 77,  
        b0 = 5,  
        r1 = 82,  
        g1 = 255,  
        b1 = 53,  
        r, g, b, percent;  
  
    // make sure we don't divide by zero!  
    percent = maxDepth ? depth / maxDepth : 0;  
    r = Math.floor(r1 + percent * (r0 - r1));  
    g = Math.floor(g1 + percent * (g0 - g1));  
    b = Math.floor(b1 + percent * (b0 - b1));  
    chaos.context.fillStyle = "rgb(" + r + "," + g + "," + b  
+ ")";
```

}

To use this, I just call `setColor(depth)` before drawing the trunk in `drawPyTree`:

```
// draw trunk  
setColor(depth);  
drawSquare(size);
```

This gives you a nicely colored tree like you can see in Figure 3.46 (assuming you are reading this on a color-capable device).



Figure 3.46. Pythagorean colors.

As in the other examples, there is still lots of unexplored territory here. Experiment and see what you can create.

Summary

In this chapter, you've created three new fractals and explored symmetry, regularity and how breaking these two things often results in more interesting, natural shapes. All of these concepts will come up in future chapters as you learn about additional types of fractal shapes.

Chapter 4: Fractal Dimensions

In this chapter, I'm going to introduce you to a couple of related fractal forms and delve a bit deeper into some of the characteristics of fractals. I'll start by examining a question first asked by "the father of fractals" himself, Benoît Mandelbrot.

Coastlines

How long is the coast of Britain?

This question was asked by Benoît Mandelbrot in a paper he wrote in 1967¹. You'll learn more about Mr. Mandelbrot in Chapter 7. For now, let's examine this question – and what it has to do with fractals.

At first, it appears to be a straightforward question. Just grab a map of Britain and measure around the perimeter. But coastlines turn out to be complicated things to measure, and the answer to the question of how long a coast is literally depends on the size of your ruler.

For example, if you measure a coastline with a one-kilometer ruler you'll be jumping right over a lot of inlets and protrusions that are less than one kilometer. If you go back and re-measure with a one-meter ruler, you'll find that your measurement will be much larger - but you'll still be skipping over features that are smaller than a meter. Using a shorter ruler will give you an even larger measurement. This will

continue down to an atomic level, at which point your measurement will be ridiculously large. At some point during this process, you will likely decide that trying to accurately measure a coastline is an exercise in futility.

This goes back to the discussion of the Koch curve in Chapter 2. Remember that, before any iterations, the length of the curve was 4 units. Successive iterations gave new lengths of 5.333, 7.111, 9.48, 12.64, 16.85 and up to 946 for 20 iterations. So the measured length of a Koch curve entirely depends on how many times it has been iterated. Infinitely iterated, it is infinitely long.

Okay, hold onto these thoughts. I'll be expanding on them later in this chapter. For now, let's make a computer-generated coastline and attempt to measure it.

The Strategy

Here's the coastline strategy:

1. Start with two points. Draw a line between them. There's your starting coastline. See Figure 4.1.
-

Figure 4.1. Simple coastline.

2. First iteration. Find the midpoint between the two original points. Randomly move it a certain amount. Now, you have two lines, most likely at some sort of angle. See Figure 4.2.

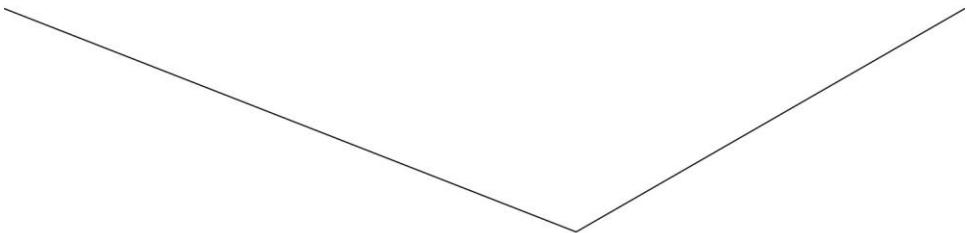


Figure 4.2. Single iteration.

3. Calling the points A, B and C, repeat step 2 for points A and B, and again for points B and C. You'll now have five points with four angled lines joining them. See Figure 4.3.

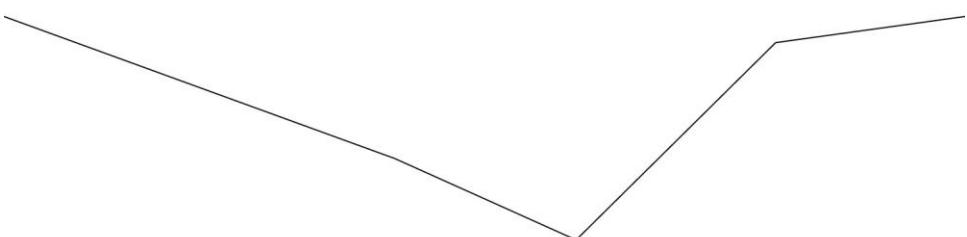


Figure 4.3. Another iteration.

4. Repeat again for each successive pair of points. This gives you nine points and eight lines, as seen in Figure 4.4.

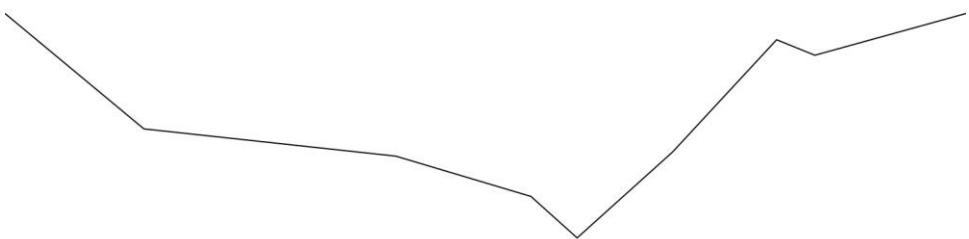


Figure 4.4. Yet another iteration.

5. Continue until the lines are so small that further iterations make no visible difference. The final coastline is shown in Figure 4.5.

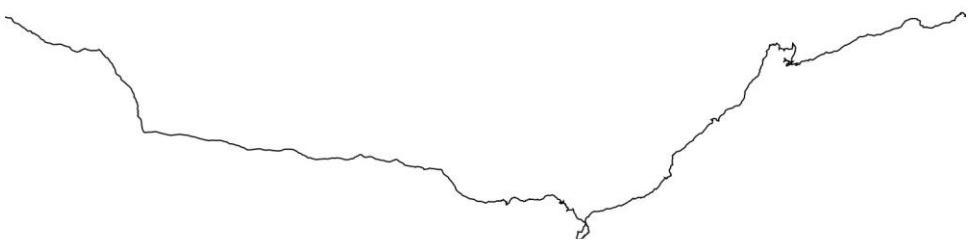


Figure 4.5. Coastline in full.

The Code

You'll need a way to represent individual points. In a less dynamic language, I'd probably make a Point class, but in JavaScript, it's easy enough to use generic objects for such purposes, like so:

```
{  
  x: 100,  
  y: 100  
}
```

Because you'll need to store a large number of points and insert new points between existing points, you can use an array to store the points. Here's the code from the file `coast1.js`:

```
window.onload = function() {  
  var points = [],  
      offset = 0,  
      scaleFactor = .5;  
  
  init();  
  
  function init() {  
  
    chaos.init();  
  
    offset = chaos.height / 2;  
  
    points.push({  
      x: 0,  
      y: chaos.height / 2  
    });  
  
    points.push({  
      x: chaos.width,  
      y: chaos.height / 2  
    });  
  
    drawCoast();  
  
    document.body.addEventListener("keyup",  
      function(event) {  
        console.log(event.keyCode);  
      }  
    );  
  }  
};
```

```

        switch(event.keyCode) {
            case 32: // space
                iterate();
                drawCoast();
                break;

            case 80: // p
                chaos.popImage();
                break;

            default:
                break;
        }
    });

}

function drawCoast() {
    chaos.clear();
    chaos.context.lineWidth = 2;
    chaos.context.beginPath();
    chaos.context.moveTo(points[0].x,
                         points[0].y);
    for(var i = 1; i < points.length; i += 1) {
        chaos.context.lineTo(points[i].x,
                             points[i].y);
    }
    chaos.context.stroke();
}

function iterate() {
    var newPoints = [];
    for(var i = 0; i < points.length - 1; i += 1) {
        var p0 = points[i],
            p1 = points[i + 1],
            newPoint = {
                x: (p0.x + p1.x) / 2,
                y: (p0.y + p1.y) / 2
            };

        newPoint.x += Math.random() * offset
                     * 2 - offset;
        newPoint.y += Math.random() * offset
                     * 2 - offset;
        newPoints.push(p0, newPoint);
    }
    newPoints.push(points[points.length - 1]);
}

```

```
    points = newPoints;
    offset *= scaleFactor;
}
}
```

As you can see, this follows the same basic structure of all the examples in the book so far. At the beginning of the file, there's a `points` array to hold the individual points. Then, an `offset` variable to specify how far a point can be randomly moved. This is assigned a value equal to half the canvas height to ensure that the points stay onscreen.

In the `init` function, two points are created – one on the left side of the screen, one on the right. The `drawCoast` function is called and the standard key handlers are set up, with a space key calling the `iterate` function and then `drawCoast` again.

The `drawCoast` function is simple enough. It clears the canvas, moves to the first point in the array and draws lines to each successive point.

The `iterate` function does all the heavy lifting. This function loops through the array, selecting each pair of points and calculating a new point named `newPoint` that is midway between them. It then alters the `x` and `y` values of this point by a random amount times `offset`. Then, it inserts the first point of the pair into new array, and then this new midpoint is also inserted into the new array. When it's done looping, it adds the last point of the original array to the new array. This `newPoints` array now contains all of the original points, plus the `offset` midpoints. The code then assigns `newPoints` to `points` so that the new array will be used for drawing the next iteration of the coast.

After `iterate` works through the whole array, it multiplies `offset` by `scaleFactor`. This reduces the amount that a point can randomly move on subsequent iterations. So, in the beginning you're creating the large features –

harbors and capes – and later iterations you’re filling in the details with little inlets and jetties.

Run the program and iterate a few times by pressing the space bar. Refresh the page to form a new random coast. A few examples are in Figures 4.6, 4.7 and 4.8.

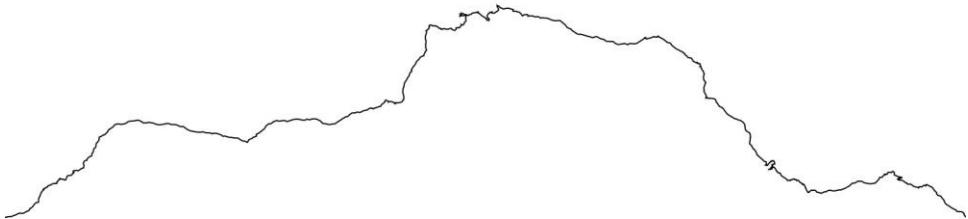


Figure 4.6.

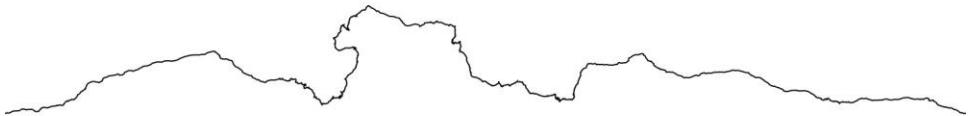


Figure 4.7.



Figure 4.8.

Already, you have a realistic coastline shape. But there are a couple of tweaks you can make. First, you can change the initial offset. I originally set it at half the canvas height to ensure that the first iterated point wouldn't go offscreen. Try making it larger, like the full canvas height. This will give you much larger overall features, such as in Figure 4.9.

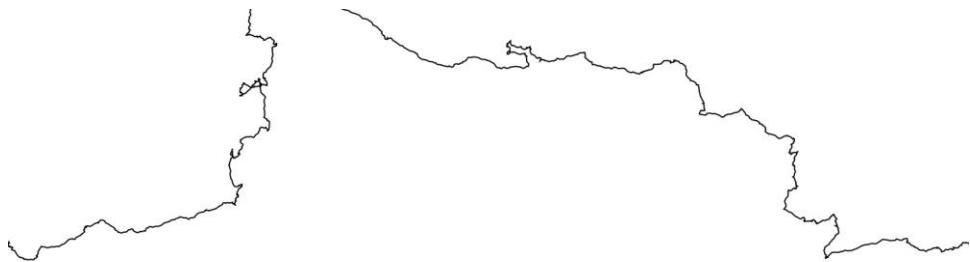


Figure 4.9. Larger initial offset.

Despite the larger initial offset, when you get down to the detailed areas, this last image has much the same look as

the earlier examples. It has the same basic “jaggedness.” You can control that with the `scaleFactor` value. Try setting `scaleFactor` to something like `.25`, and even after a bunch of iterations, you’ll get something like in Figure 4.10.

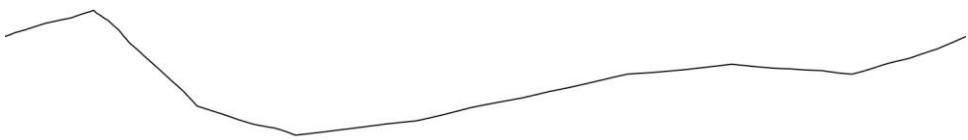


Figure 4.10. Low scaleFactor, smooth coast.

This is because the low `scaleFactor` is reducing the `offset` drastically on each iteration. So, before long, the new points are hardly being offset at all.

Set `scaleFactor` higher, to around `.625`, and you’ll get something that looks like Figure 4.11.

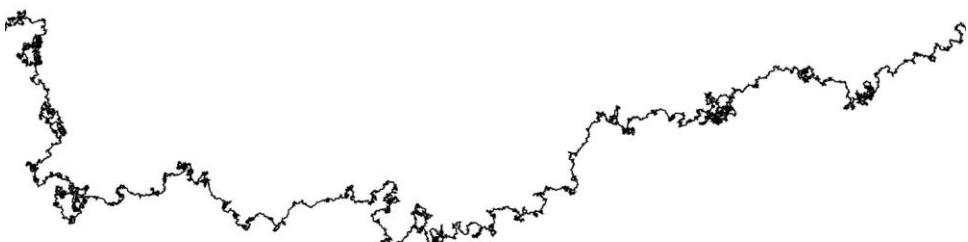


Figure 4.11. High scaleFactor, jagged coast.

Here, `scaleFactor` is not reducing `offset` nearly as much, so you're still getting lots of random change even on a small scale. If you go much higher than this for `scaleFactor`, you're going to start seeing total chaos.

Keep this quality of “jaggedness” or “roughness” in mind. It's a key feature of fractals that I'll be coming back to soon.

I'll also mention briefly that this algorithm can be used for things other than coastlines. For example, by making the line vertical and changing the colors, you can get a half-decent lightning strike. See Figure 4.12 for an example.

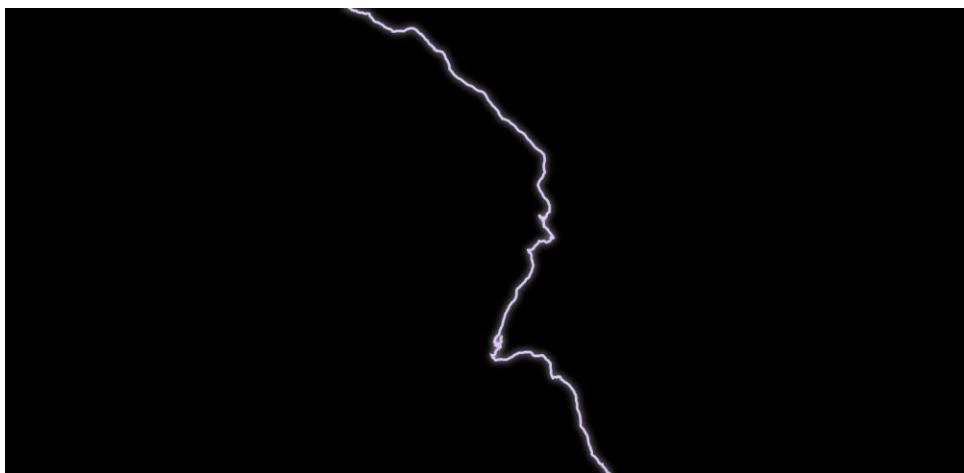


Figure 4.12. Lightning strikes.

I won't go over the code for this one, but you can find it in the file `lightning.js`.

Color

In this section, let's just add some color to make the coastline look even more realistic. It's not about changing the fractal so much as changing how it's rendered to make it look better. Choosing coloring schemes can often bring out

features in fractals that would otherwise be hidden, as you'll see quite clearly in Chapters 7 and 8.

Let's keep it simple and just color the canvas blue and fill in the "land" with a green color. The blue is easy. Just pass in the color you want the background to be when you call `chaos.clear`. Filling the land with green takes a little more work. All of this is contained in the `drawCoast` function:

```
function drawCoast() {  
    chaos.clear("#0033CC");  
    chaos.context.fillStyle = "#00CC00";  
    chaos.context.beginPath();  
    chaos.context.moveTo(points[0].x,  
                         points[0].y);  
    for(var i = 1; i < points.length; i += 1) {  
        chaos.context.lineTo(points[i].x,  
                             points[i].y);  
    }  
    chaos.context.lineTo(chaos.width,  
                        chaos.height);  
    chaos.context.lineTo(0, chaos.height);  
    chaos.context.fill();  
}
```

First, set the fill style to the green color you want to use. Draw the points as before. Then, draw another line from the last point down to the lower-right corner of the canvas, then back over to the lower-left corner. Now, when you call `fill` instead of `stroke`, a final line will automatically be drawn from the last point back to the first point, and the whole lower part of the canvas will become "land." All of these changes are in `coast2.js`. See Figure 4.13 for the result.

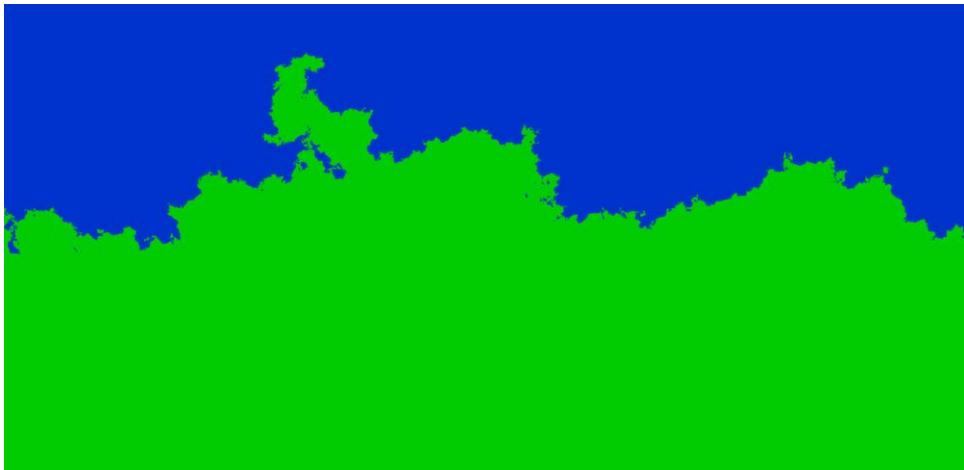


Figure 4.13. A realistic-looking map.

Of course, you could place the land on the top of the map by moving the last two points to the top right and top left of the canvas. Of course, you could generate alien-looking maps by using, well, alien-looking colors.

Islands

In the same way you turned a Koch curve into a Koch snowflake in Chapter 2, you can turn a fractal coastline into a fractal island by creating several points in a circle, or somewhat circular arrangement, and using an array of these points as the starting point for your coastline. The only trick is in making sure the first and last points are connected. The easiest way to do this is to push the first point back into the array a second time as the last point.

Here's the code in `island.js`:

```
window.onload = function() {
    var points = [],
        offset = 0,
        scaleFactor = .6;
```

```
init();

function init() {
    chaos.init();

    var initialPoints = 8,
        radius = chaos.height / 3,
        angle = 0;

    offset = chaos.height / 6;

    for(var i = 0; i < initialPoints; i += 1) {
        angle = Math.PI * 2 / initialPoints * i;
        points.push({
            x: Math.cos(angle) * radius,
            y: Math.sin(angle) * radius
        });
    }

    // put the first point back into the array
    // as the last point
    points.push(points[0]);
}

drawCoast();

document.body.addEventListener("keyup",
    function(event) {
        console.log(event.keyCode);
        switch(event.keyCode) {
            case 32: // space
                iterate();
                drawCoast();
                break;

            case 80: // p
                chaos.popImage();
                break;

            default:
                break;
        }
    });
}

function drawCoast() {
    chaos.clear("#0033CC");
    chaos.context.save();
    chaos.context.translate(chaos.width / 2,
```

```

        chaos.height / 2);
chaos.context.fillStyle = "#00CC00";
chaos.context.beginPath();
chaos.context.moveTo(points[0].x,
                     points[0].y);
for(var i = 1; i < points.length; i += 1) {
    chaos.context.lineTo(points[i].x,
                         points[i].y);
}
chaos.context.fill();
chaos.context.restore();
}

function iterate() {
    var newPoints = [];
    for(var i = 0; i < points.length - 1; i += 1) {
        var p0 = points[i],
            p1 = points[i + 1],
            newPoint = {
                x: (p0.x + p1.x) / 2,
                y: (p0.y + p1.y) / 2
            };

        newPoint.x += Math.random() * offset
                     * 2 - offset;
        newPoint.y += Math.random() * offset
                     * 2 - offset;
        newPoints.push(p0, newPoint);
    }
    newPoints.push(points[points.length - 1]);
    points = newPoints;
    offset *= scaleFactor;
}
}

```

In the `init` function, set the island to be a circle with eight points and a radius of one-third the canvas height. Finally, add the first point back into the array as the last point to close the circle. Also, note that `offset` is set to be quite a bit smaller for now. That's it for setup. The `iterate` function doesn't need to change at all. The rest of the changes are in `drawCoast`.

The major change here is that the context is saved and translated to the center of the screen so that the circle of the

island will be drawn there. Of course, you no longer need the extra points that were added to fill the lower portion of the screen. Just draw the points in the array, fill and finish up by restoring the context.

Running this code gives you Figure 4.14.

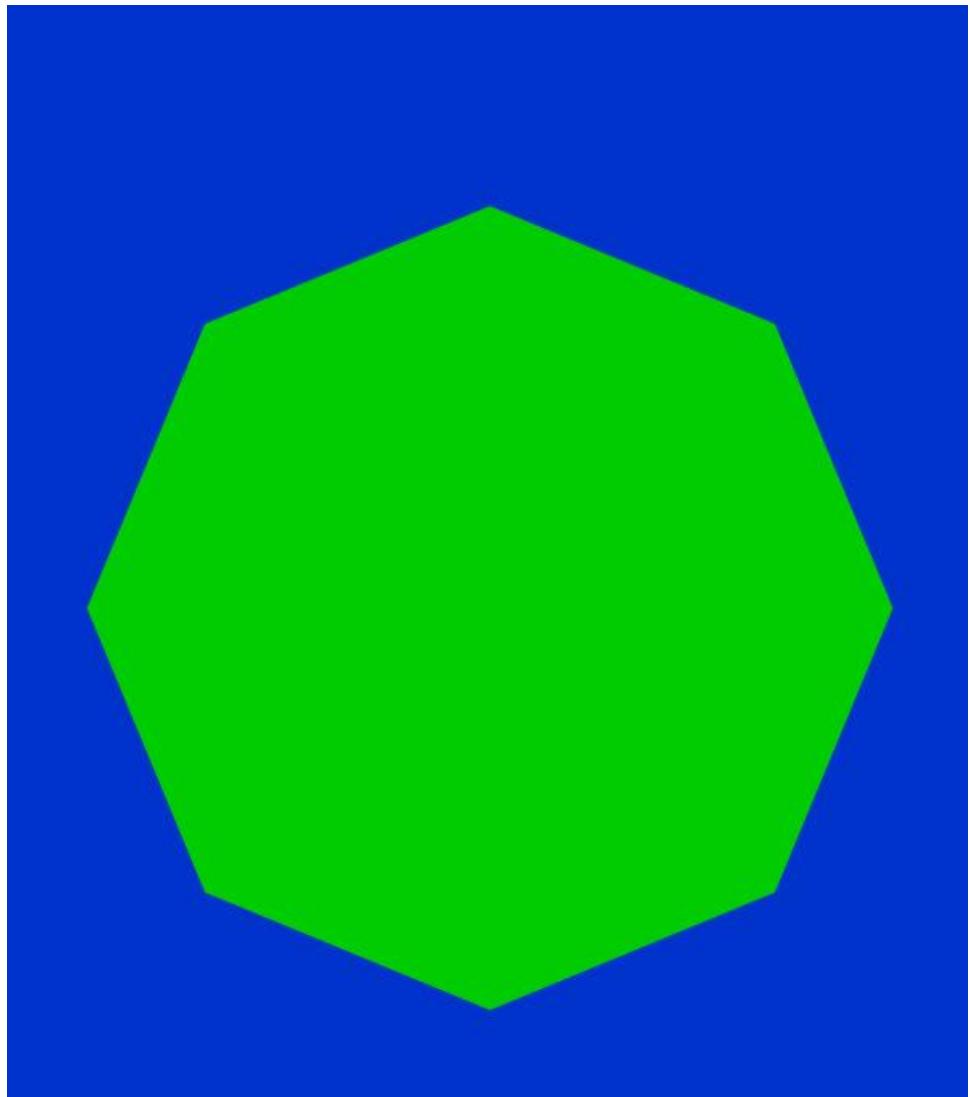


Figure 4.14. A prototypical island.

That's not very realistic yet, so let's iterate it one time to get Figure 4.15.

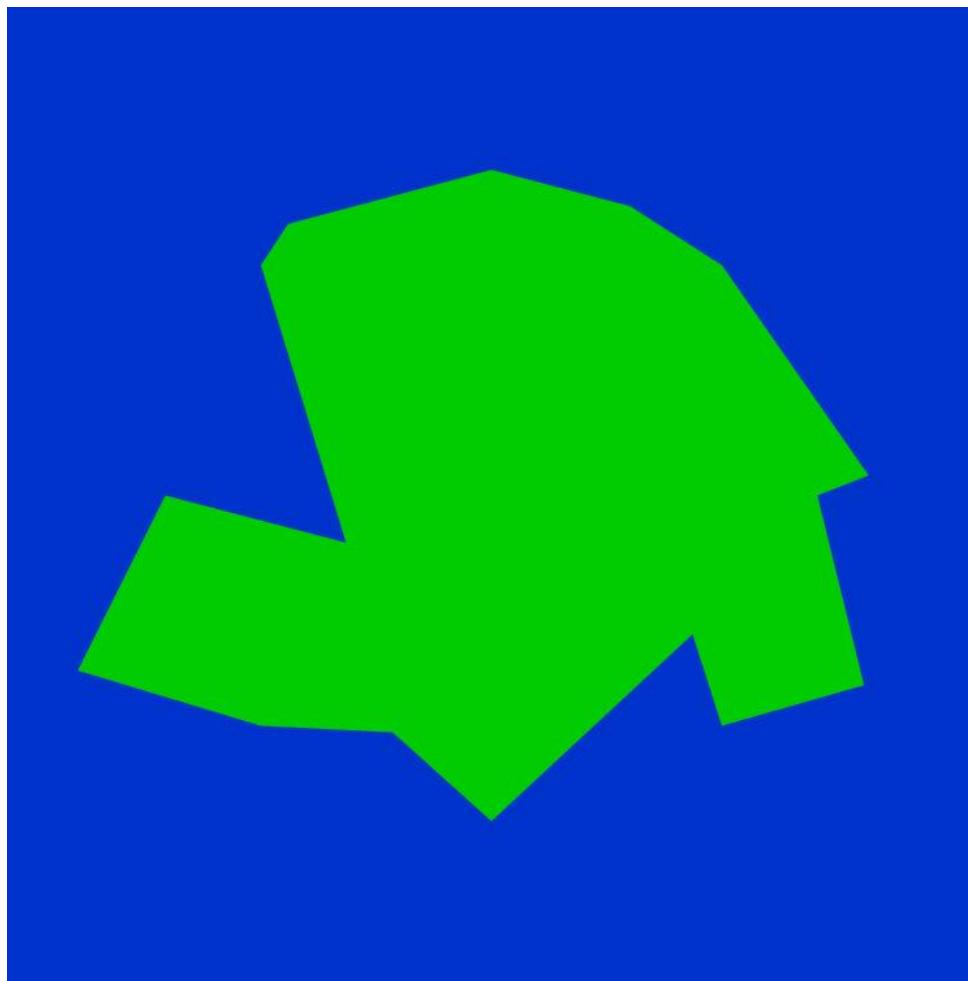


Figure 4.15. Looking a bit more islandish.

Iterating several more times gives you Figure 4.16.

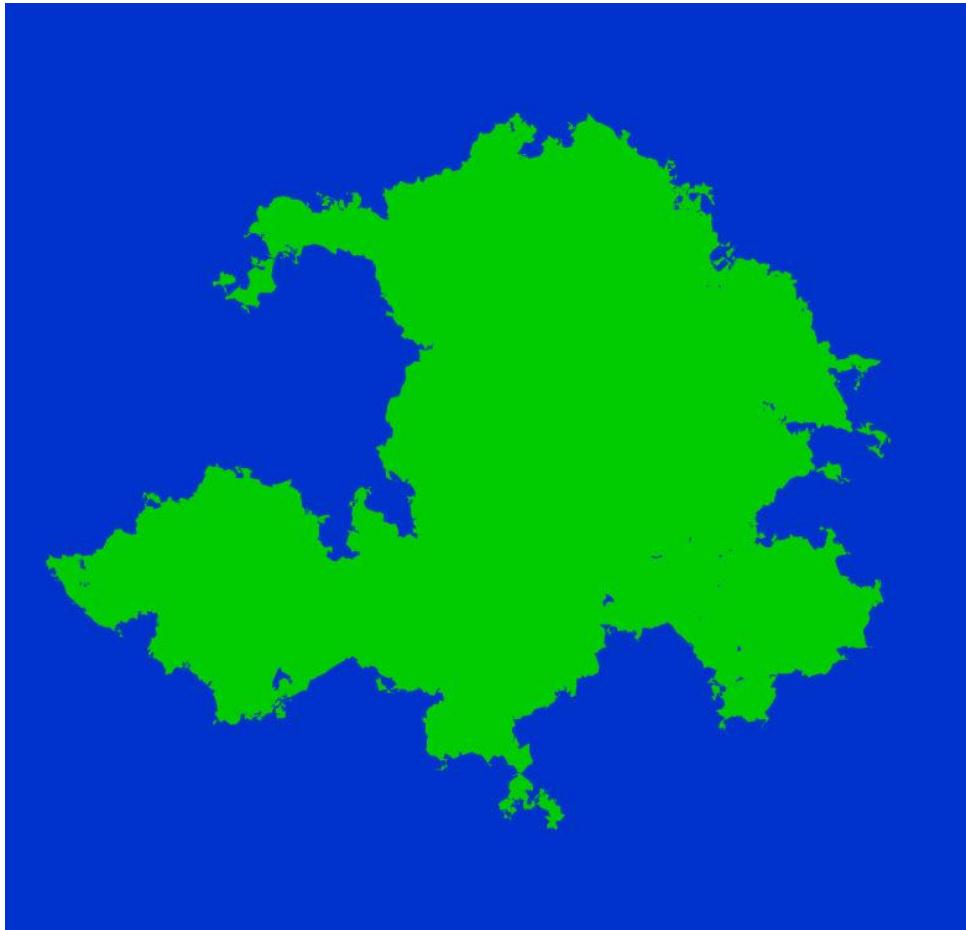


Figure 4.16. Now that's an island!

And there you have an island any mapmaker would be proud to have in his books.

Taking Measurements

Okay, I've let you make a bunch of pretty pictures, and now it's time to dive into a little more background. In the beginning of this chapter, I talked about measuring coastlines. Now that you've built a few coasts, let's measure

some of them and see what kinds of characteristics they have.

Going back to the original `coastline1.js` example, rather than considering that you are iterating to create the coastline, imagine instead that the coastline already exists and that each iteration is a new measurement of it with a smaller ruler.

Initially, the ruler is about the same width as the screen. So you take a single measurement from the start to the end. If your screen is 1600 pixels wide, you would get 1600 as a measurement. In the first iteration, the ruler is roughly half the size, and you can get two measurements. This results in a slightly more accurate – and slightly longer – measurement. Subsequent iterations use smaller and smaller rulers, getting closer to the actual shape of the coastline, and resulting in longer, more accurate measurements.

Now, this isn't a perfect analogy because each line segment is a slightly different length, depending on where the random offsets landed. If you were measuring by actually laying a ruler end to end, each line segment would be the same length. But this method will still show the concept I'm trying to get across.

I've added a bit of code to the original coastline file in `coast3.js`. It's mostly the same as `coast1.js`, so I'll just give you the differences here. First, I added a function called `measureLine`:

```
function measureLine(p0, p1) {  
  var dx = p1.x - p0.x,  
      dy = p1.y - p0.y;  
  return Math.sqrt(dx * dx + dy * dy);  
}
```

This takes two points and returns the distance between them.

Then, in the `drawCoast` function, I measure the distance between each successive pair of points, tallying the total. This will give you the length of the coastline at its current iteration.

The interesting thing about this exercise is not in any single measurement you take but in how the measurements increase across a series of iterations. And how changing the `scaleFactor` value alters this rate of increase.

For example, using a `scaleFactor` of `.25`, I got the following measurements for the first line and eight subsequent iterations:

```
1536  
1568.4  
1570.6  
1571.5  
1571.8  
1571.9  
1571.9  
1571.9
```

As you can see, the length took a bit of a jump in the first couple of iterations, but the low `scaleFactor` squashed that change quickly, with no discernible difference in length in the last four iterations.

On the other side of the spectrum, using `.65` for `scaleFactor`, I got these values:

```
1536  
1590  
1662.3  
1745.4  
1889.2  
2061.5  
2357  
2917.4  
3462.4
```

In this case, rather than leveling off, the rate of increase of length actually SPEEDS UP in later iterations.

I went ahead and tried this for several values of `scaleFactor` (.25, .35, .45, .55 and .65) and ran through eight iterations of each. I won't bore you with a table of all the results, but Figure 4.17 shows the results in a graph.

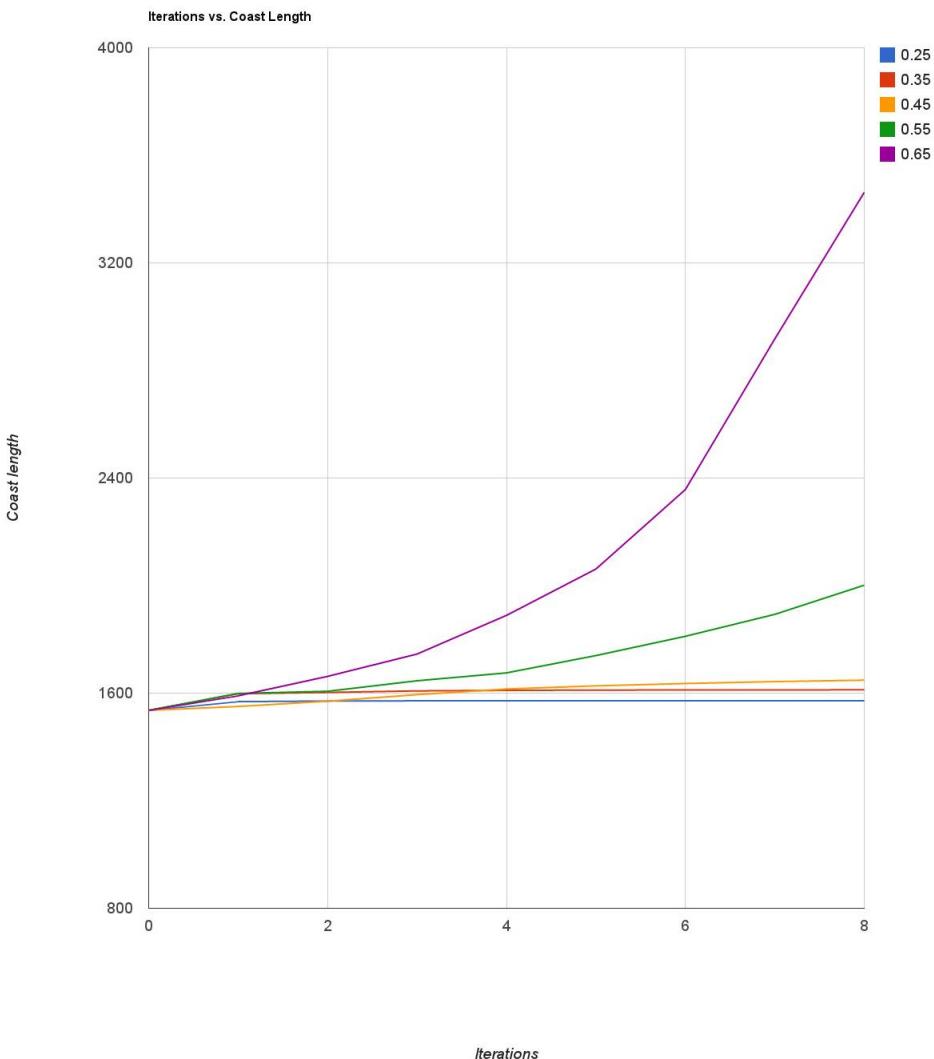


Figure 4.17. Coastline measurements.

What this shows is the number of iterations across the bottom and the measured length of the coastline on the left.

The five lines on the graph represent multiple measurements for the five different values of `scaleFactor`. The line lying on the bottom represents the .25 `scaleFactor`, and the .65 one is the one at the top that nearly jumps off the graph. This demonstrates that, in addition to just looking different, a fundamental difference exists in these coasts that has to do with how “measurable” they are.

Of course, I am not the first person to notice this feature. In the next couple of sections of this chapter, you’ll learn what other people have found out about it.

Fractal Dimensions

The Richardson Effect

Earlier in the chapter, I mentioned an article by Benoît Mandelbrot. The title of that article is HOW LONG IS THE COAST OF BRITAIN? STATISTICAL SELF-SIMILARITY AND FRACTIONAL DIMENSION. In it, he refers to work by Lewis Fry Richardson. Richardson had an impressive resume, working as a chemist, a physicist and a meteorologist at different points in his life. As a mathematician, he attempted to apply computational formulas to forecast the weather and analyze wars with the aim of preventing them. (He was a devout pacifist.)

He also made a study of coastlines by actually measuring them at different scales, as described earlier in this chapter. Through this research, he demonstrated that when you measure a coastline at smaller and smaller intervals, the results do not approach a final accurate measurement.

Instead, the length increases without limit as the ruler size decreases to zero. In other words, the coast of Britain could theoretically be said to be infinite! This seeming paradox is known as “the Richardson effect”.

Richardson also gave a formula for calculating the length of a coastline. This formula, of course, included as a parameter the length of the measuring device, but also another constant that would be between one and two. This constant is called the dimension. For a theoretically perfectly straight coast, the dimension would be one. The more jagged or rocky a coastline is, the higher the dimension would be, with a dimension of two being the limit for an extremely jagged coast.

He gave his dimension constants for various places on Earth. The coast of South Africa was one of the smoothest, with a dimension of 1.02. Australia comes in at 1.13. The border between Spain and Portugal (not a coastline) has a dimension of 1.14. Germany’s coast is 1.15, and the west coast of Britain is a whopping 1.25, one of the roughest measured coastlines on Earth.

Measuring Fractals

Mandelbrot referenced Richardson’s work in his article, and went on to extend it beyond physical coastlines and into the realm of mathematical figures, such as fractal curves. In this article, he invented the term “fractal dimension.”

You know that a line has one dimension, a plane has two and a cube has three dimensions. Mandelbrot demonstrated that certain structures, such as coastlines and mathematically constructed fractals, have a dimension that is more than one, but they fall short of being two-

dimensional figures. He called this fractional dimension between one and two “fractal dimension.” It corresponds exactly to Richardson’s dimension constant.

Furthermore, he was able to measure various known fractal shapes and find their fractal dimensions. For example, the Koch curve has a dimension of 1.26 – just a bit more than Britain’s west coast! The Sierpinski gasket, however, has a dimension of 1.585.

I’m not going to go any further into this subject here, but if it’s something that interests you, feel free to research it further.

Landscapes

Okay, that last section might have gotten a little bit heavy. Let’s lighten things up with one more quick fractal image before moving on.

The final fractal in this chapter will be a fractal landscape. Whereas the coastline fractal was a top-down view of a section of coast, or of an island, this one will be more of an elevation view of a mountain range.

Actually, the code is nearly identical to the original `coast1.js` I started out the chapter with. In fact, I feel a little like I’m cheating by presenting this as an entirely new fractal. The truth is, all you need to do is comment out or remove one line from that file, the line in the `iterate` function that says:

```
newPoint.x += Math.random() * offset * 2 - offset;
```

Remember that, in that function, you were creating a new point at the midpoint of each pair of points and then

offsetting it randomly on the x- and y-axes. In this case, you'll only be repositioning it on the y-axis. This means that all the variations will be vertical, and you'll wind up with a picture like in Figure 4.18.



Figure 4.18. Mountains.

As with the coastlines, you can adjust the initial offset value as well as the `scaledFactor` to get different types of landscapes. A low value like `.25` for `scaledFactor` will give you the type of rolling hills you can see in Figure 4.19.

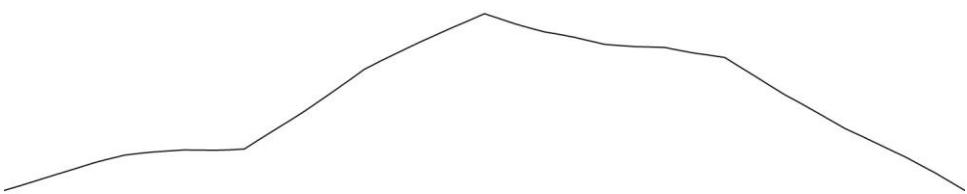


Figure 4.19. Rolling hills.

A higher value like .55 will give you rocky valleys and peaks like in Figure 4.20.



Figure 4.20. The Alps?

Going much beyond that will give you something that looks more like a stock chart than a mountain. Try your hand at coloring these pictures if you want as well.

One last thing to mention about this type of landscape-generation is that it is the actual algorithm used in many 3D landscape generation tools. Instead of starting with a single line made of two points, start with a triangle made of three points. Subdivide the triangle into smaller triangles and move the new points up or down randomly. Continue that way for a few iterations and you'll have a realistic-looking 3D landscape.

Summary

In Chapter 4, you've learned about a couple more types of fractals and some background and theory behind them. You've also learned that fractal curves can be classified by

their roughness, which corresponds to their fractal dimension – a dimension existing between one and two.

In the next chapter, you'll see how fractal order can arise from seeming chaos, and see some more of a familiar shape.

¹ Benoît Mandelbrot (1967). “How Long Is the Coast of Britain? Statistical Self-Similarity and Fractional Dimension”, *Science, New Series*, Vol. 156, No. 3775. (May 5, 1967), pp. 636-638. doi:10.1126/science.156.3775.636

Chapter 5: Order from Chaos

In this chapter, you'll learn about iterated function systems and see how, through a series of random commands, order can arise. You'll also meet another member of the fractal community, Michael F. Barnsley, who originally introduced the techniques you'll be learning about in this chapter. Barnsley is a mathematician and researcher who has written several papers and books on fractals and is a pioneer in the field of fractal image compression.

The Chaos Game

The Chaos Game is a term invented by Barnsley that describes a technique for generating various kinds of fractals. The interesting thing about this technique is that it allows for randomness in every step. But, with just a few simple rules, amazing fractal images are created.

If you have not seen this technique in action yet, I'm sure you'll be amazed at what appears on the screen the first time you run the program. No looking ahead! But, even if you do peek, I think it's still hard to believe the first few times you see it.

The Strategy

1. Start out by plotting three points on the screen. It doesn't matter where they are. In this first example, I'll arrange them in an equilateral triangle, as in Figure 5.1.



Figure 5.1. Step 1: Plot three points.

- Now, plot another random point somewhere in the general area of the first ones. I'll call this the "current point." See Figure 5.2.

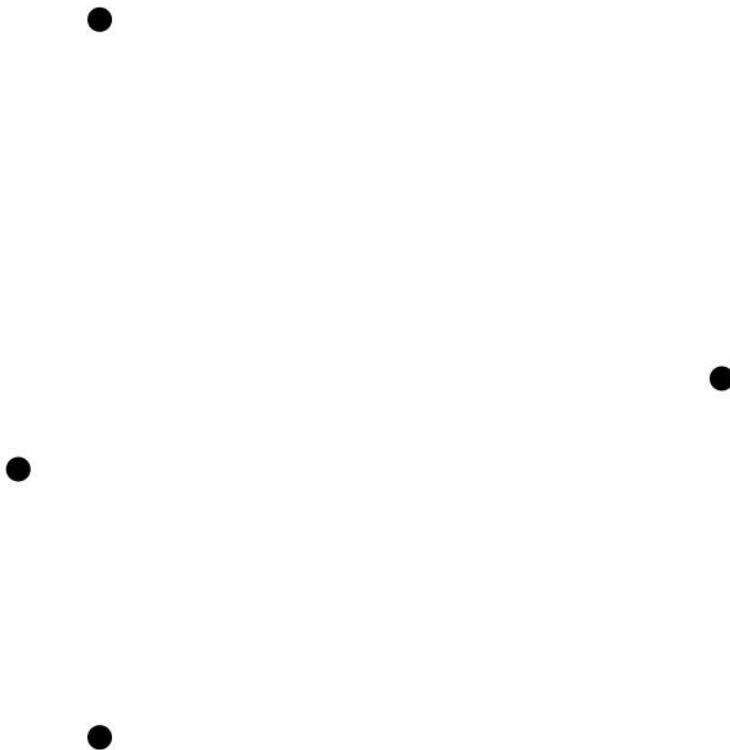


Figure 5.2. Step 2: Plot another point.

- Randomly choose one of the three points from the first step. Plot another point halfway between the current point and this random point. In Figure 5.3, I randomly chose the far-right point of the original three, so the new point goes between that and the current point.

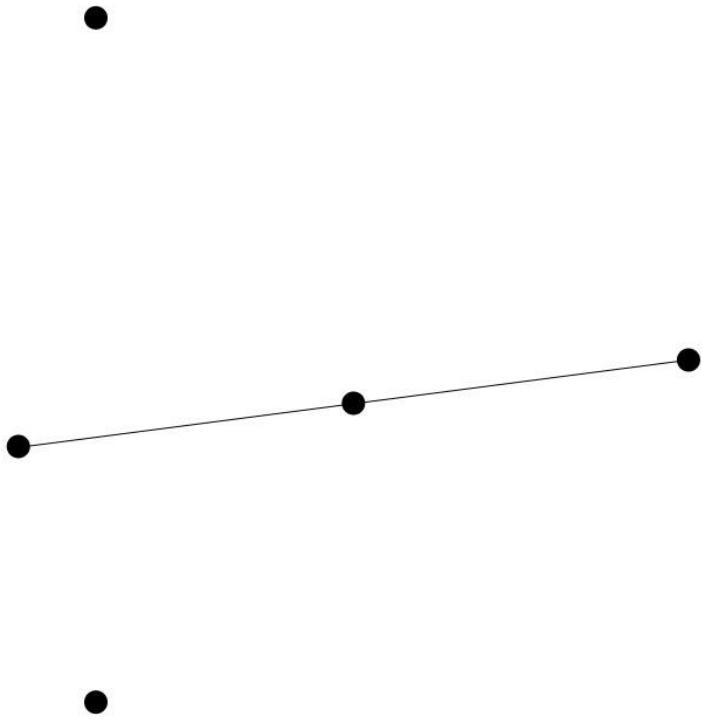


Figure 5.3. Step 3: New point between current and random original point.

4. This latest point becomes the new current point.

Again, choose a random point from the original three and plot a new point halfway between it and the current point. See Figure 5.4.

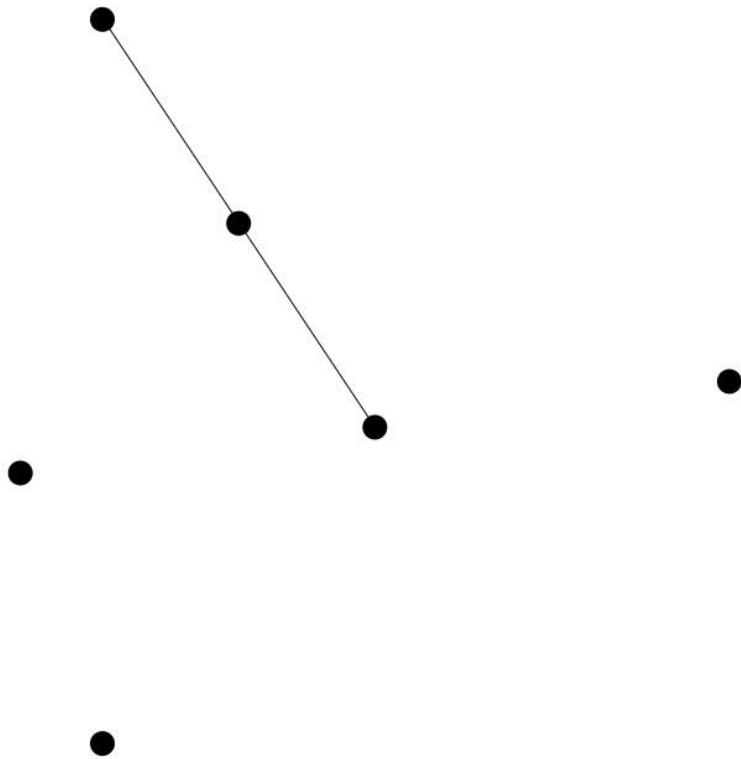


Figure 5.4. Step 4: Another halfway point.

5. Again, the last point plotted becomes the current point. Choose another random point and plot another halfway point to get to Figure 5.5.

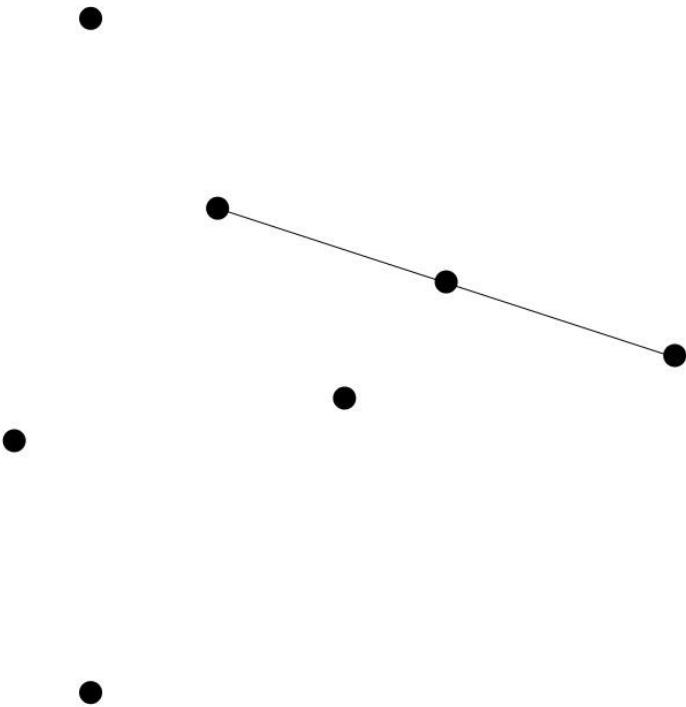


Figure 5.5. Step 5: Yet another halfway point.

And you just continue this way until something starts to take shape. Any guesses what it might be? If you want to see it in action first, stop here and run the example in files `chaosgame1.js` and `chaosgame1.html`. Otherwise, read on and we'll delve into the code that's in those files.

The Code

```
window.onload = function() {  
    var points = [],  
        numPoints = 3,  
        r = .5,
```

```
pointSize = 1,
currentPoint,
interval;

init();

function init() {

    chaos.init();

    var radius = chaos.height * .45,
        angle = 0;

    chaos.context.translate(chaos.width / 2,
                           chaos.height / 2);

    for(var i = 0; i < numPoints; i += 1) {
        angle = Math.PI * 2 / numPoints * i;
        points.push({
            x: Math.cos(angle) * radius,
            y: Math.sin(angle) * radius
        });
        setPoint(points[i]);
    }

    currentPoint = {
        x: Math.random() * radius * 2 - radius,
        y: Math.random() * radius * 2 - radius
    }
}

document.body.addEventListener("keyup",
    function(event) {
        switch(event.keyCode) {
            case 32: // space
                nextPoint();
                break;

            case 187: // +
                clearInterval(interval);
                interval = setInterval(
                    function() {
                        for(var i = 0; i < 10; i += 1) {
                            nextPoint();
                        }
                    },
                    0);
        }
    }
);
```

```

        break;

    case 189: // -
        clearInterval(interval);
        break;

    case 80: // p
        chaos.popImage();
        break;

    default:
        break;
    }
});

}

function nextPoint() {
    var randomPoint = getRandomPoint(),
        newPoint = {
            x: (currentPoint.x
                + randomPoint.x) * r,
            y: (currentPoint.y
                + randomPoint.y) * r
        };
    setPoint(newPoint);
    currentPoint = newPoint;
}

function getRandomPoint() {
    var index = Math.floor(Math.random()
        * numPoints);
    return points[index];
}

function setPoint(p) {
    chaos.context.beginPath();
    chaos.context.arc(p.x, p.y, pointSize,
        0, Math.PI * 2,
        false);
    chaos.context.fill();
}
}

```

This example has the same basic structure as examples in previous chapters, but with some significant changes. First, let's look at the variables:

```
var points = [],
    numPoints = 3,
    r = .5,
    pointSize = 1,
    currentPoint,
    interval;
```

Here, you can see an array to hold the initial points and the number of initial points to create. The `r` variable specifies where to put the new point on that line between the randomly chosen point and the current point. I told you to put it halfway between the two, so now `r` equals `.5`. This can change later.

The `pointSize` is simply the radius of the points drawn. I've set that to `10` in the images I've shown you so far so you could see what is happening. You may also want to keep `pointSize` at `10` initially so you can see where the points are being drawn. But, for final images, you'll want it at `1` or even `.5` to get better resolution.

You can guess what `currentPoint` is already, and `interval` will be used to automate the drawing.

Okay, on to the functions. The `init` function translates the context to center screen and draws the specified number of original points around that center. It then randomly creates the first current point. All points are drawn with the `setPoint` function down at the bottom of the code.

Pressing the space bar will call `nextPoint` a single time, plotting the next point. Pressing the "+" key (on the main keyboard, not the keypad) will start an interval timer, calling the `nextPoint` function many times per second. And pressing the "-" key will cancel that interval timer. You might want to

use the space bar at first to see what's being drawn slowly, but use the + key if you don't want to wait for it.

The `nextPoint` function first gets a reference to one of the initial points using the `getRandomPoint` function. It then calculates a new point midway between the current point and this new random point. It's an exact midpoint because `r` is equal to `.5`. If you change `r` to, say, `.333`, the new point will be roughly one-third of the distance between the two points. The function then plots this point and assigns it to `currentPoint` for the next iteration.

Now, it's time to see what happens when you run this. After a little more than 100 iterations, you get Figure 5.6.

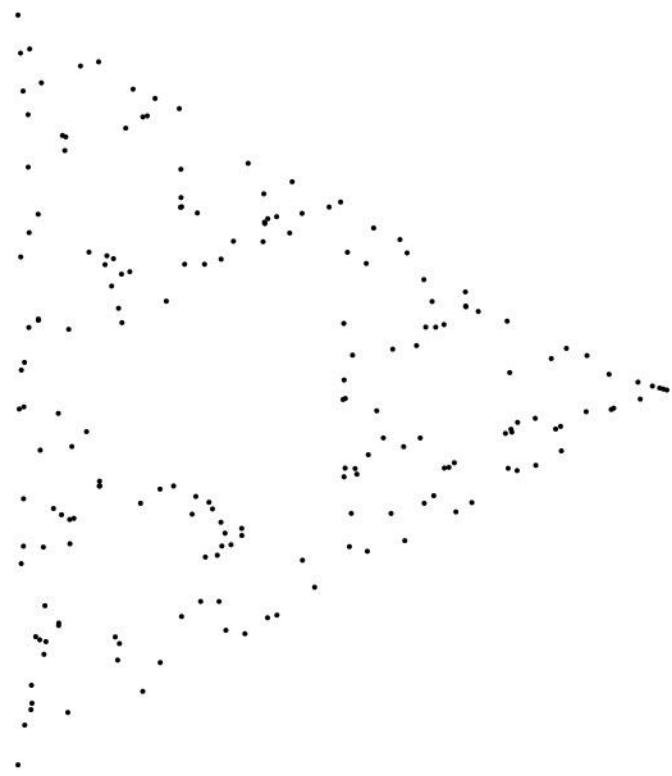


Figure 5.6. Do you see it yet?

If you hit the + key and sit back, after a few seconds, you'll see an old familiar shape. See Figure 5.7.

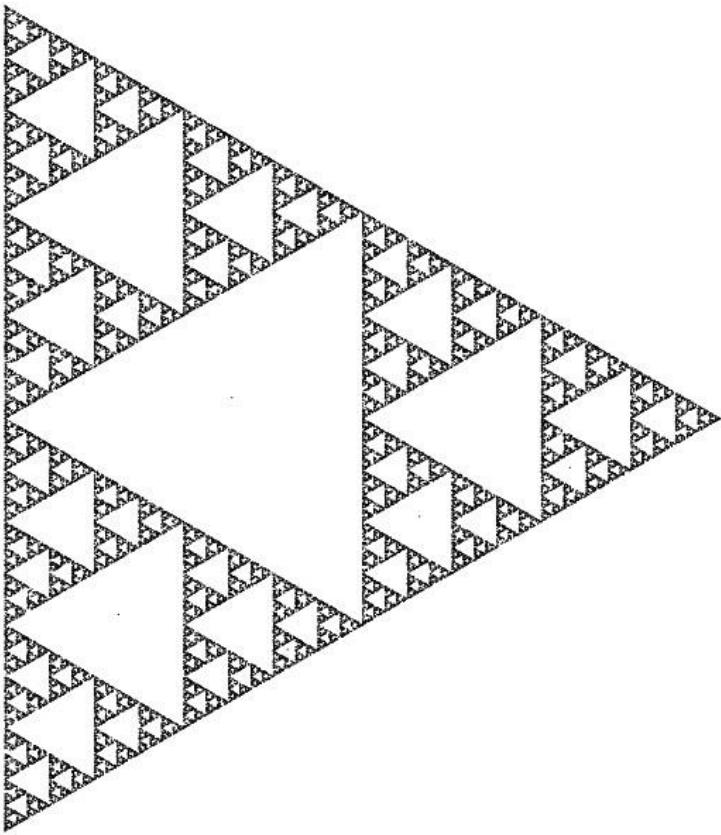


Figure 5.7. Sierpinski emerges.

So, there you have it, order from chaos. I've presented this example to a large audience several times. I usually set the interval a bit slower to give the image a few seconds to build up. There's always a collective "oooooh" sound from the audience as they realize what's forming on the screen.

Variations

There are a few ways to change the code to create different images. First, you can alter the value for r . In a sense, this controls the scale of the sub-images. Setting it at .5 gives you a perfect Sierpinski gasket. But using .4 gives you Figure 5.8.

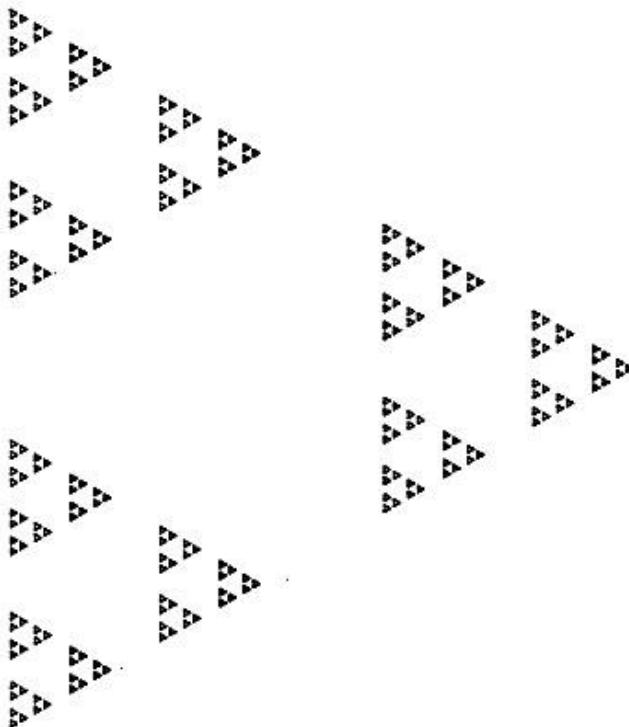


Figure 5.8. $\text{numPoints} = 3$, $r = .4$.

Here, the triangles that make up the whole fractal are not quite large enough to reach each other. Set r to .6,

however, and the triangles get a bit too big and overlap, as you can see in Figure 5.9.

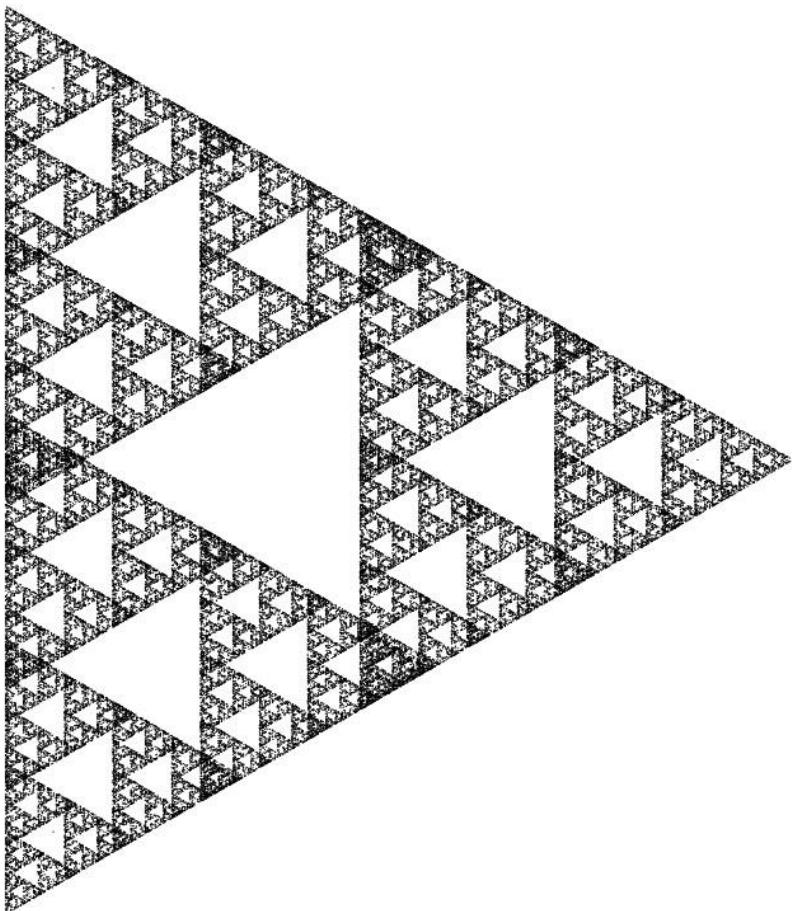


Figure 5.9. $\text{numPoints} = 3$, $r = .6$.

Next, you can change the number of initial points. If three points give you a fractal triangle, you could guess that four initial points will give you a square. And you'd be right, as Figure 5.10 shows. Note that with a larger number of points, the value for r needs to be reduced to avoid overlap.

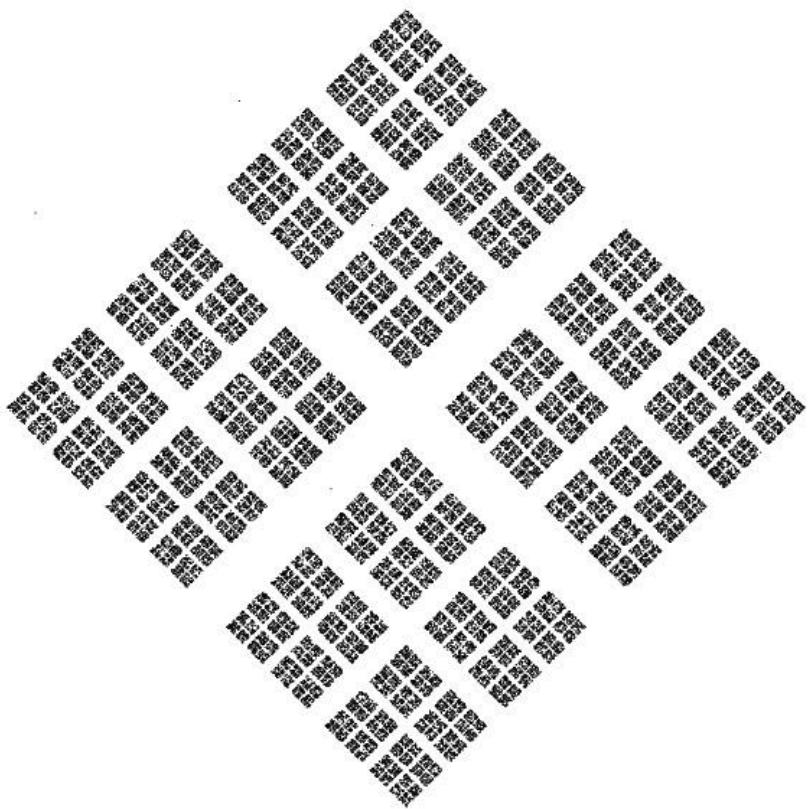


Figure 5.10. numPoints = 4, r = .4.

Five initial points will result in a pentagon made of pentagons ... made of pentagons. See Figure 5.11.

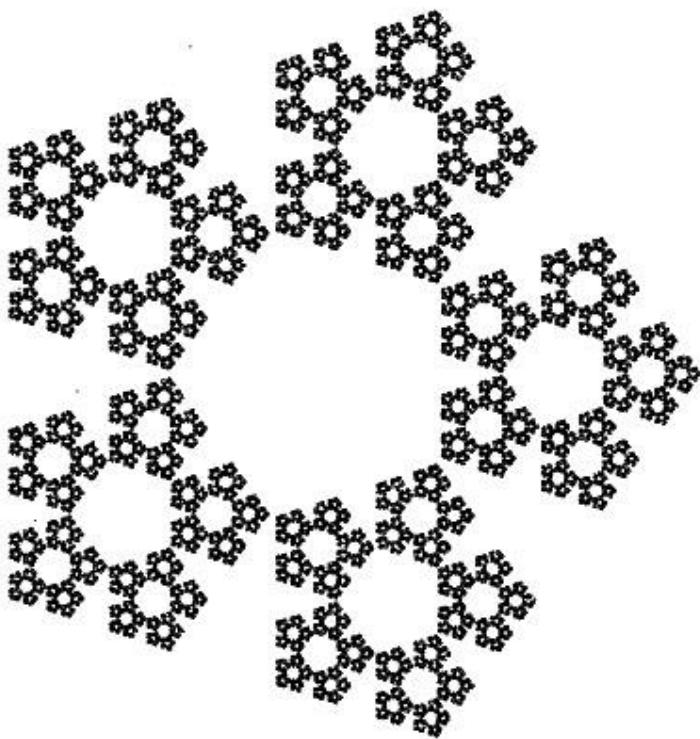


Figure 5.11. $\text{numPoints} = 5$, $r = .375$.

A fractal hexagon forms in Figure 5.12 with six initial points.

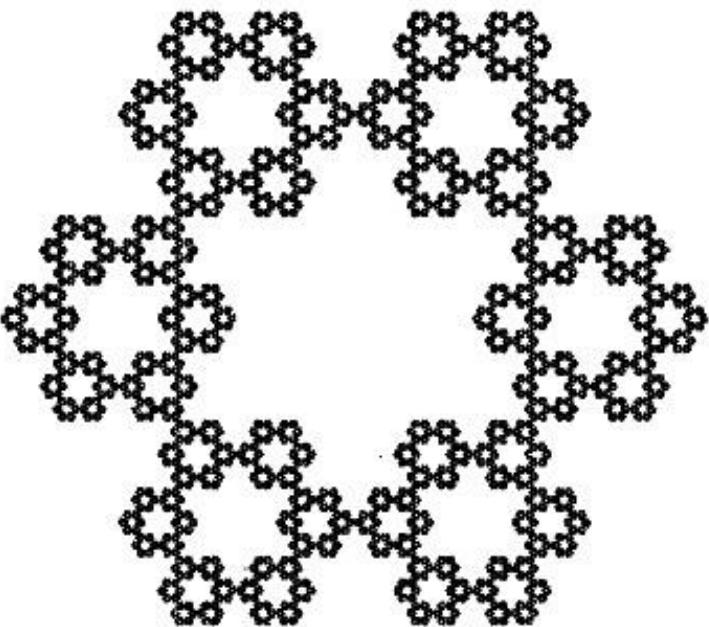


Figure 5.12. numPoints = 6, r = .333.

And why not an octagon? In Figure 5.13, I had to bump up the radius that was used to create the initial points because the low r value makes the resulting shapes quite small.

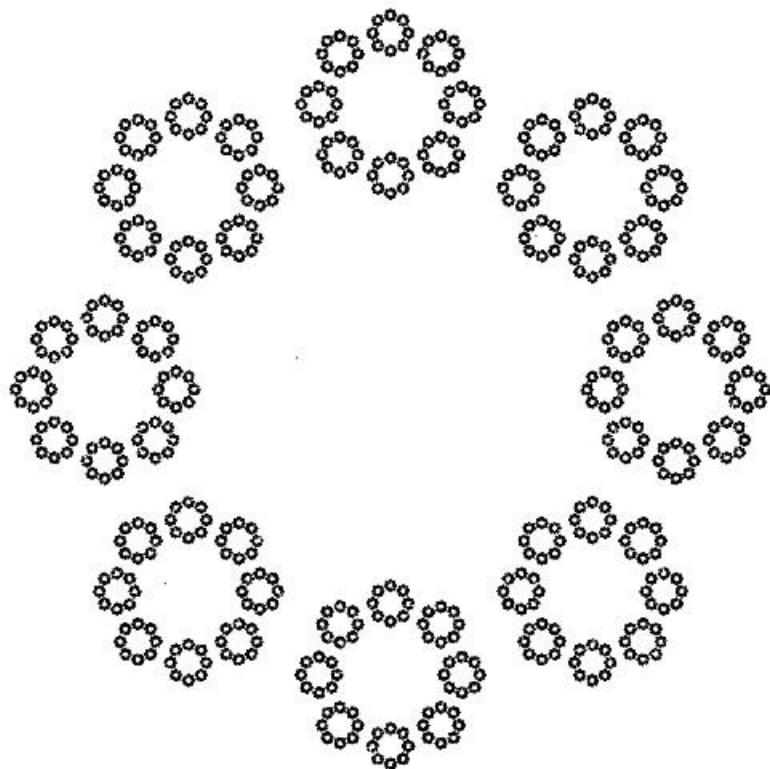


Figure 5.13. $\text{numPoints} = 8$, $r = .25$.

Random Initial Points

Earlier, in the strategy section for this example, I said that it doesn't matter where you plot the initial points. But the code you are using is always arranging them to form a regular polygon. Thus, the resulting shapes are also regular polygons.

To demonstrate my initial statement, I've created `chaosgame2.js`, which plots the initial points randomly. The only change is in the `for` loop that creates those points in the `init` function, which I'll give you here:

```
for(var i = 0; i < numPoints; i += 1) {  
    angle = Math.PI * 2 / numPoints * i;  
    points.push({  
        x: Math.cos(angle) * radius  
            + Math.random() * radius / 2  
            - radius / 4,  
        y: Math.sin(angle) * radius  
            + Math.random() * radius / 2  
            - radius / 4  
    });  
    setPoint(points[i]);  
}
```

Here, I just added a random amount to the initial placement. With three initial points, you can get images like Figures 5.14 and 5.15.

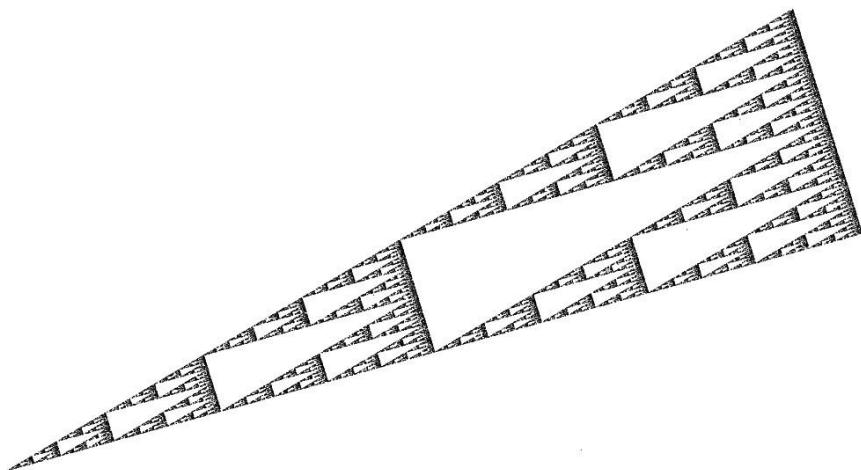


Figure 5.14.

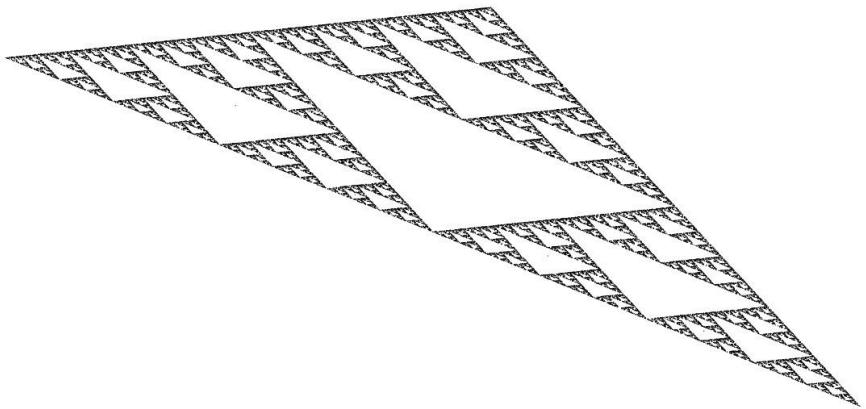


Figure 5.15.

Granted, this works best with three points. For a higher number of starting points, too much randomization causes the points to overlap and the overall shape gets lost. Still, if you don't go too crazy, you can get things like in Figure 5.16.

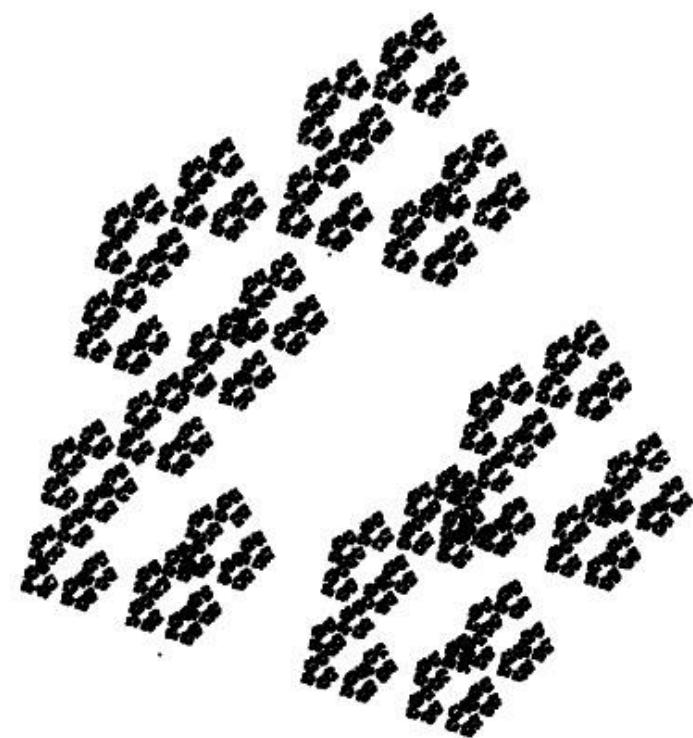


Figure 5.16. Random pentagon.

Color

There are probably many ways you can apply color to these types of images. The one I tried involved using an array of colors – one for each initial point – and setting the context's fill style to that color when the random initial point is chosen. You can see this code in `chaosgame3.js`. The only change is in the `getRandomPoint` function:

```
function getRandomPoint() {
    var index = Math.floor(Math.random()
        * numPoints);
    chaos.context.fillStyle = ["red",
        "green",
        "blue"][index];
    return points[index];
}
```

For three initial points, this will give you a three-colored Sierpinski gasket like the one you see in Figure 5.17 (if you are viewing this book in color, of course).

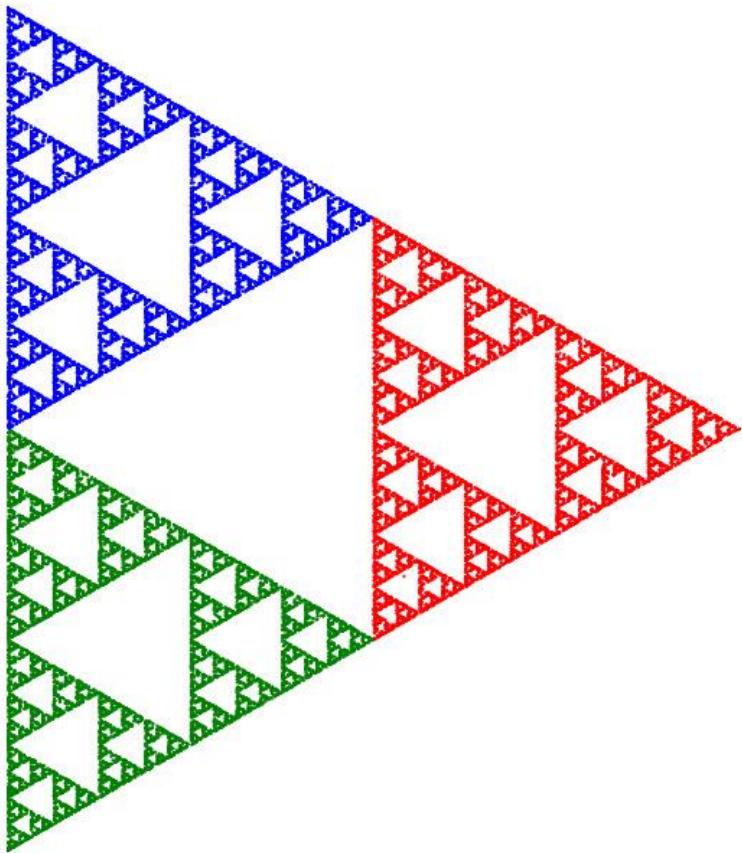


Figure 5.17. Colored Sierpinski.

I thought it was interesting that the colors aligned perfectly with the random point chosen. This perhaps gives you a bit more insight into how the shape is created.

Okay, time to move on. Try other parameters and see what kinds of images you can create. Or just use this file to go off

on some tangent and create something new. As usual, if you come up with something outrageous, I'd love to see it.

Iterated Function Systems

The next set of examples I'll cover are also a Barnsley creation: iterated function systems, or IFS fractals for short.

You could certainly say that most or all of the examples you've seen in this book (and most of the ones to come) are iterated function systems. They all have at least one function that was iterated multiple times, resulting in a fractal image. But the term IFS fractal is usually used to denote the specific method of iterating and transforming a point in order to create a fractal. The "simple shape fractals" you created in Chapter 3 were a close relative of what you'll see in this section.

The Strategy

I'll start by describing the plan for how the fractal will be created:

1. Choose a random point. Plot this point on the screen. Generally, you'll start with point values in the range of -1 to +1 on the x- and y-axes and scale the image up as needed to fit it on the screen.
2. Construct a transformation table. This is a table with seven values in each row. The first six values form a transformation matrix. I'll go into more detail about this shortly. The last value is a probability. This is a

number between 0 and 1. When the table is complete, all of the probabilities should add up to exactly 1.

3. Randomly choose one of the rows in the table based on the probabilities. For example, in a two-row table, if row 0 has a probability of .666 and row 1 has a probability of .334, on average row 0 would be chosen roughly twice as often as row 1.
4. Apply the transformation matrix from the chosen row to the point you created in the first step. This will give you a new point. Plot this point, again, scaling up as needed to fit it on the screen.
5. Repeat steps 3 and 4 thousands of times, each time transforming the last point into a new point and plotting it. See what kind of image it forms. Depending on the values in your transformation table, you just might have a fractal!

Like with the Chaos Game, it may seem that applying a randomly chosen transformation thousands of times is likely to end up in a random mess. But, as you'll see, this is not so.

Of course, not every transformation table will result in a fractal. In fact, there are probably infinitely more tables that create random noise or non-fractals than tables that create interesting fractal images, but I'll feed you a few interesting ones to get you started.

Transformation Matrices

If you're familiar with what a transformation matrix is and how one is used, feel free to skim this section.

Transformations are different changes that can be made to sets of numbers. In this chapter, the numbers are 2D coordinates, or x, y points. For 2D transformations, the most common transformations are:

1. Rotation: moving the point in a circle around another fixed point, usually 0, 0. An angle is specified for the rotation. Example: point (1, 0) rotated 90 degrees will become (0, -1). Remember that the rotation goes clockwise on canvas. This may be the opposite of coordinate systems you are used to.
2. Scaling: multiplying the x or y value (or both) by a certain amount. A scaling value of greater than one will scale the point up. From 0 to 1 it will scale the point down, and values less than one will flip the point on whatever axis the scaling is being applied.
Example: (1, 2) scaled (2, .5) will become (2, 1).
3. Translation: moving the point linearly a certain distance on the x- or y-axis. Example: (1, 1) translated (3, -2) becomes (4, -1).

There are other transformations, such as shearing or projection, but these three will do for the purposes of this chapter.

Transformations can be applied individually. In fact, you've done just that in many of the examples so far, calling `chaos.context.rotate`, `chaos.context.translate` and `chaos.context.scale`. But a more advanced way to apply multiple translations at once is by using a transformation matrix. This is an array of values that is used to specify all the transformations.

A 2D transformation matrix is usually represented by six values, sometimes labeled `a`, `b`, `c`, `d`, `e`, `f` and

sometimes a , b , c , d , tx , ty . These can be represented in matrix form like so:

$$\begin{matrix} a & c & e \\ b & d & f \end{matrix}$$

or

$$\begin{matrix} a & c & tx \\ b & d & ty \end{matrix}$$

I'll use the tx , ty variation here. To apply individual transformations in matrix form, the following forms are used:

Scaling: a controls the scaling on the x-axis, and d is scaling on the y-axis. All other values would be zero. To double a point's position or an object's size on both axes, the following matrix would be used:

$$\begin{matrix} 2 & 0 & 0 \\ 0 & 2 & 0 \end{matrix}$$

Translation: tx and ty control x translation and y translation. To move a point or object 100 units on the x-axis and -50 on the y-axis, use this matrix:

$$\begin{matrix} 1 & 0 & 100 \\ 0 & 1 & -50 \end{matrix}$$

Note that in this case, a and d are set to 1. This means that the point or object will be scaled to 100%.

Rotation: all of a through d are used in rotation. This is a little more complex than the others, and requires some trigonometry functions. To rotate a point or object by an angle of n (in radians), use a matrix, like so:

$$\begin{matrix} \cos n & -\sin n & 0 \\ \sin n & \cos n & 0 \end{matrix}$$

Often, if you're using a matrix, you'd be doing multiple transformations at the same time, but exactly how those transformations would be put in the matrix can get complex and relies on the order in which the different transformations are to take place. Rotating an object and then translating it

is not at all the same thing as translating it and then rotating it.

There are several ways to use a transformation matrix when drawing in an HTML5 canvas object. One would be to pull the values out of the matrix and use them as individual transformations – scale, rotate and translate. But why go through the trouble to put the transformations into a matrix if you’re just going to pull them back and use them individually?

The second way is to use the matrix directly on the context, passing in the matrix values through the context methods `transform` and `setTransform`. Both of these methods take six parameters, corresponding to the `a` through `ty` properties of a transformation matrix. The difference between these two methods is that `transform` applies the new transformation on top of the current transformation of the context, whereas `setTransform` clears any current transformations, resetting the context back to its original state before applying the new transformation passed in.

I’ve created several examples of this method in the file `transform.js`. I’m not going to go over it in this book, but it’s there for you to look at if you are interested.

Another way to apply transformations is to apply them mathematically to the point or points you are drawing before drawing that point. This is the method I will use in the rest of the examples in this chapter.

The formula for applying a transformation matrix to the `x`, `y` values of a point follows. Assume that `x` and `y` are the initial point values, `x1` and `y1` are the transformed values and `a` through `ty` are the matrix values.

$$\begin{aligned}x_1 &= x * a + y * b + tx \\y_1 &= x * c + y * d + ty\end{aligned}$$

Okay, that's a whole lot of theory. Time to see it in action. Bring on the examples.

Barnsley Fern

Like Messrs. Sierpiński and Koch, Mr. Barnsley also has a famous fractal named after him (something we can all aspire to). The Barnsley fern will be the first IFS fractal you'll create. Figure 5.18 shows the transformation table used for this fern.

a	b	c	d	tx	ty	p
0.00	0.00	0.00	0.16	0.00	0.00	0.01
0.85	0.04	-0.04	0.85	0.00	1.60	0.85
0.20	-0.26	0.23	0.22	0.00	1.60	0.07
-0.15	0.28	0.26	0.24	0.00	0.44	0.07

Figure 5.18. Transformation Table.

As I explained earlier, the first six columns form the transformation matrix. The last column, p , is the percentage, or the chance out of 1.0 that that particular row will be chosen. In this table, on any particular iteration, there is an 85% chance that the matrix in the second row will be chosen, a 7% chance for each of the third and fourth rows and only a 1% chance that the first row will be used.

A quick analysis should tell you that the first row scales to 0 on the x-axis and nearly 0 on the y-axis, with no other transformations. The other three rows probably have some rotation and possibly some scaling going on, with some y translation as well.

To recap the strategy, you'll be choosing a random point and plotting it, then transforming it using one of the above matrices and plotting again, choosing another matrix,

transforming, plotting, etc. Each iteration will plot one point, and they'll gradually build up into something interesting.

Here's the code from `ifs1.js`:

```
window.onload = function() {
    var table = [
        [ 0,          0,          0,      0.16,  0,  0,      0.01],
        [ 0.85,     0.04,     -0.04,  0.85,  0,  1.6,     0.85],
        [ 0.2,     -0.26,     0.23,  0.22,  0,  1.6,     0.07],
        [-0.15,     0.28,     0.26,  0.24,  0,  0.44,     0.07]];
}

var currentPoint,
    scale = 70,
    pointSize = .5 / scale,
    interval;

init();

function init() {
    chaos.init();

    chaos.context.translate(chaos.width / 2,
                           chaos.height);
    chaos.context.scale(scale, -scale);

    currentPoint = {
        x: Math.random() * 2 - 1,
        y: Math.random() * 2 - 1
    };

    setPoint(currentPoint);

    document.body.addEventListener("keyup",
        function(event) {
            switch(event.keyCode) {
                case 32: // space
                    nextPoint();
                    break;

                case 187: // +
                    clearInterval(interval);
                    interval = setInterval(
                        function() {
                            for(var i = 0; i < 10; i += 1) {
                                nextPoint();
                            }
                        }
                    );
            }
        }
    );
}
```

```

        }
    }, 0);
break;

case 189: // -
    clearInterval(interval);
break;

case 80: // p
    chaos.popImage();
break;

default:
break;
}
});

}

function nextPoint() {
var t = getRandomTransform(),
x = currentPoint.x * t[0]
+ currentPoint.y * t[1]
+ t[4];
y = currentPoint.x * t[2]
+ currentPoint.y * t[3]
+ t[5];
currentPoint.x = x;
currentPoint.y = y;
setPoint(currentPoint);
}

function getRandomTransform() {
var randomNumber = Math.random();
for(var i = 0; i < table.length; i += 1) {
    var row = table[i];
    if(randomNumber <= row[6]) {
        return row;
    }
    randomNumber -= row[6];
}
}

function setPoint(p) {
chaos.context.beginPath();
chaos.context.arc(p.x, p.y, pointSize,
                  0, Math.PI * 2, false);
chaos.context.fill();

```

```
}
```

First, an array named `table` is created. This is the exact table you just saw, converted into a JavaScript array.

A `scale` value is set and a `pointSize` set to `.5 / scale`. The reason behind this is that the `x`, `y` values for the points generated are going to be very small and this will create a tiny, barely visible drawing. You'll need to scale the context way up, 70 in this case, in order to see more than a tiny smudge on the screen. But the points you draw will also be scaled way up. So you'll preemptively scale them way down so they'll come out just right at .5 pixels on the screen.

In the `init` function, you'll translate the context to center screen and then scale it to `scale`, `-scale`. The reason for the minus there is to flip the image on the `y`-axis. The algorithm was designed to work with standard Cartesian coordinates, in which the positive `y`-axis goes up. The context's coordinate system is the reverse of this, so you need to flip it to avoid an upside-down image.

Still, in `init`, you create a random point and plot it.

Now, like with the Chaos Game example, the next point is plotted using the `nextPoint` function, and this is called each time you press the space bar, or repeatedly using `setInterval` if you press the `+` key. Let's take a closer look at this function.

```
function nextPoint() {
  var t = getRandomTransform(),
      x = currentPoint.x * t[0]
        + currentPoint.y * t[1]
        + t[4];
  y = currentPoint.x * t[2]
    + currentPoint.y * t[3]
    + t[5];
  currentPoint.x = x;
  currentPoint.y = y;
  setPoint(currentPoint);
```

}

First, it gets a row from the transformation table using `getRandomRow`. It uses the formula I showed you at the end of the last section to transform the current point, giving it new x and y values, and plots that.

All that's left is the `getRandomTransform` function, which warrants a look:

```
function getRandomTransform() {  
    var randomNumber = Math.random();  
    for(var i = 0; i < table.length; i += 1) {  
        var row = table[i];  
        if(randomNumber <= row[6]) {  
            return row;  
        }  
        randomNumber -= row[6];  
    }  
}
```

This function chooses a random number and then loops through the rows in the transformation table. If the random number is less than the probability in row 0 (0.01), it will return that row. Otherwise it will subtract the probability of that row from the random number and compare it to row 1, continuing this way until a match is found.

As an example, say the random number was 0.9. It would fail row 0 and 0.01 would be subtracted from it, making it 0.89. It would again fail row 1 and 0.85 would be subtracted, resulting in 0.04. In checking row 2, the function would see that 0.04 is indeed less than 0.07 and return row 2.

Cranking up the `pointSize` to make the points large and visible, and executing a few dozen iterations, gives you Figure 5.19.



Figure 5.19. Something is forming ...

Reverting to a smaller point size and letting the program run awhile gives you Figure 5.20, the Barnsley fern.



Figure 5.20. The Barnsley fern.

Variations

The main things to vary here are the values in the transformation table. Try changing any of the matrix values or the percentage value of one or more of the rows. Start conservatively; it doesn't take much before it devolves into total chaos. Figure 5.21 shows one example I came up with by changing just a few values. It's lost much of its "fernness," but I still find it interesting.

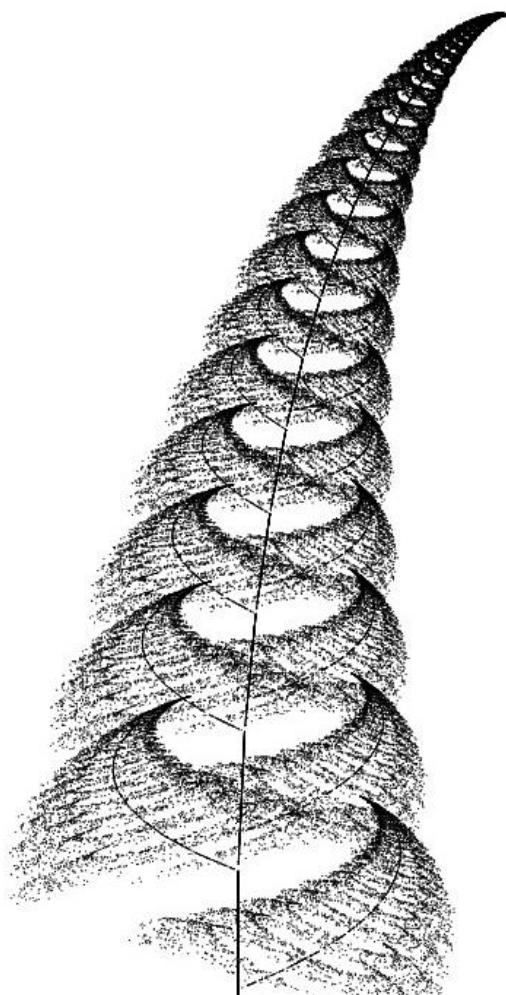


Figure 5.21. Not a Barnsley Fern.

It's a good idea to make a copy of the table and comment out the original so you can revert to it if you destroy it and get lost.

Colors

Similar to what you did in the Chaos Game example, there's a neat way to use color that will give you some insight into how the fractal is created. It involves setting the fill style of the context based on which rule is randomly chosen. It only takes a one-line difference, which I've added to the file `ifs2.js` and will show you here:

```
function getRandomTransform() {
    var randomNumber = Math.random();
    for(var i = 0; i < table.length; i += 1) {
        var row = table[i];
        if(randomNumber <= row[6]) {
            chaos.context.fillStyle = ["red",
                                      "green",
                                      "blue", "orange"][i];
            return row;
        }
        randomNumber -= row[6];
    }
}
```

As you can see, just before returning the matching row, the function now sets the fill style based on the index of the `for` loop in which the match was made. So each rule is assigned its own color. This results in the image you see in Figure 5.22.



Figure 5.22. The Barnsley fern, colored.

If you're viewing this in color (or actually ran the program), you'll see that the first large leaf on the left is rendered in blue, the bottom-right leaf is orange, the first segment of stem is red and the rest is all green. Mapping the colors you assigned to the rows in `getRandomTransform`, this tells you that row 0 creates the stem, rows 2 and 3 create the left and right leaves, and row 1 creates everything else. This can help you when altering an existing IFS fractal or trying to create your own.

Other IFS Fractals

Beyond the Barnsley fern and its variations, an infinite number of images can be produced with the IFS method described above. Of course, not all of them will be very interesting, but with a bit of work, you can find plenty that are. In the file `ifs3.js`, I've provided you with a few more tables that will produce screenshot-worthy images. You can just comment out the ones you don't want and uncomment the one you want to see. Note that in some of these examples you'll have to adjust the center point and scale in order to have the image show up onscreen and large enough to appreciate. This is done by changing the `scale` value right after the table is created, and adjusting the parameters of the call to `chaos.context.translate` in the `init` function.

Figure 5.23 shows a version of a fractal tree not too different from what you created in Chapter 3.

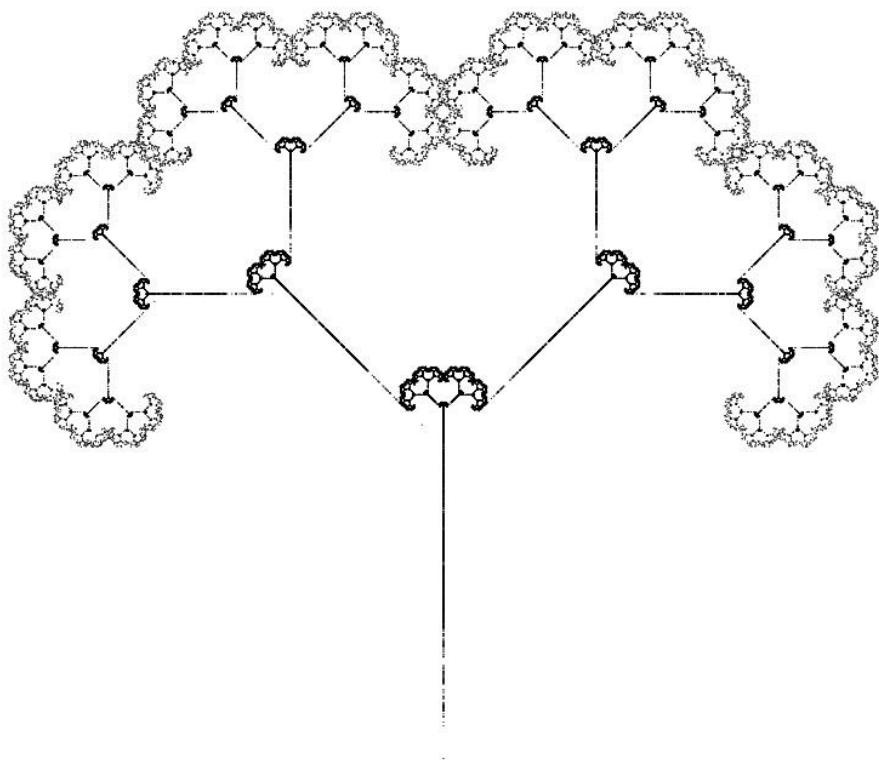


Figure 5.23. IFS tree.

And Figure 5.24 brings back our old friend, the Sierpinski gasket. It seems there are nearly an infinite number of ways to create this shape!

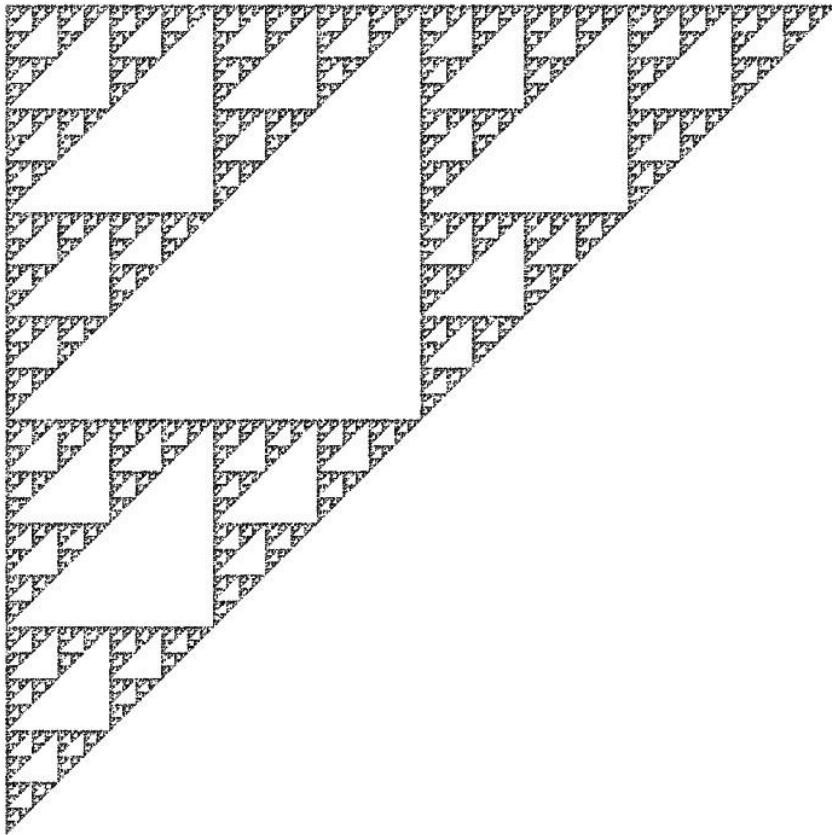


Figure 5.24. The Sierpinski gasket, yet again!

Finally, Figure 5.25 shows another variation of a fractal tree. This one is quite striking due to its realistic form. Yet, if you look closely, you can see that, despite its natural and seemingly random shape, each branch is indeed an exact copy of the tree as a whole. This is also a fun one to try the color technique on.

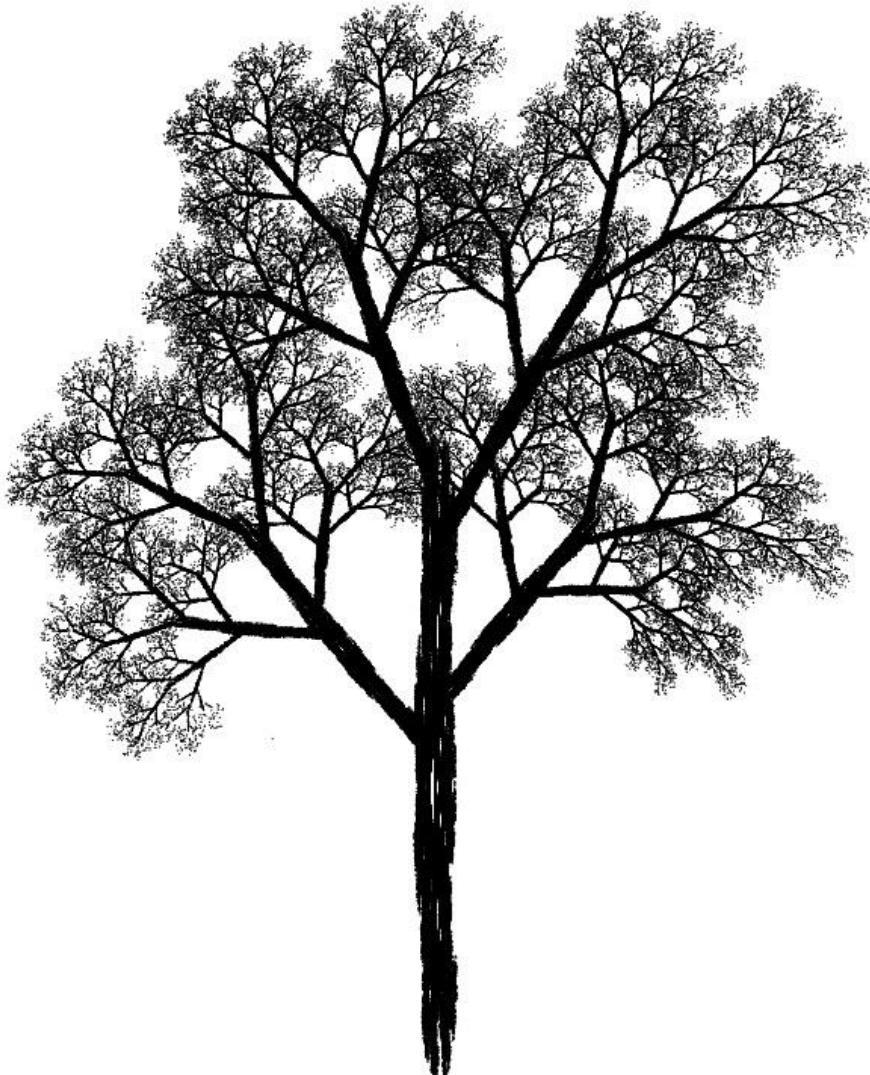


Figure 5.25. Another fractal tree.

This last image is adapted from an example by Paul Bourke, a researcher and professor at the University of Western Australia and a fractal enthusiast. I'd also count him as a master of the IFS fractal. On the fractals page of his personal site, <http://paulbourke.net/fractals/>, there are many truly amazing examples of IFS fractals, along with the

transformation tables for creating them with the exact method described in this chapter. These are great resources to examine and modify to create your own IFS fractals.

Random IFS Fractals

As I said in the previous section, infinite varieties of fractal images can be created with the IFS method. Altering existing examples is one way to discover new images, but if you are trying to come up with something unique, it can be hard to know where to start.

One strategy is to fill a translation table with random values and see what happens. While this often results in complete chaos, you'd be surprised at how often it creates a downright pretty picture. The code I created for this is in the file `ifsrandom.js`, and I'll just give you the first section here, where the random table is created:

```
window.onload = function() {
    var table = [];
    var percent = 1,
        a, b, c, d, tx, ty, p;
    for(var i = 0; i < 4; i += 1) {
        a = Math.round((Math.random() * 2 - 1)
                      * 100) / 100;
        b = Math.round((Math.random() * 2 - 1)
                      * 100) / 100;
        c = Math.round((Math.random() * 2 - 1)
                      * 100) / 100;
        d = Math.round((Math.random() * 2 - 1)
                      * 100) / 100;
        tx = Math.round((Math.random() * 4 - 2)
                      * 100) / 100;
        ty = Math.round((Math.random() * 4 - 2)
                      * 100) / 100;
        if(i < 3) {
            p = Math.round(Math.random() * percent
                          * 100) / 100;
            table.push({a: a, b: b, c: c, d: d, tx: tx, ty: ty, p: p});
        }
    }
}
```

```
    percent -= p;
}
else {
    p = Math.round(percent * 100) / 100;
}
table.push([a, b, c, d, tx, ty, p]);
console.log([
    + [a, b, c, d, tx, ty, p].join(", ")
    + ""]);
}
var currentPoint,
    scale = 70,
    pointSize = .5 / scale,
    interval;
...

```

The rest of the code is the same as what you already saw in the `ifs2.js` file. As you can see, this code loops through a number of times – in this case, four – and creates a new row in the table with random values for `a` through `ty`. All values are rounded to two places. The percentage values are the trickiest to calculate. The code here ensures the values always add up to 1 for the whole table. After calculating each row and adding it to the table, this code also outputs that row to the console. So, if you find an image that you particularly like, you can view the JavaScript console in your browser, copy and paste the values into a new file and then tweak them to refine the image further.

Figures 5.26, 5.27 and 5.28 show some purely random images created with this method.

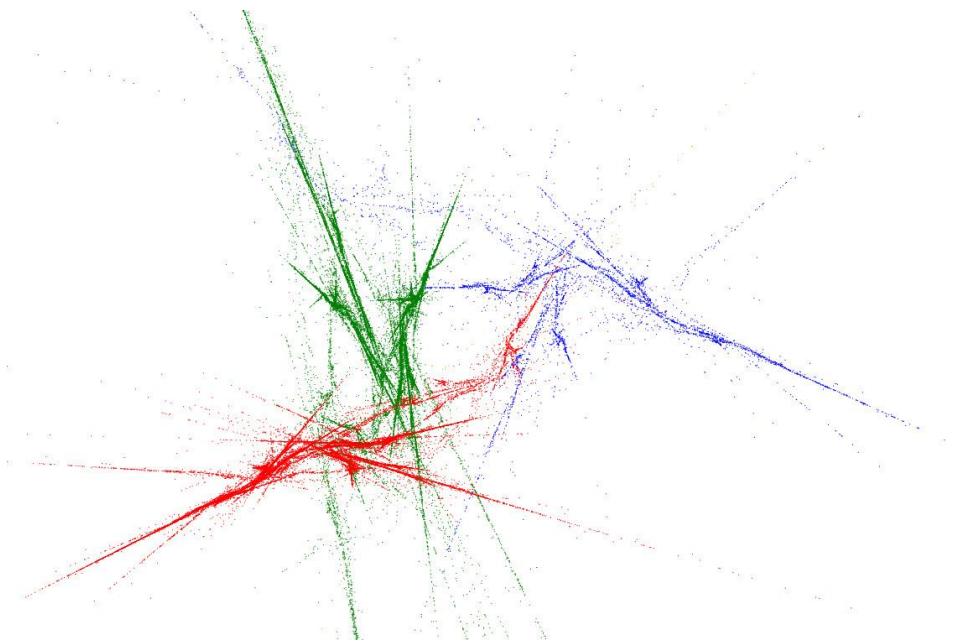


Figure 5.26.

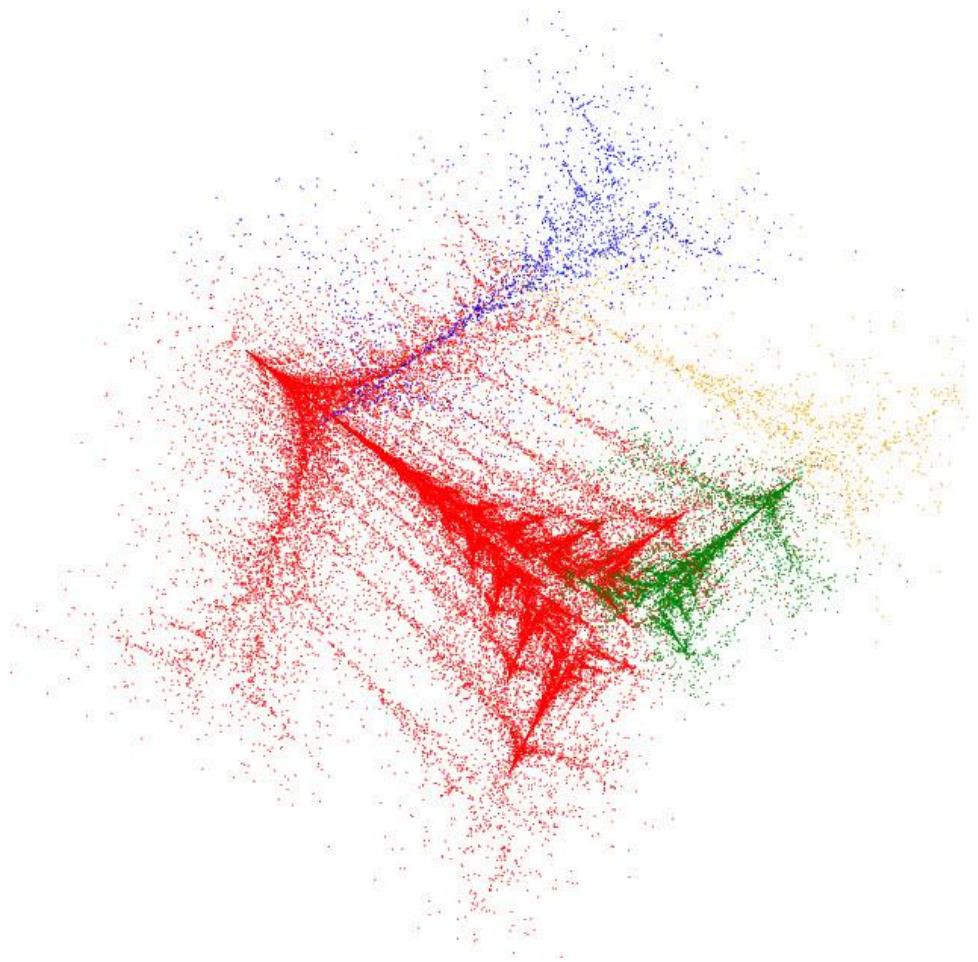


Figure 5.27.

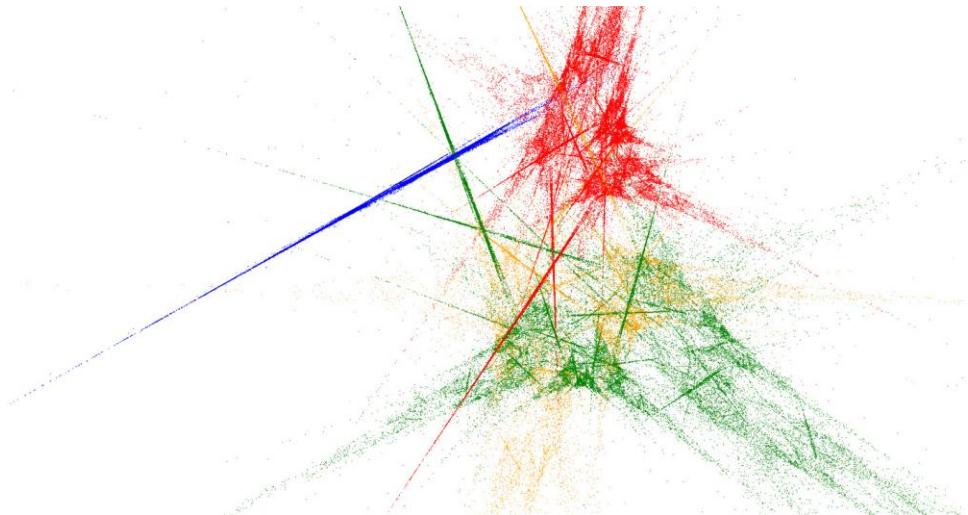


Figure 5.28.

Summary

Okay, so that's order from chaos. In this chapter, you've learned a couple of methods that start with a random point, randomly apply various rules and result in a complex, well-ordered and predictable image. Note that, although you could not predict the position of any single pixel drawn in the system, you can predict the general output of the system as a whole. This is a theme that you'll see again in future chapters. In fact, in the very next chapter, you'll see almost pure randomness being used to create a new kind of fractal image.

Chapter 6: Diffusion-Limited Aggregation

In this chapter, I'll introduce you to a new way of creating fractal forms. In Chapters 1 through 4, most of the fractals I've introduced have been very regular, with randomness only introduced as an afterthought. In Chapter 5, randomness was an integral part of the creation of the fractals you saw. For the next several pages, you'll see fractals that are created with 100% randomness. As you may imagine, this means that the fractals created are neither symmetrical nor regular. Even so, you'll see self-similarity in the final form.

Background

The concept behind diffusion-limited aggregation – or DLA as I'll sometimes call it – is that you have a bunch of particles floating around randomly in a medium. There's one motionless spot that I'll call a seed. When one of these particles hits the seed, it sticks there. Now, another particle may hit the seed, or hit that stuck particle, and it will also stick. Little by little, these particles will aggregate and build up on the seed, creating some sort of shape. This aggregation will be limited by how many particles are diffused in the medium, hence the name.

If you imagine this process being carried out for some time (and you're not already familiar with the concept of DLA), what kind of shape do you think will form? Many people imagine a random, somewhat fuzzy ball of particles being formed. What actually occurs is quite a bit more interesting.

Brownian Motion

Brownian motion describes the motion of the particles in this theoretical medium. This term is named after Robert Brown, the botanist, after he wrote about observing grains of pollen moving randomly around in a container of still water. He wondered what the cause of the motion was. Curiously enough, Brown merely asked the question and never came up with any answer. Einstein, decades later, explained that it was the force of atoms hitting the particles randomly from many different directions that caused them to move. Nevertheless, we call it Brownian motion rather than Einsteinian motion.

It's quite easy to simulate Brownian motion in code. Just give each particle a velocity and randomly vary that velocity very slightly on a regular basis. Over time, the random variation in velocity will give the particle a random, wandering path. You'll also probably want to dampen the velocity a bit so that the particles don't occasionally build up too much speed in one particular direction, which would be unrealistic.

The Strategy

This example is a bit trickier than the ones you've seen in other chapters. You'll want to show two separate things on screen: the ongoing motion of the random particles, and the static shape of a form building in the center. For this, it makes sense to use two separate, overlaid canvases. The top canvas will show the moving particles and can be cleared and redrawn over and over to show the changing positions of the moving particles. The bottom canvas will only be drawn to when a new particle joins the aggregate. This bottom canvas will also be used to do collision detection using the built-in method `getImageData` to examine the individual pixels in the canvas.

Here are the steps:

1. Create the two canvases. Draw a seed in the bottom canvas.
2. Create an array of particles. These particles will be simple JavaScript objects with properties for position (`x`, `y`) and velocity (`vx`, `vy`). Give them all random positions and zero velocity.
3. Loop through the array of particles, performing the following steps.
 - a. Check the position of the particle against the bottom canvas. If the pixel at that position is not clear, then the particle has struck the aggregate. Draw the particle at that position on the aggregate. Then reset that particle.
 - b. If the particle did not hit the aggregate, update the particle by adding a random value to its velocity,

adding the velocity to its position, and finally dampening the velocity a bit.

Repeat steps 3a and b for all particles, and repeat the whole process many times per second. Eventually, a shape will form.

A note on step 3a. If you were to just remove each particle when it hits the aggregate, you'd soon run out of particles in the medium. So each time one joins the aggregate, you can re-spawn it somewhere else, preferably somewhere near one of the edges. This will allow the aggregate to continue to grow indefinitely.

The Code

For this one, you're going to need a new HTML file as well as the usual code, as you will need to create the dual canvases and the CSS that is needed to position them. So let's start with that one. It's in the file `dla1.html`:

```
<!DOCTYPE html>
<html>
<head>
    <title>DLA 1</title>

    <style type="text/css">
        html, body {
            margin: 0px;
        }
        canvas {
            display: block;
            position: absolute;
            top: 0px;
            left: 0px;
        }
    </style>

    <script type="text/javascript"
        src="chaos.js"></script>
```

```
<script type="text/javascript"
       src="dla1.js"></script>

</head>
<body>
  <div>
    <canvas id="canvas"></canvas>
    <canvas id="particleCanvas"></canvas>
  </div>
</body>
</html>
```

Here you see that there are a few more styles added to the `canvas` object. The line `position: absolute;` tells the browser to forget about normal HTML layout rules and position the canvas exactly where you tell it. That position is 0, 0, as specified by the `top` and `left` attributes just below that.

Then, within the `body`, you can see that there is now an extra canvas with an id of `particleCanvas`. Due to the CSS just discussed, both of these canvases will be positioned in the same place, one on top of the other. The first canvas created will be on the bottom, and the next one will be on top.

Now, onto the JavaScript. I introduced the chaos utils in Chapter 1. This example, like all the others, will use the utils, but those functions were built on the assumption that the page would only contain a single canvas. I could alter the utils to allow for multiple canvases, but since this is the only example in the book that uses more than one canvas, I decided to leave the utils as they were. It will just mean a bit more code will have to go into the file for this example:

```
window.onload = function() {
  var particles = [],
      numParticles = 5000,
      pCanvas = document.getElementById("particleCanvas"),
      pContext = pCanvas.getContext("2d"),
      imageData;
```

```
init();

function init() {
    chaos.init();

    // set the canvas smaller than full size
    chaos.setSize(500, 500);

    // set the particle canvas to the same size
    pCanvas.width = chaos.width;
    pCanvas.height = chaos.height;

    // draw the seed
    chaos.context.fillRect(chaos.width / 2 - 2,
                          chaos.height / 2 - 2,
                          4, 4);

    makeParticles();

    setInterval(update, 0);

    document.body.addEventListener("keyup",
        function(event) {
            switch(event.keyCode) {
                case 80: // p
                    chaos.popImage();
                    break;

                default:
                    break;
            }
        });
}

function makeParticles() {
    for(var i = 0; i < numParticles; i += 1) {
        var p = {
            x: Math.random() * chaos.width,
            y: Math.random() * chaos.height,
            vx: 0,
            vy: 0
        }
        particles.push(p);
    }
}

function update() {
```

```

// grab the current pixels of the aggregate canvas
imageData = chaos.context.getImageData(
    0, 0, chaos.width, chaos.height).data;

// clear the particle canvas
pContext.clearRect(0, 0,
    chaos.width, chaos.height);

// update all particles
for(var i = 0; i < numParticles; i += 1) {
    var p = particles[i];
    updateParticle(p);
}
}

function updateParticle(p) {
    // check if this particle is hitting the aggregate (see
text)
    var x = Math.round(p.x),
        y = Math.round(p.y),
        pixel = imageData[(y * chaos.height + x)
            * 4 + 3],
        hit = pixel > 0;

    if(hit) {
        // draw this particle on the aggregate
        // and respawn it
        chaos.context.fillRect(p.x, p.y, 1, 1);
        respawn(p);
    }
    else {
        // randomize velocity a bit
        p.vx += Math.random() * .1 - .05;
        p.vy += Math.random() * .1 - .05;
        // update position
        p.x += p.vx;
        p.y += p.vy;
        // dampen motion
        p.vx *= .99;
        p.vy *= .99;

        // if offscreen, wrap around to the other side
        if(p.x > chaos.width) {
            p.x -= chaos.width;
        }
        else if(p.x < 0) {
            p.x += chaos.width;
        }
    }
}

```

```

        if(p.y > chaos.height) {
            p.y -= chaos.height;
        }
        else if(p.y < 0) {
            p.y += chaos.height;
        }

        // draw current particle on particle canvas
        pContext.fillRect(p.x, p.y, 1, 1);
    }
}

function respawn(p) {
    // reset to a random position,
    // either along top edge or left side
    if(Math.random() < .5) {
        p.x = Math.random() * chaos.width;
        p.y = 0;
    }
    else {
        p.x = 0;
        p.y = Math.random() * chaos.height;
    }
}
}
}

```

There's quite a bit of code here, but it's commented and I'll explain each part here. First, the variables:

```

var particles = [],
    numParticles = 5000,
    pCanvas = document.getElementById("particleCanvas"),
    pContext = pCanvas.getContext("2d"),
    imageData;

```

You have an array of particles, and you'll be creating 5,000 of them. `pCanvas` is a reference to the `particleCanvas` element and `pContext` is the 2D drawing context for that canvas.

The `imageData` variable will be used to hold the pixel data for the aggregate canvas, as you'll see shortly.

The `init` function is called as usual, and here are the interesting parts of that:

```

chaos.init();

// set the canvas smaller than full size

```

```
chaos.setSize(500, 500);

// set the particle canvas to the same size
pCanvas.width = chaos.width;
pCanvas.height = chaos.height;

// draw the seed
chaos.context.fillRect(chaos.width / 2 - 2,
                      chaos.height / 2 - 2,
                      4, 4);

makeParticles();

setInterval(update, 0);
```

After calling `chaos.init`, set the size of the main canvas to 500 by 500 pixels. This example is very processor-intensive and can take a very long time for the image to build up. So, by limiting the size of the canvas, you'll see results a lot faster and will be less likely to melt your CPU. The next couple of lines set the size of the particle canvas to the same size as the aggregate canvas. The seed is then drawn, particles are initialized and `setInterval` is called to run the `update` function repeatedly. Passing in 0 here will let it run as fast as possible. This is not the best way to accomplish animation in JavaScript. A much better way is to use the `requestAnimationFrame` function that is in many newer browsers. But, as it is implemented slightly differently in some browsers, and not at all in others, I've kept things simple here by using code that will work in just about any browser. I highly suggest you look into `requestAnimationFrame` when you get a chance, though.

The `makeParticles` function is straightforward. It creates the specified amount of particles, with a random x, y and zero velocity, and puts them into the `particles` array.

The function called `update` is the one that is run via `setInterval`. Most of it is obvious, but one line needs explaining:

```
imageData = chaos.context.getImageData(0, 0, chaos.width,  
chaos.height).data;
```

The context method `getImageData` takes `x`, `y`, `width` and `height` arguments and returns a special object that has a few interesting properties. The one you are concerned with is the `data` property. This is what is assigned to `imageData`. This `data` property is essentially an array filled with pixel color values in a special format. Hold that thought for just a moment.

The `updateParticles` function is the real meat of this example. It is called by `update` for every single particle in the `particles` array. In this function, you first see:

```
var x = Math.round(p.x),  
    y = Math.round(p.y),  
    pixel = imageData[(y * chaos.width + x)  
                    * 4 + 3],  
    hit = pixel > 0;
```

This rounds the current pixel's position to whole number values. Now, you'll need to use this `x` and `y` to find the value of the pixel at that position. As I just mentioned, the pixel values are now stored in `imageData`. The way they are stored is key to figuring out the next couple of lines. Each color component (red, green, blue and alpha) of each pixel is stored as a separate integer in the array. So the first four elements in the `imageData` array are the red, green, blue and alpha values of the first pixel in the upper-right corner of the canvas, which is to say, at location 0, 0. The next four values will be the location of the pixel at 1, 0, and so on across the first row of pixels and then downwards towards the bottom.

So, to find the index of a particular pixel, you multiply the current `y` value times the width of the canvas and add the current `x` value. Work this out on paper if it doesn't make sense. But, because each pixel actually takes up four spaces in the array, you have to multiply that value by four.

Now, you'll be pointed at the element that holds the red value for the pixel in question.

The red value probably isn't very useful in this case. The canvas is initialized with all zeros for the red, green, blue and alpha of every pixel. But since the default drawing color is black, that means the red, green and blue value of a "filled" pixel could also be zero. The component that will be different is the alpha value, which will go from zero for an empty spot on the canvas up to 255 for a completely opaque black pixel. Adding three to the index just calculated gives you the alpha value of the pixel under scrutiny. If that alpha is greater than zero, you have a hit. Thus:

```
pixel = imageData[(y * chaos.width + x) * 4 + 3],  
hit = pixel > 0;
```

The rest of the `updateParticle` function is fairly clear and well-commented. If there's a hit, it draws a rectangle on the aggregate canvas at the particle's current location. If not, it applies some simple Brownian motion to the particle and draws it to the particle canvas.

You might be thinking that, if the pixel you just checked is not clear, drawing a rectangle there will just be drawing over an already colored area. But, in practice, there is antialiasing happening each time you draw a rectangle. The edges of the aggregate shape are actually a bit "fuzzy," neither fully clear nor fully opaque. So it all works out and the form does slowly build up.

All that's left is the `respawn` function:

```
function respawn(p) {  
    // reset to a random position,  
    // either along top edge or left side  
    if(Math.random() < .5) {  
        p.x = Math.random() * chaos.width;  
        p.y = 0;  
    }  
    else {  
        p.x = 0;
```

```
    p.y = Math.random() * chaos.height;  
}  
}
```

This uses a random number to execute the `if` or `else` block equally 50% of the time. This resets the position of the particles either along the top or left edge of the canvas. Why there? Well, you don't want to respawn right in the middle of the aggregate. So this code keeps you far away at the edges of the virtual tank holding the medium. Why only the top and left edges? Well, if you look again at the `updateParticle` function, you'll see that if a particle goes off any edge it wraps over onto the opposite edge. Due to the random velocity of the particles, there's a 50% chance that a particle placed on the top edge will move upwards and reappear on the bottom edge. The same goes for particles on the left edge appearing on the right. So setting the particles on just two edges effectively distributes them over all four edges.

DLA in Action

Now, finally, what happens when you run this code? At first, you'll get what you see in Figure 6.1.

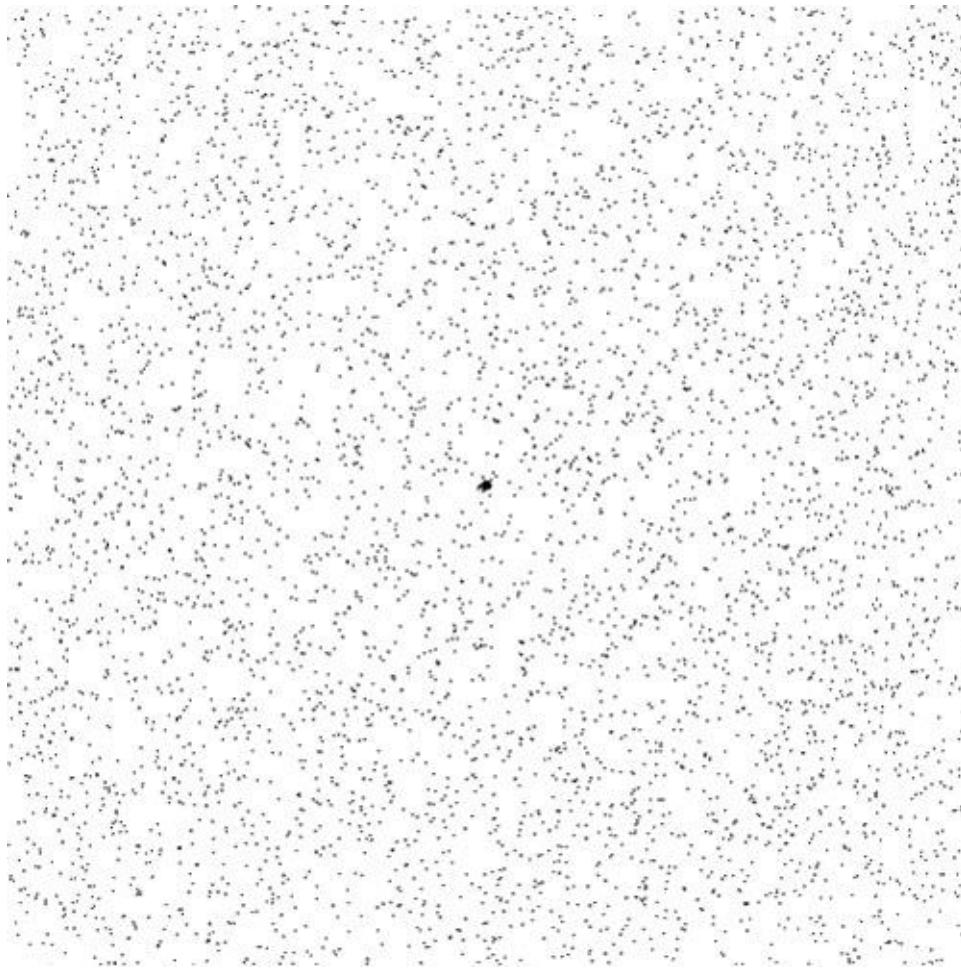


Figure 6.1. 5,000 particles and a seed.

This shows the seed in the center and all the particles floating about randomly. You can see that a few have already stuck. Of course, if you press the p key to save the image using the chaos utils, only the aggregate canvas will be saved, giving you something more like Figure 6.2.

Figure 6.2. Just a seed, starting to grow.

After a few seconds, you'll see more and more particles have hit the seed and are starting to build up, giving you Figure 6.3.



Figure 6.3. After a few seconds.

As this goes on, the shape takes form as in Figure 6.4. And, after a long while, Figure 6.5.



Figure 6.4. A minute or so.

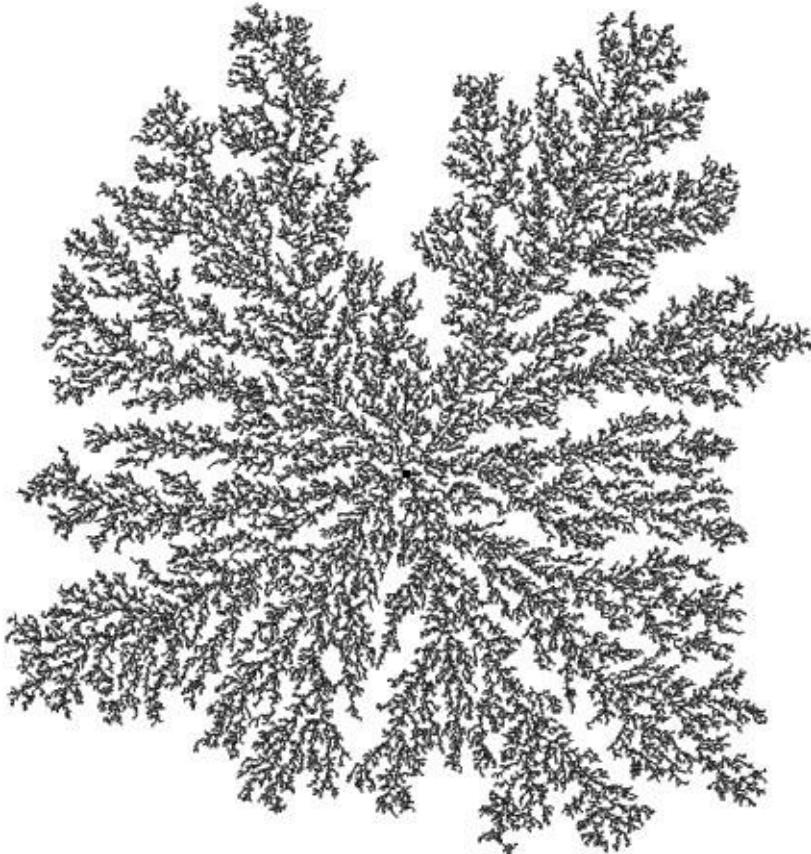


Figure 6.5. Well-grown aggregate.

What you see here is not a random blob but a branching pattern that looks a lot like lichen growing on a rock, ice crystals on a window or maybe some kind of coral growth. These are all classic, natural fractal patterns seen in nature. Note the branching reminiscent of the fractal trees created in earlier chapters. And how those branches are made of smaller branches, similar to the whole.

How does this happen? As particles hit the seed, they expand the size of the aggregate. New particles are more

likely to hit the pieces that are sticking out, so longer branches are formed. Eventually, the spaces between branches become like long, narrow tunnels. If a particle does happen to slip into one of those tunnels, its random motion will most likely cause it to crash into one of the sides and stick. Most of the tunnels will become closed off, or nearly so. The aggregate continues to grow outward, leaving fractal trails behind.

Variations

There are many variations you can try here. The most obvious is to change the position and/or shape of the seed. In the file `dla2.js`, instead of having a small seed in the center of the canvas, I inverted the whole thing. First, I drew a rectangle covering the whole canvas and then cut out a circle from the center of it:

```
// draw the seed
chaos.context.beginPath();
chaos.context.rect(0, 0,
    chaos.width, chaos.height);
chaos.context.arc(chaos.width / 2,
    chaos.height / 2,
    chaos.width / 2 - 10,
    0, Math.PI * 2, true);
chaos.context.fill();
```

The key here is that last argument to the `arc` method. This causes the circle to be drawn counterclockwise. Because the rectangle is drawn clockwise, the two will cancel each other out and the circle will be subtracted from the rectangle, but only if they are drawn in the same `beginPath` to `fill` block.

Next, I changed the `makeParticles` and `respawn` functions so that new and respawned particles always occur at the exact center of the canvas:

```
function makeParticles() {  
    for(var i = 0; i < numParticles; i += 1) {  
        var p = {  
            x: chaos.width / 2,  
            y: chaos.height / 2,  
            vx: 0,  
            vy: 0  
        }  
        particles.push(p);  
    }  
}  
  
function respawn(p) {  
    p.x = chaos.width / 2;  
    p.y = chaos.height / 2;  
}
```

This one takes a while to build up, as the particles have to find their way from the very center of the circle to the very edges, but the result is worth the wait, as you can see in Figure 6.6.

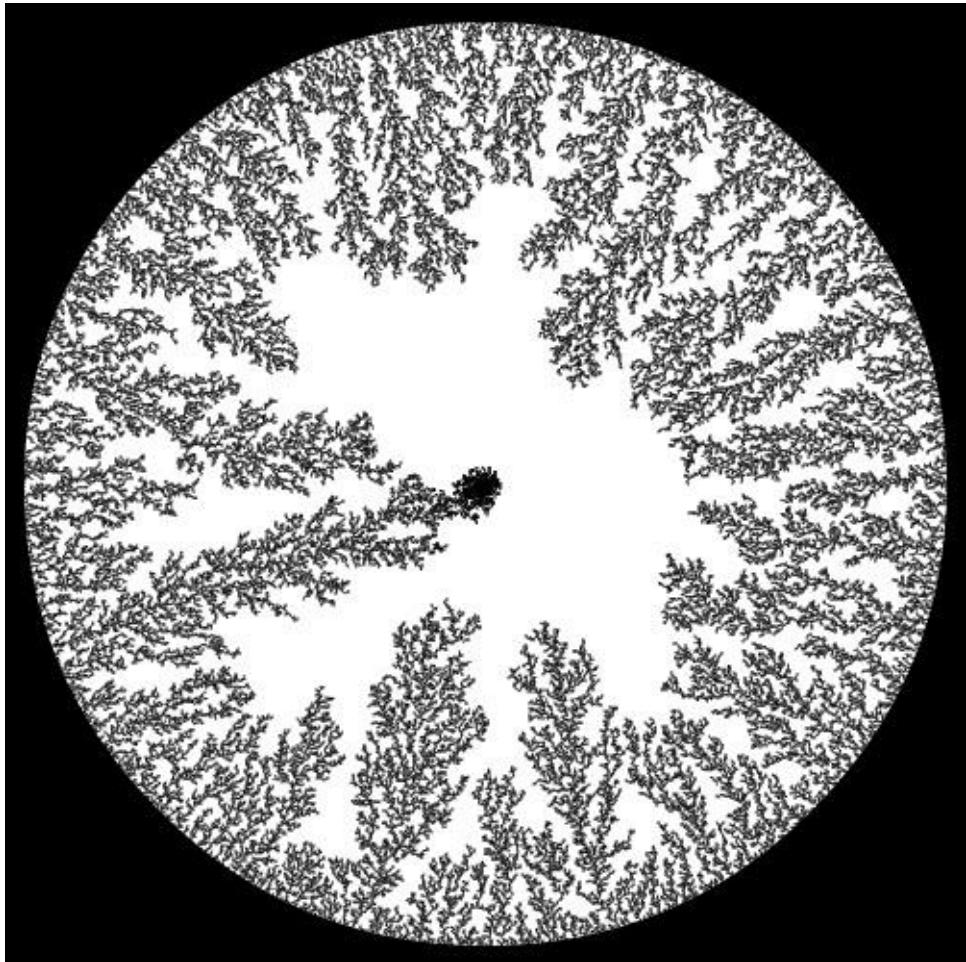


Figure 6.6. Circular aggregation.

The next thing I tried was adding in some gravity to the mix in the file `dla3.js`. This required several changes. Starting back with the code from `dla1.js`, the first change I made was to draw the seed at the bottom center of the canvas:

```
// draw the seed
chaos.context.fillRect(chaos.width / 2 - 10,
                      chaos.height - 2,
                      20, 2);
```

In the `updateParticle` function, I added a small amount to the particle's `vy` property:

```

function updateParticle(p) {
    // check if this particle
    // is hitting the aggregate (see text)
    var x = Math.round(p.x),
        y = Math.round(p.y),
        pixel = imageData[(y * chaos.width + x)
            * 4 + 3],
        hit = pixel > 0;

    if(hit) {
        // draw this particle on the aggregate
        // and respawn it
        chaos.context.fillRect(p.x, p.y, 1, 1);
        respawn(p);
    }
    else {
        // a bit of gravity
        p.vy += .005;
        // randomize velocity a bit
        p.vx += Math.random() * .1 - .05;
        p.vy += Math.random() * .1 - .05;
        // update position
        p.x += p.vx;
        p.y += p.vy;
        // dampen motion
        p.vx *= .99;
        p.vy *= .99;

        // if offscreen, wrap around to the other side
        if(p.x > chaos.width) {
            p.x -= chaos.width;
        }
        else if(p.x < 0) {
            p.x += chaos.width;
        }
        if(p.y > chaos.height) {
            respawn(p);
        }
        else if(p.y < 0) {
            p.y += chaos.height;
        }

        // draw current particle on particle canvas
        pContext.fillRect(p.x, p.y, 1, 1);
    }
}

```

It's a very small amount, .005 at best. This is often overcome by the random y velocity added right after that, but it puts enough of a bias on the motion that the particles all eventually sink slowly towards the bottom of the screen.

Also notice that, if the particle goes off the bottom of the screen, I call `respawn` instead of just wrapping it around back to the top. This is because I changed the `respawn` function a bit:

```
function respawn(p) {  
    p.x = Math.random() * chaos.width;  
    p.y = 0;  
    p.vx = 0;  
    p.vy = 0;  
}
```

This not only respawns all particles at a random position on the top edge of the canvas but zeros out their velocity. Otherwise, particles might keep falling faster and faster as gravity built up their y velocity.

The result of these changes can be seen in Figure 6.7.

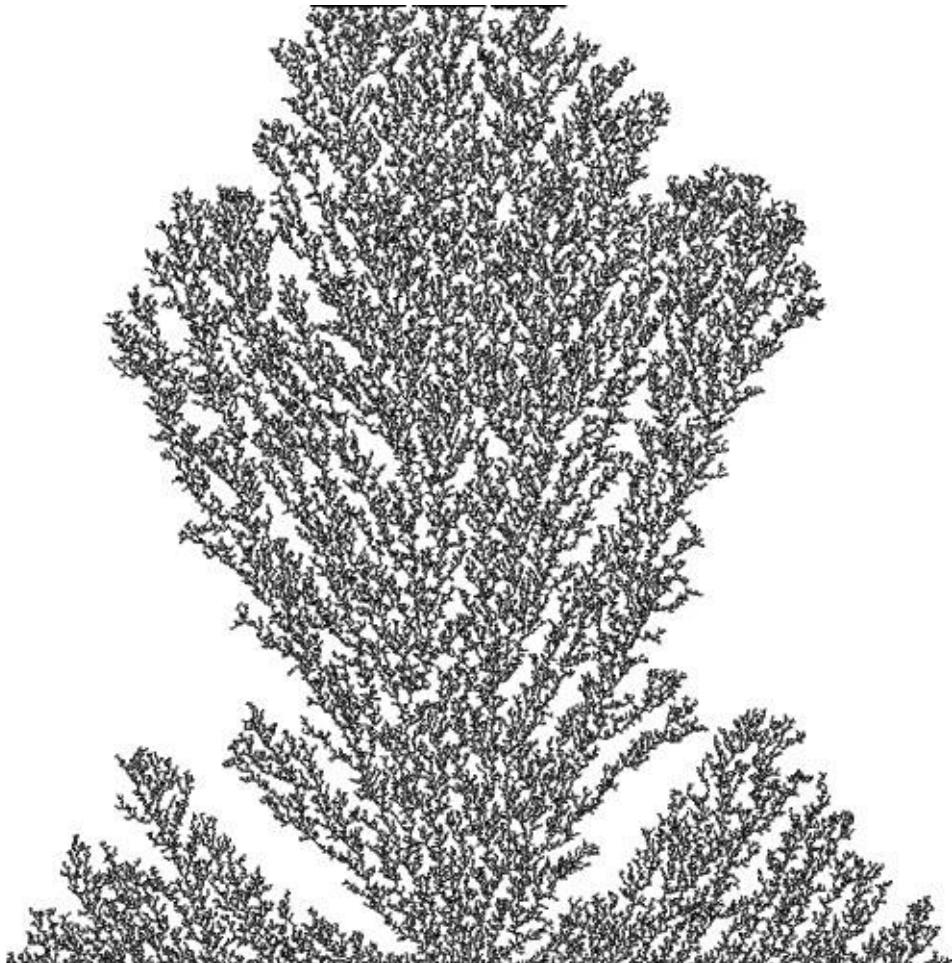


Figure 6.7. Aggregation with gravitation.

This looks even more like some kind of seaweed or coral formation. Playing off that idea, I decided to play with color a bit and used a line of text as a seed. In the file `dla4.js`, I created the seed, like so:

```
// draw the seed
chaos.context.fillStyle = "#003366";
chaos.context.font = "70px Arial bold";
chaos.context.fillText("THE DEEP", 50,
                      chaos.height - 50);

// set color to bright green for aggregated particles
chaos.context.fillStyle = "#00CC00";
```

The text was made with a dark blue color, but the fill style is set to a brighter green for any subsequently drawn particles on the aggregate canvas. I also reduced the gravity to .002 in `updateParticle`:

```
// a bit of gravity  
p.vy += .002;
```

You can see the result in Figure 6.8.



Figure 6.8. Seaweed?

The lower gravity keeps the particles generally falling, so most of the aggregate builds up on top of the text. But there's enough random floating around on the underside that some sticks down there as well.

Summary

I've only scratched the surface of the possible variations here. Try different color schemes, or maybe a couple of different types of particles, each of which only stick to a specific type of seed. Instead of just gravity, maybe add a "current" as an added velocity on the x-axis, and see what that does to the shape of the structures that form.

Chapter 7: The Mandelbrot Set

I figure there's a good chance you turned straight to this page as soon as you got this book. If so, I can't blame you. Mandelbrot fractals ARE the quintessential fractal images. These are what you see on club flyers or psychedelic posters in the head shop. These are the structures that I spent hours exploring when I got my first Commodore Amiga computer. Heck, I even put one on the cover of this book. So, yeah, Mandelbrots are cool. And you can code your own. In JavaScript! For me, the amazing thing is that, even running on JavaScript in the browser, they render many orders of magnitude faster than the ones I played with back in my Amiga days. So let's dive in.

Introduction

On the off chance that you have no idea what I'm talking about, Mandelbrot fractals are a very complex and visually stimulating form of fractal. Again, an example is on the cover, and more are within this chapter. These fractals are made with an exact algorithm, so there is no randomness involved. But you can zoom into the shapes to such a degree that, even within the limitations of a run-of-the-mill computer, it would be like examining a small country inch by inch. Each section of these images has its own unique style and form. As you zoom into a section, you'll see the same basic form repeat and morph into similar but slightly transformed shapes. But you probably know all this, which

is why you turned straight to this chapter. So let's get into some background and theory, then finally write some code.

Background

The Mandelbrot set is a set of values that satisfy a certain equation. It was named after Benoît Mandelbrot, a mathematician often referred to as “the father of fractals”. I briefly mentioned him in Chapter 4 when discussing fractal dimensions.

Mandelbrot coined the word “fractal” and wrote several books on the subject. He was one of the first to explore fractal forms using computers. In 1979, a pair of mathematicians named Robert Brooks and Peter Matelski created the first computer-generated image of the set that would eventually be known as the Mandelbrot set. The set was further researched by two other mathematicians, Adrien Douady and John Hubbard. It was these last two who named it after Mandelbrot.

Who actually “discovered” the Mandelbrot set is a contentious question. Mandelbrot claimed that he did it himself, but others have a good claim on it as well. Certainly, Mandelbrot dug deepest into the properties of the set and popularized it, along with his other works on fractals.

Figure 7.1 shows what the Mandelbrot set looks like when graphed out in two dimensions.

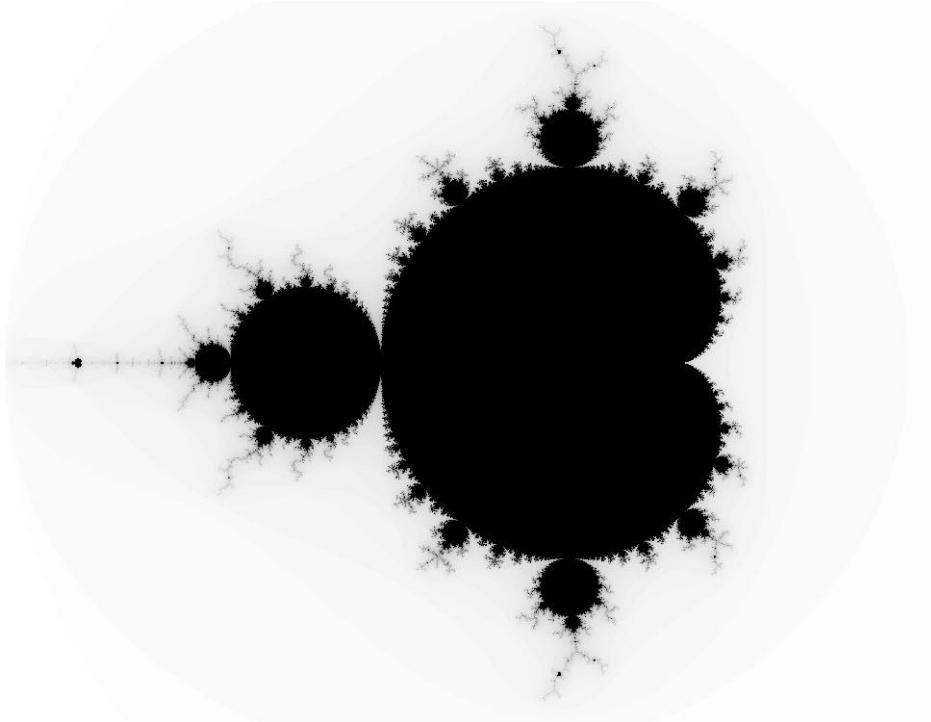


Figure 7.1. The Mandelbrot set.

Now, a brief description of how this set is derived. Take a function and feed some numbers into it. This function returns a value. Feed that value back into the function to get a new value. Repeat this many times. One of two things happens:

1. The values returned shoot off towards infinity.
2. The values stabilize and stay small.

The initial input values that result in number 2 are part of the set. The inputs that result in number 1 are not. Generally, to create a shape such as you see in Figure 7.1, the pixels representing in-set values are colored black. The pixels resulting in values that go infinite are colored using different color schemes, usually indicating how quickly they went towards infinite values. To understand what input values to

use, and how to map these to pixels, you're going to need to know about complex numbers.

Complex Numbers

To understand complex numbers, you need to understand imaginary numbers, and to understand those, you have to understand the imaginary unit i . The symbol is just the letter “ i ” in italics. The value of i is the square root of -1. Or, to put it another way, i times i equals -1. (In some fields, such as electronics, where i may be used to represent some other value such as electrical current, the symbol j is used for the imaginary unit instead.)

Now, if you know anything about math, you'll know that if you square any number (multiply it times itself) you'll get a positive number. So there is no real number that will give you -1 when squared. Thus, we call it imaginary.

Despite it being a bit hard to grasp at first, once you use the symbol i to represent this value it works just fine in all kinds of formulas and has proven itself quite useful in real-world applications, such as electronics, fluid dynamics and quantum mechanics, to name a few.

So i is the imaginary unit. Then, you have multiples of i , such as $2i$, $7i$ or $3.14i$. These are imaginary numbers. A value like $2i$ is simply 2 times the square root of -1.

Then, you have complex numbers. These are the combination of a real number (anything non-imaginary) and an imaginary number. An example is $3 + 4i$. Here, 3 is the real part and $4i$ is the imaginary part. Another example would be $5 - 7i$, in which case you would say 5 is the real

part and $-7i$ is the imaginary part. (Imaginary numbers can be negative.)

Graphing Complex Numbers

I'm going to assume you are familiar with a 2D graph such as what is shown in Figure 7.2. This is also known as the Cartesian plane.

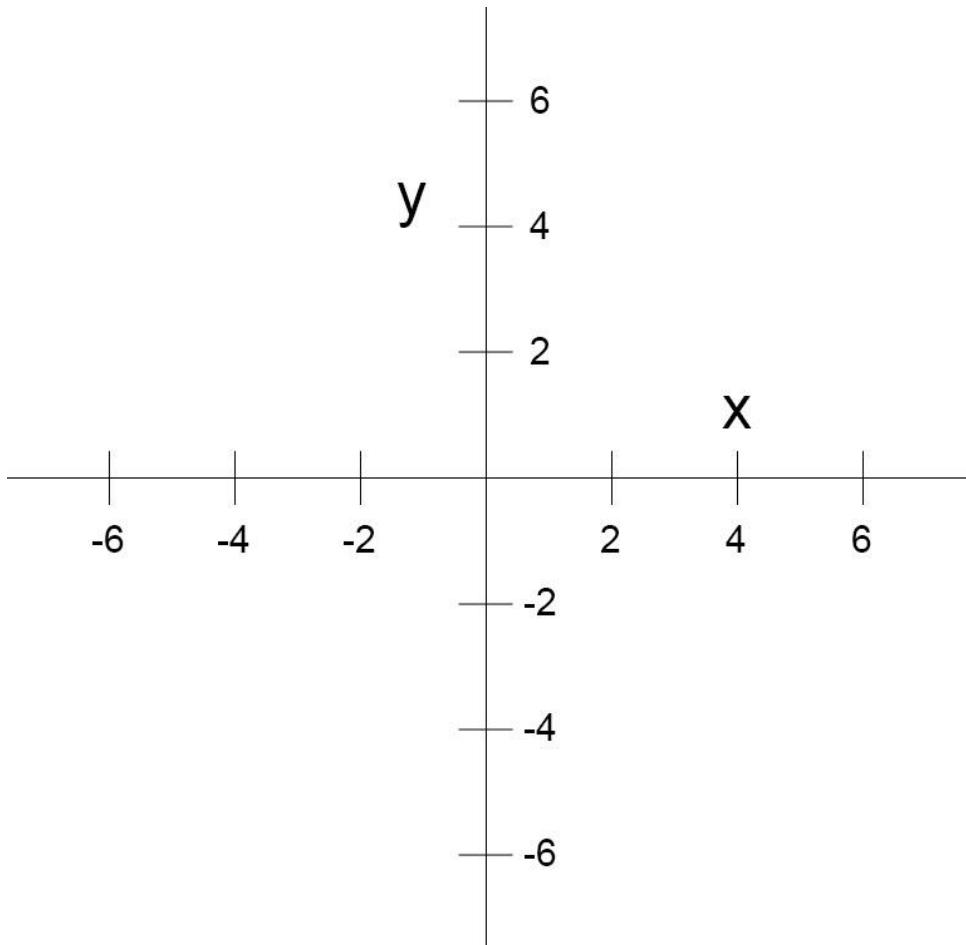


Figure 7.2. The Cartesian plane.

In a Cartesian plane, the horizontal axis is usually labeled “x” and the vertical axis is labeled “y.” Values increase on the x-axis as they go left to right and increase on the y-axis as they go from bottom to top. Note that this is different from many common computer screen coordinate systems, including HTML’s canvas, in which the y-axis is reversed.

You can plot pairs of x, y values on a Cartesian plane. For example, the value 3, 4 would be plotted at 3 on the x-axis and 4 on the y-axis, as shown in Figure 7.3.

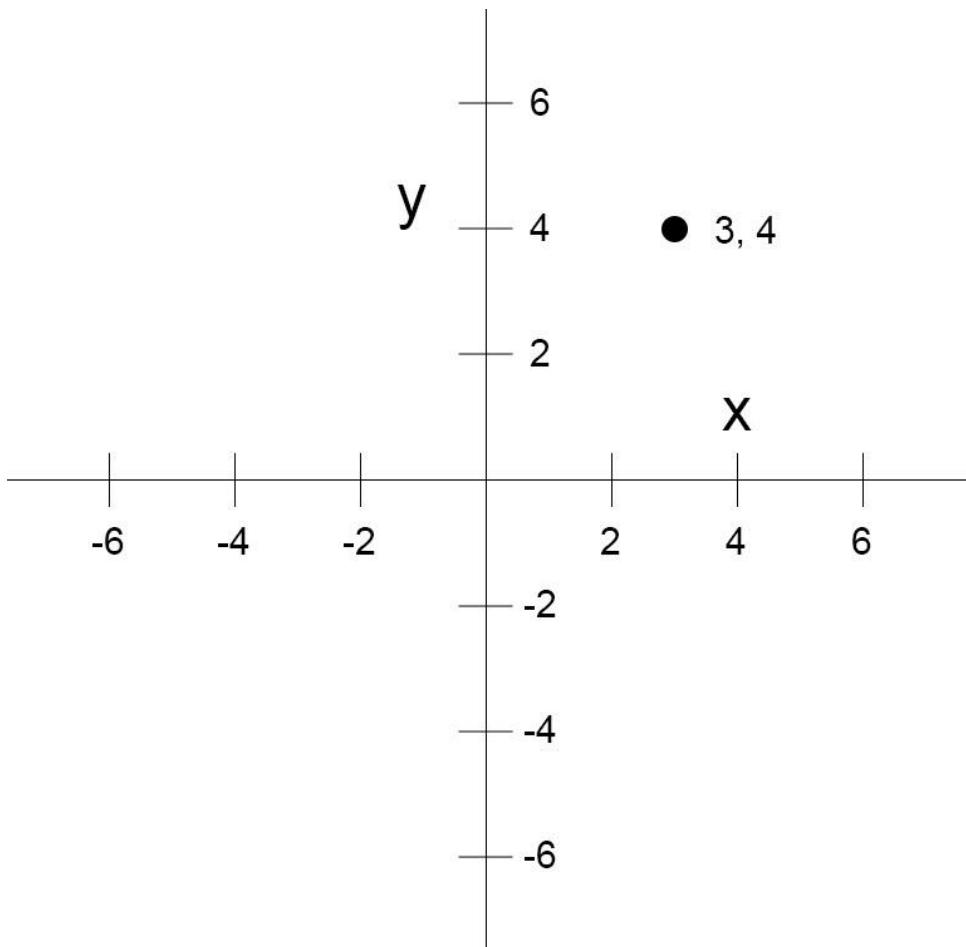


Figure 7.3. The value 3, 4 plotted.

Simple enough, right? Well, you can do the same thing with complex numbers. As you've seen, complex numbers have two components: the real part and the imaginary part. If you use the horizontal axis to represent the real component of a complex number, and the vertical axis to represent the imaginary part, you have what is known as the complex plane, as seen in Figure 7.4.

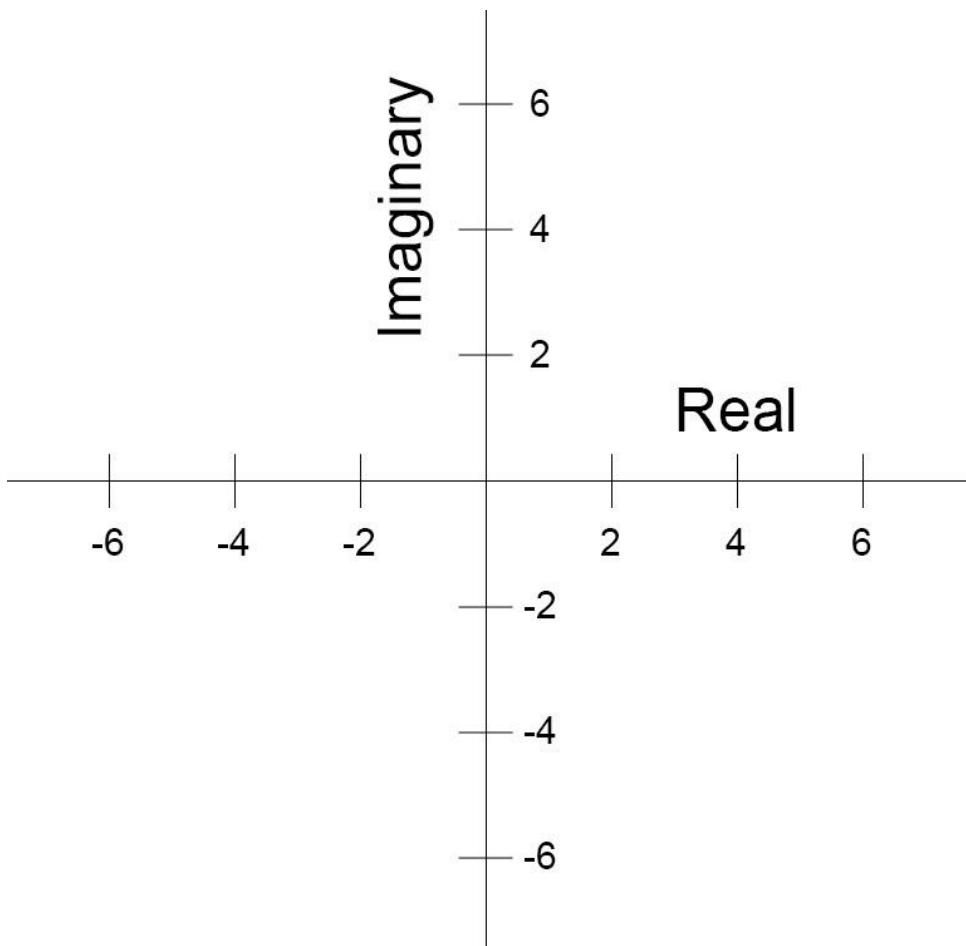


Figure 7.4. The complex plane.

Now, you can take a complex number, such as $4 - 3i$, and graph 4 on the real axis and $-3i$ on the imaginary axis, as you can see in Figure 7.5.

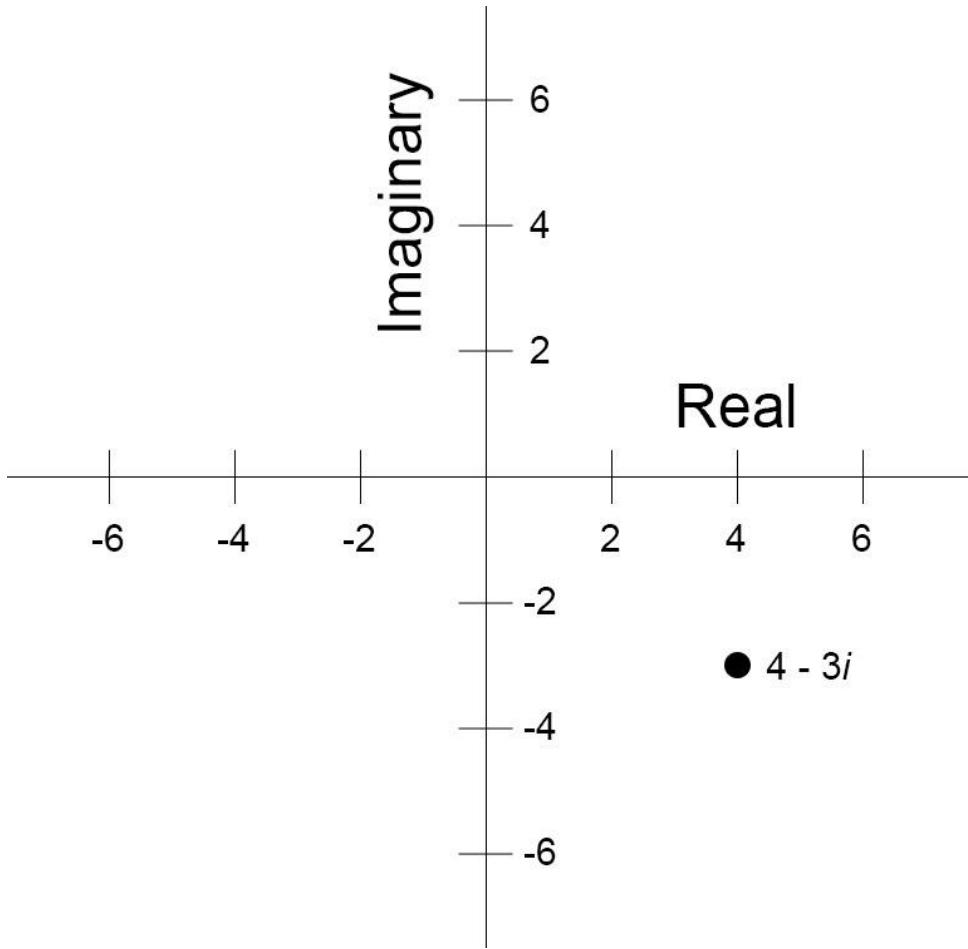


Figure 7.5. The value $4 - 3i$ plotted on the complex plane.

Now that you understand this much, I can tell you that the values fed into the formula that creates the Mandelbrot set are complex numbers and the image you get is plotted on the complex plane. This is made more clear in Figure 7.6.

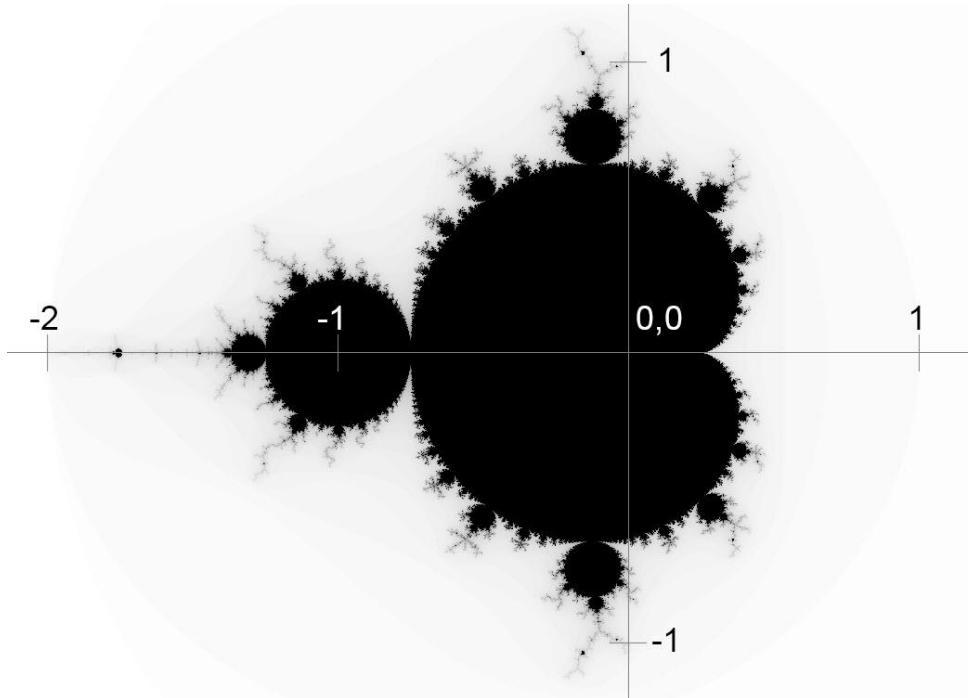


Figure 7.6. The Mandelbrot set on the complex plane.

Alright, with all that knowledge in place, you're ready to proceed with the formula that will create the Mandelbrot set.

The Mandelbrot Formula

The formula that creates the Mandelbrot set is:

$$z_1 = z^2 + c$$

It seems deceptively simple. The values z and c are both complex numbers. c is a constant and is initially set to around $-2 + i$. Recalling Figure 7.6, this is the top-right corner of the portion of the complex plane holding the Mandelbrot set. In the first iteration, z is set equal to $0 + 0i$. The value of z is squared and added to c . This gives you a

new complex number, which becomes the value for z_1 . Then z_1 is fed back into the equation as z , with c remaining the same, giving you a new z_1 .

Eventually, you'll notice that z_1 is getting larger and larger, and it's obvious that it's just going to shoot off into infinity. At this point you know that the complex number in c is NOT part of the Mandelbrot set. You can then color that point on the plane somehow. Usually, the color is chosen based on how many iterations of the formula it took to determine it was not part of the set.

You move onto the next point in the complex plane that you want to test, maybe something like $-1.99 + i$. At some point, you'll find that the values you are getting for z_1 are not going off towards infinity. Even after many iterations, they are remaining in a stable, low range. When you see that, you know that the value in c IS part of the Mandelbrot set. Usually, this is colored black.

You continue working this way through all points on the chosen portion of the complex plane down to the lower-right corner at around $1 - i$. When you're done, you'll have a nice picture of the Mandelbrot set!

A couple of obvious questions may come to mind at this point:

1. How many iterations do you perform before making a decision as to whether or not a point is part of the set?
2. What value would have to be reached to determine that a number is going off to infinity, and is, thus, not a part of the set?

As for the first question, there is no specific number of iterations that should be done. In fact, most Mandelbrot-

generating programs allow you to adjust the number of iterations. A higher number of iterations will give you more detail, especially useful as you zoom closer and closer into complicated areas.

As for how to determine if a number is getting too big, and not part of the set, there is a simple answer to this. Actually, the theory behind it is a bit more complex, but the application is simple. If you find that the magnitude of the complex number Z has gone above 2, then the number will not be in the Mandelbrot set. Now, what do I mean by “magnitude of the complex number?”

If you picture a complex number plotted on the complex plane, as in Figure 7.5, then imagine a vector pointing from the origin $0 + 0i$, that vector’s length will be the magnitude of the complex number, as in Figure 7.7.

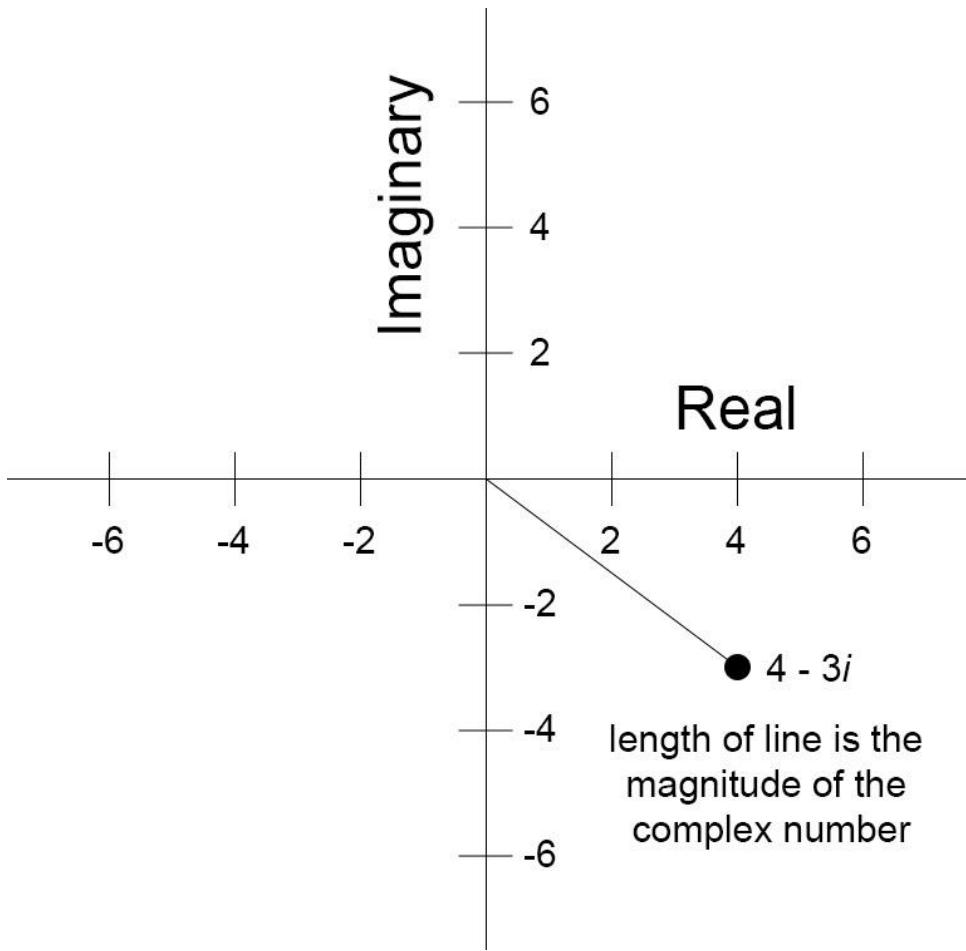


Figure 7.7. The magnitude of a complex number.

The magnitude is also sometimes called the MODULUS, or ABSOLUTE VALUE, of the complex number.

Mathematically, you can find it by using the Pythagorean theorem. If you have a complex number $a + bi$, the magnitude is the square root of a squared plus b squared.

So, if this magnitude is greater than 2, you know the value is not part of the set. You'll see that this is easy once you put it into code.

That's it. Now let's write some code.

Mandelbrot Code

This example is going to be far more complex than anything else you've seen in this book so far, so I'm going to take it one step at a time.

First, the top part of the file, which shouldn't have anything shocking:

```
window.onload = function() {
    var currentX = 0,
        imageData,
        stripWidth = 50,
        minR = -2,
        maxR = 1,
        minI = -1.2,
        maxI = 1.2,
        maxIter = 100,
        interval,
        dr, di,
        aspectRatio;

    init();

    function init() {

        chaos.init();

        aspectRatio = chaos.width / chaos.height;
        imageData =
            chaos.context.getImageData(
                0, 0,
                chaos.width,
                chaos.height);
        renderFull();

        document.body.addEventListener("keyup",
            function(event) {
                // console.log(event.keyCode);
                switch(event.keyCode) {
                    case 80: // p
                        chaos.popImage();
                        break;

                    default:
                }
            }
        );
    }
}
```

```
        break;
    }
});
```

Starting out, there are several variables, most of which I will explain as I go through the other sections. Notable are `minR`, `maxR`, `minI` and `maxI`. These refer to the minimum and maximum values on the real and imaginary axes of the complex plane and form the area of the plane that will be rendered. The Mandelbrot set fits nicely between -2 and 1 on the real axis and goes just past -1 and 1 on the imaginary axis.

The `init` function is called, calling `chaos.init` as usual. It takes note of the aspect ratio of the canvas and gets the image data for the canvas. It then calls `renderFull` before setting up the usual key handler:

```
function renderFull() {
    currentX = 0;
    adjustWidth();
    dr = (maxR - minR) / chaos.width;
    di = (maxI - minI) / chaos.height;

    interval = setInterval(renderStrip, 0);
}
```

The `renderFull` function calls `adjustWidth`, which I'll explain in just a minute. Next, it calculates values for `dr` and `di`. These stand for delta real and delta imaginary. They represent the size of a single pixel on the portion of the complex plane being represented. For example, you are currently viewing from -2 to 1 on the real axis. That's a distance of 3. If your canvas width is 1000, then `dr` will be $3 / 1000$ or .003. In other words, if you move one pixel left or right on the canvas, you are moving .003 on the real axis.

All this is so that you can calculate what point on the complex plane you are dealing with when you begin to render a single pixel.

After this, the function sets up an interval that will call the function `renderStrip` over and over as fast as possible:

```
function adjustWidth() {  
    var w = maxR - minR,  
        h = maxI - minI,  
        newW = h * aspectRatio,  
        diff = newW - w;  
  
    minR -= diff / 2;  
    maxR += diff / 2;  
}
```

The whole purpose of the `adjustWidth` function is to change the width of the portion of the complex plane you are viewing so that it matches the aspect ratio of the canvas. With the default maximum and minimum real and imaginary axis values, the aspect ratio would be $3/2$. Your canvas will usually have an aspect ratio closer to $4/3$. Without matching the two, your image will wind up stretched out, something like in Figure 7.8.

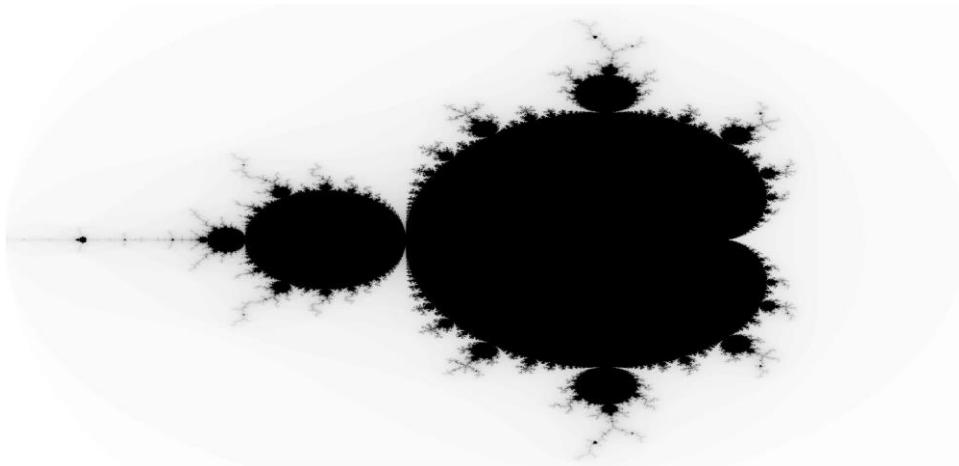


Figure 7.8. Unmatched aspect ratios.

What this function does is find the current width and height of the plane portion. It then calculates the ideal width based on the height times the correct aspect ratio and stores this

in `newW`. It then takes the difference between these two widths and adds half to the left and half to the right.

Now that the aspect ratio is correct, it's time to render:

```
function renderStrip() {
    var x, y, h, color, index,
        w4 = chaos.width * 4,
        iData = imageData.data;

    // work across the strip horizontally
    for(x = currentX; x < currentX + stripWidth; x += 1) {
        index = x * 4;
        // render all the pixels in this vertical column
        for(y = 0, h = chaos.height; y < h; y += 1) {
            color = mandel(x, y);
            iData[index] = color.red;
            iData[index + 1] = color.green;
            iData[index + 2] = color.blue;
            iData[index + 3] = 255;
            index += w4;
        }
        if(x > chaos.width) {
            clearInterval(interval);
            break;
        }
    }
    chaos.context.putImageData(imageData, 0, 0,
                                currentX, 0,
                                stripWidth,
                                chaos.height);
    currentX += stripWidth;
}
```

The `renderStrip` function is what gets called over and over. Its name comes from the fact that it's going to render a single vertical strip of the image at a time. The size of this strip is controlled by the `stripWidth` variable at the start of the file, which is initially set to 50.

The reason for rendering the image in strips is a compromise. On the one hand, you could render the entire image one pixel at a time. The problem with that is that the `putImageData` method is VERY slow. The less you call it

the better. To render each pixel, you'd be calling it many thousands of times.

On the other hand, you could just calculate all the pixels, set their values into the image data and, when you're done, call `putImageData` a single time. That, indeed, is the fastest way to do it. The problem there is that you could be sitting there for up to several seconds looking at a blank screen, then suddenly the image would appear all at once.

The compromise is to calculate a chunk of the image, display it, calculate another chunk, display that and so on. This limits the number of calls to `putImageData` to a dozen or two, and lets you see the progress of the image as it builds from left to right.

The `currentX` value is the x position for the start of the current strip. The `for` loop goes from that to `currentX + stripWidth`. It then works down each column of pixels, calling the `mandel` function for each `x, y` value. This returns a color object that is pushed into the image data array. If the right edge of the canvas is hit, it breaks out of the loop and kills the interval.

Finally, once the strip is fully calculated, you can call `putImageData`, writing that strip to the canvas. Note that all of the optional arguments to `putImageData` are used so that only that one section of pixels is drawn to the canvas. The less data you are pushing each time, the quicker it happens.

Now, onto the `mandel` function where all the magic happens:

```
function mandel(x, y) {  
  var cr = minR + x * dr,  
      ci = minI + y * di,  
      zr = 0,  
      zi = 0,  
      iter, zrl, zil;
```

```

for(iter = 0; iter < maxIter; iter += 1) {
    zr1 = zr * zr - zi * zi + cr;
    zi1 = 2 * zr * zi + ci;
    zr = zr1;
    zi = zi1;
    if(zr * zr + zi * zi > 4) {
        var shade = 255
        - Math.floor(iter / maxIter * 255);
        return {
            red: shade,
            green: shade,
            blue: shade
        }
    }
}

return {
    red: 0,
    green: 0,
    blue: 0
}
}

```

Because JavaScript (like most languages) does not have a complex number type, this function just stores the real and imaginary parts of each complex number as a separate variable. So you have `cr` and `ci` to represent the real and imaginary parts of the complex constant `c` and `zr` and `zi` for the complex variable `z`.

Next, the `for` loop iterates through from 0 to the maximum number of iterations, which is set to 100. Remember that the Mandelbrot formula is:

$$z_1 = z^2 + c$$

Again, as you don't have a complex number type, there is no way to simply square and add complex numbers. The first two lines of the `for` loop show the squaring applied to individual components and the addition of the constant. To see how I got there, let's look at squaring a complex number. You have a number like:

$a + bi$

Squaring it is multiplying it by itself:

$$(a + bi)(a + bi)$$

To multiply two complex numbers, multiply each component in one by each component in the other and add them up:

$$a^2 + 2abi + b^2i^2$$

But i^2 equals -1 by definition, so you really have:

$$a^2 + 2abi - b^2$$

So, you have a real part: $a^2 - b^2$

And an imaginary part: $2abi$

Translated to our code, that's the real part:

`zr * zr - zi * zi.`

And the imaginary part:

`2 * zr * zi.`

You add `cr` and `ci` to those and the result is assigned to two temporary values, `zr1` and `zi1`, and then put back into `zr` and `zi`.

The next `if` statement checks to see if the result is going infinite. Remember that to do that you need to find the magnitude of the complex number and see if it is greater than 2. By squaring 2 (to become 4), you can drop the calculation of the square root.

If the value for `z` is going infinite, this means `c` is not in the Mandelbrot set. The code calculates a `shade` value based on the number of iterations that it took to get to this point and assigns this to the red, green and blue values of a color

object. It then returns this object, ending the function. This will create a gray-scale image. I'll discuss other coloring algorithms later.

If the maximum number of iterations is reached, and the value for z is still in range, then c IS in the Mandelbrot set. You color the current pixel black by returning 0 for the red, green and blue components of the color.

And that's all. If all goes well, the image should render from left to right in under a second, giving you the image you see in Figure 7.1.

You can play with the `stripWidth` variable to see how that changes the rendering speed. Change the maximum and minimum real and imaginary values to specify what portion of the complex plane will be rendered. If you want to jump ahead, try out some other methods of generating color values.

One thing to note here is that, technically, the image is rendered upside down. I didn't do any correction for the fact that, in canvas, the y-axis is reversed. Fortunately, the Mandelbrot set has horizontal symmetry – the top half is a mirror image of the bottom. So for simply rendering images, it makes no difference.

Now, changing values in a source file and reloading your browser is not a very efficient way to zoom into the set. Next up, let's create an interactive zooming mechanism.

Zooming In

There are a number of possible ways to allow a user to zoom in to the image. I decided to allow the user to click on

a point with the mouse and drag out a rectangle over the area to zoom in on. Rather than trying to draw this rectangle on the canvas itself, I added a new div with the id of `zoom` to the HTML file itself. Here is `mandelbrot2.html`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Mandelbrot 2</title>

    <style type="text/css">
        html, body {
            margin: 0px;
        }
        canvas {
            display: block;
        }
        #zoom {
            background-color: rgba(255, 0, 0, 0.25);
            width: 0px;
            height: 0px;
            position: absolute;
        }
    </style>

    <script type="text/javascript"
        src="chaos.js"></script>
    <script type="text/javascript"
        src="mandelbrot2.js"></script>

</head>
<body>
    <div>
        <canvas id="canvas"/>
    </div>
    <div id="zoom"></div>
</body>
</html>
```

Note also that I've added some CSS to make the div a transparent red color and zero size. It's also positioned absolutely, so its location can be set to any point on the screen.

In the file, `mandelbrot2.js`, there are three new variables defined at the top of the file:

```
var currentX = 0,
    imageData,
    stripWidth = 50,
    minR = -2,
    maxR = 1,
    minI = -1.2,
    maxI = 1.2,
    maxIter = 100,
    interval,
    dr, di,
    aspectRatio,
    zoomDiv,
    zoomX,
    zoomY;
```

`zoomDiv` will be a reference to the new div you just saw in the HTML file, and `zoomX` and `zoomY` will hold the initial point clicked on when zooming.

There are two changes in the `init` function:

```
function init() {

    chaos.init();

    zoomDiv = document.getElementById("zoom");
    aspectRatio = chaos.width / chaos.height;
    imageData = chaos.context.getImageData(0, 0,
                                            chaos.width, chaos.height);
    renderFull();

    document.body.addEventListener("mousedown",
                                  onMouseDown);

    document.body.addEventListener("keyup",
        function(event) {
            // console.log(event.keyCode);
            switch(event.keyCode) {
                case 80: // p
                    chaos.popImage();
                    break;

                default:
                    break;
            }
        });
}
```

First, grab a reference to the new div using `getElementById` and store it in `zoomDiv`. Then, add a listener for `mousedown`, which will call a new `onMouseDown` function. Let's look at that function next:

```
function onMouseDown(event) {  
    clearInterval(interval);  
    zoomX = event.clientX;  
    zoomY = event.clientY;  
    zoomDiv.style.left = zoomX + "px";  
    zoomDiv.style.top = zoomY + "px";  
    document.body.addEventListener("mousemove",  
        onMouseMove);  
    document.body.addEventListener("mouseup",  
        onMouseUp);  
}
```

The `onMouseDown` function first stops rendering, if it's happening, by calling `clearInterval`. It then notes the location of the click by storing the x and y values in `zoomX` and `zoomY`. It positions the zoom div at that location and starts listening for mouse-move and mouse-up events. First, the handler for mouse move:

```
function onMouseMove(event) {  
    zoomDiv.style.width = event.clientX - zoomX  
        + "px";  
    zoomDiv.style.height = event.clientY - zoomY  
        + "px";  
}
```

This one is simple. It subtracts the initial click point from the current mouse position and sizes the zoom div to fit in between those two points. The result is that you are dragging out a rectangle. Then there is the mouse-up handler:

```
function onMouseUp(event) {  
    var x = event.clientX,  
        y = event.clientY;  
  
    document.body.removeEventListener("mousemove",  
        onMouseMove);  
    document.body.removeEventListener("mouseup",  
        onMouseUp);  
    zoomDiv.style.width = "0px";
```

```

zoomDiv.style.height = "0px";
if(x < zoomX || y < zoomY) {
    return;
}

maxR = minR + dr * x;
maxI = minI + di * y;
minR = minR + dr * zoomX;
minI = minI + di * zoomY;
renderFull();
}

```

There's a bit more going on here. First, remove the listeners for mouse move and mouse up. They'll only be active after a mouse down. Then, return the zoom div to zero size so it disappears.

Now, it's possible that the user dragged up or to the left or both. In this case, you'll have a negative rectangle. It's certainly possible to handle this case. It would also require some changes in `onMouseMove` to show the rectangle. But, in this simple example, I chose to just call `return`, canceling the whole zoom. I figure it gives the user an “out” if he starts to zoom and changes his mind. I'll leave it to you if you want to implement reverse rectangle dragging.

Assuming the zoom is not canceled, new values are calculated for min and max R and I. Remember the current `minR` and `minI` will be located at 0, 0 on the canvas. Multiplying `x` by `dr` and adding it to the current `minR` gives you the new `maxR`. The rest of the values are calculated similarly. Note that you need to do the maximum values first, as they depend on the un-zoomed minimum values.

Once you have your new complex plane values all set, you're ready to render a zoomed image!

Another point to note is that this code does not try to restrain the zoom rectangle to the current canvas aspect ratio. Because `adjustWidth` is called at the beginning of `renderFull`, the `minR` and `maxR` values will be changed to

correct the aspect ratio. Thus, the rectangle you drag out may not be exactly what you see in the final zoom. In fact, if you draw a short, wide rectangle, parts may be cropped off the left and right in the final zoom, as it is all based on the height. Again, I leave this as an exercise for you if you are interested in developing the application further.

Now, you can zoom in to your heart's content, and get images like in Figure 7.9, 7.10 and 7.11.

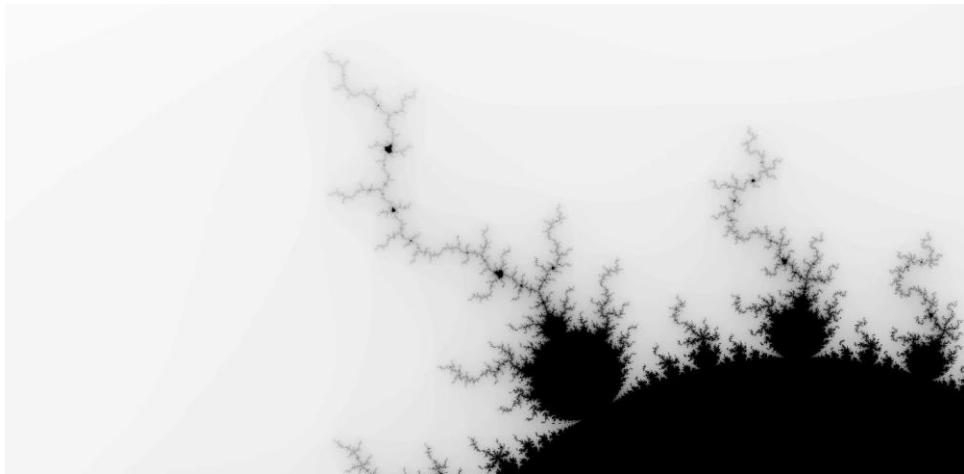


Figure 7.9. Zooming in.

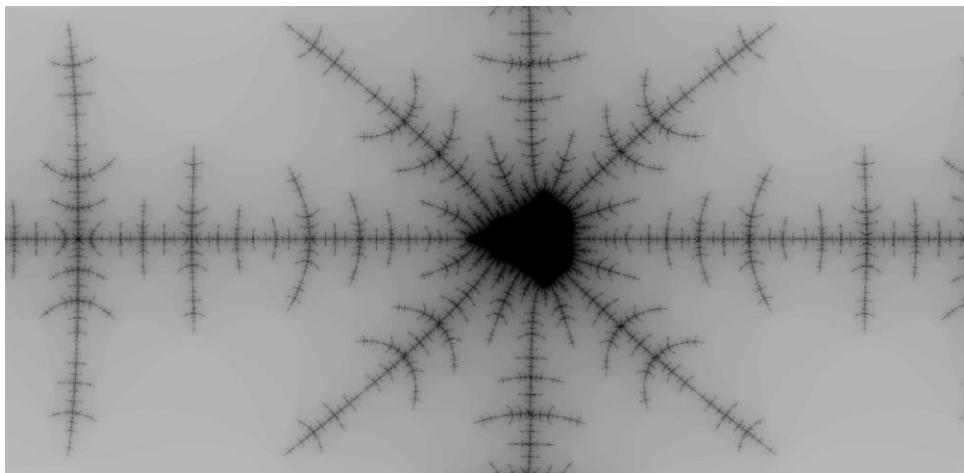


Figure. 7.10.

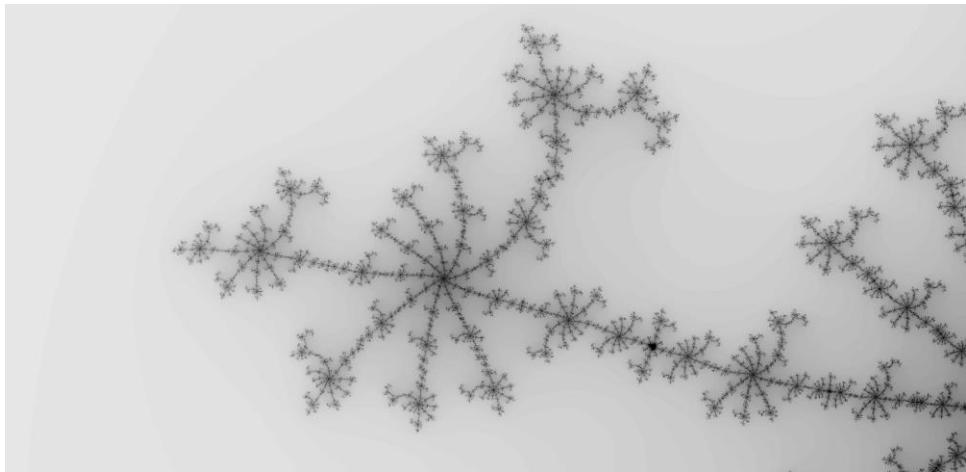


Figure. 7.11.

In fact, I could happily fill the rest of this book with such images, but I'll hold myself back.

Actually, when I said you could zoom in to your heart's content, I was lying. You can zoom in to the limits of your computer's capabilities. Eventually, you're going to see something like in Figure 7.12.

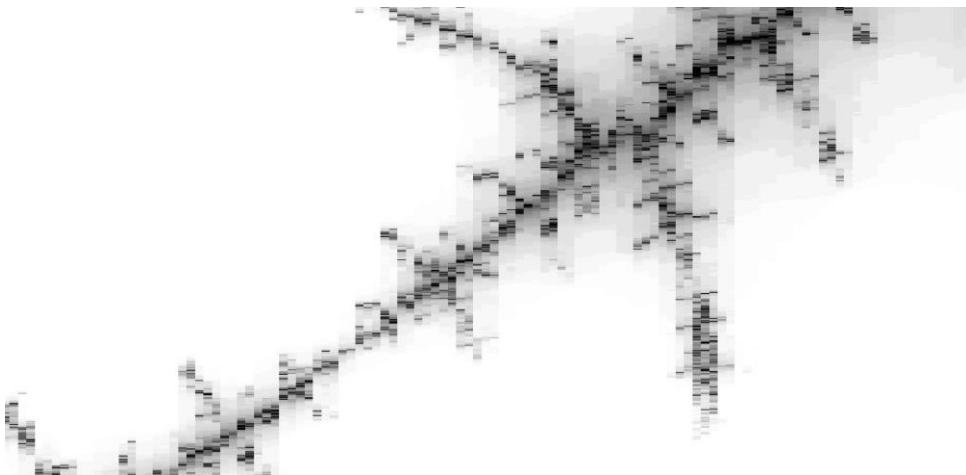


Figure 7.12. You've hit the zoom limit.

The image becomes blocky and pixelated. The reason for this is that you've zoomed into the point where the

differences in minimum and maximum values on the two axes become extremely small. Then, you're trying to split that difference over maybe 1000 or more pixels on the screen. The computer lacks the precision to differentiate such small changes, and so many adjacent pixels get the same values. There are probably tricks to get around this, but I'm not sure it's worth it; you'll have plenty to explore with the zoom available.

More Iterations!

You might notice that, as you zoom in to certain areas, the image begins to get very dark and the black areas that are part of the Mandelbrot set begin to get wider and thicker. Part of this is due to the gray-scale color scheme being used, but it's also because of the relatively low value (100) you're using for maximum iterations. After just 100 iterations, a point may not have escaped to infinity, and it will be counted as part of the set and colored black. But, maybe with a few more iterations, it would have escaped. So it's nice to be able to adjust the value of `maxIter` on the fly. Let's use some keyboard shortcuts for that.

The file `mandelbrot3.js` adds a couple more blocks to the switch statement that handles key events, one for the up key and one for the down.

```
document.body.addEventListener("keyup",
  function(event) {
    switch(event.keyCode) {
      case 80: // p
        chaos.popImage();
        break;

      case 38: // up
        maxIter += 20;
        clearInterval(interval);
```

```

        renderFull();
        break;

    case 40: // down
        maxIter -= 20;
        clearInterval(interval);
        renderFull();
        break;

    default:
        break;
    }
});

```

These either increase or decrease `maxIter` by 20, stop rendering and begin a new render with the new iteration value. As an example of how this changes the image, Figure 7.13 shows a portion of the Mandelbrot set zoomed in at the default number of iterations.



Figure 7.13. Getting muddy.

As you can see, it's getting rather dark and muddy. But hitting the up key a few times increases the number of iterations, lightens the image up and brings out a tremendous amount of detail, as you can see in Figure 7.14.

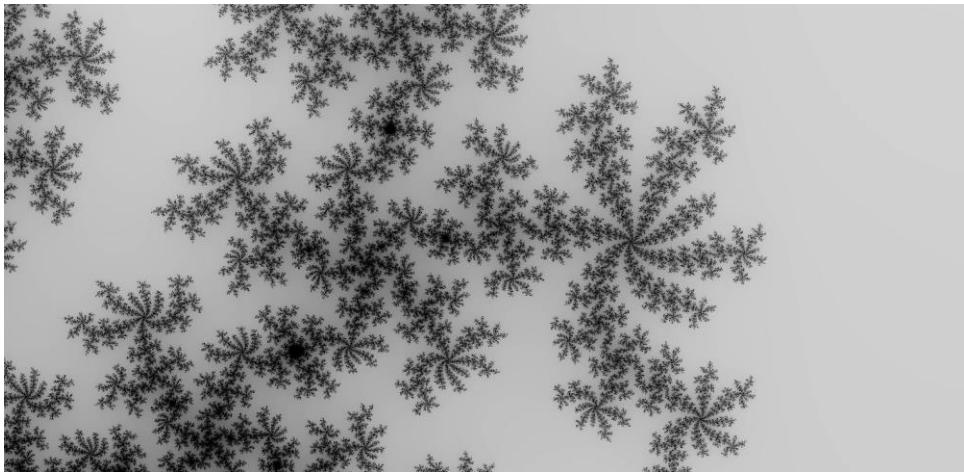


Figure 7.14. More iterations, more detail.

Another experiment you might want to try is automatically increasing the maximum iterations based on the level of zoom.

Coloring Mandelbrots

Thus far, you've been using a simple color scheme that goes from white to black based on the number of iterations it takes to exclude a particular value from the Mandelbrot set. There's been a lot going on with other parts of the file, so I've kept that part of it simple. It looks half-decent, but you can step it up a notch by introducing some new color schemes.

For these changes, see the file `mandelbrot4.js`. First, there are some new variables:

```
var currentX = 0,  
    imageData,  
    stripWidth = 50,  
    minR = -2,  
    maxR = 1,
```

```
minI = -1.2,
maxI = 1.2,
maxIter = 100,
interval,
dr, di,
aspectRatio,
zoomDiv,
zoomX,
zoomY,
colors,
colorA = [255, 196, 0],
colorB = [0, 0, 0],
numColors = 20;
```

The plan is to have two base colors: `colorA` and `colorB`. And an array of color values: `colors`. The code will interpolate between the two base colors in a number of steps, `numColors`, storing the intermediate values in the array. Here, I start with a kind of golden-orange color and go down to black in 20 steps. In the `init` function, I call `initColors`:

```
function init() {

    chaos.init();

    initColors();
    zoomDiv = document.getElementById("zoom");
    aspectRatio = chaos.width / chaos.height;
    imageData = chaos.context.getImageData(0, 0,
                                             chaos.width, chaos.height);
    renderFull();
```

This function interpolates the colors and adds them to the `colors` array. Here it is:

```
function initColors() {
    var redRange = colorB[0] - colorA[0],
        greenRange = colorB[1] - colorA[1],
        blueRange = colorB[2] - colorA[2];

    colors = [];
    for(i = 0; i < numColors; i += 1) {
        colors.push({
            red: colorA[0]
                + Math.floor(redRange
                    / numColors * i),
```

```

        green: colorA[1]
            + Math.floor(greenRange
                / numColors * i),
        blue: colorA[2]
            + Math.floor(blueRange
                / numColors * i)
    })
}
}
}

```

This calculates the difference between each component on the two base colors and fills the `colors` array with values that smoothly ramp between them. The final change is in the `mandel` function:

```

function mandel(x, y) {
    var cr = minR + x * dr,
        ci = minI + y * di,
        zr = 0,
        zi = 0,
        iter, zrl, zil;

    for(iter = 0; iter < maxIter; iter += 1) {
        zrl = zr * zr - zi * zi + cr;
        zil = 2 * zr * zi + ci;
        zr = zrl;
        zi = zil;
        if(zr * zr + zi * zi > 4) {
            return colors[iter % numColors];
        }
    }

    return {
        red: 0,
        green: 0,
        blue: 0
    }
}

```

As before, if this function finds that a value is not in the Mandelbrot set it returns a calculated color value. Now, though, it will return a value from the `colors` array.

Remember that there are only 20 colors in the array at this point, and 100 iterations by default. Therefore, it uses the modulus operator `%` to reuse values from the array. For example, if the iterations were at 25, `iter % numColors` would

be $25 \% 20$, which equals 5. So the 5th color element would be used. When the colors array goes from light to dark, as it does in this case, it gives you some very striking contrast in the image. See Figure 7.15.

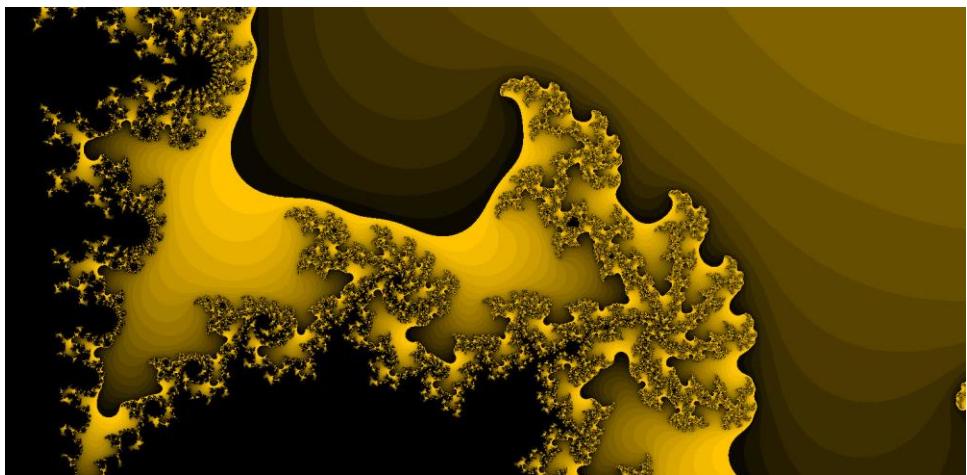


Figure 7.15. Better coloring for better fractals.

Rather than forcing you to type in color values by hand, I created a couple of other functions to generate color schemes:

```
function randomizeColors() {  
    colorA = [Math.random() * 255,  
              Math.random() * 255,  
              Math.random() * 255];  
    colorB = [Math.random() * 32,  
              Math.random() * 32,  
              Math.random() * 32];  
    initColors();  
}  
  
function grayScale() {  
    colorA = [255, 255, 255];  
    colorB = [0, 0, 0];  
    initColors();  
}
```

The `randomizeColors` function does just that: it makes `colorA` and `colorB` random. I kept `colorB` dark, as I like the effect, but try using other values in there.

The `grayScale` function sets a color gradation from white to black. But, because it has a limited number of color values, this time you'll get a contrast effect unlike the smooth gradation you saw in the earlier examples in this chapter.

Both of these functions are called based on key presses:

```
document.body.addEventListener("keyup",
  function(event) {
    switch(event.keyCode) {
      case 80: // p
        chaos.popImage();
        break;

      case 38: // up
        maxIter += 20;
        clearInterval(interval);
        renderFull();
        break;

      case 40: // down
        maxIter -= 20;
        clearInterval(interval);
        renderFull();
        break;

      case 67: // c
        clearInterval(interval);
        randomizeColors();
        renderFull();
        break;

      case 90: // z
        clearInterval(interval);
        numColors -= 2;
        numColors = Math.max(numColors, 2);
        initColors();
        renderFull();
        break;

      case 88: // x
        clearInterval(interval);
        numColors += 2;
        initColors();
        renderFull();
        break;
    }
  }
)
```

```

        case 71: // g
            clearInterval(interval);
            grayScale();
            renderFull();
            break;

        default:
            break;
    }
});

```

The `c` key calls `randomizeColors` and the `g` key calls `grayScale`. You'll see that I also added two other keys in there: `z` and `x`. These lower and raise the number of colors and then recalculate the `colors` array and re-render. Having more colors in the array makes for a smoother gradient in the image. If you have as many or more colors than you have iterations, you'll wind up with one smooth gradient, like you had in all the images earlier in the chapter. The fewer colors you have, the more bands or layers will show up as the iterations loop over them repeatedly. You can see an example of the same image with more colors in Figure 7.16, and fewer colors in Figure 7.17.

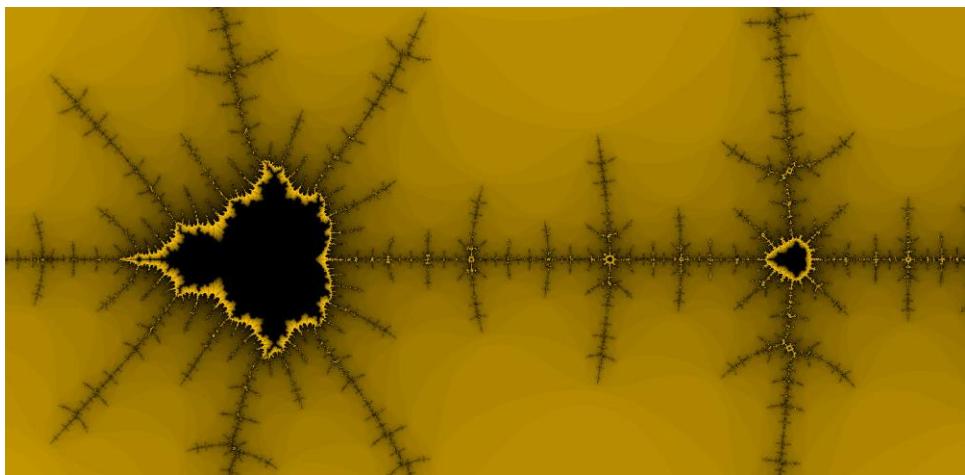


Figure 7.16. More colors, smoother gradient.

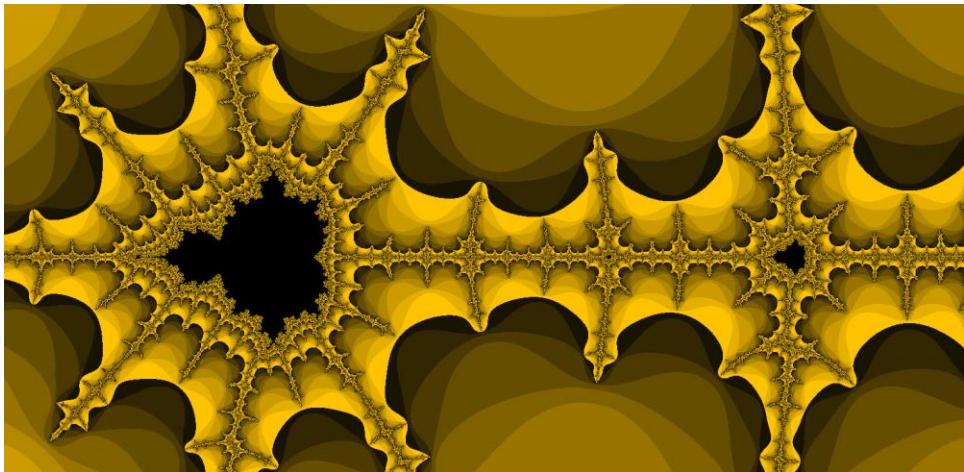


Figure 7.17. Fewer colors, more bands.

Other than the number of colors, these images are exactly the same.

Adjusting all these values – colors, number of colors and maximum iterations – you have in your hands a very powerful tool for exploring the Mandelbrot set. On any decent computer, it will be powerful enough to render even deeply zoomed images within seconds. Back in the '90s, rendering fractals on my Commodore Amiga, I would have killed for the rendering speed you now have in your browser. In truth, this chapter probably took a day or two longer to write than it should have, due to my playing with this program and creating cool images rather than writing the text.

Just to make sure we're on the same page, here is the current code in full for `mandelbrot4.js`:

```
window.onload = function() {
    var currentX = 0,
        imageData,
        stripWidth = 50,
        minR = -2,
        maxR = 1,
        minI = -1.2,
        maxI = 1.2,
```

```
maxIter = 100,
interval,
dr, di,
aspectRatio,
zoomDiv,
zoomX,
zoomY,
colors,
colorA = [255, 196, 0],
colorB = [0, 0, 0],
numColors = 20;

init();

function init() {

    chaos.init();

    initColors();
    zoomDiv = document.getElementById("zoom");
    aspectRatio = chaos.width / chaos.height;
    imageData = chaos.context.getImageData(0, 0,
                                            chaos.width, chaos.height);
    renderFull();

    document.body.addEventListener("mousedown",
                                  onMouseDown);

    document.body.addEventListener("keyup",
        function(event) {
            switch(event.keyCode) {
                case 80: // p
                    chaos.popImage();
                    break;

                case 38: // up
                    maxIter += 20;
                    clearInterval(interval);
                    renderFull();
                    break;

                case 40: // down
                    maxIter -= 20;
                    clearInterval(interval);
                    renderFull();
                    break;

                case 67: // c
                    break;
            }
        }
    );
}
```

```

        clearInterval(interval);
        randomizeColors();
        renderFull();
        break;

    case 90: // z
        clearInterval(interval);
        numColors -= 2;
        numColors = Math.max(numColors, 2);
        initColors();
        renderFull();
        break;

    case 88: // x
        clearInterval(interval);
        numColors += 2;
        initColors();
        renderFull();
        break;

    case 71: // g
        clearInterval(interval);
        grayScale();
        renderFull();
        break;

    default:
        break;
    }
});

}

function onMouseDown(event) {
    clearInterval(interval);
    zoomX = event.clientX;
    zoomY = event.clientY;
    zoomDiv.style.left = zoomX + "px";
    zoomDiv.style.top = zoomY + "px";
    document.body.addEventListener("mousemove",
                                  onMouseMove);
    document.body.addEventListener("mouseup",
                                  onMouseUp);
}

function onMouseMove(event) {
    zoomDiv.style.width = event.clientX
                        - zoomX + "px";
    zoomDiv.style.height = event.clientY

```

```

        - zoomY + "px";
}

function onMouseUp(event) {
    var x = event.clientX,
        y = event.clientY;

    document.body.removeEventListener("mousemove",
                                     onMouseMove);
    document.body.removeEventListener("mouseup",
                                     onMouseUp);
    zoomDiv.style.width = "0px";
    zoomDiv.style.height = "0px";
    if(x < zoomX || y < zoomY) {
        return;
    };

    maxR = minR + dr * x;
    maxI = minI + di * y;
    minR = minR + dr * zoomX;
    minI = minI + di * zoomY;
    renderFull();
}

function randomizeColors() {
    colorA = [Math.random() * 255,
              Math.random() * 255,
              Math.random() * 255];
    colorB = [Math.random() * 32,
              Math.random() * 32,
              Math.random() * 32];
    initColors();
}

function grayScale() {
    colorA = [255, 255, 255];
    colorB = [0, 0, 0];
    initColors();
}

function initColors() {
    var redRange = colorB[0] - colorA[0],
        greenRange = colorB[1] - colorA[1],
        blueRange = colorB[2] - colorA[2];

    colors = [];
    for(i = 0; i < numColors; i += 1) {
        colors.push({

```

```

        red: colorA[0]
          + Math.floor(redRange
            / numColors * i),
        green: colorA[1]
          + Math.floor(greenRange
            / numColors * i),
        blue: colorA[2]
          + Math.floor(blueRange
            / numColors * i)
      })
    }
}

function renderFull() {
  currentX = 0;
  adjustWidth();
  // one pixel's width on complex plane
  dr = (maxR - minR) / chaos.width;
  // one pixel's height on complex plane
  di = (maxI - minI) / chaos.height;

  interval = setInterval(renderStrip, 0);
}

function adjustWidth() {
  // width on complex plane
  var w = maxR - minR,
    // height on complex plane
    h = maxI - minI,
    // width with correct aspect ratio
    newW = h * aspectRatio,
    // difference to equal new width
    diff = newW - w;

  minR -= diff / 2; // add half difference to left
  maxR += diff / 2; // and half to right
}

function renderStrip() {
  var x, y, h, color, index,
    w4 = chaos.width * 4,
    iData = imageData.data;

  // work across the strip horizontally
  for(x = currentX;

```

```

        x < currentX + stripWidth;
        x += 1) {
index = x * 4;
// render all the pixels in this vertical column
for(y = 0, h = chaos.height; y < h; y += 1) {
    color = mandel(x, y);
    iData[index] = color.red;
    iData[index + 1] = color.green;
    iData[index + 2] = color.blue;
    iData[index + 3] = 255;
    index += w4;
}
if(x > chaos.width) {
    clearInterval(interval);
    break;
}
}
chaos.context.putImageData(imageData, 0, 0,
    currentX, 0, stripWidth, chaos.height);
currentX += stripWidth;
}

function mandel(x, y) {
var cr = minR + x * dr,
    ci = minI + y * di,
    zr = 0,
    zi = 0,
    iter, zrl, zil;

for(iter = 0; iter < maxIter; iter += 1) {
    zrl = zr * zr - zi * zi + cr;
    zil = 2 * zr * zi + ci;
    zr = zrl;
    zi = zil;
    if(zr * zr + zi * zi > 4) {
        return colors[iter % numColors];
    }
}

return {
    red: 0,
    green: 0,
    blue: 0
}
}
}

```

Hacking

Now that you've learned how to render the Mandelbrot set the right way, it's time to break it!

By "break it" I mean to start playing with the formulas and see what the code creates when you add things, remove things, change things or move things around. All this will happen in the section of the `mandel` function that does the iterations:

```
for(iter = 0; iter < maxIter; iter += 1) {  
    zr1 = zr * zr - zi * zi + cr;  
    zi1 = 2 * zr * zi + ci;  
    zr = zr1;  
    zi = zi1;  
    if(zr * zr + zi * zi > 4) {  
        return colors[iter % numColors];  
    }  
}
```

There are all kinds of things you can try in here, especially in the lines that define `zr1` and `zi1`. Just try swapping around some values or operators. Or even throw in some other math operations. One of my favorites defines `zi1` as:

```
zi1 = Math.sin(2 * zr * zi) + ci;
```

This gives you Figure 7.18.

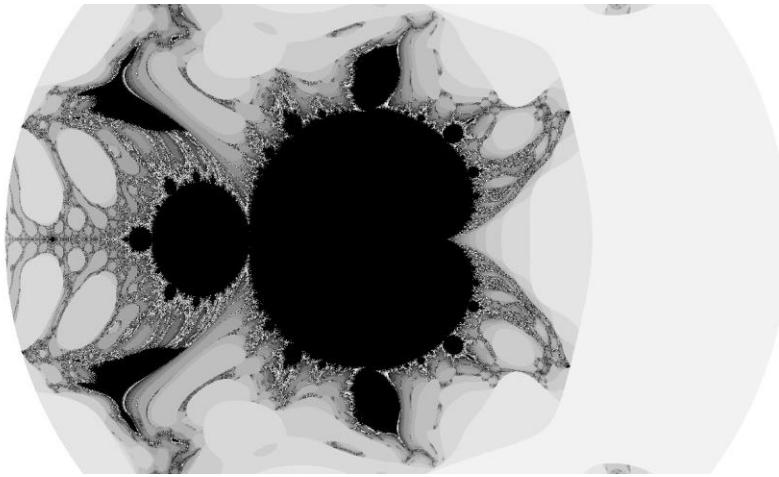


Figure 7.18. Messing with the Mandelbrot.

This immediately looks bizarre, as if some virus has attacked the set. It gets even more strange when you start zooming in and tweaking the iterations and number of colors, though. Some nice patterns form in the left-hand “needle.” See Figure 7.19.

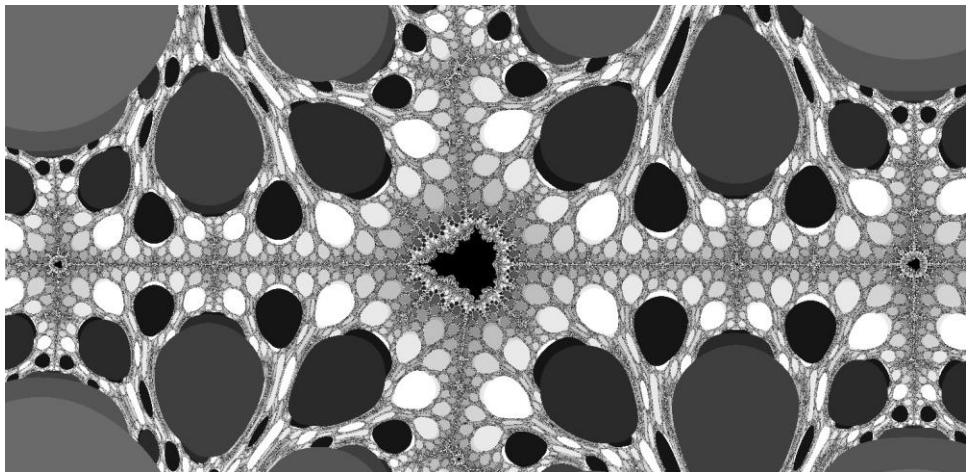


Figure 7.19. Not your run-of-the-mill Mandelbrot.

I imagine you can come up with many different variations. Most of them will be chaotic messes. If you stumble onto something particularly interesting, though, let me know!

There are also some well-known variations of the formula. For example, instead of squaring z each iteration, cube it, or raise it to even higher powers. See if you can work those out.

Perhaps the most famous alternative Mandelbrot is called the “Buddhabrot.” One look at Figure 7.20 will tell you how it got its name.

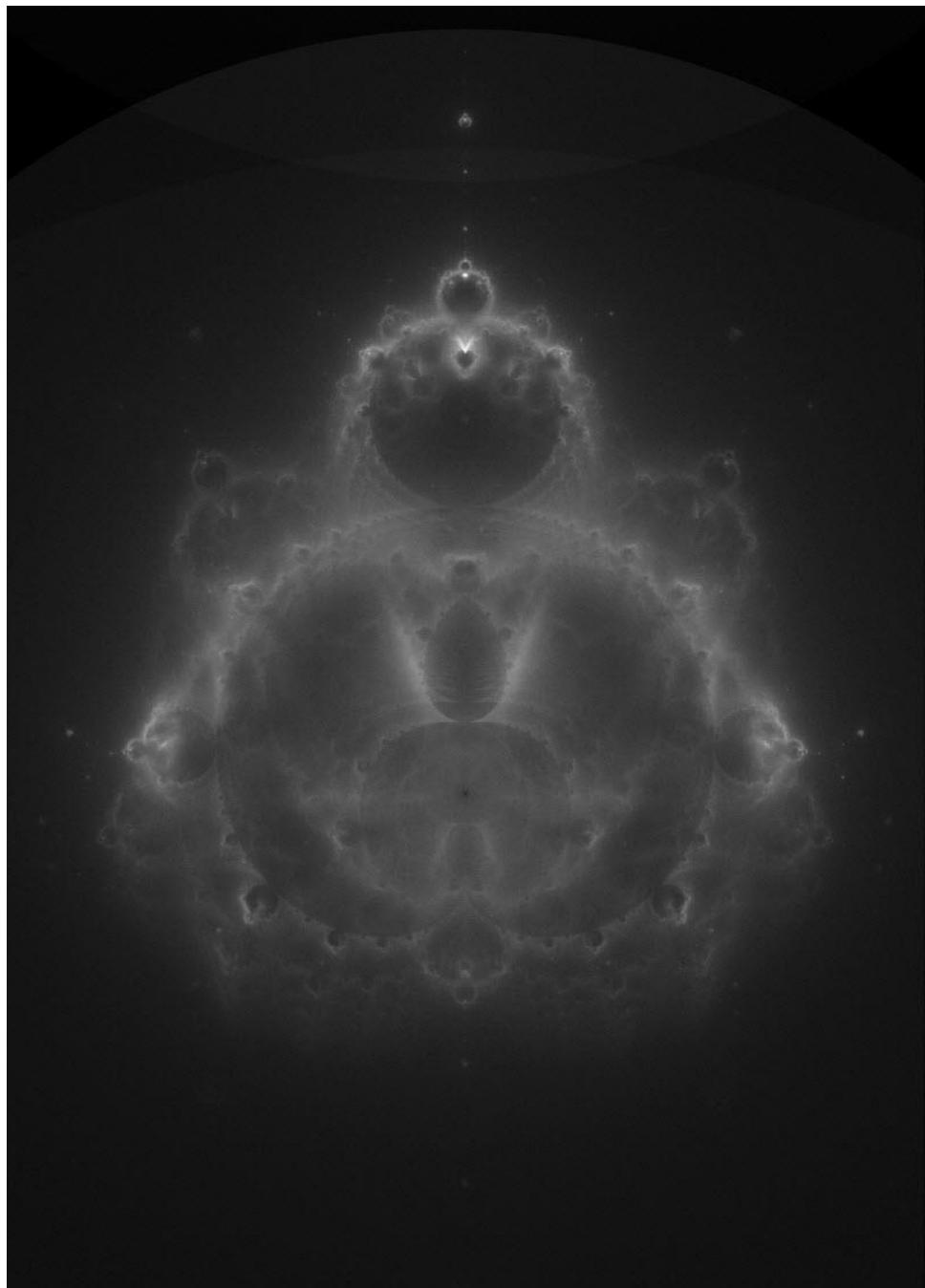


Figure 7.20. Buddhabrot.

Actually, this is made with the standard Mandelbrot formula, but another method is used for rendering the points (and the image is rotated 90 degrees). I'm not going to cover the Buddhabrot here, as it's a more complex and a much more CPU-intensive process than the regular Mandelbrot, and I'm not convinced it's worth doing in JavaScript. Figure 7.20 was rendered with JavaScript, but it took ages. However, you can find the rendering method easily enough and give it a go yourself.

Summary

Whew! This has been quite a chapter. I hope it has given you a decent understanding of the process of rendering the Mandelbrot set. You now have in your hands a far more powerful program for rendering these images than you could have easily found a decade ago. It is powerful, not only because it is faster, but also because you have full access and understanding of every line of the source code and can tweak it to your will. If you get half as much enjoyment out of it as I did creating and using it myself, I'll be happy.

Next, I'll cover the Julia set. This is a fractal set very closely related to the Mandelbrot set. In fact, I originally planned to cover both in the same chapter. But this chapter has grown large enough, so I'll shift the Julias over one and give you a shorter, easier chapter to follow this one.

Chapter 8: Julia Sets and Fatou Dusts

Now let's look at another group of fractal images, those created by rendering Julia sets. This will be a relatively brief chapter, as all of the groundwork was laid in Chapter 7. You'll find that fractals based on Julia sets look quite similar to those made with the Mandelbrot set, especially once you zoom in a bit. This is because Julia sets are very closely related to the Mandelbrot set. In fact, you'll be using the same formula, just applying it in a different way. Figure 8.1 shows a typical Julia fractal.

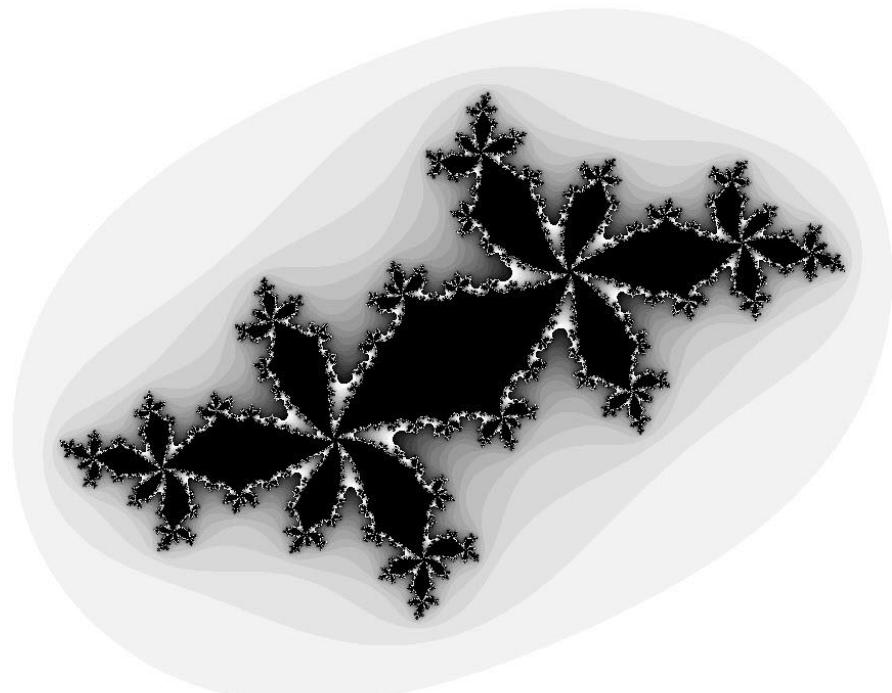


Figure 8.1. A typical Julia fractal.

You'll also learn about Fatou sets and their relationship to Julia sets as well as a type of Julia set sometimes called a Fatou dust.

Background

The Julia and Fatou Sets were named after two French mathematicians, Gaston Julia and Pierre Fatou, pioneers in the field of complex dynamics in the early 1900s. Some claim that they were in fact the ones to “discover” the Mandelbrot set, long before Mandelbrot himself had anything to do with it. Certainly, they were dealing with the same equations, and it's likely that they ran across it. But,

without access to computers, they probably had only the tiniest glimpse of the set's amazing properties.

Theory

You can calculate the Julia set of any complex function of a dynamical system. You don't have to understand exactly what that means. The important point is that there is no single Julia set. Each function will generate a different Julia set and a complementary Fatou set. Many of these sets will be fractals. This chapter will only deal with Julia sets based on the function that generates the Mandelbrot set:

$$z_1 = z^2 + c$$

When used to generate Julia sets, this function will almost always generate fractals.

So how is it that the same formula can result in such different images? Again, it's how you apply that formula. When rendering the Mandelbrot set, you choose a point on the complex plane to render. You then set c equal to that point and z to $0 + 0i$ and begin iterating the function. When you choose a new point to render, you change c to equal that point and reset z to 0 and iterate again.

To render a Julia fractal, you choose a complex number for c and hold it constant **THROUGHOUT THE ENTIRE PROCESS** of rendering the image. In other words, a single value for c is chosen at the beginning of the program, and it never changes. Instead, z varies for each point on the plane. If you are rendering a pixel that represents a point on the complex plane, $-2 + 1.6i$, then that becomes the initial value for z . When you are done with that point and you

move onto the next one, the next point is used as the value for z in the next set of iterations, but c never changes.

So, rather than a single image, there are infinite Julia fractals. Every value you choose for c creates a unique Julia set and a unique image.

Definitions

Using the same rendering techniques used to create the Mandelbrot images, you will again wind up with some pixels colored black (those that did not escape to infinity) and some rendered with whatever color scheme you are using (those that did go infinite).

Remember that the Mandelbrot set is the set of all those points that did not go to infinity – all the black pixels. In the world of Julia fractals, that collection of black pixels is sometimes called the “filled-in Julia set”.

This is a bit different from the actual Julia set itself, which is technically the border between the black area and the colored area. Think of it as a very squiggly line, like a Koch snowflake.

Finally, the Fatou set is every point that is NOT on the border. This includes all the points outside the border and all the points inside the border (the filled-in Julia set).

So, technically, when you see an image of a Julia fractal, most of what you are actually seeing (the black interior and the colored exterior) is the complimentary Fatou set.

I will say that there seems to be a bit of confusion on these terms if you start searching the Internet for definitions. Many of the definitions are far too technical to absorb without

several readings (and even then leave you scratching your head), some are rather misleading and questionable, and a couple are flat-out wrong. Honestly, for the purposes of playing and making interesting renderings, the exact definitions are not extremely important, but I figure it's worth the effort to know and use the proper terms.

Connectedness

Let me take a slight detour here and discuss the concept of connectedness. It has been proven that the Mandelbrot set is “connected.”. This means that, if you take an image of the Mandelbrot set and choose any point in the set (any black pixel), you will be able to move to any other point in the set without leaving the set. You might need to zoom in very closely, and it might be a nearly infinite, twisting, turning path, but you can always “get there from here.” I’ve never looked at the proof of this, and have no illusions that I would vaguely understand it if I did. I just trust that smarter people than myself have validated it.

This also means that if you drew a line to represent the border of the set, you’d have a single, closed loop with a single area inside and an area outside that loop.

Now, when you start rendering different Julia fractals, you may see that the black parts of the image are very obviously connected, or very obviously DISCONNECTED. For some images, it may not be very clear until you zoom in – and in some cases not even then. Figures 8.2 and 8.3 show some obvious examples.

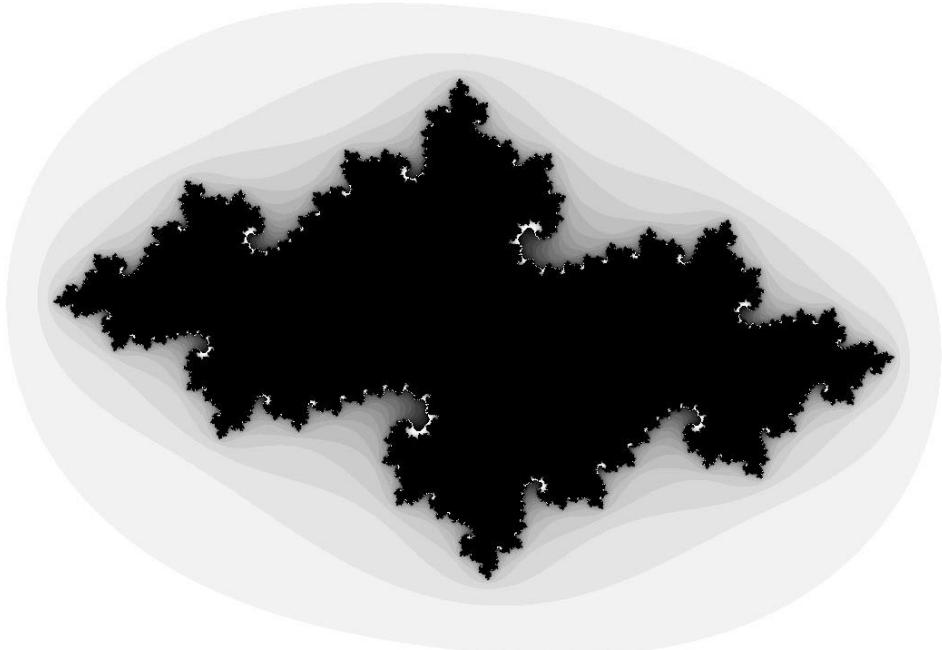


Figure 8.2. Obviously connected.

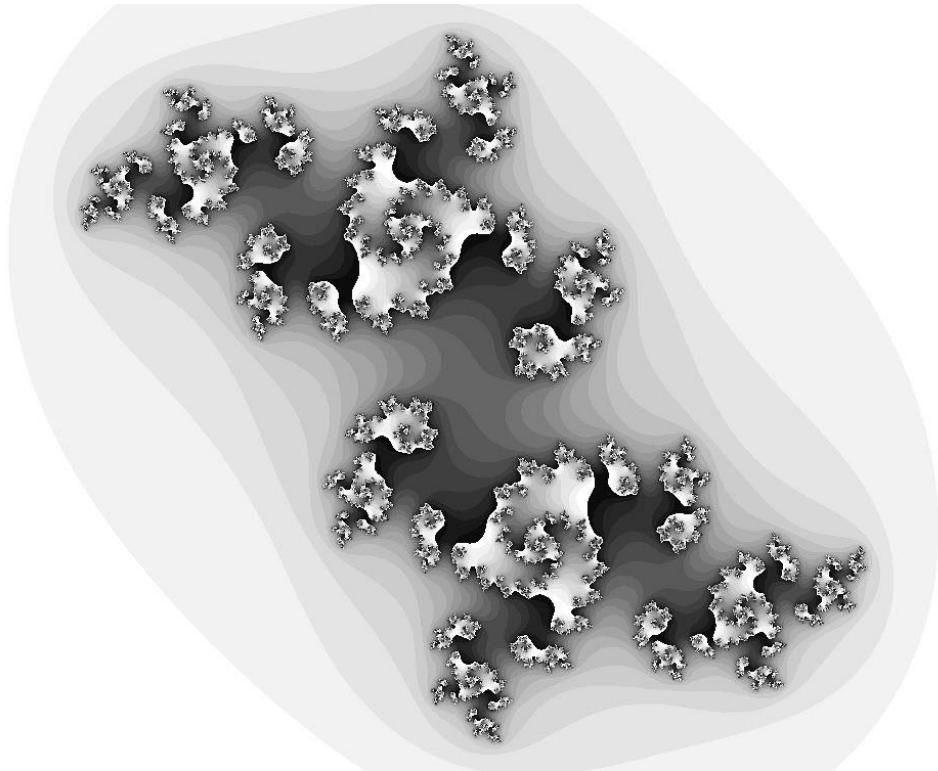


Figure 8.3. Obviously disconnected.

When the areas of black are very tiny and very disconnected, the fractal is sometimes called a “Fatou Dust.” Figure 8.4 should make the reason for the name obvious.

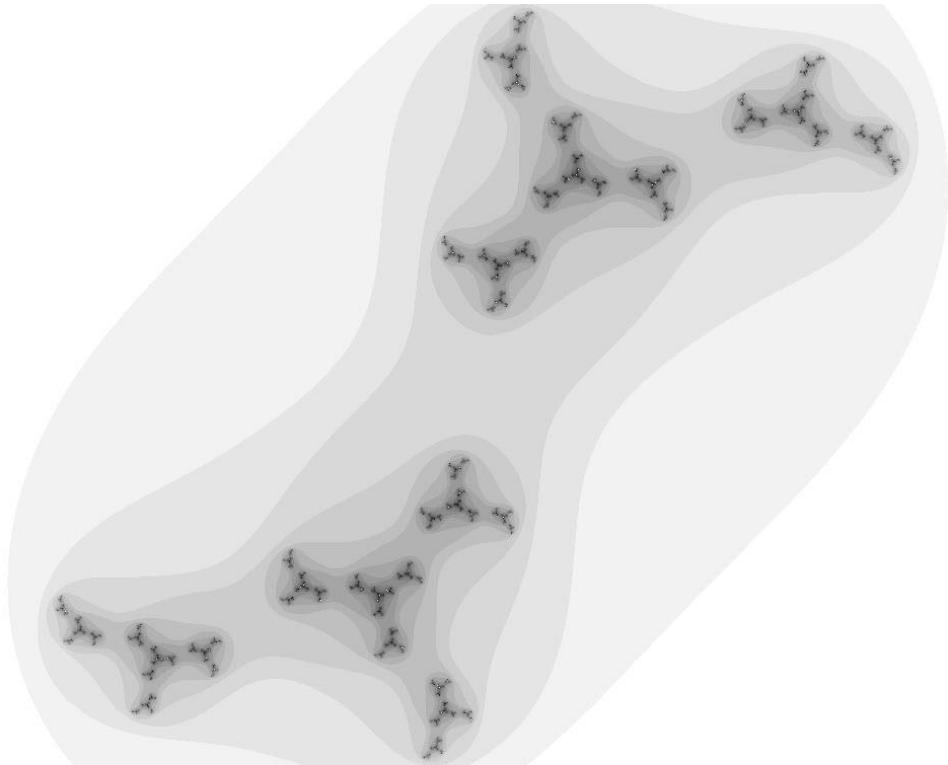


Figure 8.4. Fatou dust.

Realize that, for disconnected Julia sets, the border is not a single, connected line, but one for each disconnected region. And there are no longer just two components (inner and outer) of the Fatou Set but multiple components – one for each region. Actually, when you start zooming into these disconnected shapes, you'll see that "multiple" in this sense probably means "infinite."

I think that the term "Fatou dust" is one of the things that has caused some confusion in the definitions. It seems that some have taken this to mean connected sets are Julia sets and disconnected sets are known as Fatou sets. Just remember that the Julia set is the line representing the border of the shape, regardless of whether it is connected or disconnected.

The Code

As I said earlier, the function for rendering a Julia set is the same one you used in Chapter 7 to render the Mandelbrot set, with just a few changes in how it's used. The `c` constant is constant for the whole image, and `z` represents the point being rendered at the start of each set of iterations. These are very easy changes to make. Only a few lines of code to change in the whole program. Here's `julia1.js`:

```
window.onload = function() {
    var currentX = 0,
        imageData,
        stripWidth = 50,
        minR = -1,
        maxR = 1,
        minI = -1.4,
        maxI = 1.4,
        maxIter = 100,
        dr, di,
        aspectRatio,
        zoomDiv,
        zoomX,
        zoomY,
        colors,
        colorA = [255, 255, 255],
        colorB = [0, 0, 0],
        numColors = 20,
        cr = .125,
        ci = -.612;

    init();

    function init() {
        chaos.init();

        initColors();
        zoomDiv = document.getElementById("zoom");
        aspectRatio = chaos.width / chaos.height;
        imageData = chaos.context.getImageData(0, 0,
                                                chaos.width, chaos.height);
        renderFull();
    }
}
```

```
document.body.addEventListener("mousedown",
    onMouseDown);

document.body.addEventListener("keyup",
    function(event) {
        // console.log(event.keyCode);
        switch(event.keyCode) {
            case 80: // p
                chaos.popImage();
                break;

            case 38: // up
                maxIter += 20;
                clearInterval(interval);
                renderFull();
                break;

            case 40: // down
                maxIter -= 20;
                clearInterval(interval);
                renderFull();
                break;

            case 67: // c
                clearInterval(interval);
                randomizeColors();
                renderFull();
                break;

            case 90: // z
                clearInterval(interval);
                numColors -= 2;
                numColors = Math.max(numColors, 2);
                initColors();
                renderFull();
                break;

            case 88: // x
                clearInterval(interval);
                numColors += 2;
                initColors();
                renderFull();
                break;

            case 71: // g
                clearInterval(interval);
                grayScale();
                renderFull();
        }
    }
);
```

```

        break;

    default:
        break;
    }
});
}

function onMouseDown(event) {
    clearInterval(interval);
    zoomX = event.clientX;
    zoomY = event.clientY;
    zoomDiv.style.left = zoomX + "px";
    zoomDiv.style.top = zoomY + "px";
    document.body.addEventListener("mousemove",
                                  onMouseMove);
    document.body.addEventListener("mouseup",
                                  onMouseUp);
}

function onMouseMove(event) {
    zoomDiv.style.width = event.clientX - zoomX
                        + "px";
    zoomDiv.style.height = event.clientY - zoomY
                          + "px";
}

function onMouseUp(event) {
    var x = event.clientX,
        y = event.clientY;

    document.body.removeEventListener("mousemove",
                                    onMouseMove);
    document.body.removeEventListener("mouseup",
                                    onMouseUp);
    zoomDiv.style.width = "0px";
    zoomDiv.style.height = "0px";
    if(x < zoomX || y < zoomY) {
        return;
    };

    maxR = minR + dr * x;
    maxI = minI + di * y;
    minR = minR + dr * zoomX;
    minI = minI + di * zoomY;
    renderFull();
}

```

```

function randomizeColors() {
    colorA = [Math.random() * 255,
              Math.random() * 255,
              Math.random() * 255];
    colorB = [Math.random() * 32,
              Math.random() * 32,
              Math.random() * 32];
    initColors();
}

function grayScale() {
    colorA = [255, 255, 255];
    colorB = [0, 0, 0];
    initColors();
}

function initColors() {
    var redRange = colorB[0] - colorA[0],
        greenRange = colorB[1] - colorA[1],
        blueRange = colorB[2] - colorA[2];

    colors = [];
    for(i = 0; i < numColors; i += 1) {
        colors.push({
            red: colorA[0] + Math.floor(redRange
                                         / numColors * i),
            green: colorA[1] + Math.floor(greenRange
                                         / numColors * i),
            blue: colorA[2] + Math.floor(blueRange
                                         / numColors * i)
        })
    }
}

function renderFull() {
    currentX = 0;
    adjustWidth();
    // one pixel's width on complex plane
    dr = (maxR - minR) / chaos.width;
    // one pixel's height on complex plane
    di = (maxI - minI) / chaos.height;

    interval = setInterval(renderStrip, 0);
}

function adjustWidth() {
    // width on complex plane
    var w = maxR - minR,

```

```

// height on complex plane
h = maxI - minI,

// width with correct aspect ratio
newW = h * aspectRatio,

// difference to equal new width
diff = newW - w;

minR -= diff / 2;      // add half difference to left
maxR += diff / 2;      // and half to right
}

function renderStrip() {
    var x, y, h, color, index,
        w4 = chaos.width * 4,
        iData = imageData.data;

    // work across the strip horizontally
    for(x = currentX;
        x < currentX + stripWidth;
        x += 1) {
        index = x * 4;
        // render all the pixels in this vertical column
        for(y = 0, h = chaos.height; y < h; y += 1) {
            color = julia(x, y);
            iData[index] = color.red;
            iData[index + 1] = color.green;
            iData[index + 2] = color.blue;
            iData[index + 3] = 255;
            index += w4;
        }
        if(x > chaos.width) {
            clearInterval(interval);
            break;
        }
    }
    chaos.context.putImageData(imageData, 0, 0,
                                currentX, 0,
                                stripWidth, chaos.height);
    currentX += stripWidth;
}

function julia(x, y) {
    var zr = minR + x * dr,
        zi = minI + y * di,
        iter, zr1, zi1;

```

```

for(iter = 0; iter < maxIter; iter += 1) {
    zr1 = zr * zr - zi * zi + cr;
    zil = 2 * zr * zi + ci;
    zr = zr1;
    zi = zil;
    if(zr * zr + zi * zi > 4) {
        return colors[iter % numColors];
    }
}

return {
    red: 0,
    green: 0,
    blue: 0
}
}
}

```

Now, I'll outline the changes between this file and the `mandelbrot4.js` file you ended the last chapter with:

1. The addition of two variables in the outer layer, `cr` and `ci`:

```

var currentX = 0,
    imageData,
    stripWidth = 50,
    minR = -1,
    maxR = 1,
    minI = -1.4,
    maxI = 1.4,
    maxIter = 100,
    dr, di,
    aspectRatio,
    zoomDiv,
    zoomX,
    zoomY,
    colors,
    colorA = [255, 255, 255],
    colorB = [0, 0, 0],
    numColors = 20,
    cr = .125,
    ci = -.612;

```

2. The change of `minR` from -2 to -1, as Julia fractals will generally be centered – unlike the Mandelbrot set, which sits to the left of center.
3. Changing the name of the `mandel` function to `julia`. This also needs to be changed in the `renderStrip` function where it is called:

```
for(y = 0, h = chaos.height; y < h; y += 1) {  
    color = julia(x, y);  
    ...
```

4. Changing the first couple of lines in `julia` to assign the current point to `zr` and `zi` instead of `cr` and `ci`:

```
function julia(x, y) {  
    var zr = minR + x * dr,  
        zi = minI + y * di,  
        iter, zrl, zil;  
    ...
```

That's all! Now, instead of rendering the Mandelbrot set you'll be rendering a Julia set. For the particular values set for `cr` and `ci`, you will get the image in Figure 8.5.

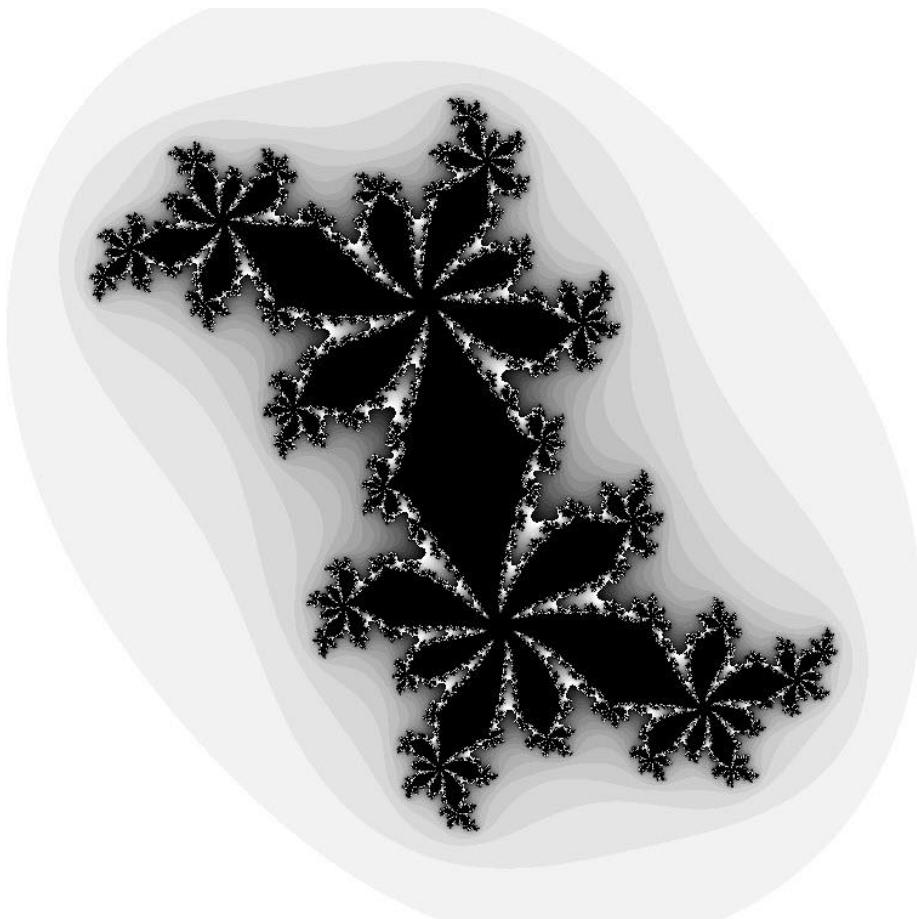


Figure 8.5. A Julia fractal.

Since the program is still the same as the last Mandelbrot program, you'll still have all the zooming and coloring functions you had there, now applied to Julia fractals.

One thing I want to acknowledge is that, like the Mandelbrot images in the previous chapter, the Julia images created by this program will be upside down due to the canvas y-axis being upside down. However, for Julia fractals, this is a more significant change, as they do not have the same symmetry as the Mandelbrot set. Again, I could have corrected this in the code, but it would have added

complexity. It was my choice to leave the code as simple and understandable as possible.

Another thing to note about Figure 8.5 is that it is obviously connected. From any one black point in the image, you could draw a path to any other black point without going through any non-black point. This might look questionable for some of the more detailed areas, but if you zoom into them, you'll see there is actually a clear path. You can increase the maximum iterations, which will close the gaps somewhat. But you'll always be able to offset that with more zooming. Figure 8.6 shows a highly zoomed in area even after the iterations have been raised considerably.

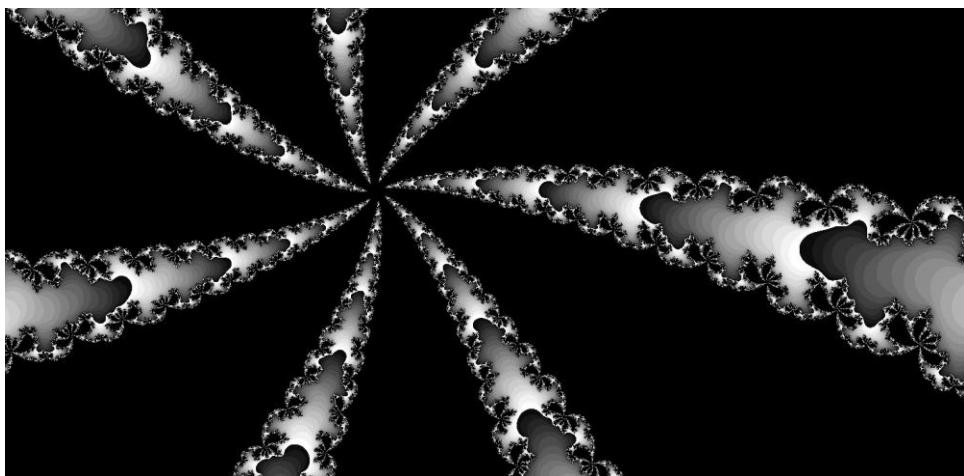


Figure 8.6. Connectivity, zoomed in.

With this particular image, it's easy to zoom into any portion of the image and prove to yourself conclusively that it's connected.

Now, let's take another value for c :

```
...  
cr = .038,  
ci = .655;
```

This gives you the image in Figure 8.7.

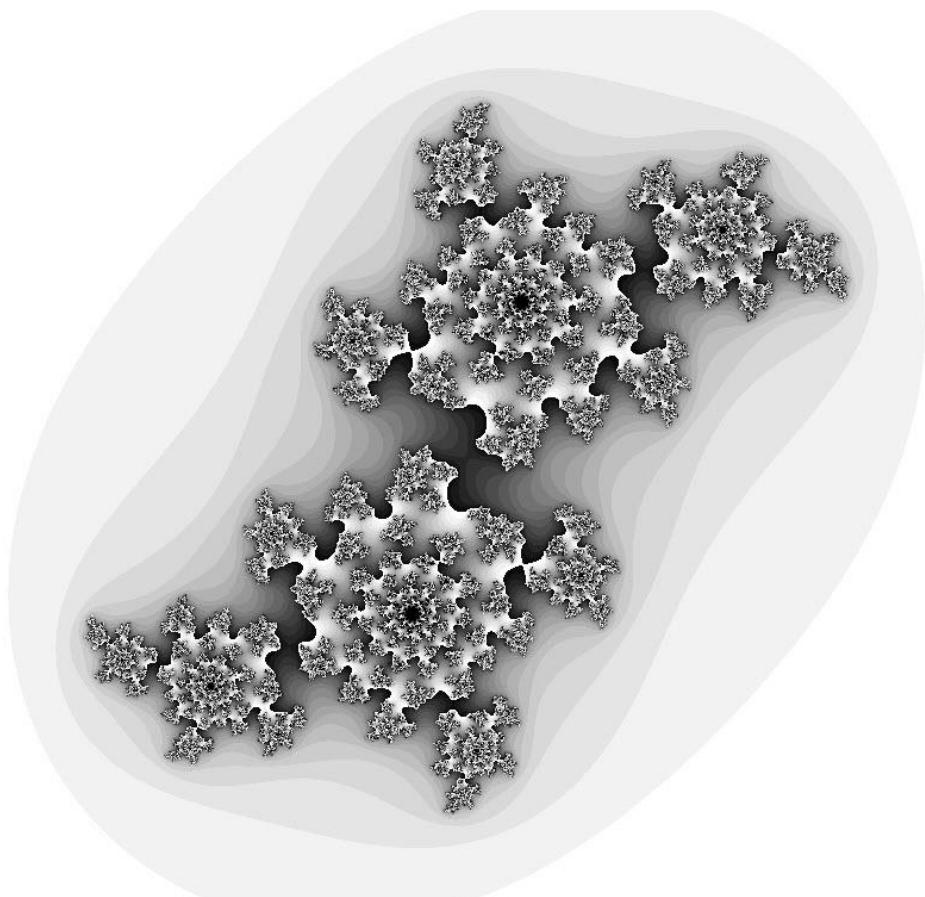


Figure 8.7. Another Julia fractal.

Do you think this set is connected? The main black areas are the circular areas in the beginning of each “island.” Zoom into one of those areas a little and you’ll get something like in Figure 8.8.

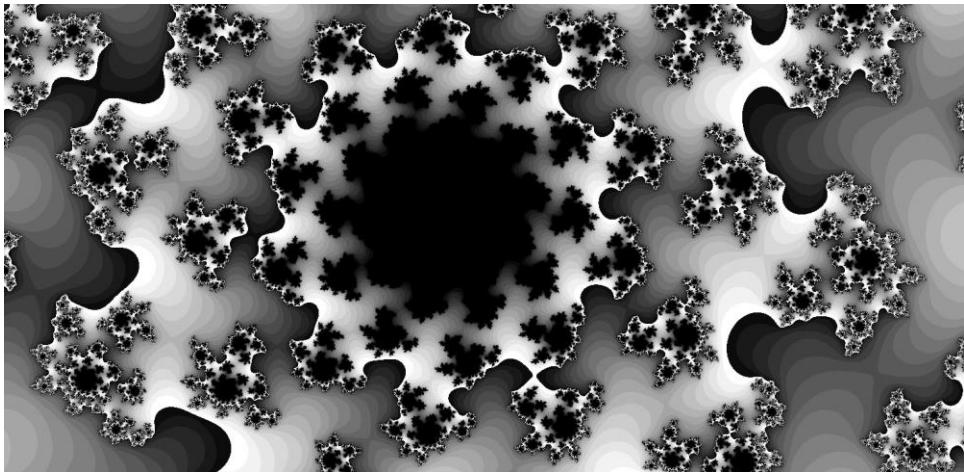


Figure 8.8. Not very connected.

Here you see very distinct and separated black blobs. No matter how you look or zoom into the areas between blobs, it's plain that they aren't connected. In fact, with a small bump in zoom, the blobs themselves fall apart into distinct smaller blobs, as you can see in Figure 8.9.

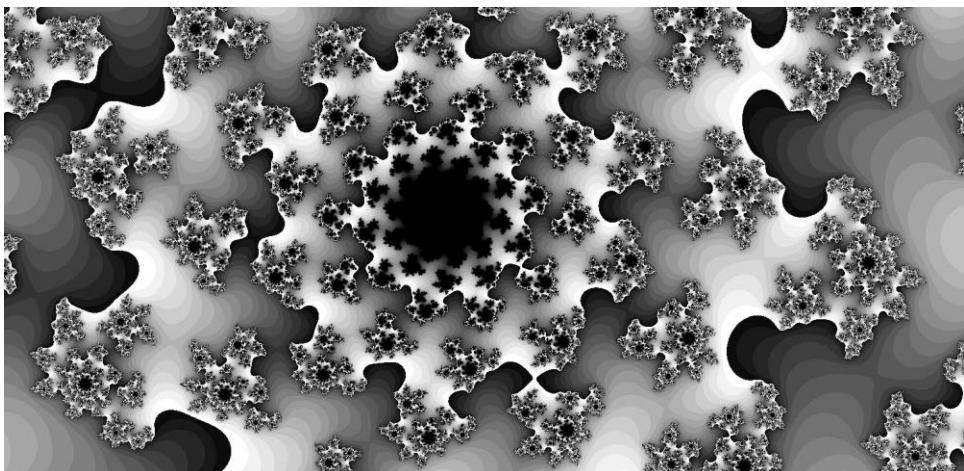


Figure 8.9. Definitely not connected.

Play around with this for a while until you are convinced that this set is not connected.

Now it's not very practical to explore Julia images by manually typing in complex numbers, saving the file and refreshing the browser over and over. So let's improve this program – and then discover an interesting tidbit about the connectivity of Julia sets and their relationship to the Mandelbrot set.

Interactive Julia Rendering

Okay, you need a user interface to enter a complex number, preferably not by typing it in. A mouse click would be nice. You've already created code that translates a screen position to a complex number in Chapter 7. Why not reuse that to allow a user to enter a complex number? And, while you're reusing Chapter 7 code, why not render the Mandelbrot set too? Then, you can see what kind of Julia fractal is generated based on where you click on or around the Mandelbrot set. If there are any relationships there, that will be a good way to see them.

The strategy will be to first render the Mandelbrot set as you did all through Chapter 7. But, once the image is clicked on, you'll convert that click location to a point on the complex plane and render a Julia image using that number as the constant. So you'll need both the `mandel` and `julia` functions, and a few changes to the mouse handlers. Here's the code in full for `julia2.js`:

```
window.onload = function() {
    var currentX = 0,
        imageData,
        stripWidth = 50,
        minR = -2,
        maxR = 1,
        minI = -1.4,
        maxI = 1.4,
        maxIter = 100,
        interval,
```

```
dr, di,
aspectRatio,
zoomDiv,
zoomX,
zoomY,
colors,
colorA = [255, 255, 255],
colorB = [0, 0, 0],
numColors = 20,
cr,
ci;

init();

function init() {

    chaos.init();

    initColors();
    zoomDiv = document.getElementById("zoom");
    aspectRatio = chaos.width / chaos.height;
    imageData = chaos.context.getImageData(0, 0,
                                            chaos.width, chaos.height);
    renderMandel();

    document.body.addEventListener("click",
                                  onClick);
}

function addKeyListeners() {
    document.body.addEventListener("keyup",
        function(event) {
            // console.log(event.keyCode);
            switch(event.keyCode) {
                case 80: // p
                    chaos.popImage();
                    break;

                case 38: // up
                    maxIter += 20;
                    clearInterval(interval);
                    renderFull();
                    break;

                case 40: // down
                    maxIter -= 20;
                    clearInterval(interval);
                    renderFull();
            }
        }
    );
}
```

```

        break;

    case 67: // c
        clearInterval(interval);
        randomizeColors();
        renderFull();
        break;

    case 90: // z
        clearInterval(interval);
        numColors -= 2;
        numColors = Math.max(numColors, 2);
        initColors();
        renderFull();
        break;

    case 88: // x
        clearInterval(interval);
        numColors += 2;
        initColors();
        renderFull();
        break;

    case 71: // g
        clearInterval(interval);
        grayScale();
        renderFull();
        break;

    default:
        break;
    }
});
```

}

```

function onClick(event) {
    cr = minR + event.clientX * dr,
    ci = minI + event.clientY * di,
    minR = -1;
    maxR = 1;
    renderFull();
    document.body.removeEventListener("click",
                                    onClick);
    document.body.addEventListener("mousedown",
                                 onMouseDown);
    addKeyListeners();
}
```

```

function onMouseDown(event) {
    clearInterval(interval);
    zoomX = event.clientX;
    zoomY = event.clientY;
    zoomDiv.style.left = zoomX + "px";
    zoomDiv.style.top = zoomY + "px";
    document.body.addEventListener("mousemove",
        onMouseMove);
    document.body.addEventListener("mouseup",
        onMouseUp);
}

function onMouseMove(event) {
    zoomDiv.style.width = event.clientX - zoomX
        + "px";
    zoomDiv.style.height = event.clientY - zoomY
        + "px";
}

function onMouseUp(event) {
    var x = event.clientX,
        y = event.clientY;

    document.body.removeEventListener("mousemove",
        onMouseMove);
    document.body.removeEventListener("mouseup",
        onMouseUp);

    zoomDiv.style.width = "0px";
    zoomDiv.style.height = "0px";
    if(x < zoomX || y < zoomY) {
        return;
    };

    maxR = minR + dr * x;
    maxI = minI + di * y;
    minR = minR + dr * zoomX;
    minI = minI + di * zoomY;
    renderFull();
}

function randomizeColors() {
    colorA = [Math.random() * 255,
              Math.random() * 255,
              Math.random() * 255];
    colorB = [Math.random() * 32,
              Math.random() * 32,
              Math.random() * 32];
    initColors();
}

```

```

}

function grayScale() {
    colorA = [255, 255, 255];
    colorB = [0, 0, 0];
    initColors();
}

function initColors() {
    var redRange = colorB[0] - colorA[0],
        greenRange = colorB[1] - colorA[1],
        blueRange = colorB[2] - colorA[2];

    colors = [];
    for(i = 0; i < numColors; i += 1) {
        colors.push({
            red: colorA[0] + Math.floor(redRange
                / numColors * i),
            green: colorA[1] + Math.floor(greenRange
                / numColors * i),
            blue: colorA[2] + Math.floor(blueRange
                / numColors * i)
        });
    }
}

function renderFull() {
    currentX = 0;
    adjustWidth();
    // one pixel's width on complex plane
    dr = (maxR - minR) / chaos.width;
    // one pixel's height on complex plane
    di = (maxI - minI) / chaos.height;

    interval = setInterval(renderStrip, 0);
}

function adjustWidth() {
    // width on complex plane
    var w = maxR - minR,
        // height on complex plane
        h = maxI - minI,
        // width with correct aspect ratio
        newW = h * aspectRatio,
        // difference to equal new width
}

```

```

diff = newW - w;

minR -= diff / 2;      // add half difference to left
maxR += diff / 2;      // and half to right
}

function renderStrip() {
  var x, y, h, color, index,
    w4 = chaos.width * 4,
    iData = imageData.data;

  // work across the strip horizontally
  for(x = currentX;
    x < currentX + stripWidth;
    x += 1) {
    index = x * 4;
    // render all the pixels in this vertical column
    for(y = 0, h = chaos.height; y < h; y += 1) {
      color = julia(x, y);
      iData[index] = color.red;
      iData[index + 1] = color.green;
      iData[index + 2] = color.blue;
      iData[index + 3] = 255;
      index += w4;
    }
    if(x > chaos.width) {
      clearInterval(interval);
      break;
    }
  }
  chaos.context.putImageData(imageData, 0, 0,
                            currentX, 0,
                            stripWidth,
                            chaos.height);
  currentX += stripWidth;
}

function julia(x, y) {
  var zr = minR + x * dr,
    zi = minI + y * di,
    iter, zr1, zi1;

  for(iter = 0; iter < maxIter; iter += 1) {
    zr1 = zr * zr - zi * zi + cr;
    zi1 = 2 * zr * zi + ci;
    zr = zr1;
    zi = zi1;
    if(zr * zr + zi * zi > 4) {

```

```

        return colors[iter % numColors];
    }
}

return {
    red: 0,
    green: 0,
    blue: 0
}
}

function renderMandel() {
    var x, y, w, h, color,
        index = 0,
        iData = imageData.data;

    adjustWidth();
    // one pixel's width on complex plane
    dr = (maxR - minR) / chaos.width;
    // one pixel's height on complex plane
    di = (maxI - minI) / chaos.height;

    for(y = 0, h = chaos.height; y < h; y += 1) {
        for(x = 0, w = chaos.width; x < w; x += 1) {
            color = mandel(x, y);
            iData[index] = color.red;
            iData[index + 1] = color.green;
            iData[index + 2] = color.blue;
            iData[index + 3] = 255;
            index += 4;
        }
    }
    console.log("done");
    chaos.context.putImageData(imageData, 0, 0);
}

function mandel(x, y) {
    var cr = minR + x * dr,
        ci = minI + y * di,
        zr = 0,
        zi = 0,
        i, zrl, zil;

    for(iter = 0; iter < maxIter; iter += 1) {
        zrl = zr * zr - zi * zi + cr;
        zil = 2 * zr * zi + ci;
        zr = zrl;
        zi = zil;
    }
}

```

```
        if(zr * zr + zi * zi > 4) {
            return colors[iter % numColors];
        }
    }

    return {
        red: 0,
        green: 0,
        blue: 0
    }
}

}
```

Now, let's look at what changed.

First, in the variable listing, there are no initial values set for `cr` and `ci`. They will get set when the user clicks.

Also, `minR` is set back to -2, the usual value for the Mandelbrot set.

Next, some changes to the `init` function. The handler for `mousedown` is removed and, instead, a handler for the `click` event is set. This prevents the user from zooming in on the Mandelbrot. You can change this if you want, but it would have added complexities that I'd rather avoid at this point.

The `init` function also gets rid of all the key-handling code. This is moved to another function called `addKeyListeners` for the same reason as the removal of the mouse-down handler.

The final change in this section is that, instead of calling `renderFull`, `init` calls another new function, `renderMandel`.

Then, you come to the `onClick` function.

```
function onClick(event) {
    cr = minR + event.clientX * dr,
    ci = minI + event.clientY * di,
    minR = -1;
```

```
maxR = 1;
renderFull();
document.body.removeEventListener("click",
                                  onClick);
document.body.addEventListener("mousedown",
                               onMouseDown);
addKeyListeners();
}
```

This sets the value for `cr` and `ci` in a way that you should be familiar with now. It resets `minR` and `maxR` to -1 and 1, good values for a Julia fractal, and calls `renderFull` to begin the rendering process. It removes the click handler, since it's done with that, and adds the mouse-down and key handlers so you can zoom, change colors, etc. once you're done rendering.

The last changes come at the end of the file with the new function, `renderMandel`, and the return of the familiar `mandel` function itself.

The `renderMandel` function is quite similar to `renderStrip` but renders the full Mandelbrot set in a single pass. Because it's only done a single time at program launch, while fully zoomed out, it's safe enough to render the set all at once.

So the flow is:

1. Program renders a non-interactive Mandelbrot set with `renderMandel`.
2. User clicks somewhere on the Mandelbrot set.
3. The `onClick` function converts the click to a number on the complex plane.
4. The `onClick` function calls `renderFull` to render the Julia fractal using the chosen complex number as a constant.

5. The `onClick` function enables full interactivity with the newly rendered fractal.

Now, you can click all over the place and render thousands of different Julia fractals.

Mandelbrot and Julia: The Connectedness Connection

Okay, now you've had a chance to render a whole bunch of new fractals. Have you noticed anything about where you click on the Mandelbrot set and the resulting Julia image?

To compare, I've marked a few points on the Mandelbrot set in Figure 8.10, along with Julias that were created by clicking in that general area.

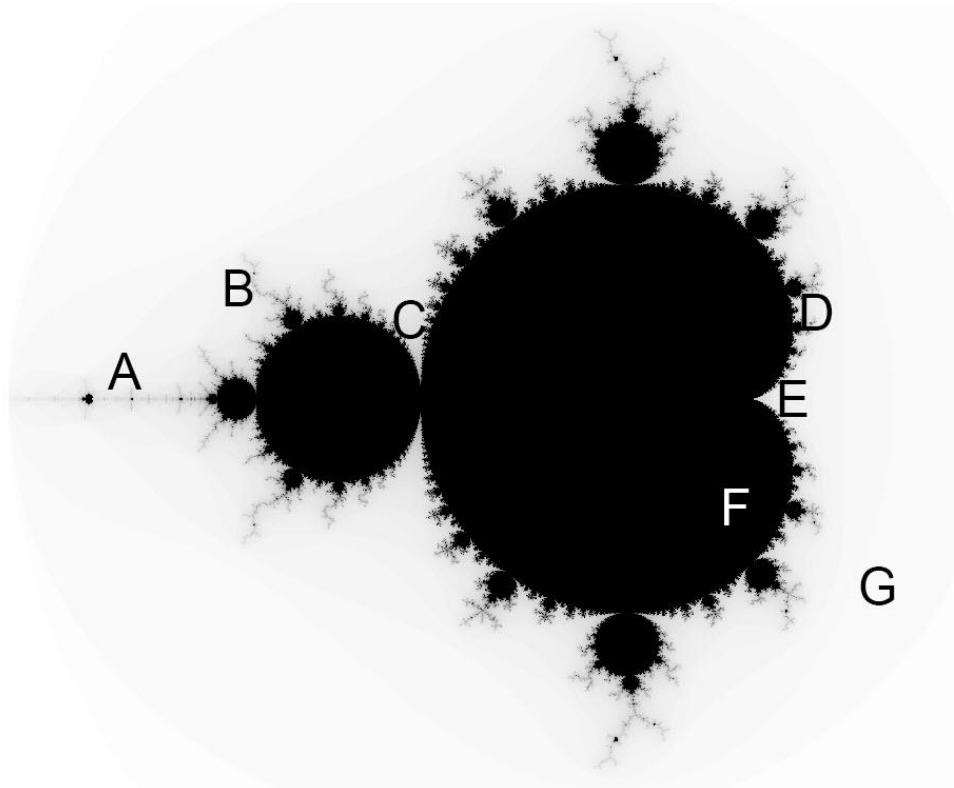


Figure 8.10. Clicking around in different areas.

Figure 8.11 comes from the area around point A, which is the long, narrow needle on the left. As you can see, this is a long, narrow fractal.

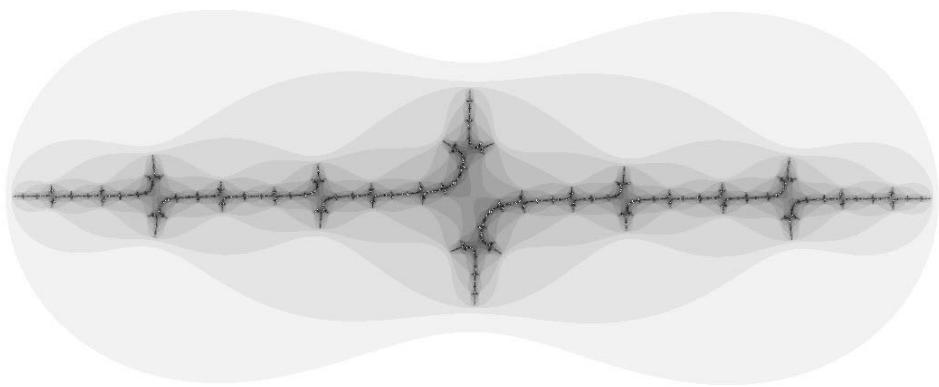


Figure 8.11. Area A.

Figure 8.12 comes from area B, one of the long, twisty tendrils. It has the same kind of form.

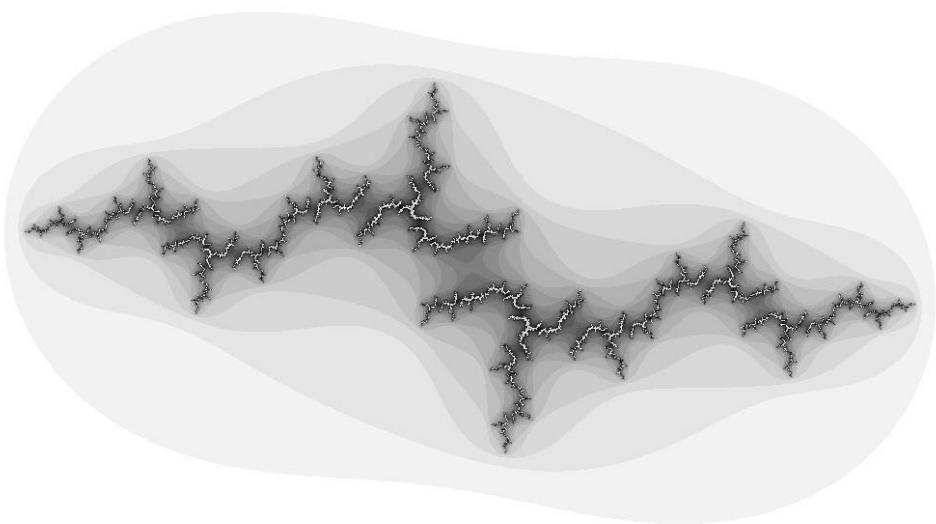


Figure 8.12. Area B.

Figure 8.13 is from area C. The resemblance here isn't so obvious, but you can probably see it.

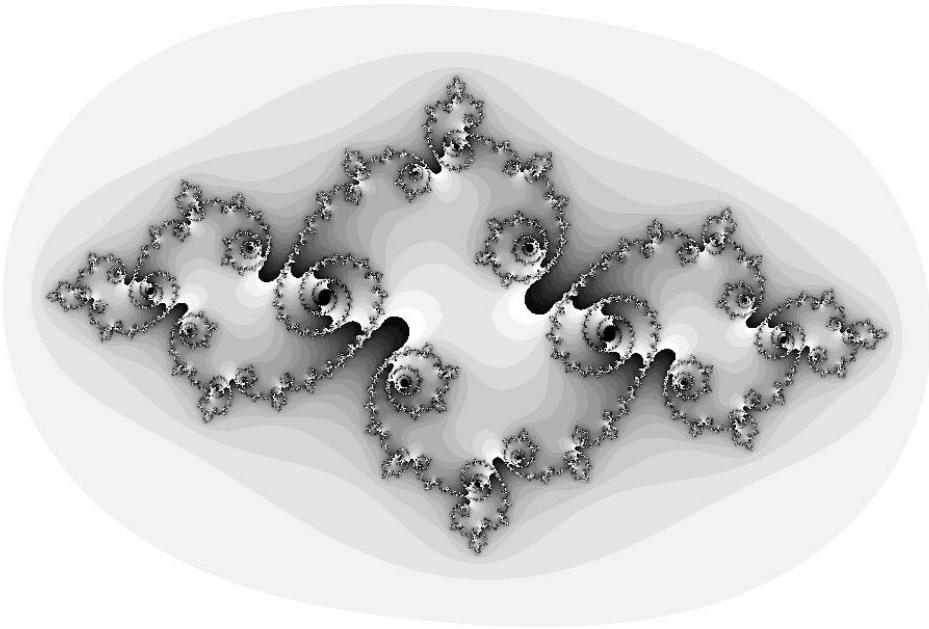


Figure 8.13. Area C.

If you zoom into area D, you will see shapes that look much like Figure 8.14, which comes from that area.

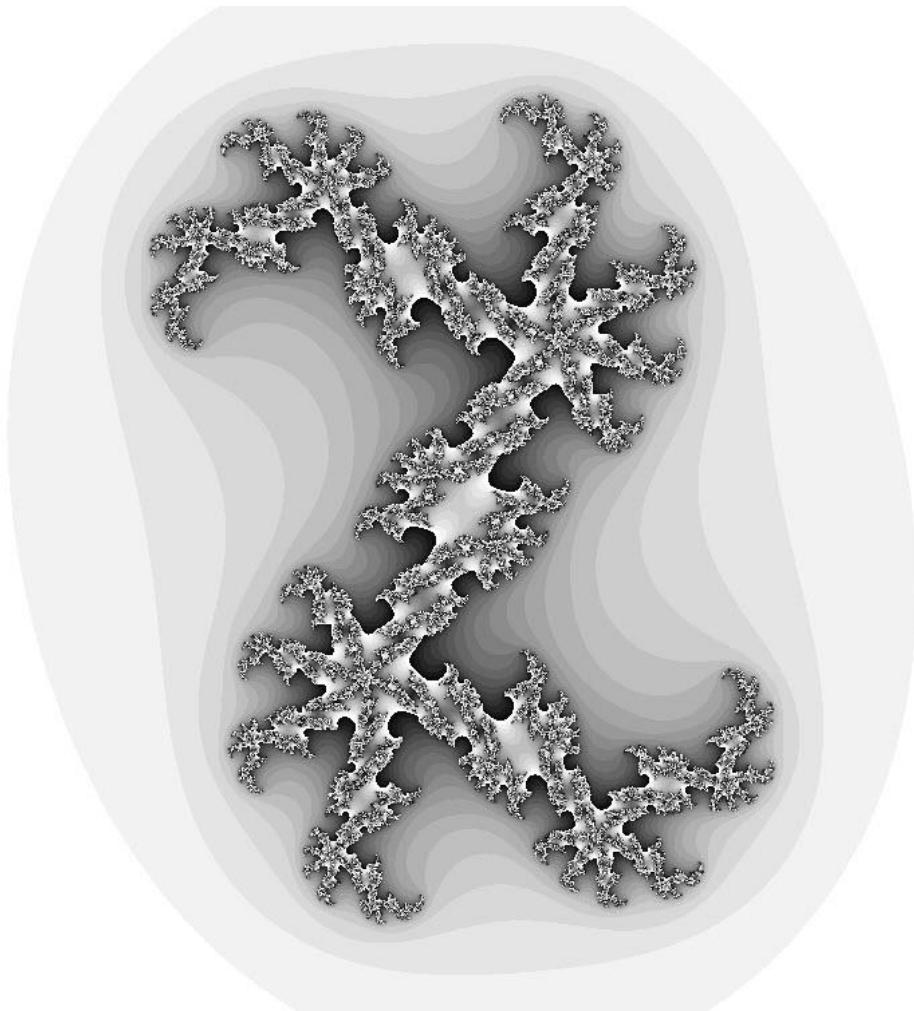


Figure 8.14. Area D.

Figure 8.15 does not show the greatest resemblance, but maybe you can see some similarity by zooming into area E.

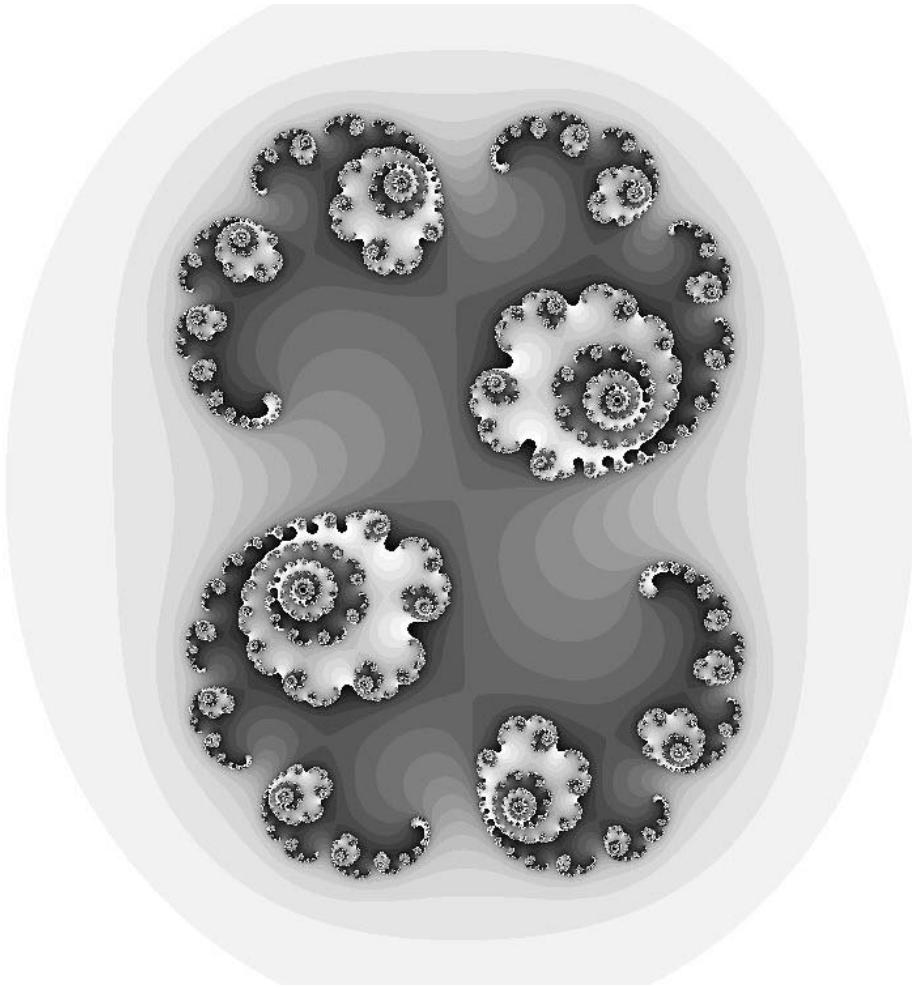


Figure 8.15. Area E.

With the above examples, you can see there is sometimes a great resemblance between the location in the Mandelbrot set and the resulting Julia fractal. Sometimes it's not so obvious, but it's hard to deny there is a relationship there. In fact, the Mandelbrot set is often called a map or index of the different possible Julia fractals.

Now, points A through E were all chosen from around the border of the Mandelbrot set. But Figure 8.16 comes from

well within the set in area F. This is very obviously a connected Julia.

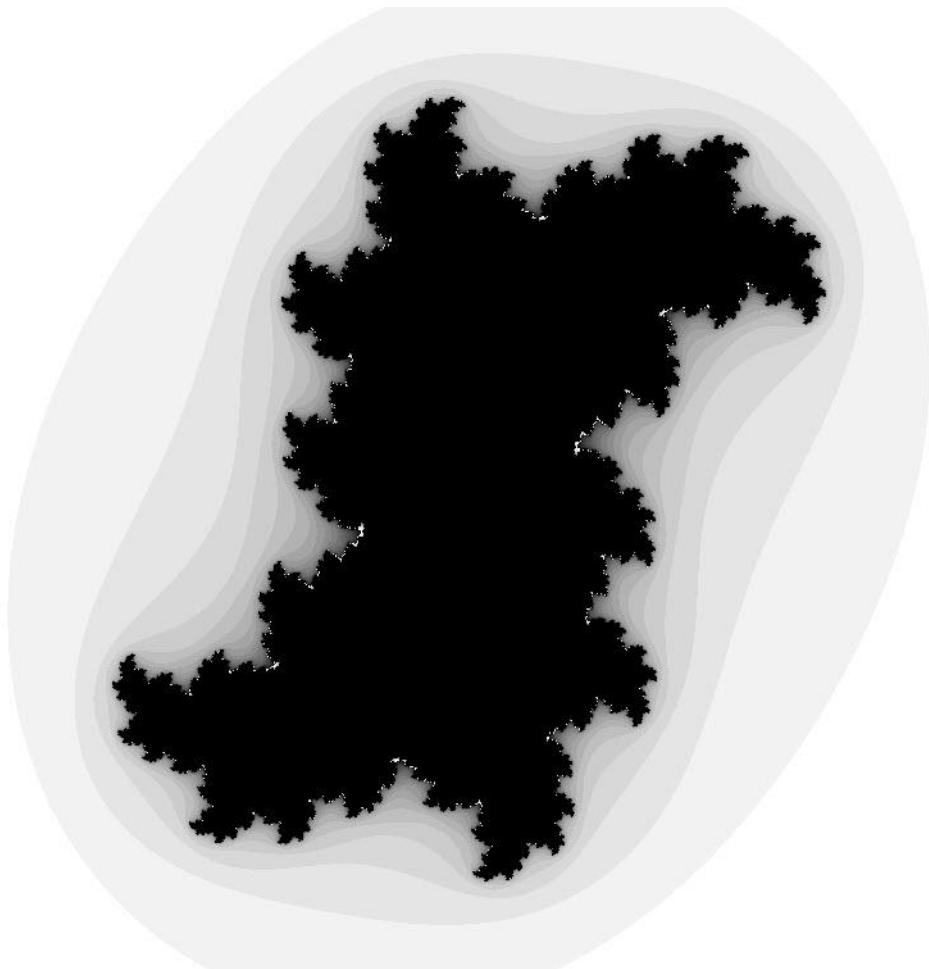


Figure 8.16. Area F.

Figure 8.17 comes from area G, well outside the Mandelbrot set. It can easily be identified as disconnected. In fact, you can call it a Fatou dust.

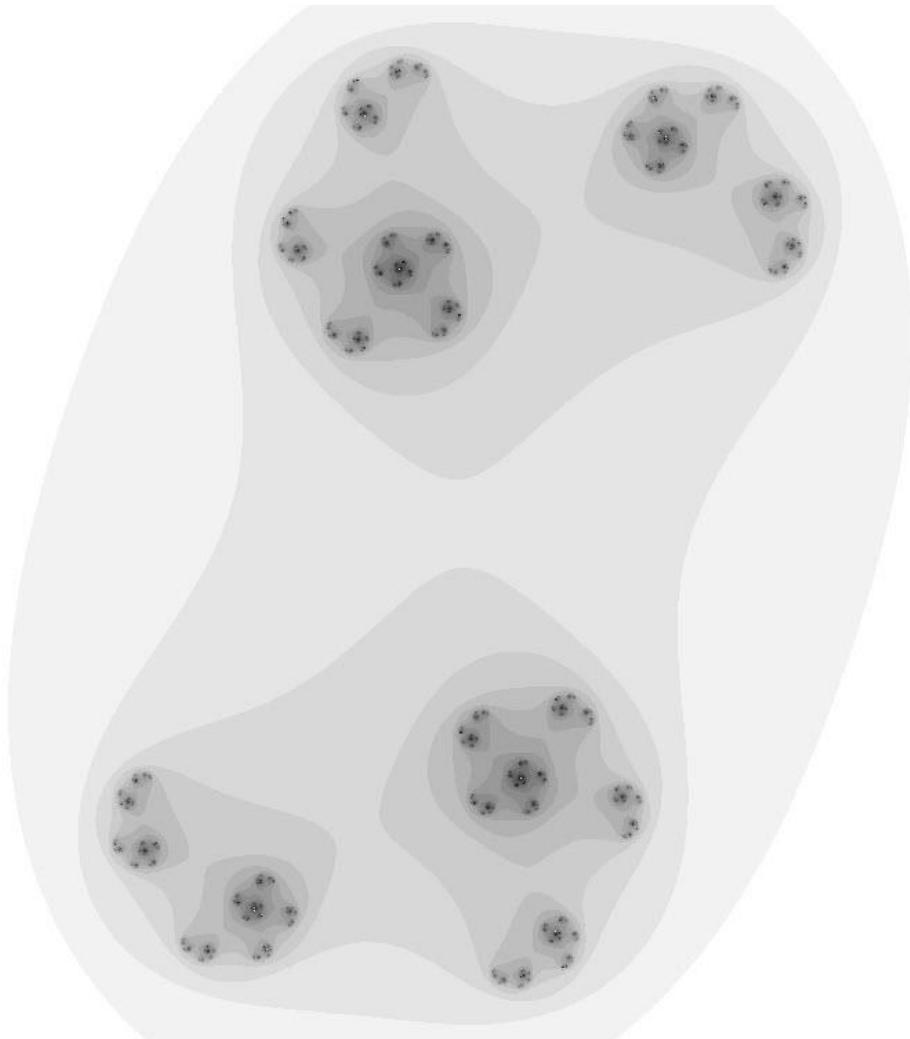


Figure 8.17. Area G.

This correlation will hold true no matter where you click. Clicking within the Mandelbrot set will give you connected Julias. Clicking outside of it will give you disconnected ones. And now you can see even more deeply the relationship between these three sets.

Summary

I hope this was a fun, shorter chapter for you. Not a whole lot of brand new code to wrap your head around, but some new, hopefully surprising concepts. And of course, a new way to make intriguing images with code!

In Chapter 9, I'll be leading you on a new adventure as you learn about chaos theory and strange attractors.

Chapter 9: Chaos Theory and Strange Attractors

In this chapter, I'm going to discuss the concepts of chaos, attractors and a special kind of attractor called the strange attractor. The connections between these subjects and fractals may not be obvious at first, but by the end of the chapter I think you'll see the connections. Without further ado, let's dive into chaos.

Chaos Theory

In everyday language, the word “chaos” simply means lack of order or complete confusion. In the subject of chaos theory however, the term takes on a bit of a different nuance. To sum up chaos theory in one line, I would say:

VERY SMALL CHANGES IN INPUT RESULT IN DRASTICALLY LARGE CHANGES IN OUTPUT.

Say you have a system consisting of one or more parts that can interact. You know the initial state of the system, and there are specific rules that determine how that state will change over time. This is a deterministic system. Given the exact same starting state, it will always wind up in the exact same state after a certain amount of time.

An example is an object operating in a gravitational field. If you know the initial position and velocity of the object, and you know the rules of gravity, you can predict the exact path of that object and exactly where it will land.

However, some deterministic systems can become chaotic. Particularly systems that have some kind of feedback loop or that have multiple interacting objects, where one object's state affects the other object's state and vice versa. These don't have to be very complex systems either. Take a double pendulum – a classic chaotic example. A single pendulum is deterministic and easily predictable. So predictable that they have been used to keep accurate time for centuries. When you know the starting position and length of a pendulum and the force of gravity, you can predict very accurately what it's going to do thereafter: how far it's going to swing, how long each swing will take, etc.

But hang another pendulum on the bottom of the first pendulum and it becomes nearly impossible to predict what's going to happen. The bottom pendulum influences the motion of the top one and vice versa. It's still a deterministic system. There's no artificial randomness or magic involved. It's just too complex to predict what state it's going to be in after more than a few swings.

Initially, you might think, well, yes, it's a very complex motion, but since it is deterministic, you COULD predict it. You just need a more powerful computer to analyze the data. Theoretically, yes, that's correct. But, as I'll show you before the end of the next section, unpredictability is inescapable in the real world. It doesn't matter how powerful a computer you have, the effects of chaos will always come back to bite you.

Population Models

A popular method of demonstrating chaos theory is the population model.

Say you have a population of animals in a closed environment. These simulations were initially used for fish in a body of water, but let's use deer in a forest. Each year, some of the deer families will have little fawns, increasing the population. And let's say that the rate at which they reproduce is well-documented. It's a percentage thing – the more deer there are, the more babies will be born. The population will increase by some known percent.

Also, each year some of the older or weaker deer will die off. Because this forest is a finite space, the death rate is proportional to the population. If the population is relatively low, the death rate will be relatively low. However, when the forest becomes overcrowded, there will not be enough food and more deer will die. There is a theoretical maximum population of deer that the forest can support, and this is a known value.

Given those two factors – reproduction and death rate – and assuming the rules I just gave you are exact and unerring, you have a deterministic system. If you know how many deer there are at a given point in time, you can directly calculate how many deer will be born and how many deer will die in the next year. And you should be able to predict some years into the future how many deer will be in the forest, right?

Let's figure out the formulas and see.

Call the population in a given year p and the next year p_1 . Rather than give discrete numbers, let's say that the

maximum number of deer the forest can handle is 1. A population of 0 would be extinction. So p would be somewhere between 0 and 1.

The rate at which the deer reproduce will be r . This will be a decimal number above 1. An r of 1.1 means that the population will increase by 10% each year. A value of 2 means that the population would double each year. The equation is:

$$p_1 = p * r * (1 - p)$$

The $p * r$ part is the birth rate and is easy enough to understand. The $(1 - p)$ part is the death rate. As an example, say the population after calculating reproduction was 0.1. The death rate would be $1 - 0.1$ or 0.9. It might be better to call this the survival rate because this means that 90% of the deer will survive and the new population will be 0.09. But say the population was higher, like 0.6. Then, only 40% of the deer would survive and the new population would be 0.24. If the population somehow reached that theoretical maximum, 1.0, then this factor would be 0 and result in total deer annihilation. That's why it's a theoretical maximum. In the real world, this would not happen.

Now, rather than work this all out manually with columns of numbers, let's create a graph to see it visually. The code is in the file `population1.js`.

```
window.onload = function() {
    var p = 0.25,
        r = 1.5,
        year = 0,
        interval,
        xRes = 20;

    init();

    function init() {
```

```

chaos.init();

interval = setInterval(oneYear, 0);
document.body.addEventListener("keyup",
    function(event) {
        switch(event.keyCode) {
            case 80: // p
                chaos.popImage();
                break;

            default:
                break;
        }
    });
}

function oneYear() {
    var oldX,
        newX,
        oldY,
        newY;

    oldX = year * xRes;
    oldY = chaos.height - p * chaos.height,

    p = p * r * (1 - p);
    year += 1;

    newX = year * xRes;
    newY = chaos.height - p * chaos.height;

    chaos.context.beginPath();
    chaos.context.moveTo(oldX, oldY);
    chaos.context.lineTo(newX, newY);
    chaos.context.stroke();
    console.log(p);

    if(newX >= chaos.width) {
        clearInterval(interval);
    }
}
}

```

First, let's look at the variables. The starting population is represented by `p` at .25. The reproduction rate `r` is 1.5 and you'll start at year 0.

The `init` function simply starts an interval that will call the `oneYear` function as quickly as possible. That's where the action happens.

First, this function calculates an `oldX` and `oldY` to find the current population on the graph. It then increments the year by one and calculates a `newX` and `newY` and finishes up by drawing a line between the two. When it reaches the edge of the canvas, it stops the interval from running. Note that you can change `xRes` to alter the x scale of the graph, stretching it out or compacting it on the x-axis. For now, 20 is a good value for `xRes`, as it will show you year-to-year changes clearly.

So all this program really does is take the population and graph it out over the next bunch of years. Run the program and you'll get a graph like in Figure 9.1.

Figure 9.1. Population graph: $r = 1.5$.

From this, you can see that the relatively low death rate, along with a reproductive rate of 1.5 is going to allow the population to increase for the first few years. The population will reach an equilibrium at about .333 and stay there.

Change `r` to 2 and you'll see that the population rises a little faster and levels off at .5. For `r` values between 1 and 2, you can predict the final stable population with the equation $(r - 1) / r$.

Now, try 2.5 for `r` and you'll see something a bit interesting. The population rises quickly, reaches a peak and then falls back down. It rises again, though not as far, and then goes

up and down several times before finally settling on .6. See Figure 9.2.



Figure 9.2. Population graph: $r = 2.5$.

As you continue to increase the value for r , this settling down period takes longer and longer. Figure 9.3 shows what happens when r is at 2.9.

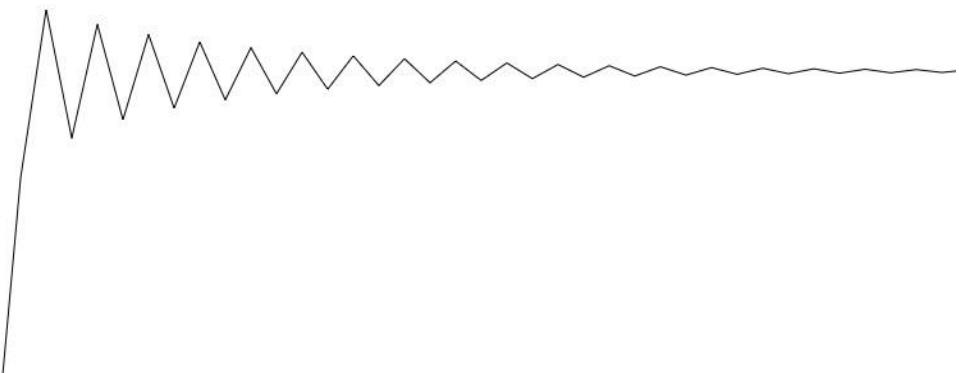


Figure 9.3. Population graph: $r = 2.9$.

Now, a strange thing happens when you go just above 3.0. The graph NEVER settles down. Well, that is, it never settles down to a SINGLE value. As you can see in Figure 9.4, which has a value of 3.1 for r , the graph is now stably oscillating between the two values .764 and .558. Those two values will never converge. The graph will go back and forth between them forever. Going back to the deer forest analogy, the hunters in that area would probably soon

discover that one year there would be lots of deer and the next year not so many.

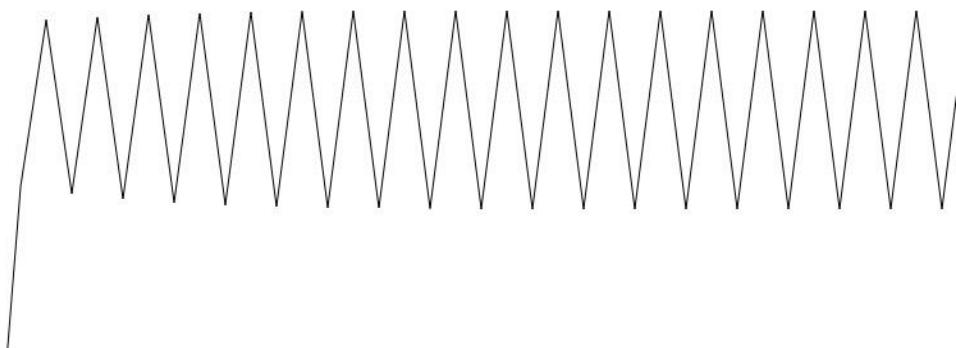


Figure 9.4. Population graph: $r = 3.1$.

As you continue to increase r slightly, eventually you'll see another interesting occurrence. Change r to 3.5 and you'll see the graph in Figure 9.5.

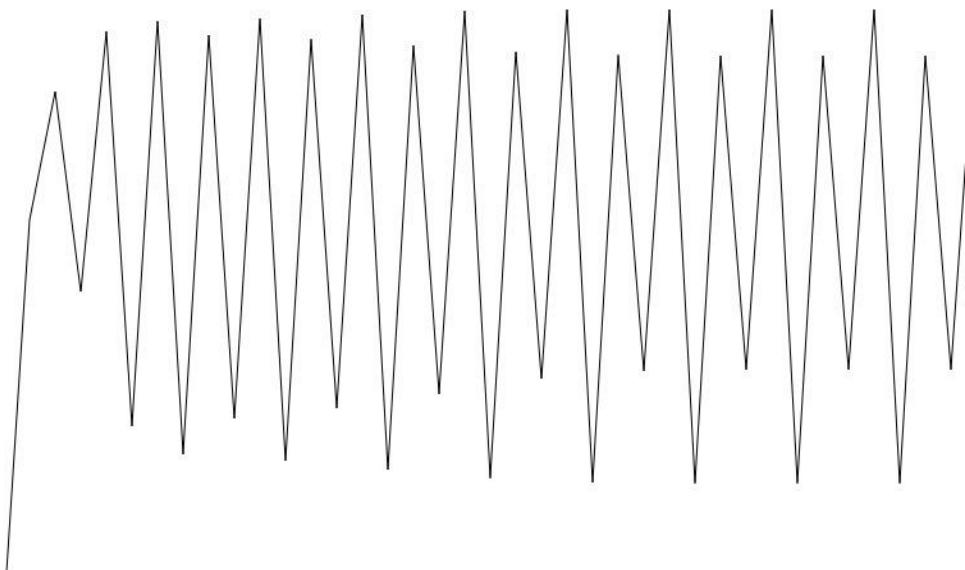


Figure 9.5. Population graph: $r = 3.5$.

Now, the graph has settled on four points. The population is now on a four-year cycle. If you increase r slowly and

carefully, you should be able to find a point where it splits into eight and maybe even 16 stable points.

The next interesting change comes when r goes just over 3.57. At that point, there is no more pattern. The graph has entered a chaotic state. Figure 9.6 shows an r of 3.58. The x_{Res} was also reduced to 5 to show more years on the graph.

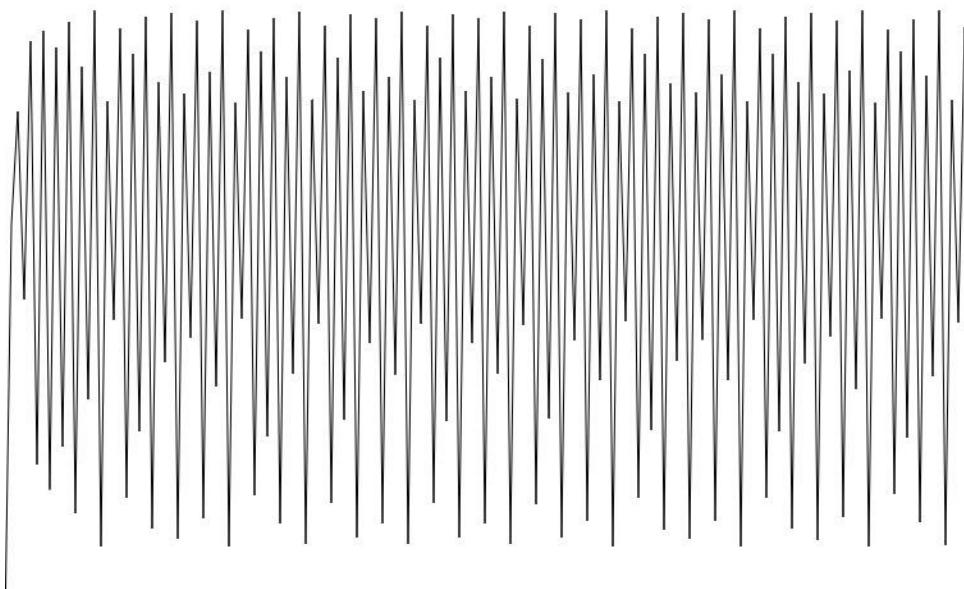


Figure 9.6. Population graph: $r = 3.58$.

There is no pattern here. At first, you think you might see one, but when you look closer, it doesn't repeat exactly. You may think it might repeat eventually. But it won't. It is chaotic.

At this point, there is no real way to predict the population "n" years into the future, other than calculating it out year by year. This is one factor in chaos, but there's another important factor mentioned in the beginning of the chapter:

VERY SMALL CHANGES IN INPUT RESULT IN DRASTICALLY LARGE CHANGES IN OUTPUT.

Let's talk about deer again. I just said there'd be no way to predict the deer population several years into the future except by calculating it out. Okay, so you go into the forest, you count all the deer and you do your calculations. You should be all set, right? If you missed a deer or two, well, you should still be pretty good, right? Not according to chaos theory.

You have your population value set at .25 right now. Say that represents 25,000 deer. It's a big forest. You count them and you're off by a single, small deer. You've counted 25,001. Let's see what happens.

The file `population2.js` simulates this scenario:

```
window.onload = function() {
    var p = 0.25,
        r = 3.7,
        year = 0,
        interval,
        xRes = 2;

    init();

    function init() {

        chaos.init();

        while(year < chaos.width / xRes) {
            oneYear();
        }
        chaos.context.translate(0, chaos.height / 2);
        p = 0.25001;
        year = 0;
        while(year < chaos.width / xRes) {
            oneYear();
        }
        document.body.addEventListener("keyup",
            function(event) {
                switch(event.keyCode) {
```

```

        case 80: // p
            chaos.popImage();
            break;

        default:
            break;
    }
});
```

}

```

function oneYear() {
    var oldX,
        newX,
        oldY,
        newY;

    oldX = year * xRes;
    oldY = chaos.height / 2 - p * chaos.height / 2,

    p = p * r * (1 - p);
    year += 1;

    newX = year * xRes;
    newY = chaos.height / 2 - p * chaos.height / 2;

    chaos.context.beginPath();
    chaos.context.moveTo(oldX, oldY);
    chaos.context.lineTo(newX, newY);
    chaos.context.stroke();
    console.log(p);

    if(newX >= chaos.width) {
        clearInterval(interval);
        console.log("done");
    }
}
```

}

I'm not going to go into the code very deeply here. It's largely the same as the earlier version. The key lines are here:

```

while(year < chaos.width / xRes) {
    oneYear()
}
chaos.context.translate(0, chaos.height / 2);
```

```
p = 0.25001;  
year = 0;  
while(year < chaos.width / xRes) {  
    oneYear()  
}
```

First, calculate the population into the future with an initial `p` of 0.25 and graph it. Then, do it all again with `p` equal to 0.25001. You can see how the two graphs relate in Figure 9.7.

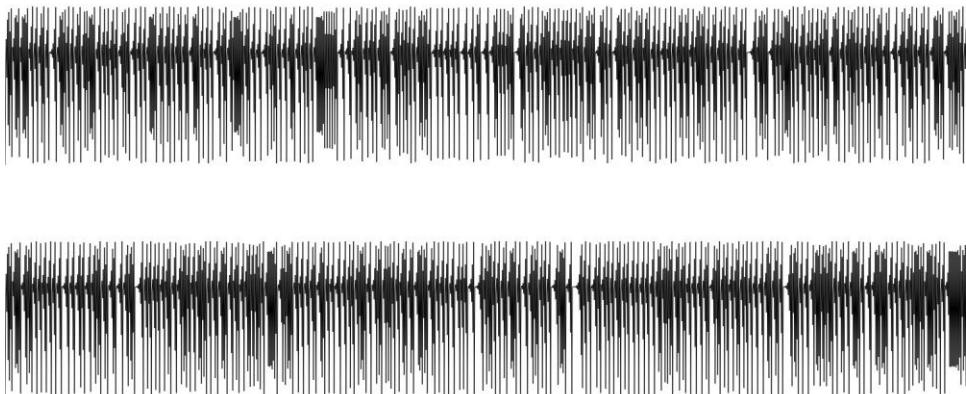


Figure 9.7. Small changes in input.

I've shortened the `xRes` to 1 to show as many years as possible. For a short while, the two graphs are tracking together. But, before too long, they diverge. And, after that, they are very different graphs. For a miscount of a single deer. I think anyone would agree that .00001 is a very small change in input. And you can easily see that the output is drastically different. But even a much smaller change could eventually wind up with just as much unpredictable output.

The Butterfly Effect

This concept of small changes in input creating drastic changes in output is often called the “butterfly effect”, which takes its name from a lecture given by Philip Merilees in 1972: “Does the flap of a butterfly’s wings in Brazil set off a tornado in Texas?” Similar sayings had gone around before, but the butterfly stuck.

The idea is what the last code example just demonstrated – that a small change, such as the flapping of a butterfly’s wings, can have enormous consequences down the line. At least in chaotic systems.

The inclusion of a tornado in the title of the lecture is apt, as weather is a great example of a chaotic system. At one point, people believed that our inability to predict the weather more than a couple days into the future would eventually be solved by highly accurate sensors and powerful computers – and larger numbers of both. However, here we are in the 21st century, with a massive network of amazingly powerful computers, and really in no better shape when it comes to predicting the weather. The truth is, due to the chaotic nature of weather, you could blanket the earth in highly accurate sensors and have the most powerful computers available churning through the data and making predictions, and the results still wouldn’t be perfect. Weather iterates every fraction of a second, and a mis-measurement of a fraction of a percent would throw things off rather quickly.

Logistic Map and Bifurcation Diagram

You might be thinking that this is all rather interesting, but what is it doing in a book about fractals? Fair question. I think this section will answer that question.

Let's look again at that population formula from the previous section:

$$p_1 = p * r * (1 - p)$$

This equation is also known as the LOGISTIC MAP, and it has applications and interest beyond predicting animal populations. An interesting way of viewing the logistic map is via a BIFURCATION DIAGRAM. This is a way of viewing the changes that the system goes through as the input changes. The word “bifurcation” means “splitting in two.” As you’ve seen, as r increased, the stable population went from a single value to two values, four, eight and shortly after into chaos. The bifurcation diagram will show you these changes very clearly.

The strategy is to start with a low value for r and run the logistic map equation for many iterations. You’ll ignore the first 100 or so iterations to allow the output values to settle down and then plot the remainder on a graph. You then move onto a slightly higher value for r and repeat the process. The values for r will be plotted along the x-axis and output values on the y-axis. Starting with an r of 2 and moving to about 3.3, you get the graph you see in Figure 9.8.

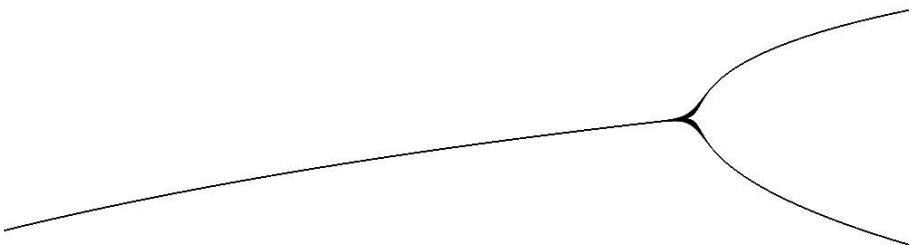


Figure 9.8. The first bifurcation.

You can see that, as r increases, the stable population slowly increases as well. When r passes 3, the population bifurcates into two alternating, stable values. This coincides with what you saw in Figure 9.4.

The code for this example is in the file `bifurcation1.js`:

```
window.onload = function() {
    var p = 0.5,
        minR = 2,
        maxR = 4,
        minP = 0,
        maxP = 1,
        dr, dp,
        interval,
        x;

    init();

    function init() {
```

```

chaos.init();

renderFull();

document.body.addEventListener("keyup",
    function(event) {
        switch(event.keyCode) {
            case 80: // p
                chaos.popImage();
                break;

            default:
                break;
        }
    });
}

function renderFull() {
    chaos.clear();
    x = 0;
    // one pixel's width on the r axis
    dr = (maxR - minR) / chaos.width;
    // one pixel's height on the p axis
    dp = (maxP - minP) / chaos.height;
    interval = setInterval(iterate, 0);
}

function iterate() {
    p = .5;
    for(var i = 0; i < 200; i += 1) {
        oneYear(i);
    }
    x += 1;
    if(x >= chaos.width) {
        clearInterval(interval);
    }
}

function oneYear(year) {
    var r = minR + x * dr,
        y;
    p = p * r * (1 - p);
    if(year > 100) {
        y = chaos.height - (p - minP) / dp;
        chaos.context.fillRect(x, y, 1, 1);
    }
}

```

}

The code is very similar to the Mandelbrot examples from Chapter 7. First, the variables:

```
var p = 0.5,  
    minR = 2,  
    maxR = 4,  
    minP = 0,  
    maxP = 1,  
    dr, dp,  
    interval,  
    x;
```

The `p` is population. Then, you have values for minimum and maximum `r` values and `p` values for graphing.

Again, `r` will be on the x-axis and `p` on the y-axis.

The `dr` and `dp` variables are deltas indicating the size of a single pixel on each axis.

The `renderFull` function clears the canvas, sets `x` to 0, calculates `dr` and `dp`, and starts the interval that will call the `iterate` function.

The `iterate` function sets `p` back to .5 each time, and iterates the `oneYear` function 200 times. It then increments `x` and checks to see if it's gotten to the end of the canvas.

The `oneYear` function is much the same as you saw in the population examples. Here, though, it calculates `r` on the fly based on the value of `x`. It skips the first 100 years to allow the population to settle down, and then plots a single pixel for each population value it finds for the rest of the iterations.

When you run this program and let it finish, you'll have Figure 9.9, the bifurcation diagram for the logistic map.

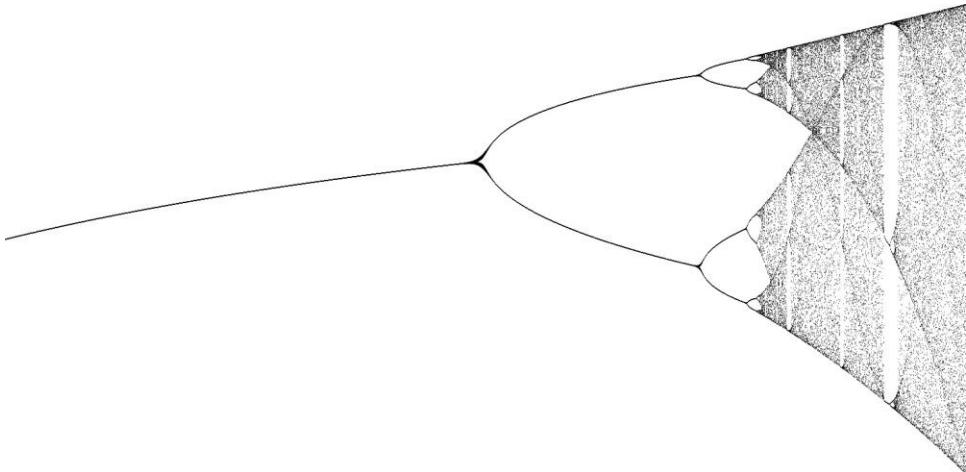


Figure 9.9. Bifurcation diagram.

This is a classic figure that you'll see in any text on chaos theory. You can plainly see where the values split into two, four and eight values. Soon thereafter, the diagram goes into chaos. But you'll also see that there are several vertical strips in the right side of the diagram where there are brief windows of order within the chaos. In the largest of these windows, you can see that the population is converging on three values. And, if you are sharp-eyed, you should be able to see those three bifurcate into six before diving back into a chaotic realm.

In addition, you should be able to see the self-similar, fractal nature of this diagram. It even looks a bit like the fractal tree you created earlier in the book. Perhaps you're thinking this would be interesting to zoom in on. Perhaps that is exactly what the code in `bifurcation2.js` allows you to do:

```
window.onload = function() {  
    var p = 0.5,  
        minR = 2,  
        maxR = 4,  
        minP = 0,  
        maxP = 1,  
        dr, dp,  
        x,
```

```
interval,
zoomDiv,
zoomX,
zoomY,
maxIter = 200;

init();

function init() {
    chaos.init();

    zoomDiv = document.getElementById("zoom");
    renderFull();

    document.body.addEventListener("mousedown",
                                  onMouseDown);

    document.body.addEventListener("keyup",
        function(event) {
            switch(event.keyCode) {
                case 80: // p
                    chaos.popImage();
                    break;

                case 38: // up
                    clearInterval(interval);
                    maxIter += 500;
                    renderFull();
                    break;

                case 40: // down
                    clearInterval(interval);
                    maxIter += 100;
                    renderFull();
                    break;

                default:
                    break;
            }
        });
}

function renderFull() {
    chaos.clear();
    x = 0;
```

```

// one pixel's width on complex plane
dr = (maxR - minR) / chaos.width;
// one pixel's height on complex plane
dp = (maxP - minP) / chaos.height;
interval = setInterval(iterate, 0);
}

function iterate() {
p = .5;
for(var i = 0; i < maxIter; i += 1) {
    oneYear(i);
}
x += 1;
if(x >= chaos.width) {
    clearInterval(interval);
}
}

function oneYear(year) {
var r = minR + x * dr,
    y;
p = p * r * (1 - p);
if(year > 100) {
    y = chaos.height - (p - minP) / dp;
    chaos.context.fillRect(x, y, 1, 1);
}
}

function onMouseDown(event) {
    clearInterval(interval);
    zoomX = event.clientX;
    zoomY = event.clientY;
    zoomDiv.style.left = zoomX + "px";
    zoomDiv.style.top = zoomY + "px";
    document.body.addEventListener("mousemove",
                                    onMouseMove);
    document.body.addEventListener("mouseup",
                                    onMouseUp);
}

function onMouseMove(event) {
    zoomDiv.style.width = event.clientX - zoomX
                        + "px";
    zoomDiv.style.height = event.clientY - zoomY
                        + "px";
}

function onMouseUp(event) {

```

```

var x = event.clientX,
    y = event.clientY;

document.body.removeEventListener("mousemove",
                                  onMouseMove);
document.body.removeEventListener("mouseup",
                                  onMouseUp);

zoomDiv.style.width = "0px";
zoomDiv.style.height = "0px";
if(x < zoomX || y < zoomY) {
    return;
}

maxR = minR + dr * x;
maxP = minP + dp * (chaos.height - zoomY);
minR = minR + dr * zoomX;
minP = minP + dp * (chaos.height - y);

console.log(minP, maxP);
console.log(minR, maxR);
renderFull();
}

}

```

I won't go into this code in very much detail. It is all directly adapted from the Mandelbrot zooming code you saw extensively in Chapters 7 and 8. You click and drag to create a zoom rectangle. (Note that you'll also need the changes that are in `bifurcation2.html` to enable this zoom rectangle.) This sets new values for `minR`, `maxR`, `minP` and `maxP`. Then, a new call to `renderFull` uses those new values. Also, the up and down cursor keys change the `maxIter` value, which causes each value of `r` to be iterated more or less. When zooming in, you'll probably want to increase the maximum iterations to view more detail. One thing I left out was the `adjustWidth` function, which means that the shape of the rectangle you draw to zoom will alter the aspect ratio of the next render. I see this as more of a feature in this case; it allows for separate zooming on each axis.

Now, you can really dig into this diagram and fully appreciate its fractal nature. As you zoom in, you'll see that there are actually very many of those "windows of order" in the chaos, each one of them having several stable values that each begin bifurcating, creating a copy of the whole diagram. Figure 9.10 is an example.

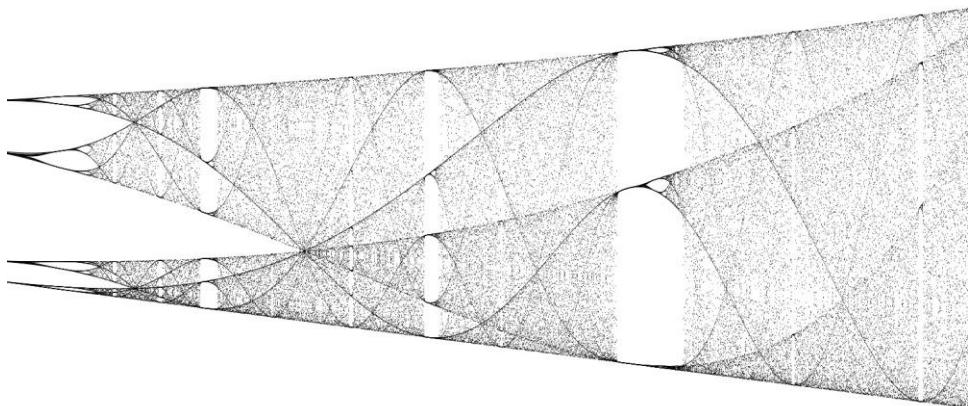


Figure 9.10. Bifurcation diagram, zoomed in.

Using this new zoom feature, you also exactly isolate the point where chaos begins. It happens just about when x reaches 3.56995, known as the "Feigenbaum point" and named after Mitchell J. Feigenbaum, a mathematical physicist specializing in chaos theory. If you zoom into those areas right after the first few bifurcations, you'll see something like in Figure 9.11.

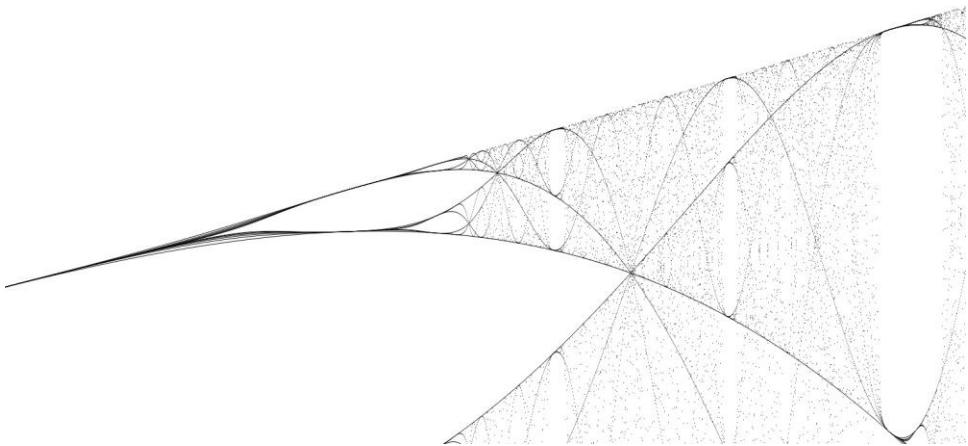


Figure 9.11. The emergence of chaos.

At this point, the graph ceases simply to split into separate branches. The branches cross each other, and the whole diagram goes into a wild oscillation pattern.

The bifurcation map doesn't have quite the diversity or beauty of Mandelbrot or Julia fractals. But, personally, I still find myself lost exploring it for long periods of time. I'm always amazed how such a simple deterministic equation can result in such beautiful complexity. And the interplay of chaos and order is fascinating.

Next up, I'll expand on these same ideas as I introduce you to the world of strange attractors.

Strange Attractors

I guess any discussion of strange attractors should begin with a definition of an attractor, and follow up by explaining the difference between a “normal” attractor and a strange one.

An attractor is simply a value that an iterated function with certain input will tend to have as an output. The output will be “attracted” to that value. You’ve already seen plenty of examples of this in the first part of this chapter. Using the logistic map equation, a population with a reproductive rate r of 1.5 will settle on .333... So you can say that value is the attractor for that equation with that given input.

An equation can, of course, have more than one value as an attractor, as you’ve also seen already. You wouldn’t say that the equation has two attractors but that it has an attractor of period 2, or a period-2 attractor. In other words, an attractor can be a set of values rather than just a single number. In fact, an attractor can be a multi-dimensional set of values.

David Ruelle and Floris Takens, a pair of mathematicians studying non-linear equations relating to fluid dynamics, discovered that attractors could sometimes form fractal patterns. They coined the term “strange attractor” to describe this phenomenon.

The Lorenz Attractor

One of the most famous strange attractors is the Lorenz attractor, discovered by Edward Lorenz, a mathematician and meteorologist. In 1963, Lorenz was working on a model for atmospheric conditions. As I’ve already mentioned, weather is a chaotic system. Lorenz’s model was a 3D model with three interacting variables. These three variables have rather involved technical definitions, but they relate to the viscosity, thermal conductivity, temperature differences and size ratios of the space defined by the model. Fortunately, you don’t really need to understand all

these concepts in order to reproduce the equations in a program.

When Lorenz supplied various values to these variables and plotted the output values over time, he came up with the shape you see in Figure 9.12.

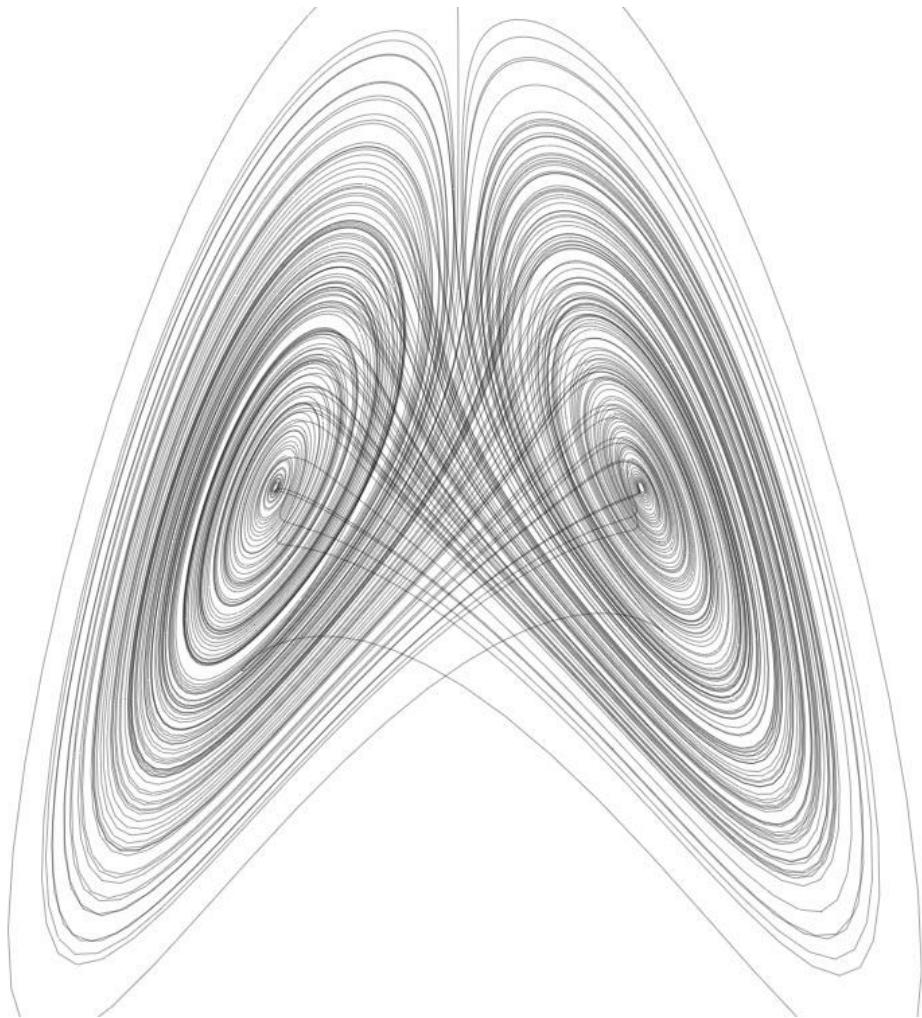


Figure 9.12. The Lorenz attractor.

This has a beautiful form that some find looks like a butterfly, a strange coincidence when you consider the

connection between chaos, attractors and the just-mentioned butterfly effect.

But it's even more interesting to see the Lorenz attractor in action, so let's look at the formulas and then get coding.

Again, the model is in 3D, so the output of the equations will relate to x , y and z coordinates in 3D space. In the first example, you'll just be plotting a two-dimensional view, so you can ignore one of the coordinates when rendering. But you'll still need to calculate them all on each iteration, as all three values interact in the formulas. I'll use the variables a , b and c to represent the three variables in the Lorenz equations. These are usually represented by Greek letters, but unless you have a Greek keyboard, the variables a , b and c will be easier to type.

Here are the formulas as given by Lorenz (de-Greeked):

$$\begin{aligned} \frac{dx}{dt} &= a * (y - x) \\ \frac{dy}{dt} &= x * (c - z) - y \\ \frac{dz}{dt} &= x * y - b * z \end{aligned}$$

The $\frac{d}{dt}$ in $\frac{dx}{dt}$ again stands for delta, or change. So it's the change in x over a particular elapsed time. Same for the other equations. Because you'll mostly be interested in the change in the value of x , y or z , you can rearrange the equations, like so:

$$\begin{aligned} dx &= (a * (y - x)) * dt \\ dy &= (x * (c - z) - y) * dt \\ dz &= (x * y - b * z) * dt \end{aligned}$$

And since this is the CHANGE in the value of a particular variable, it's the amount you'll add to the existing value of that variable. So you can re-write it further:

$$\begin{aligned} x1 &= x + (a * (y - x)) * dt \\ y1 &= y + (x * (c - z) - y) * dt \\ z1 &= z + (x * y - b * z) * dt \end{aligned}$$

And this is the exact code you'll use in the next file, `lorenz1.js`:

```

window.onload = function() {
    var x, y, z, a, b, c,
        scale = 16;

    init();

    function init() {

        chaos.init();
        chaos.context.translate(chaos.width *.5,
                               -chaos.height *.1);
        chaos.context.lineWidth = 0.35;

        x = Math.random() -.5;
        y = Math.random() -.5;
        z = Math.random() -.5;

        a = 20;
        b = 8 / 3;
        c = 28;

        setInterval(iterate, 0);

        document.body.addEventListener("keyup",
            function(event) {
                switch(event.keyCode) {
                    case 80: // p
                        chaos.popImage();
                        break;

                    default:
                        break;
                }
            });
    }

    function iterate() {
        chaos.context.beginPath();
        chaos.context.moveTo(x * scale, z * scale);
        lorenz();
        chaos.context.lineTo(x * scale, z * scale);
        chaos.context.stroke();
    }

    function lorenz() {
        var x1, y1, z1,

```

```

dt = .01;

x1 = x + (a * (y - x)) * dt;
y1 = y + (x * (c - z) - y) * dt;
z1 = z + (x * y - b * z) * dt;
x = x1;
y = y1;
z = z1;
}
}

```

This is a relatively simple script. The variables are what I just discussed, with the addition of `scale` to control how big the image will be.

In the `init` function, the context is translated to a particular location. This just uses values I found by trial and error to center the image on the canvas. By default, the image would end up centered horizontally, but well below center vertically. So I moved it up a bit on the y-axis. Feel free to change this to better display the image on your screen. The initial values of `x`, `y` and `z` are randomized, and `a`, `b` and `c` are set to the original values specified by Lorenz.

The function `iterate` is run on an interval. This does a `moveTo` to the existing point, calls the `lorenz` function to get the next point and draws a line to it. As I said earlier, you'll be rendering in 2D, so just ignore one of the coordinates. I found that rendering the `x` and `z` coordinates looked nice, but try other combinations. Also, the coordinate value is multiplied by `scale` to make it large enough to fill the canvas. You may need to change `scale` to have the image better fit your screen.

The `lorenz` function simply applies the formula you just saw to the existing values of `x`, `y` and `z`. Note that you'll need to apply the results to some temporary variables first and then back to the original variables. Otherwise, you'll be changing values in the middle of the overall calculation.

And that's it. Run the file and you should get an image somewhat similar to Figure 9.12. Because it is running on an interval, you'll see the image slowly forming in real time. Notice how the path it traces orbits two separate points. And realize that, although you are rendering in 2D, the model itself exists in 3D space and the path is orbiting two 3D points.

As the path orbits around these two points, it will never take the exact same path. It may appear to do so for a loop or two, but it will soon diverge. There is no repeated pattern in the loops it forms.

Of course, you can try altering the values for a , b and c . The default values are not magical. Similar values will usually give you very similar results, but not always. One interesting experiment is to change the value for c to 15. You should get something similar to Figure 9.13. Note that you may get the mirror image of this figure depending on the random placement of the initial point.

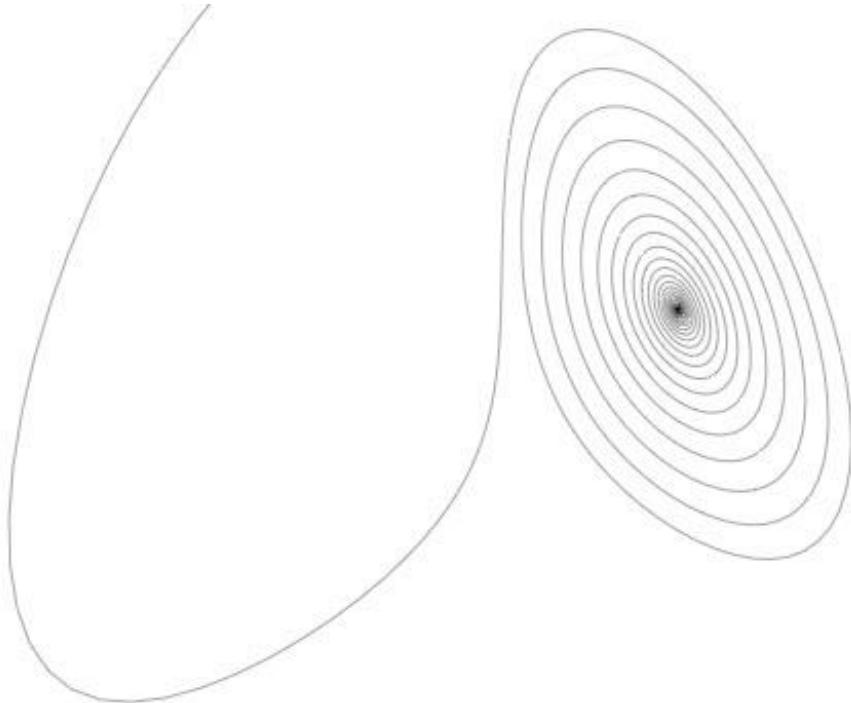


Figure 9.13. Lorenz with $c = 15$.

Here, you see the path is attracted to and eventually converges on a single point. No chaos here. But, when you increase c to 18, something interesting happens. The path starts orbiting one point, but rather than converging on it, the radius of the orbit increases until it finally flies off and starts to orbit the other point briefly. It will then go back and forth between the two orbits.

Another interesting one to try is setting c to 50. At first this seems like it's going to do the usual Lorenz attractor thing, but after about 20 loops, it settles into a single path and stays there forever. See Figure 9.14.

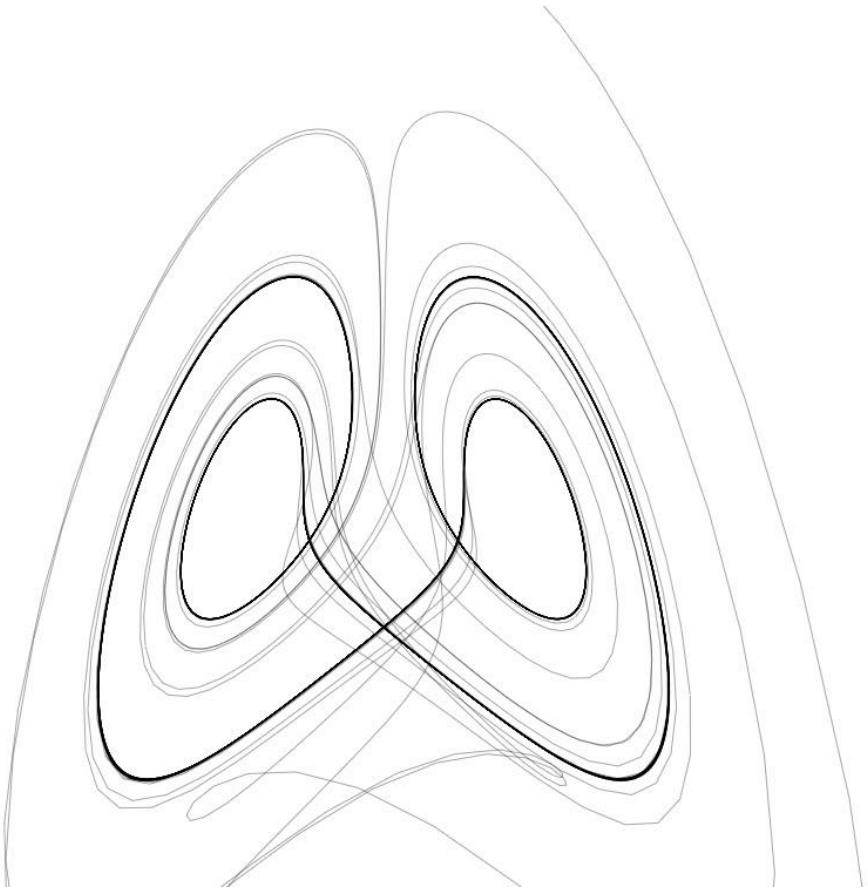


Figure 9.14. Lorenz with $c = 50$.

Try playing with other values and plotting other pairs of dimensions to get a different view on the attractor's form. There's not a whole lot of variety here, but you might find something interesting.

Lorenz 3D

To really appreciate the Lorenz attractor, you should view it in 3D. The file `lorenz2.js` takes care of that:

```
window.onload = function() {
```

```
var x, y, z, a, b, c,
    scale = 8,
    points = [],
    numPoints = 5000;

init();

function init() {

    chaos.init();
    chaos.context.lineWidth = 0.35;

    x = Math.random() - .5;
    y = Math.random() - .5;
    z = Math.random() - .5;

    a = 20;
    b = 8 / 3;
    c = 28;

    createPoints();
    requestAnimationFrame(display);

    document.body.addEventListener("keyup",
        function(event) {
            switch(event.keyCode) {
                case 80: // p
                    chaos.popImage();
                    break;

                default:
                    break;
            }
        });
}

function createPoints() {
    for(var i = 0; i < numPoints; i += 1) {
        lorenz();
        points.push({
            x: x,
            y: y,
            z: z
        });
    }
}
```

```

function display() {
    var p,
        pers,
        f1 = 300,
        sx, sy;

    chaos.clear();
    chaos.context.save();
    chaos.context.translate(chaos.width *.5,
                           chaos.height * .5);

    for(var i = 0; i < numPoints; i += 1) {
        p = points[i];
        rotate(p);
        pers = f1 / (f1 + p.z);
        sx = p.x * pers * scale;
        sy = p.y * pers * scale;
        chaos.context.fillRect(sx, sy,
                               pers * 2,
                               pers * 2);
    }
    chaos.context.restore();
    requestAnimationFrame(display);
}

function rotate(p) {
    var sinx = Math.sin(.002),
        cosx = Math.cos(.002),
        siny = Math.sin(0.01),
        cosy = Math.cos(0.01),
        y1 = p.y * cosx - p.z * sinx,
        z1 = p.z * cosx + p.y * sinx;
        x1 = p.x * cosy - z1 * siny,
        z2 = z1 * cosy + p.x * siny;
    p.x = x1;
    p.y = y1;
    p.z = z2;
}

function lorenz() {
    var x1, y1, z1,
        dt = .01;

    x1 = x + (a * (y - x)) * dt;
    y1 = y + (x * (c - z) - y) * dt;
    z1 = z + (x * y - b * z) * dt;
    x = x1;
}

```

```
    y = y1;
    z = z1;
}
}
```

This has a few new features. I've moved the `scale` down to 8, created a `points` array and set `numPoints` to 5000. You may need to lower that if your computer can't handle that many points.

In the `init` function, the call to `setInterval(iterate, 0)` has been changed to:

```
createPoints();
requestAnimationFrame(display);
```

The `createPoints` function calls `lorenz` 5,000 times and pushes the value of `x`, `y` and `z` onto an object and into the `points` array on each iteration. You now have an array of pre-calculated points. The `display` and `rotate` functions rotate these points around the origin and display them on the screen each interval.

It's a bit beyond the scope of this book to go into 3D rendering methods, but it's all right there if you want to try to figure out what's going on. Note that this could probably be further optimized as well, but it serves the purpose for this demonstration.

I also decided to use `requestAnimationFrame`, as it is a far more optimized way of doing animation on those browsers that support it, which should be most modern browsers. If the example doesn't work for you, change the `requestAnimationFrame` line to:

```
setInterval(display, 60/1000);
```

Also, remove the `requestAnimationFrame` line from the end of the `display` function. Alternately, do a search for "requestAnimationFrame polyfill," which should give you info on how to make a more robust function that will support all possible browsers.

One way or the other, you should be able to get the Lorenz attractor up and rotating around in 3D. Plenty of values to mess around with here to create all kinds of other effects. Have fun with it.

Other Attractors

The Lorenz attractor is relatively simple. There are other strange attractors that have far more variety. I've programmed many different kinds of 3D strange attractors in several different languages over the years. There's nothing I'd love more than to present you with some of those examples here. Unfortunately, most of these require many thousands of points to build an interesting image – from 10,000 to 100,000 or more. And rotating that many points in real time in plain JavaScript using canvas just isn't feasible.

I'm sure some JavaScript guru could come up with something that runs half-decently, but one of the goals of this book is to have the code be understandable, and to not go off into some side discussion of crazy JavaScript optimization hacks.

Therefore, I've dug up a couple of 2D attractors that are simple enough to program and will produce amazing images. You won't be able to rotate them in 3D, but I think you'll still enjoy some of the images they create.

I got both of these from Paul Bourke's website. You should remember Professor Bourke from his IFS fractals in Chapter 5. The first one is attributed to Clifford Pickover, another name you should know if you are into fractals, math and computer-generated, math-based images. This is the page from Bourke's site:

<http://paulbourke.net/fractals/clifford/>

On that page are the formulas and many sample images, along with the parameters that created them. The code to create them is in `pickover.js` and is largely repurposed from the first Lorenz example:

```
window.onload = function() {
    var x, y, a, b, c, d,
        scale = 150,
        maxPoints = 300000,
        pointCount = 0,
        interval;

    init();

    function init() {

        chaos.init();
        chaos.context.translate(chaos.width *.5,
                               chaos.height *.5);
        chaos.context.lineWidth = 0.35;

        x = Math.random() - .5;
        y = Math.random() - .5;

        a = Math.random() * 4 - 2;
        b = Math.random() * 4 - 2;
        c = Math.random() * 4 - 2;
        d = Math.random() * 4 - 2;
        console.log("a:", a);
        console.log("b:", b);
        console.log("c:", c);
        console.log("d:", d);

        interval = setInterval(iterate, 0);

        document.body.addEventListener("keyup",
            function(event) {
                switch(event.keyCode) {
                    case 80: // p
                        chaos.popImage();
                        break;

                    default:
                        break;
                }
            }
        );
    }
}
```

```

        }
    });
}

function iterate() {
    for(var i = 0; i < 100; i += 1) {
        pickover();
        chaos.context.fillRect(x * scale, y * scale,
            .25, .25);
        pointCount += 1;
        if(pointCount > maxPoints) {
            clearInterval(interval);
            alert("done");
            return;
        }
    }
}

function pickover() {
    var x1, y1, z1;

    x1 = Math.sin(a * y) + c * Math.cos(a * x);
    y1 = Math.sin(b * x) + d * Math.cos(b * y);
    x = x1;
    y = y1;
}
}

```

This example is in 2D, so there is only an `x` and `y`, no `z`. The formulas also have four parameters, so the variables for those are `a` through `d`. Rather than hard-coding any specific values for the parameters, the `init` function randomizes all of them.

It can take several hundred thousand iterations to generate a decent image with this formula, so the `iterate` function speeds things up by calling the `pickover` function 100 times each time it is called. Each time it draws a small rectangle at the current `x`, `y` position. If you draw too many points, the drawing will just become a black blob, so the `maxPoint` variable provides a way to stop rendering after a certain number of points. I've found 300,000 to be about right, but feel free to change that.

Finally, the `pickover` function applies the formula and generates new `x` and `y` values.

Each time you refresh the page, you'll get a new image. Most of the time, you'll just see a few points or a small squiggly line. But, maybe one time in ten or so, you'll see something cool, like in Figures 9.15, 9.16 or 9.17.

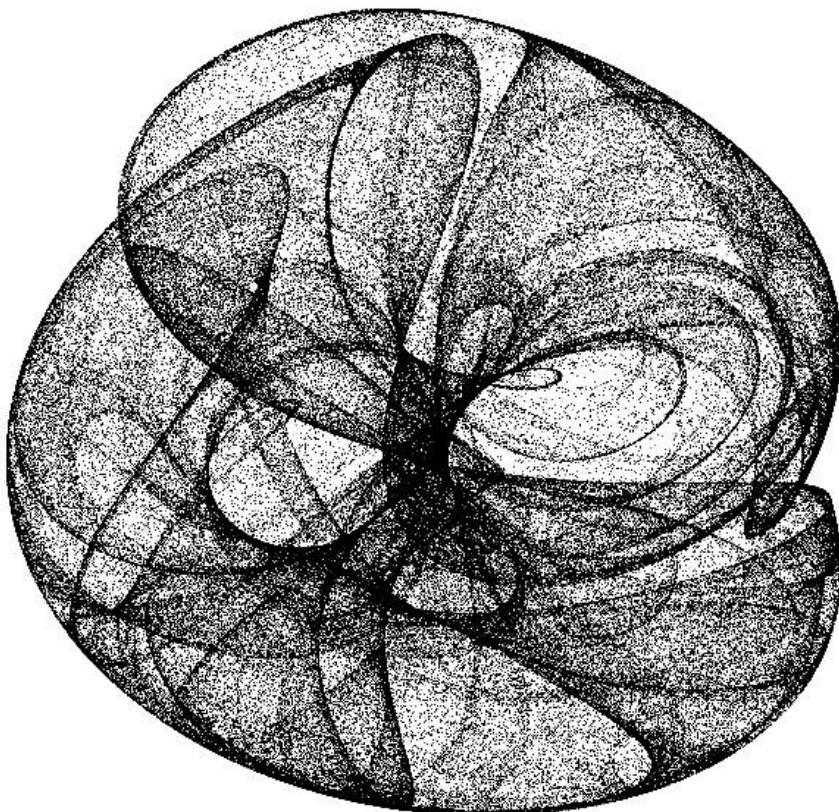


Figure 9.15. Strange attractor.

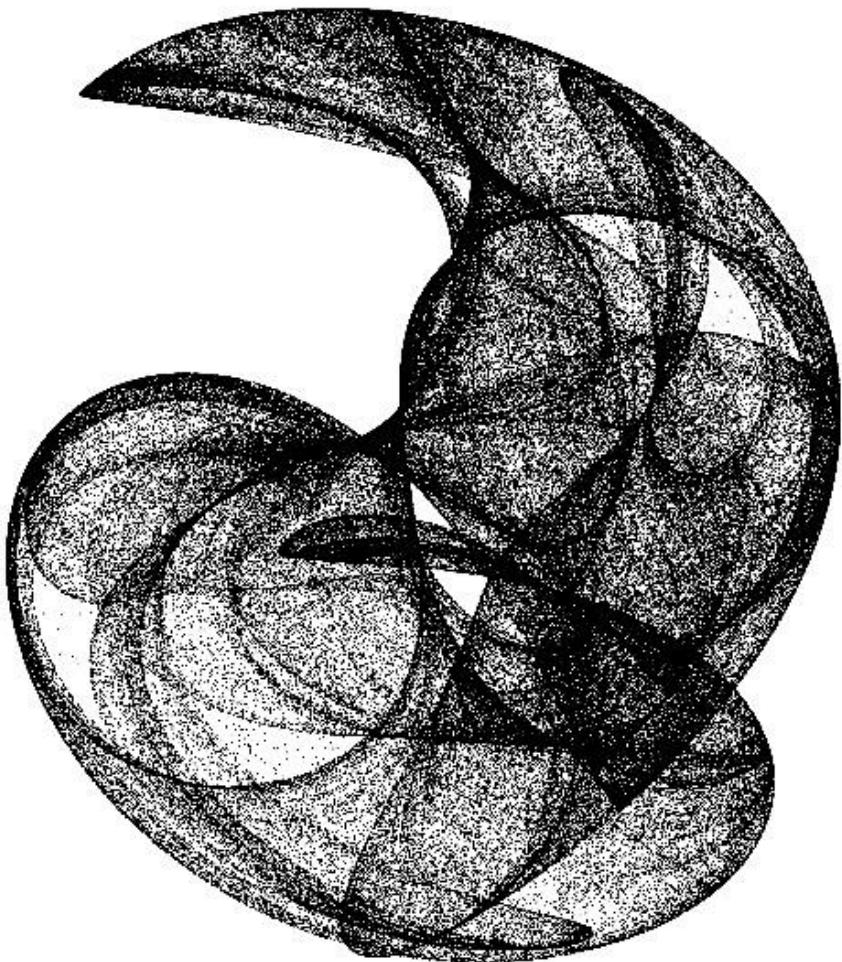


Figure 9.16. Strange attractor.

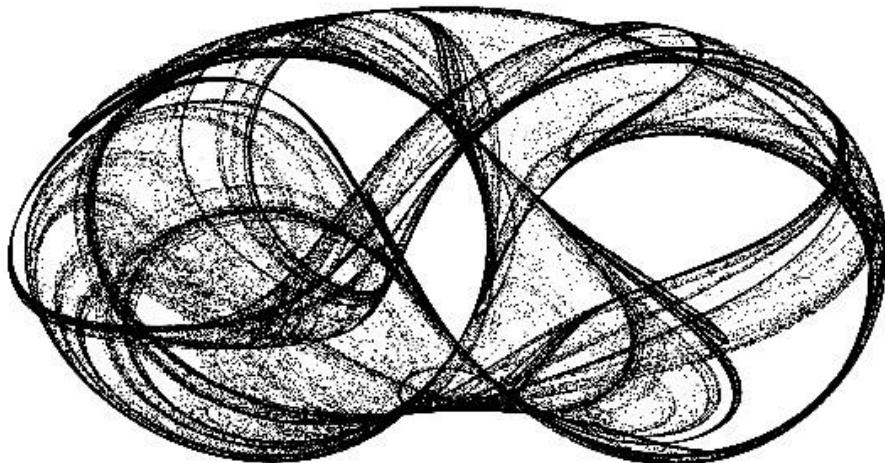


Figure 9.17. Strange attractor.

The more interesting ones are most likely strange attractors. Especially those that continue to grow and get darker until the maximum is reached. The ones that seem to finish in a second or two have converged on a set of points or a path. The strange attractors are in chaos mode and will never settle down, though they will keep building on the same basic form.

The next attractor is the Peter de Jong attractor, found on this page:

<http://paulbourke.net/fractals/peterdejong/>

The file for this is `dejong.js` and is identical to `pickover.js`, except that the `pickover` function is renamed `dejong` and contains the following formula:

```
x1 = Math.sin(a * y) + c * Math.cos(b * x);  
y1 = Math.sin(c * x) + d * Math.cos(d * y);
```

This is only a few characters different from the Pickover version. I'm not really sure what the qualitative difference in the images is. I feel like this version creates images that

tend to be a bit more wispy or stringy, but I wouldn't swear to it.

Both of these last examples write out the values for the parameters so that, if you find an image you like, you can check your console and note the values and hard-code them in another file to experiment with values near them. Slight changes to one or more of the values can enhance (or totally ruin) the image.

You might even want to create some kind of user interface in HTML to control the parameters. It's a project waiting for you to attack it.

Also, look deeper into Paul Bourke's fractal site at <http://paulbourke.net/fractals/> and you'll find many other fractal and strange attractor formulas that you can easily implement in much the same way as these last examples. And there are even more available if you take the time to do a broader search for them.

Summary

Well, you've gotten a tour from order through chaos and back again safely. More fractal tools in your belt. And more to come!

Chapter 10: L-Systems

In Chapter 10, you're going to learn about Lindenmayer Systems, or L-systems as they are usually called. Although not specifically designed for creating fractals, you can certainly use L-systems for such purposes. Let's start by finding out what an L-system is and how the subject came about.

Background

L-systems were developed in the 1960s by Aristid Lindenmayer. Lindenmayer was a biologist studying the various stages of plant cell growth. His L-systems were an attempt to create a language that could codify and describe this development.

In a nutshell, an L-system is a set of symbols combined to form a string. The string is then transformed into another, usually longer, string by the application of various rules that replace symbols with other symbols. By itself, that's all an L-system does – generate strings of symbols. But you can use those symbols to represent graphical drawing commands, in which case you wind up with a powerful fractal-creation language.

Intro to L-Systems

An L-system consists of three elements:

1. The alphabet or vocabulary. This is the set of valid symbols that can be used in the system. Sometimes you might see the alphabet broken down into variables, which are symbols that will be replaced, and constants, which will be copied over as-is when transforming a string.
2. The initiator or seed. This is the initial string of symbols that a particular system starts with.
3. The set of rules that are applied to transform the string.

In addition, for creating fractals you'll need some kind of graphics interpreter that will iterate through the string and perform a graphics operation for any symbols that are mapped to graphics operations. Generally, this is similar to the "turtle graphics" from the Logo programming language. A turtle graphics system has a virtual turtle with a position and orientation. You can tell the turtle to move forward or rotate a certain number of degrees. The turtle also has a pen, so you can tell it to draw a line while it is moving. In some implementations, the pen can have attributes, such as width or color. The system used here will only draw one width and one color, but you could extend it rather easily.

With these few simple commands, you can build programs that create quite complex graphics. In the L-system version, some of the symbols in the system vocabulary will map to turtle commands. Other symbols are not mapped to commands, only used to help form the structure of the string.

Ignoring the graphics part for now, let's see an example of an L-system in action.

The sample system will have two symbols: A and B.

The initiator will be a string of a single character: A.

There will be two rules:

1. Whenever you see an A, replace it with AB.
2. Whenever you see a B, replace it with A.

This gives you the following sequence:

A

AB

ABA

ABAAB

ABAABABA

And so on. This particular system was Lindenmayer's system for modeling the growth of algae.

Because L-systems need two separate, somewhat complex components – the L-system itself and the graphics interpreter – I thought it would be best to build these one at a time. First, you'll create an L-system that transforms strings and then plug in the graphics interpretation piece.

Coding the L-System

In the first pass at this, you will just create an L-system that transforms strings. Rather than getting fancy with graphics,

this version will just use `console.log` to trace the string out to the JavaScript console. The code can be found in `lindenmayer1.js`:

```
window.onload = function() {
    var vocab,
        initiator,
        string,
        rules = {},
        interval,
        iterations = 0,
        maxIter;

    init();

    function init() {
        chaos.init();
        algae();
        string = initiator;
        interval = setInterval(transform, 1000);
        document.body.addEventListener("keyup",
            function(event) {
                switch(event.keyCode) {
                    case 80: // p
                        chaos.popImage();
                        break;

                    default:
                        break;
                }
            });
    }

    function algae() {
        vocab = "AB";
        initiator = "A";

        rules["A"] = "AB";
        rules["B"] = "A";

        maxIter = 7;
```

```

}

function transform() {
    var i,
        char,
        rule,
        newString = "";

    console.log(string);

    iterations += 1;
    if(iterations > maxIter) {
        clearInterval(interval);
    }

    for(i = 0; i < string.length; i += 1) {
        char = string.charAt(i);
        rule = rules[char];
        if(rule) {
            newString += rule;
        }
        else {
            newString += char;
        }
    }
    string = newString;
}
}

```

Variables are created for the standard parts of the L-system: the vocabulary, initiator, rules and the string that is to be transformed. Note that no values are assigned to these yet. Also, the program will iterate using `setInterval`, so it has a few variables to be able to keep track of iterations and stop the program when a set number of iterations have been executed.

The `init` function calls the `algae` function which defines the L-system itself. Using this strategy, you can now create and insert new L-systems into the program by simply adding a new function that defines the various parts of the system. The rest of the code can remain unchanged.

The `algae` function defines the vocabulary, initiator, rules and maximum iterations. Actually, the `vocab` variable will never be used in the program, but it's nice to define it if only for the sake of documentation. A more robust system might even check the initiator and all the rules to ensure that all the symbols used were part of the vocabulary, but I'll leave that for you to implement if you want.

The `rules` variable is a generic JavaScript object. An individual rule is added to it using array notation. This gives you an easy way to access rules and get their replacement values.

After the `algae` function is called, the `string` variable is set to equal `initiator`. Then, `setInterval` is called, which will run `transform` once every second.

The `transform` function logs the current string, and checks to see if the maximum iterations have been reached. If so, it ends the program. If not, it proceeds to string transformation.

It would be nice to use regular expressions here, but when you have multiple rules, this would get messy. You need to apply each rule in sequence. Thus, each successive rule would be operating on a partially transformed string, rather than the original string. So `transform` loops through the string character by character, doing the replacements and creating a brand new string. Note that if no rule exists for a particular symbol, then the symbol itself is copied to the output string. When it is done, it assigns the value of this new string back to `string`.

When you run this program and open your JavaScript console, you should see the following strings logged, one per second:

A
AB

```
ABA
ABAAB
ABAABABA
ABAABABAABAAB
ABAABABAABAABABAABABA
ABAABABAABAABABAABAABAAB
```

This matches what Lindenmayer himself got, so I'll count it as a success and move onto the next section.

You are free to experiment with different L-systems at this point if you want, but unless your mind works a bit differently from mine, you probably won't find logging out long strings of characters very interesting, so let's get some graphics going.

Coding the Graphics Interpreter

Next up is the L-system code along with the graphics interpretation code. Now, some – but not all – of the symbols will be used to trigger functions, such as drawing, moving or turning. So, in addition to a `rules` object, there will be a `commands` object. This will be set up the same way, with array notation, but will hold links to functions that can be called. Here's the code in full for `lindenmayer2.js`:

```
window.onload = function() {
  var vocab,
    initiator,
    string,
    rules = {},
    commands = {},
    system,
    angle,
    turnAngle,
    x, y,
    stack,
    size = 10,
    maxIter = 1;
```

```

init();

function dragon() {
    vocab = "+-FXY";
    initiator = "FX";

    rules["X"] = "X+YF+";
    rules["Y"] = "-FX-Y";

    commands["F"] = draw;
    commands["+"] = right;
    commands["-"] = left;
    turnAngle = 90;

    angle = 0;
    x = chaos.width * .5;
    y = chaos.height * .5;
}

function init() {

    chaos.init();

    system = dragon;
    iterate();
    render();

    document.body.addEventListener("keyup",
        function(event) {
            switch(event.keyCode) {
                case 80: // p
                    chaos.popImage();
                    break;

                case 38: // up
                    maxIter += 1;
                    iterate();
                    render();
                    break;

                case 40: // down
                    maxIter -= 1;
                    maxIter = Math.max(1, maxIter);
                    iterate();
                    render();
            }
        }
    );
}

```

```
        break;

    case 90: // z
        size -= 1;
        size = Math.max(1, size);
        iterate();
        render();
        break;

    case 88: // x
        size += 1;
        iterate();
        render();
        break;

    default:
        break;
    }
});
```

}

```
function iterate() {
    stack = [];
    system();
    string = initiator;
    for(var i = 0; i < maxIter; i += 1) {
        transform();
    }
}
```

}

```
function transform() {
    var i,
        char,
        rule,
        newString = "";

    for(i = 0; i < string.length; i += 1) {
        char = string.charAt(i);
        rule = rules[char];
        if(rule) {
            newString += rule;
        }
        else {
            newString += char;
        }
    }
    string = newString;
    console.log(string);
```

```
}

function render() {
    var i, char, command;

    chaos.clear();
    chaos.context.beginPath();
    chaos.context.moveTo(x, y);
    for(i = 0; i < string.length; i += 1) {
        char = string.charAt(i);
        command = commands[char];
        if(command) {
            command();
        }
    }
    chaos.context.stroke();
}

function move() {
    x += Math.cos(angle * Math.PI / 180) * size;
    y += Math.sin(angle * Math.PI / 180) * size;
    chaos.context.moveTo(x, y);
}

function draw() {
    x += Math.cos(angle * Math.PI / 180) * size;
    y += Math.sin(angle * Math.PI / 180) * size;
    chaos.context.lineTo(x, y);
}

function left() {
    angle -= turnAngle;
}

function right() {
    angle += turnAngle;
}

function push() {
    stack.push({
        x: x,
        y: y,
        angle: angle
    })
}

function pop() {
    var state = stack.pop();
```

```

        if(state) {
            x = state.x;
            y = state.y;
            angle = state.angle;
            chaos.context.moveTo(x, y);
        }
    }
}

```

A few more variables have been added at the top. I'll cover those as I go through the rest of the code.

The `algae` function has been changed to a `dragon` function. This will create a relatively simple but impressive fractal shape:

```

function dragon() {
    vocab = "+-FXY";
    initiator = "FX";

    rules["X"] = "X+YF+";
    rules["Y"] = "-FX-Y";

    commands["F"] = draw;
    commands["+"] = right;
    commands["-"] = left;
    turnAngle = 90;

    angle = 0;
    x = chaos.width * .5;
    y = chaos.height * .5;
}

```

In addition to `vocab`, `initiator` and `rules`, this now sets `commands` and several other variables.

The `commands` object holds references to functions that will be called for various symbols when the string is interpreted. Note that not all symbols have a function associated with them. Symbols may call a `left` or `right` function to turn the orientation of the drawing object. (I'll call it the turtle from here on out.) The `turnAngle` variable determines how many degrees the turtle will turn when one of those functions is called.

The last three, `angle`, `x` and `y`, set the initial location and orientation of the turtle.

The `init` function sets the `system` variable equal to the `dragon` function. This is so that later you can easily swap out other L-system functions by changing this one line of code. After that, `iterate` and `render` are called.

First, the `iterate` function.

```
function iterate() {  
    stack = [];  
    system();  
    string = initiator;  
    for(var i = 0; i < maxIter; i += 1) {  
        transform();  
    }  
}
```

This creates an array called `stack`. I'll cover that later. It calls `system`, which is now equal to the `dragon` function. This creates that particular L-system and positions the turtle. It then calls `transform` a number of times, limited by `maxIter`, currently set to 1. The `transform` function is much the same as it was, but simplified a little since it does not need to deal with keeping track of iterations and stopping the interval. So, at this point, the string has been iterated a single time.

Now `render` is called:

```
function render() {  
    var i, char, command;  
  
    chaos.clear();  
    chaos.context.beginPath();  
    chaos.context.moveTo(x, y);  
    for(i = 0; i < string.length; i += 1) {  
        char = string.charAt(i);  
        command = commands[char];  
        if(command) {  
            command();  
        }  
    }  
    chaos.context.stroke();  
}
```

The `render` function clears the canvas, starts a path and moves to the current turtle location. It runs through all the symbols in the string. If a symbol is associated with a command, that command is executed.

Finally, `render` finishes up by calling `stroke` on the context, drawing whatever shape has been created.

The drawing command functions

are `move`, `draw`, `left`, `right`, `push` and `pop`. The first four should be easy enough to figure out. I'll go over `push` and `pop` later, along with that `stack` variable I put off.

When you run this program, you should see a small upside-down and inverted "L" shape in the center of the canvas. If you open up the JavaScript console, you'll see the current string:

`FX+YF+`

Analyzing this, first you see `F`. This is mapped to the `draw` function, which causes the turtle to move forward, drawing one step. A step happens to be 10 pixels, based on the value of the `size` variable. In this case, "forward" means to the right because `angle` is currently 0.

Next is `x`, which has no command associated with it. Then `+`, which is mapped to the `right` function, causing the turtle to turn 90 degrees to the right.

The `y` symbol does nothing, `F` again moves forward (now in the downward direction) and `+` turns to the right 90 degrees again. And you're done.

Now, look at the key-handler section of the code. The up and down keys will increment and decrement the value of `maxIter` and then call `iterate` and `render` again. Pressing the up key will cause the shape to be transformed twice and redrawn. You should have a little squiggle on the screen. Hit

up several more times and you should see something like in Figure 10.1.

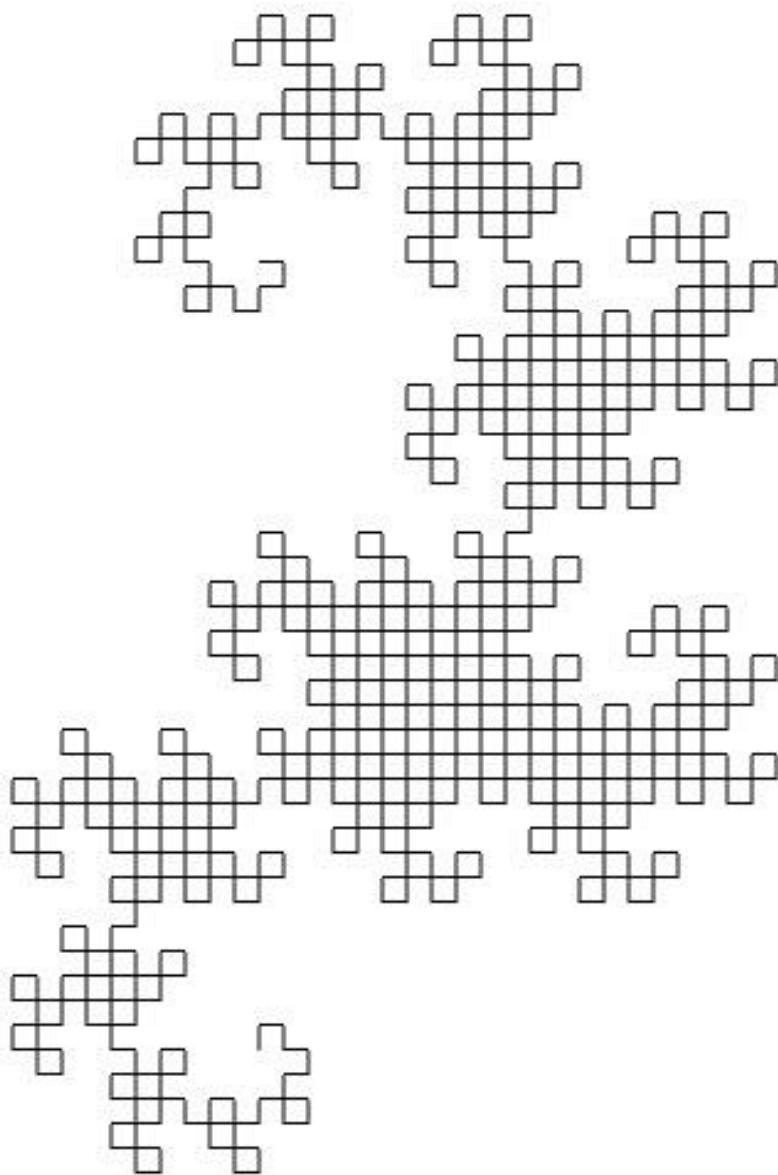


Figure 10.1. A dragon?

Hitting the up key a couple more times will likely make it too big to fit on the screen. You can also hit the “z” and “x” keys to decrease and increase the `size` variable. This makes each step smaller or larger, and thus changes the size of the shape as a whole. Figure 10.2 shows a larger dragon with a very small step.

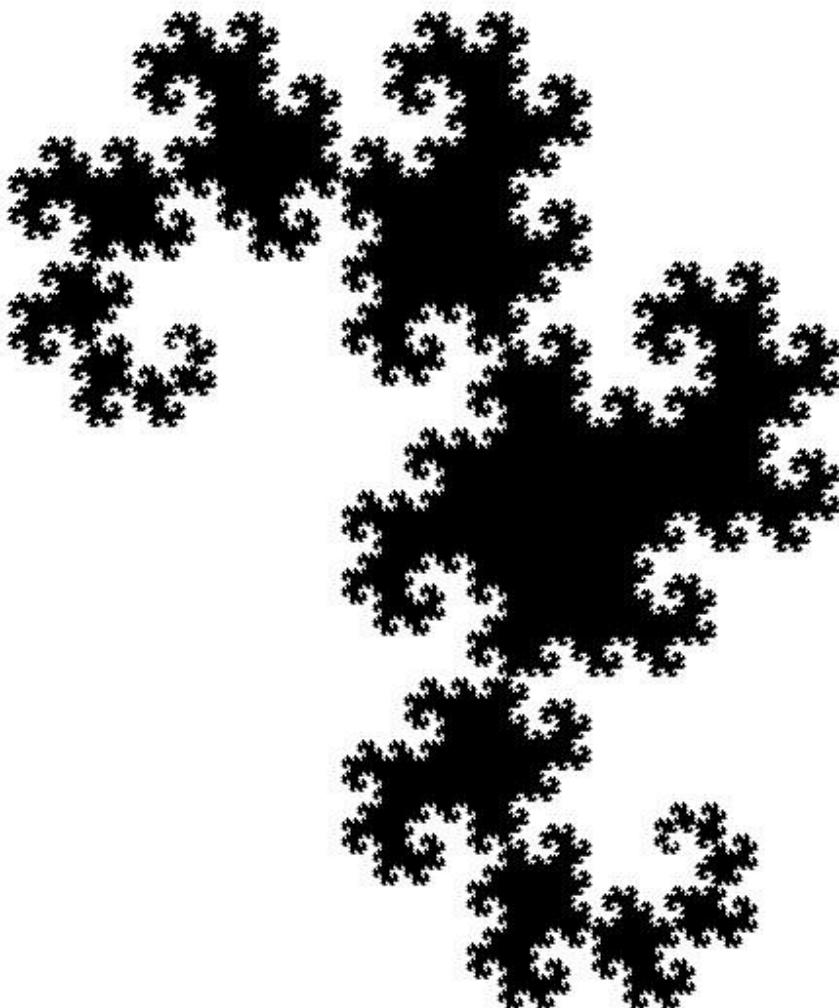


Figure 10.2. A dragon!

Advanced Commands: Push and Pop

Now, let's look at those items I said I'd cover later.

The dragon fractal is drawn in one long, single line. But there are times when you might want to do a branching action and draw multiple lines from a single point. Drawing a fractal tree is a good example of why you would want to do such a thing.

You could try to turn 180 degrees and go back to the previous point, then draw another line from there. But there's another way, using the `push` and `pop` functions.

The `push` function takes the current values for `x`, `y` and `angle` and puts them onto an object that it pushes into the `stack` array. You can then perform any number of other commands – moving, drawing, turning, etc. – and then call `pop`. This takes the last object out of the stack and copies the `x`, `y` and `angles` from it back into the system. This has the effect of resetting the turtle to the same state it was in when you called `push`. You can then execute another series of commands and draw another line or set of lines from the same point.

To see this in action, add the following `tree` function to the file:

```
function tree() {  
    vocab = "AB[]";  
    initiator = "A";  
  
    rules["A"] = "B[-A]+A";  
    rules["B"] = "BB";  
  
    commands ["A"] = draw;  
    commands ["B"] = draw;  
    commands ["["] = push;  
    commands ["]"] = pop;  
    commands ["+"] = right;
```

```
commands["-"] = left;
turnAngle = 45;

angle = -90;
x = chaos.width * .5;
y = chaos.height;
}
```

Also, set the `system` variable to `tree` in the `init` function:

```
system = tree;
```

Use the up key to iterate this a few times and maybe reduce the `size` a bit, and you should get something like in Figure 10.3.

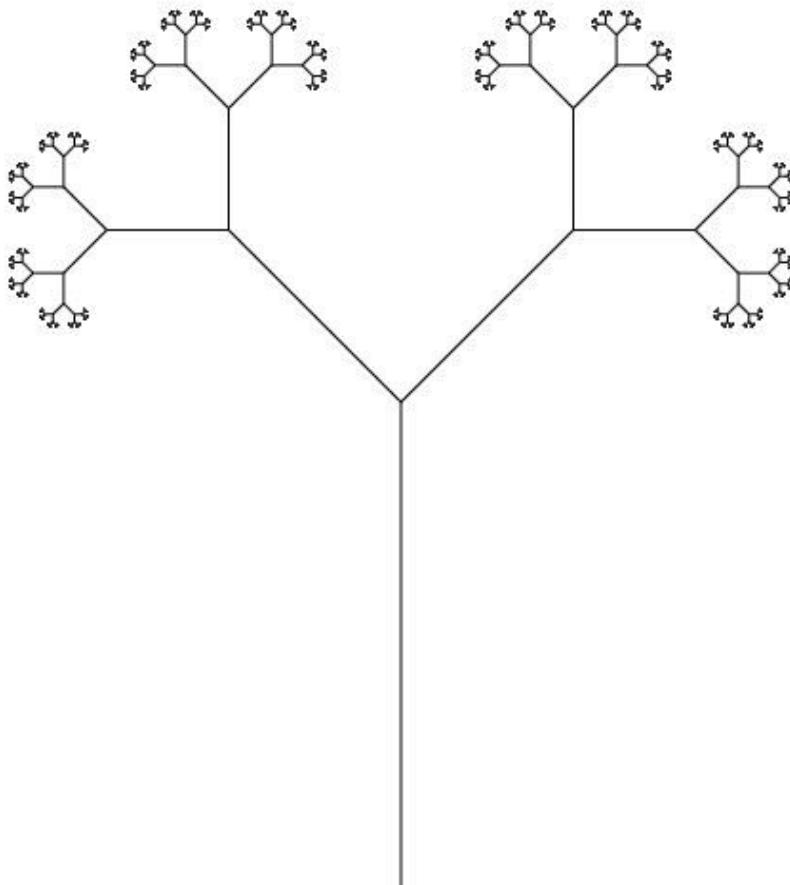


Figure 10.3. Fractal tree, revisited.

I've added several other system functions to the file as well. All you have to do is make sure the function is there and change the `system` variable to point to that function.

There's a `sierpinski` function:

```
function sierpinski() {  
    vocab = "AB+-";  
    initiator = "A-B-B";  
  
    rules["A"] = "A-B+A+B-A";  
    rules["B"] = "BB";  
  
    commands["A"] = draw;  
    commands["B"] = draw;  
    commands["+"] = right;  
    commands["-"] = left;  
    turnAngle = 120;  
  
    angle = 0;  
    x = chaos.width * .333;  
    y = chaos.height * .1;  
}
```

I imagine you can guess what shape this makes, but it's interesting to see the familiar fractal created with yet another method. See Figure 10.4.

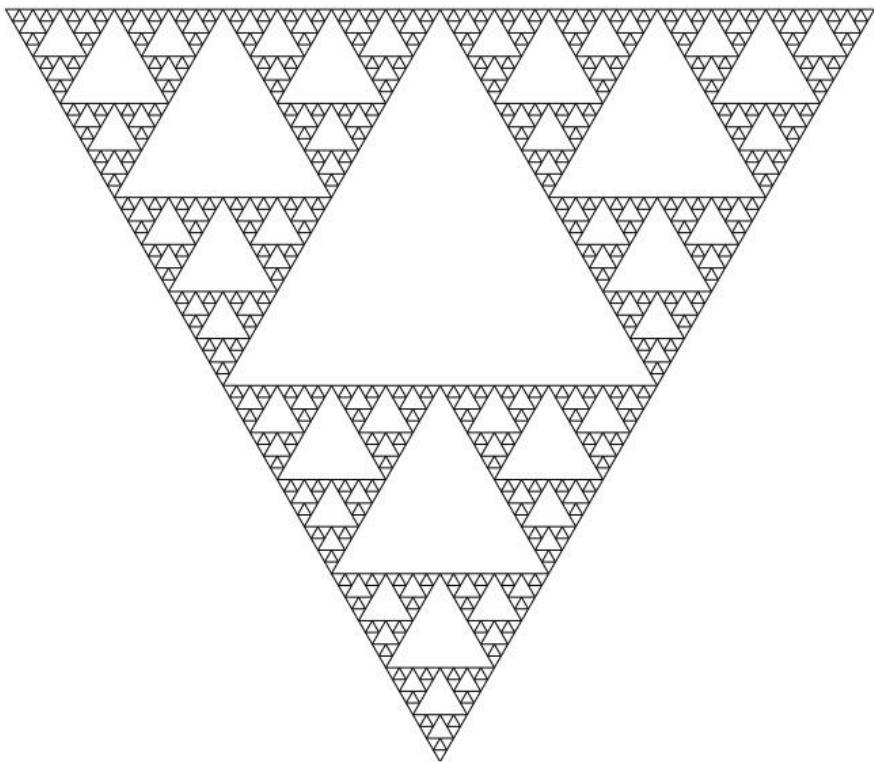


Figure 10.4. Not another Sierpinski!

Another example that uses the `push` and `pop` functions is the `plant` function:

```
function plant() {  
    vocab = "XF+-[]";  
    initiator = "X";  
  
    rules["X"] = "F-[ [X]+X]+F[+FX]-X";  
    rules["F"] = "FF";  
  
    commands["X"] = draw;  
    commands["F"] = draw;  
    commands["+"] = right;  
    commands["-"] = left;  
    commands["["] = push;  
    commands["]"] = pop;
```

```
turnAngle = 25;  
  
angle = -90;  
x = chaos.width * .5;  
y = chaos.height;  
}
```

This creates Figure 10.5.

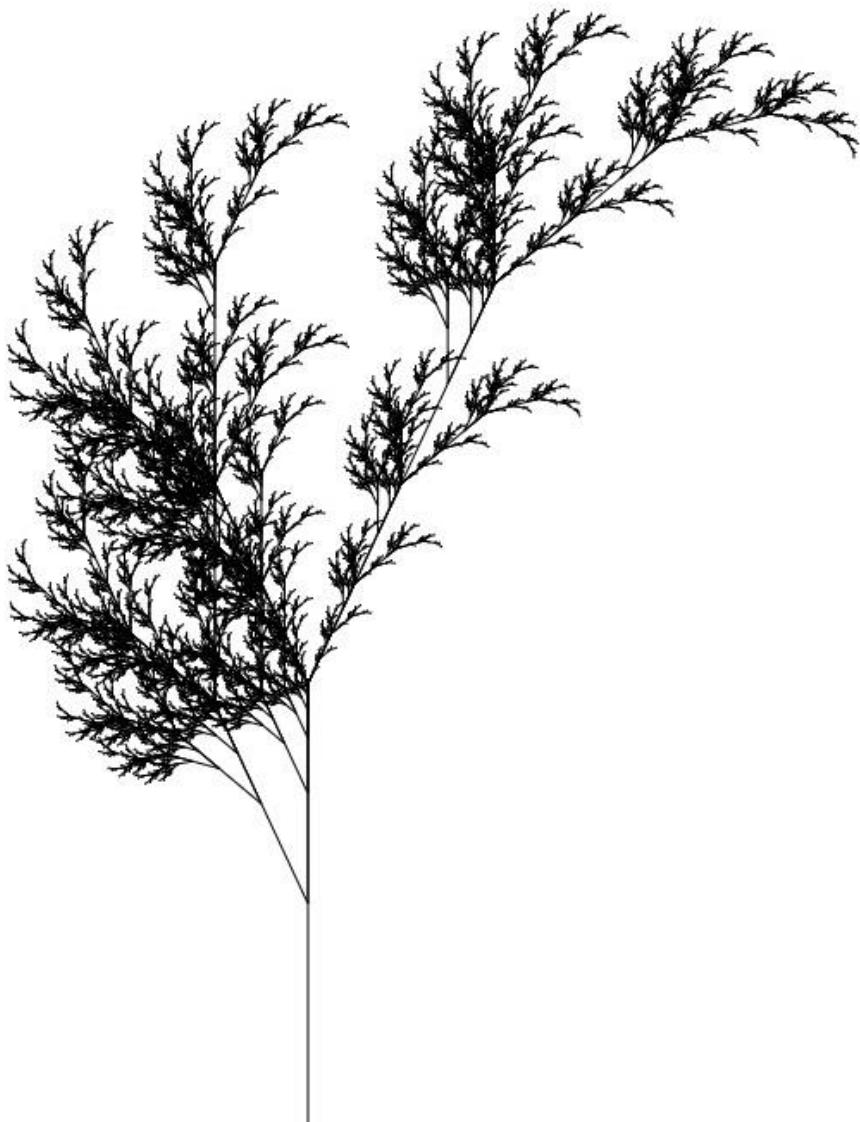


Figure 10.5. Fractal plant.

Another old friend, the Koch snowflake is formed by the `koch` function and seen in Figure 10.6:

```
function koch() {
    vocab = "FX+-";
    initiator = "F++F++F";

    rules["F"] = "F-F++F-F";
    rules["X"] = "FF";

    commands["X"] = draw;
    commands["F"] = draw;
    commands["+"] = right;
    commands["-"] = left;

    turnAngle = 60;

    angle = 0;

    x = chaos.width * .2;
    y = chaos.height * .3;
}
```

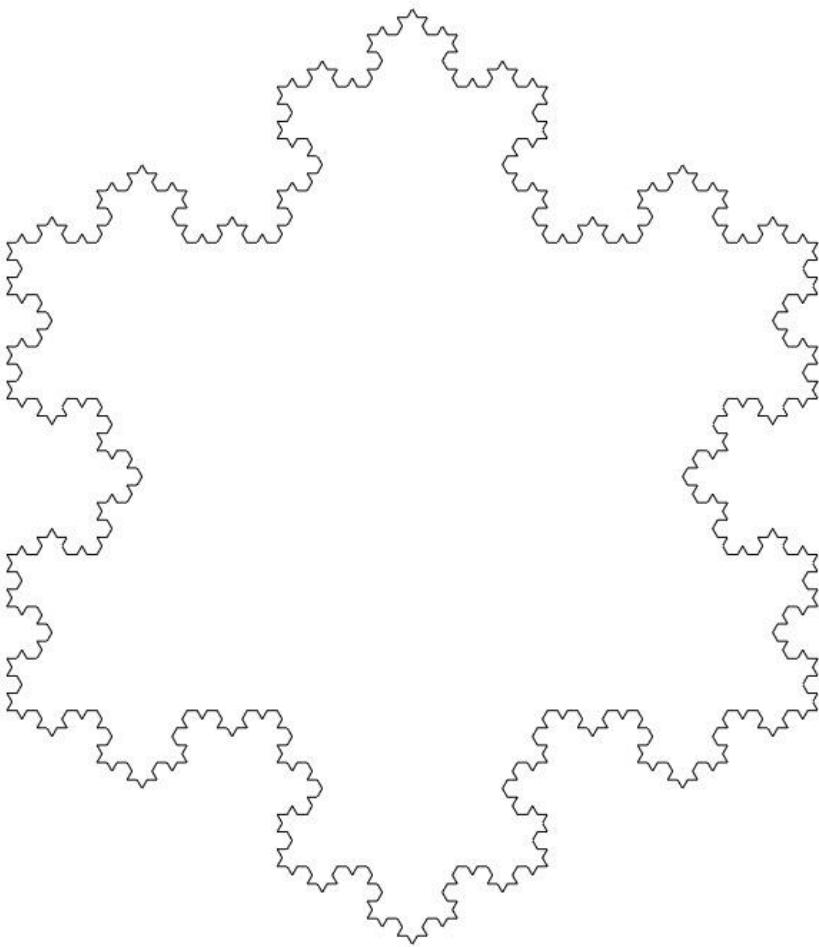


Figure 10.6. Koch snowflake, L-system style.

The `hilbert` function creates the “Hilbert space-filling curve,” first created by the mathematician David Hilbert in the late 1800s and seen in Figure 10.7.

```
function hilbert() {  
  vocab = "ABF+-";  
  initiator = "A";  
  
  rules["A"] = "+BF-AFA-FB+";  
  rules["B"] = "-AF+BFB+FA-";
```

```
commands["F"] = draw;  
commands["+"] = right;  
commands["-"] = left;  
  
turnAngle = 90;  
  
angle = 0;  
  
x = chaos.width * .1;  
y = chaos.height * .1;  
}
```

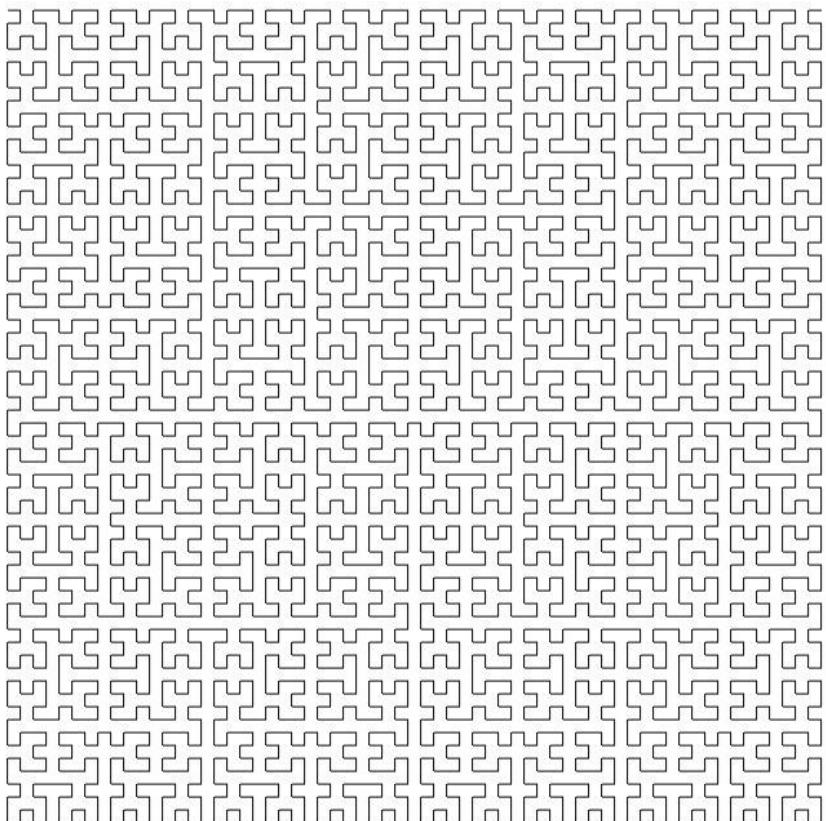


Figure 10.7. Hilbert curve.

Finally, the `gosper` function creates the Gosper curve, named for Bill Gosper, yet another mathematician. You can see this one in Figure 10.8.

```
function gosper() {  
    vocab = "XYF+-";  
    initiator = "X";  
  
    rules["X"] = "X+YF++YF-FX--FXFX-YF+";  
    rules["Y"] = "-FX+YFYF++YF+FX--FX-Y";  
  
    commands["F"] = draw;  
    commands["+"] = right;  
    commands["-"] = left;  
  
    turnAngle = 60;  
  
    angle = 0;  
  
    x = chaos.width * .8;  
    y = chaos.height * .8;  
}
```

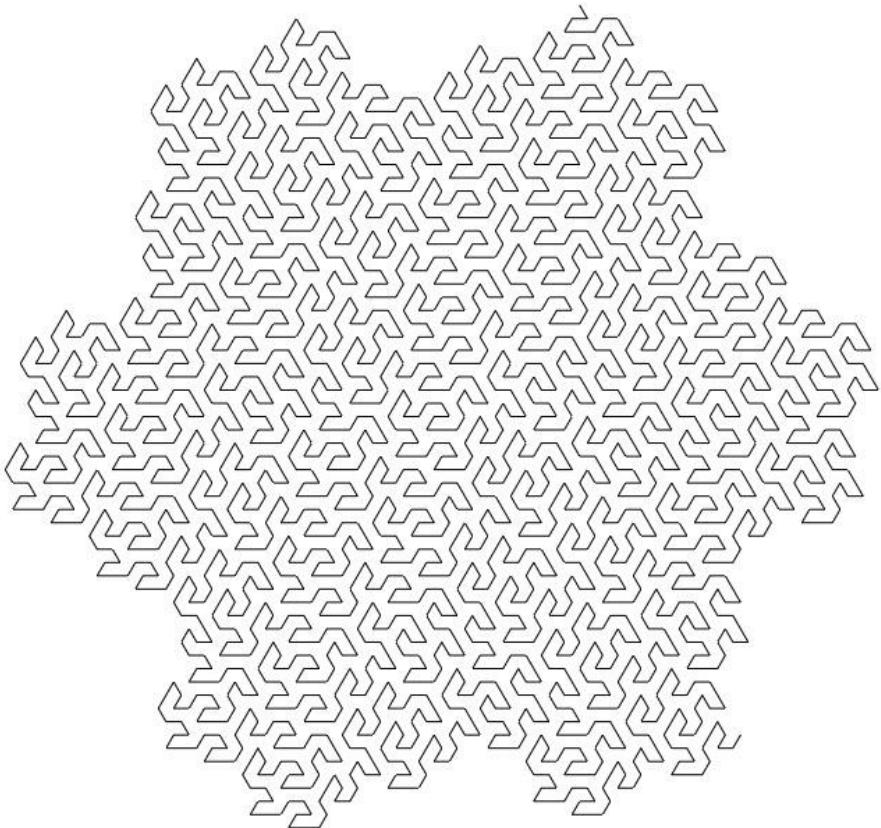


Figure 10.8. Gosper curve.

I said “finally,” but this is hardly an extensive list of all possible L-systems. The examples I gave in this chapter were adapted from L-systems I found on Wikipedia and other sites across the Web. You’ll find many different types of trees and plants and space filling curves. As long as the examples you find have the requisite parts of an L-system – the vocabulary, initiator and rules – you should be able to figure out how to fit them into this program. Realize that the specific symbols used in an example are not very important. Some of the examples I’ve given have used A and B while others have used F and x and y . It doesn’t really matter, as long as you are consistent. In general, though, you’ll

see + and - used for turning and [and] used for pushing and popping, so it's helpful to stick to that convention.

Further Explorations

Run these examples. Look at the strings and try to figure out how the rules created those particular shapes. Alter the rules and see what happens. Don't forget to look for the fractal nature of the shapes. Do some searching, find some other L-systems and adapt them for the program. Maybe even try to create your own from scratch!

One idea might be to add some new rules that change the color of the lines. Or create a user interface for entering systems without having to edit the source code each time. No lack of things that can be done starting with this simple file. Have fun!

Summary

You've made it through another chapter and learned yet another fractal-creating technique. One more technique to go – cellular automata – and then on to a look at fractals in the real world.

Chapter 11: Cellular Automata

Okay, you've made it to the last code-heavy chapter! For the next few pages, I'll be discussing cellular automata. Like the L-systems in Chapter 10, cellular automata are not solely mechanisms for generating fractals, but fractals and chaotic systems arise out of them often enough to make them worth a discussion. First, some background.

Background

The concept of cellular automata goes back to the 1940s and two mathematicians, Stanislaw Ulam and John von Neumann. Stephen Wolfram probably did more work with the subject than anyone previous, publishing an epic (and surprisingly popular) volume on the subject named *A NEW KIND OF SCIENCE* in 2002. Wolfram is also the mind behind the software package Mathematica, the Wolfram Alpha answer engine and the Wolfram MathWorld website. All amazing resources in themselves.

Initially, the concept of cellular automata, or CA for short, was explored as a means to investigate the question of whether one could create a machine that would be capable of reproducing itself. However, the subject wound up having many additional applications. Cellular automata systems contain units, or cells, that can be in certain states. Various rules are applied to the cells to change their individual states, and thus the state of the system as a whole. Even

with a few simple rules, very complex behavior sometimes results. If you’re thinking that this sounds somewhat similar to L-systems and strange attractors, you wouldn’t be far off. All of these start with an initial system that is iteratively transformed based on known rules, and can result in very complex fractal results.

Probably the most common cellular automata system is CONWAY’S GAME OF LIFE, created by the mathematician John Conway. Based on the initial configuration, various forms can arise. Some can travel, while some are self-sustaining and even self-replicating. There are even instances of a sort of symbiosis, where different forms work together to sustain themselves and create new forms. Quite fascinating, even if it doesn’t exactly fit into the subject of this book.

In the rest of this chapter, you’ll be recreating some of the patterns explored by Wolfram in A NEW KIND OF SCIENCE, which are rife with fractals, and then exploring some other systems that also occasionally produce fractal forms.

Wolfram Rules

In this section, I’ll cover a very tiny portion of the concepts presented in A NEW KIND OF SCIENCE. If the subject interests you, pick up the book. It will keep you busy for weeks, if not months.

The first cellular automata that Wolfram presents are one-dimensional systems, viewed as horizontal rows of cells. Each cell can have two states – on or off. The first row is drawn at the top of the screen with “on” cells rendered as black and “off” as white.

Each iteration looks at the state of each cell in the row and its two neighbors. It compares the state of those three cells to the current rule. A RULE in this system is a list of all possible patterns that those three cells could form and an output for each pattern. This output determines the state of the cell in the next iteration.

Applying the rule to all the cells in the row creates a new row with a different state. This newly transformed row is drawn below the first one. This continues for many iterations. Thus, the y-axis represents time, showing the changing state of the system over each iteration.

For example, say the system contains the following data initially:

0 0 1 0 0

This is a small system with only five cells. The center cell is on and the others are all off.

The rule in this case says that if either one of a cell's neighbors are on, but not both, then the cell will go to the on state. Otherwise, it will go or stay off.

Going through the row cell by cell, you see that the first cell has no ON neighbors so stays off. The second cell has one ON neighbor so goes on. The third has none, fourth has one and fifth has none. Drawing the new state below the old, you get:

0 0 1 0 0
0 1 0 1 0

Applying the rules a couple more times gives you:

0 0 1 0 0
0 1 0 1 0
1 0 0 0 1
0 1 0 1 0

You can see a pattern is starting to form. If you're not sure what it is, you will see it very shortly. Let's do some rough coding for this one. Here is the file `cellular1.js`:

```
window.onload = function() {
    var currentRow,
        nextRow,
        interval,
        y = 0;

    init();

    function init() {
        var index;

        chaos.init();

        index = Math.round(chaos.width / 2);
        currentRow = [];
        currentRow[index] = true;

        interval = setInterval(iterate, 0);

        document.body.addEventListener("keyup",
            function(event) {
                switch(event.keyCode) {
                    case 80: // p
                        chaos.popImage();
                        break;

                    default:
                        break;
                }
            });
    }

    function iterate() {
        var cell, i;

        renderCurrentRow();

        // apply rule
        nextRow = [];
        for(i = 0; i < chaos.width; i += 1) {
            if(currentRow[i - 1] && !currentRow[i + 1]) {
                nextRow[i] = true;
            }
        }
    }
}
```

```

    }
    else if(!currentRow[i - 1] && currentRow[i + 1]){
        nextRow[i] = true;
    }
}
currentRow = nextRow;
y += 1;
if(y >= chaos.height) {
    clearInterval(interval);
}
}

function renderCurrentRow() {
    for(var x = 0; x < chaos.width; x += 1) {
        if(currentRow[x]) {
            chaos.context.fillRect(x, y, 1, 1);
        }
    }
}
}
}

```

First, the variables `currentRow` and `nextRow` will be arrays that will hold the values of all the cells.

The `init` function sets the cell in the center of the row to `true` and starts an interval that will call `iterate`.

The `iterate` function first renders the current row by calling `renderCurrentRow`, which just draws a small rectangle for each cell that is on. The function then loops through each cell in the current row and checks the state of its neighbors. If the left neighbor is on and the right is off – or vice versa – the cell in `nextRow` is set to 1. Now that `currentRow` has been processed and `nextRow` holds the results, you can assign the array in `nextRow` to `currentRow` and increment `y`. If you haven't reached the bottom of the canvas, the next iteration will draw the new row to screen and do the same thing all over again.

Running this, you get Figure 11.1.

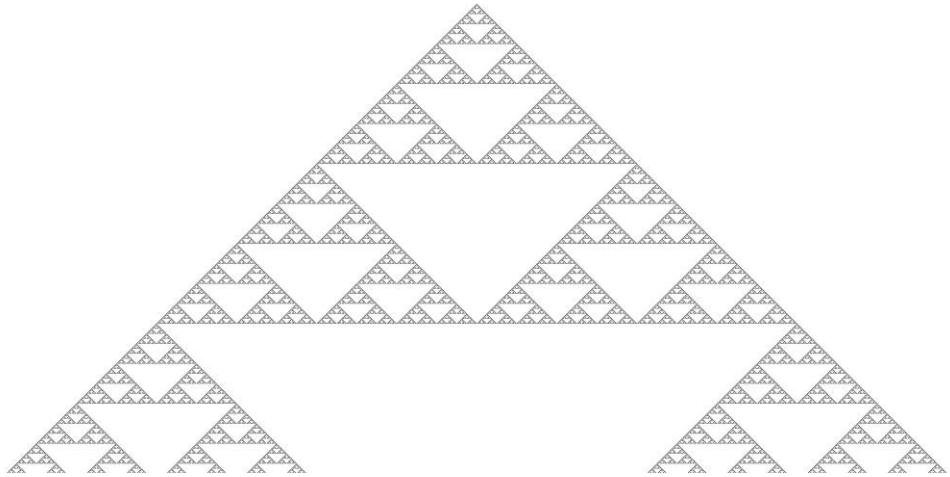


Figure 11.1. I can't believe it's another Sierpinski.

Yes, even I'm getting a little tired of this one. It seems to pop up in just about every possible method of making fractals. You'll probably see it a whole lot more in the images you render yourself throughout this chapter, but that will be the last picture of it that I'll include in this book.

More Robust Rules

Now, defining rules with textual descriptions is cumbersome and hard to program. Fortunately, there is a better way. You have three cells that you are comparing – the cell itself and the cell's two neighbors. Each of these can be in one of two states – on or off or true or false or 1 or 0. For ease of writing, let's use 1 and 0. Here are all the possible combined states of those three cells:

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0

1 1 1

Eight possible states, and if you've ever dealt with binary numbers, you'll recognize that I just wrote out the binary representations of the numbers 0 through 7. This is very convenient, as you'll soon see.

Next, for each possible state, you need to say whether a particular cell should go on or off. For example, in the example you just coded, the cell should go on if the state of it and its neighbors is the second, fourth, fifth or seventh state. Those states have a single neighbor on. The others have either two neighbors on or both off.

You could represent this rule as follows:

1	1	1	0	1	0	1	0	0	1	0	0	0	0	0	0
0	1	0	1	1	0	1	0	0	1	0	1	0	0	0	0

This shows the state of the three cells on top, and the result on the bottom. Even more simply, you could label the states as 0 through 7 and write:

7	6	5	4	3	2	1	0
0	1	0	1	1	0	1	0

If the cells are in state 6, 4, 3 or 1, the cell goes on. Otherwise, it's off. Now, you've probably noticed that I listed these in reverse order – from 7 to 0. I did it that way because this creates another binary number. An 8-bit number this time. You can now label each column for its binary value:

128	64	32	16	8	4	2	1
0	1	0	1	1	0	1	0

Add these up and you see that the binary number 01011010 is equal to decimal 90. And it's no coincidence that this is "Rule 90" in Wolfram's catalog. Thus, every rule has a number, and you can figure out mathematically what all the possible states are for that particular rule. You can also deduce that for this type of system there are 256 possible rules. In A NEW KIND OF SCIENCE, Wolfram discusses

many of these rules and gives thumbnail images for the results of all 256.

Now, let's write the more robust code that will render all these rules. This is `cellular2.js`:

```
window.onload = function() {
    var currentRow,
        nextRow,
        interval,
        cols,
        cellSize = 2,
        y = 0,
        rule = 30;

    init();

    function init() {
        var index;

        chaos.init();

        cols = chaos.width / cellSize;

        index = Math.round(cols / 2);
        currentRow = [];
        currentRow[index] = 1;

        interval = setInterval(iterate, 0);

        document.body.addEventListener("keyup",
            function(event) {
                switch(event.keyCode) {
                    case 80: // p
                        chaos.popImage();
                        break;

                    default:
                        break;
                }
            });
    }

    function iterate() {
        var state,
            i, left, center, right;
```

```

render.CurrentRow();

// apply rule
nextRow = [];
for(i = 0; i < cols; i += 1) {

    // left neighbor
    if(i == 0) {
        left = currentRow[cols - 1] || 0;
    }
    else {
        left = currentRow[i - 1] || 0;
    }

    // the cell itself
    center = currentRow[i];

    // right neighbor
    if(i == cols - 1) {
        right = currentRow[0] || 0;
    }
    else {
        right = currentRow[i + 1] || 0;
    }

    state = left << 2 | center << 1 | right;
    if(rule & (1 << state)) {
        nextRow[i] = true;
    }
}
currentRow = nextRow;
y += 1;
if(y >= chaos.height / cellSize) {
    clearInterval(interval);
}
}

function render.CurrentRow() {
    for(var x = 0; x < cols; x += 1) {
        if(currentRow[x]) {
            chaos.context.fillRect(x * cellSize,
                                  y * cellSize,
                                  cellSize,
                                  cellSize);
        }
    }
}

```

There are many changes here, but most of them are in the `iterate` function.

First, though, I added variables for `cols` and `cellSize`. These will allow you to render the cells at different sizes, not always a single pixel. Some of the patterns look better when they are zoomed in a bit. The default cell size will be 2 pixels. Also, the rule in play can now simply be specified by a number from 0 to 255.

The `init` method has not changed much. It now needs to calculate how many columns are in a row because, if `cellSize` is more than one, fewer cells will fit on the canvas. All calculations will now use `cols` where before they used `chaos.width`.

Also, there's a slight change in the `render.CurrentRow` function to allow for different-sized cells.

Now, for the serious changes. The `iterate` function renders the current row and loops through the current rows as it did before, but inside of that loop things go very differently now.

There are variables for each cell – `left`, `center` and `right`. In general, these will be indexed by `i - 1`, `i` and `i + 1`. There's one more complication. Theoretically, the world containing the cells would extend infinitely to the left and the right. But you can't have an infinite array. So the first and last cells in the row only have a single neighbor. The usual way to handle this is to have the world “wrap around.” In other words, for the first cell in the row, its left-hand neighbor is the last cell. And the last cell's right-hand neighbor is the first cell. It's not unlike many video games where, if your character goes off the screen in one direction, it reappears on the opposite side.

The not-very-elegant code to handle this is:

```
// left neighbor
```

```
if(i == 0) {
    left = currentRow[cols - 1] || 0;
}
else {
    left = currentRow[i - 1] || 0;
}
```

And similar for the right neighbor. Realize that an “on” cell will contain a value of 1, but an “off” cell will, by default, have a value of `null`, not 0. By using the logical OR operator, `||`, it forces `left`, `center` and `right` to have values of either 1 or 0.

After all those `ifs` and `else`s, you have the state of each of the three cells. You need to transform that into one of the eight states, 0 through 7, or binary 000 through 111. The next line handles that:

```
state = left << 2 | center << 1 | right;
```

The `<<` operator is the “bit-shift-left” operator. This takes the binary representation of a number and shifts its bits the specified number of places to the left. For example, if the variable `left` contains 1, in binary that is 001. Shifting that to the left two places makes it 100. If `center` contains 1, it gets shifted one place to the left, making it 010. The `right` value does not get shifted. If any of these values are 0, the shifting has no effect.

The `|` operator is the “binary OR” operator. This compares the binary representation of two numbers and does an OR on each digit. In other words, if that digit is 1 in this number OR that number it will be 1 in the output.

It’s easier to show this than describe it.

If `left`, `center` and `right` contained 1, 0 and 1, then the shifting would make them 100, 000 and 001. A binary OR is best visualized vertically:

```
100
000
001
```

101

For each column, if the value in row one OR row two OR row three is 1, the result is 1.

The result of all that is that the `state` value will contain binary 101, or decimal 5. Now that I've possibly numbed your mind with binary operators, I'll tell you that you could replace that line with:

```
state = left * 4 + center * 2 + left;
```

But, since you're dealing with binary, I decided to do it the binary way. And there are a couple more binary operators to come. The next block determines if the current state of the three cells is contained in the rule:

```
if(rule & (1 << state)) {  
    nextRow[i] = true;  
}
```

First, look at the `1 << state` part. Going with the example that the `state` variable now contains the value 5, this is shifting 1 to the left five times, which gives you 100000 in binary. It then does a binary AND with the `rule` variable. Say you are using Rule 90, which you've already seen is 01011010. A binary AND compares columns of two binary numbers, and the output for a column is 1 if BOTH numbers have a 1 in that column. Visually:

01011010	
00100000	

00000000	

Here, none of the columns has two 1s. So the result is 0. The `if` statement fails and the cell is not set to 1.

But let's say the cells you were checking had the values 1, 0 and 0. This would make `state` equal to 100 in binary or 4 in decimal. You do `1 << 4` and you get 10000. Now, the binary AND is:

01011010	
00000000	

00000000	

```
00010000  
-----  
00010000
```

Column 5 has two 1s. The output is non-zero.
The `if` statement passes, and the new cell gets assigned 1.

It's a bit complex for the brain if this is the first time you're dealing with binary, but it makes for compact code. Now, rather than defining rules with long chains of "if this and that but not the other," you just have to assign a number from 0 to 255 to the `rule` variable, and it will all just work.

Figure 11.2 shows Rule 30, one of Wolfram's favorites, rendered out.

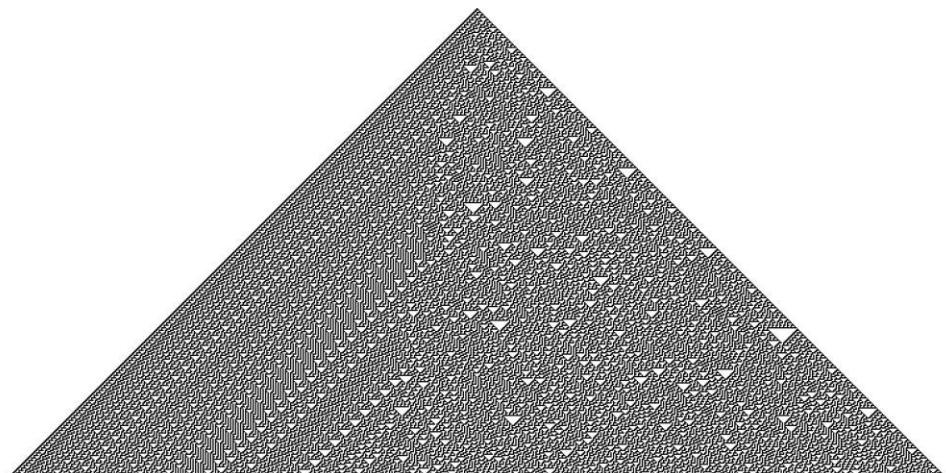


Figure 11.2. Order and chaos.

This is an interesting image, indeed. To fully appreciate this one, you'll probably want to run the code and see it full screen. The image has many of the same characteristics you learned about in Chapter 10 on chaos theory and strange attractors; it's created by deterministic rules, and yet there are areas of complete chaos, interspersed with areas of order. The left side has an obvious pattern, but the right side is completely random. But it's not just random dots; there are hundreds of various-sized blank triangles

appearing like little windows of order. No matter how long and how many rows you rendered here, there would be no discernible repeating pattern. There's also an obvious self-similarity in the large and small triangles that are continuously repeated.

Another interesting rule is Rule 124. This will produce Figure 11.3, which is almost reminiscent of diffusion-limited aggregation from Chapter 6.

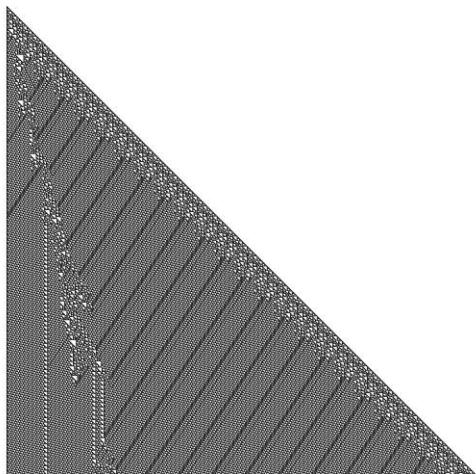


Figure 11.3. More chaos.

With this code, you can specify any of the 256 rules and see what they look like for yourself. But I'll give you a quick verbal preview. You're going to see many straight and diagonal lines, a lot of pyramids with regular patterns, a bunch more Sierpinskis in various orientations, a few other interesting fractal shapes and some chaotic patterns.

At first, you might think 256 rules is a lot, but it won't take long before you've seen most of what there is to see. So let's create a system that supports more rules.

More Neighbors

Using one cell and two neighbors, you are limited to eight possible states, which creates a maximum of 256 rules. What will happen if you expand your concept of neighbors to include, not only directly adjacent cells, but the cells adjacent to those as well?

This will give you five cells, each with two states, for a total of 32 states instead of just eight. If you're not familiar with how binary works, you might be surprised at how many rules this makes possible. With eight states, you can use an 8-bit number, which has 256 possible values. With 32 states, you have a 32-bit number, which gives you ... wait for it ... 4,294,967,296 possible rules. Yeah, over four billion. That should give you some variety!

There's not too much that has to change to support this. Here's the code from `cellular3.js`:

```
window.onload = function() {
    var currentRow,
        nextRow,
        interval,
        cols,
        cellSize = 2,
        y = 0,
        rule = Math.floor(Math.random()
                          * Math.pow(2, 32));

    console.log("rule: ", rule);

    init();

    function init() {
        var index;

        chaos.init();
        cols = chaos.width / cellSize;
```

```

index = Math.round(cols / 2);
currentRow = [];
currentRow[index] = 1;

// uncomment to enter a specific rule number
// rule = 666987049;

interval = setInterval(iterate, 0);

document.body.addEventListener("keyup",
    function(event) {
        switch(event.keyCode) {
            case 80: // p
                chaos.popImage();
                break;

            default:
                break;
        }
    });
}

function iterate() {
    var state,
        i, left2, left1, center, right1, right2;

    renderCurrentRow();

    // apply rule
    nextRow = [];
    for(i = 0; i < cols; i += 1) {
        if(i == 0) {
            left2 = currentRow[cols - 2] || 0;
            left1 = currentRow[cols - 1] || 0;
        }
        else if(i == 1) {
            left2 = currentRow[cols - 1] || 0;
            left1 = currentRow[0] || 0;
        }
        else {
            left2 = currentRow[i - 2] || 0;
            left1 = currentRow[i - 1] || 0;
        }

        center = currentRow[i];

        if(i == cols - 1) {
            right1 = currentRow[0] || 0;
        }
    }
}

```

```

        right2 = currentRow[1] || 0;
    }
    else if(i == cols - 2) {
        right1 = currentRow[i + 1] || 0;
        right2 = currentRow[0] || 0;
    }
    else {
        right1 = currentRow[i + 1] || 0;
        right2 = currentRow[i + 2] || 0;
    }

    state = left2 << 4 |
            left1 << 3 |
            center << 2 |
            right1 << 1 |
            right2;

    if(rule & (1 << state)) {
        nextRow[i] = true;
    }
}
currentRow = nextRow;
y += 1;
if(y >= chaos.height / cellSize) {
    clearInterval(interval);
}
}

function render.CurrentRow() {
    for(var x = 0; x < cols; x += 1) {
        if(currentRow[x]) {
            chaos.context.fillRect(x * cellSize,
                                  y * cellSize,
                                  cellSize,
                                  cellSize);
        }
    }
}
}
}

```

First of all, I doubt you want to enter long random numbers up to four billion by hand each time. So the rule number is now randomized at the start of the program.

```
rule = Math.floor(Math.random() * Math.pow(2, 32));
```

The rule number is then logged in case you see an image you like and want to note down the rule that created it. In

the `init` function, you can set a specific rule before setting the interval, and that rule will be used instead of the randomly generated one.

The rest of the changes are in the `iterate` function. Instead of `left` and `right`, there's

now `left2`, `left1` and `right1`, `right2` to cover two neighbors on each side instead of one. The wrap-around logic is a bit more complex as well, because there could be one or two neighbors wrapping around on each side. Finally, the shifting and ORing that calculates the `state` value now deals with two neighbors on each side, combining a total of five bits into a 32-bit number. The rest works all the same.

Fire it up and start refreshing your browser. You'll still see many single lines, repeating patterns and, yes, more Sierpinskis, but you'll see some really interesting stuff in there as well. Figures 11.4, 11.5 and 11.6 show a few interesting ones I ran across.

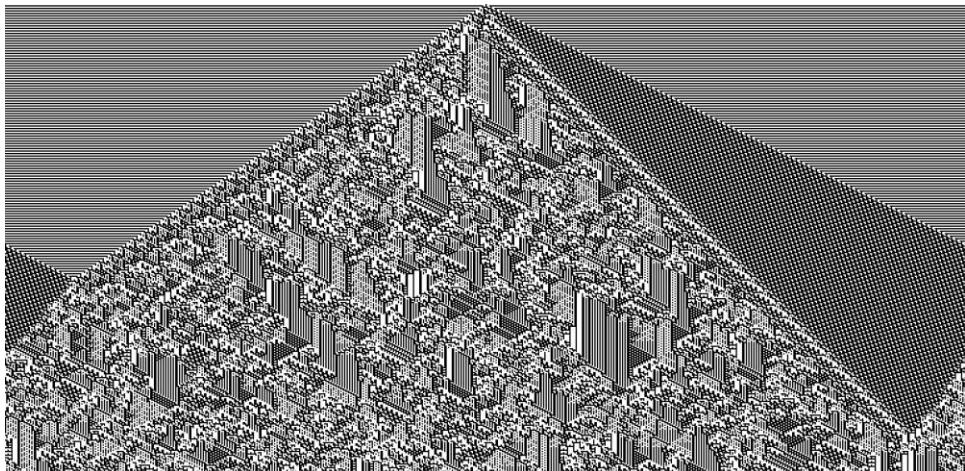


Figure 11.4. Rule 988197457.

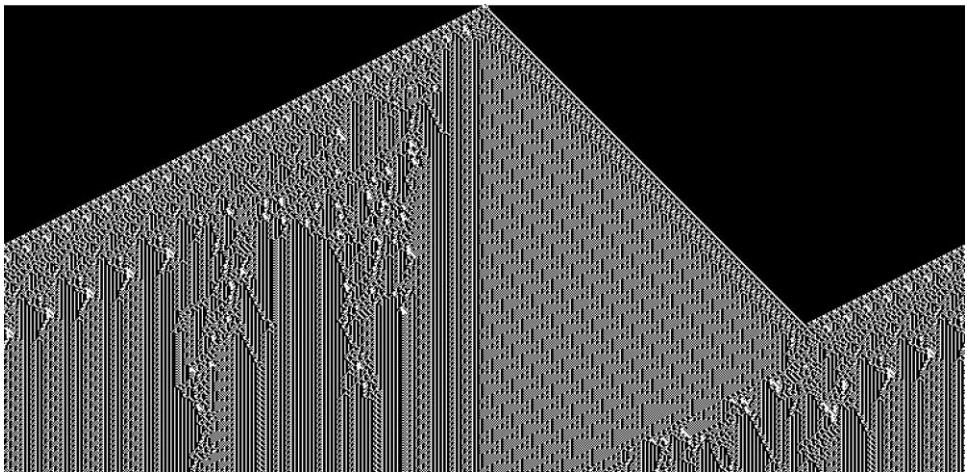


Figure 11.5. Rule 2530535241.

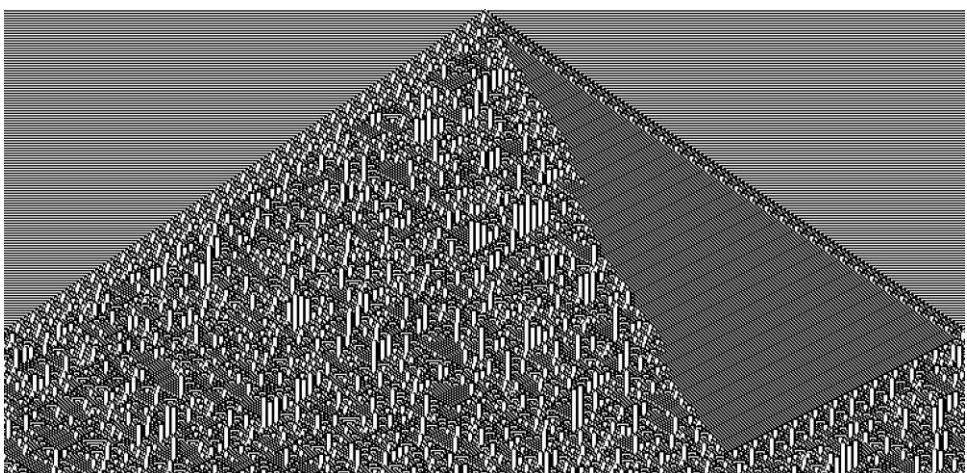


Figure 11.6. Rule 2123739367.

A lot of these seem almost to be 3D scenes that remind me of old isometric video games like Sim City or Populous. Anyway, there you have three rules. I'll leave the other 4,294,967,293 for you to explore. Don't get too caught up in it, though – there's more to explore in this subject before you're through here.

2D Cellular Automata

Although the images you created so far in this chapter have been 2D images, they are actually created with 1D cellular automata. Remember that the y-axis represents the change in the system over time. In this section, you'll create actual 2D systems.

First, let's look at all the different ways of classifying cellular automata.

Classifying Cellular Automata

First, there is the dimension. You've already seen one-dimensional CA. This is a single row of cells, and you can use another dimension to represent time. In a two-dimensional CA, you'll have a 2D grid of cells. Time will not be represented as such. You'll just have to update and redraw the entire grid on each iteration. This allows for animation. Conway's Game of Life is an example of this.

There are, of course three-dimensional cellular automata, with a 3D grid of cells. That's well beyond the scope of this book, but worth looking into if you're interested.

After dimensionality, there is the number of possible states for each cell. The examples you've worked with so far have had two states – on and off. The rest of the examples in the book will also be two-state examples. But it's definitely possible to have cells that can be in any number of possible states. Von Neumann's first cellular automata had cells with 29 possible states!

Finally, there is the neighborhood factor. Actually, there are multiple factors here. In the 1D fractals, you've considered directly adjacent neighbors and neighbors expanded out to two places. It's possible, of course, to expand out further, but realize that three places on either side would give you seven cells capable of 128 possible states, resulting in 2^{128} or

340,282,366,920,938,463,463,374,607,431,768,211,456 possible rules. (That's 340 undecillion.) Go for it!

In 2D systems, you can consider the eight cells located directly around a particular cell, or the 16 surrounding those as well. Most 2D CAs use only direct neighbors. As seen below, visually, the A is the cell and the Bs are its neighbors:

B B B
B A B
B B B

Of course, one cell plus eight neighbors means 512 possible states, resulting in an even more astronomical number of rules. So usually 2D CA uses what are called "totalistic" rules. Totalistic means that you don't care WHICH cells are on; you only care HOW MANY cells are on. So for nine cells you could have from zero to nine cells on. That's 10 states and 1,024 rules, which is far more manageable.

But the usual variation of totalistic rules is called "outer totalistic." This means you are mostly concerned with the state of the neighbors. That's 9 possible states (zero to eight). But you then consider whether the center cell is on or off. That gives you a total of 18 states, which is 2^{18} or 262,144 rules. A decent variety there.

And there's even another variation of neighborhoods. When you take all eight neighbors into account, that is known as a "Moore neighborhood," named after Edward Moore, a

computer science and mathematics professor. Then, there's the "von Neumann neighborhood," which only considers orthogonal neighbors, like so:

```
B  
B A B  
B
```

That's five cells, allowing for 32 states and four-billion-something rules if you count all of the possibilities. Or, if you go totalistic, six states (zero to five) or 64 possible rules. And outer totalistic gives you five states with the center on, five with it off for 10 total states and 1,024 rules.

Whew! A lot of different types of cellular automata, eh? Where do you even start? I'll go through a simple one with you and then leave you to explore more complex systems on your own.

Coding 2D Cellular Automata

The example I'm going with is a two-state outer totalistic von Neumann neighborhood cellular automaton. I just love saying that. Makes me feel smart. This means:

1. Each cell can be on or off.
2. The state is determined by the total neighbor count, plus the state of the center cell.
3. The neighbors considered will be the top, bottom, left and right only.
4. Four neighbors gives you five states (zero to four) when the center cell is on and five when it is off. Ten total states = 2^{10} or 1,024 possible rules.

The code in `cellular4.js` has some significant changes, as it now needs to render the entire grid on each iteration. Because of this, one change I've made is to remove the interval and replace it by a key handler. The `iterate` function will be called when you hit the space bar:

```
window.onload = function() {
    var currentGrid,
        nextGrid,
        cols,
        rows,
        cellSize = 5,
        rule = Math.floor(Math.random() * 1024);

    console.log("rule: ", rule);

    init();

    function init() {

        chaos.init();

        // uncomment to set a specific rule
        // rule = 50;

        cols = Math.floor(chaos.width / cellSize);
        rows = Math.floor(chaos.height / cellSize);

        currentGrid = createGrid();

        currentGrid[Math.round(cols / 2)][Math.round(rows / 2)] = 1;

        renderCurrentGrid();

        document.body.addEventListener("keyup",
            function(event) {
                switch(event.keyCode) {
                    case 80: // p
                        chaos.popImage();
                        break;

                    case 32: // space
                        iterate();
                        break;
                }
            }
        );
    }
}
```

```

        default:
            break;
    }
})
}

function iterate() {
    var state,
        x, y;

    renderCurrentGrid();

    nextGrid = createGrid();

    for(x = 0; x < cols; x += 1) {
        for(y = 0; y < rows; y += 1) {
            state = getState(x, y);
            if(rule & (1 << state)) {
                nextGrid[x][y] = 1;
            }
        }
    }
    currentGrid = nextGrid;
}

function createGrid() {
    var grid = [];
    for(var i = 0; i < cols; i += 1) {
        grid[i] = [];
    }
    return grid;
}

function getState(x, y) {
    var state = 0;
    // left
    if(x == 0) {
        state += currentGrid[cols - 1][y] || 0;
    }
    else {
        state += currentGrid[x - 1][y] || 0;
    }
    // right
    if(x == cols - 1) {
        state += currentGrid[0][y] || 0;
    }
    else {
        state += currentGrid[x + 1][y] || 0;
    }
}

```

```

    }
    // top
    if(y == 0) {
        state += currentGrid[x][rows - 1] || 0;
    }
    else {
        state += currentGrid[x][y - 1] || 0;
    }
    // bottom
    if(y == rows - 1) {
        state += currentGrid[x][0] || 0;
    }
    else {
        state += currentGrid[x][y + 1] || 0;
    }
    if(currentGrid[x][y]) {
        state += 5;
    }
    return state;
}

function renderCurrentGrid() {
    chaos.clear();
    for(var x = 0; x < cols; x += 1) {
        for(var y = 0; y < rows; y += 1) {
            if(currentGrid[x][y]) {
                chaos.context.fillRect(x * cellSize,
                                      y * cellSize,
                                      cellSize,
                                      cellSize);
            }
        }
    }
}
}

```

The variables have changed a bit, as you'll need grids instead of rows. The `init` function has largely the same functionality. Because you'll be creating grids often, there's a `createGrid` function that returns an empty 2D array. Again, the interval is removed, replaced by a key handler for the space bar. The center cell in the grid is turned on.

The `iterate` function is surprisingly simple. It renders the current grid and creates the next grid. Then, it loops through

every cell in the grid, calling `getState` for that particular cell. If that state is in the current rule, it sets the cell on in the next grid.

The `getState` function uses some brute force logic to get a count of the neighbors in the on state, taking care of wrapping around as needed. Remember that cells that are not on will generally have a null value rather than 0, so the code uses `|| 0` to make sure you are adding either a 0 or 1 to the state. When the neighbors are all counted, you'll have a value from 0 to 4 in the `state` variable. If the center cell is on, another 5 is added to that. So it's 0 to 4 for center cell off and 5 to 9 for center cell on. A total of 10 states.

Run this one and press the space bar a few times. You'll see a pattern form and start to grow. Or maybe not. Many of the rules result in a black or white screen or maybe a single cell in the center of the screen. Refresh your browser to try another random rule. Eventually, you'll find something interesting. Some examples I found are in Figures 11.7, 11.8 and 11.9.

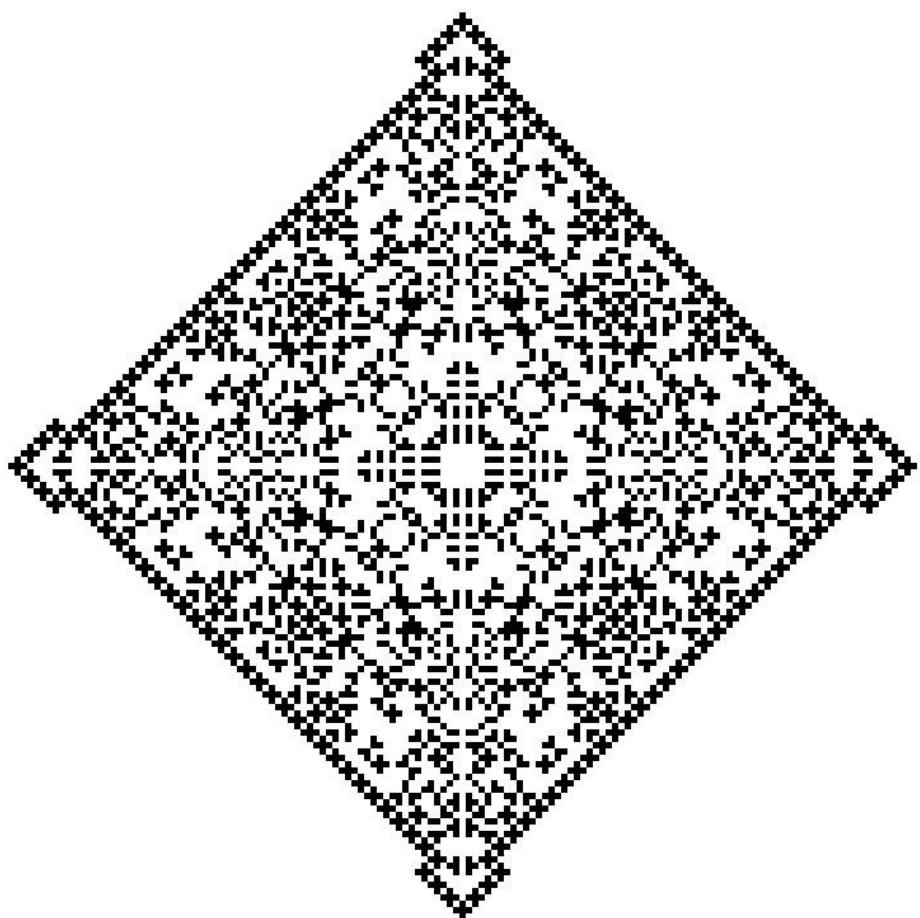


Figure 11.7. Rule 50.

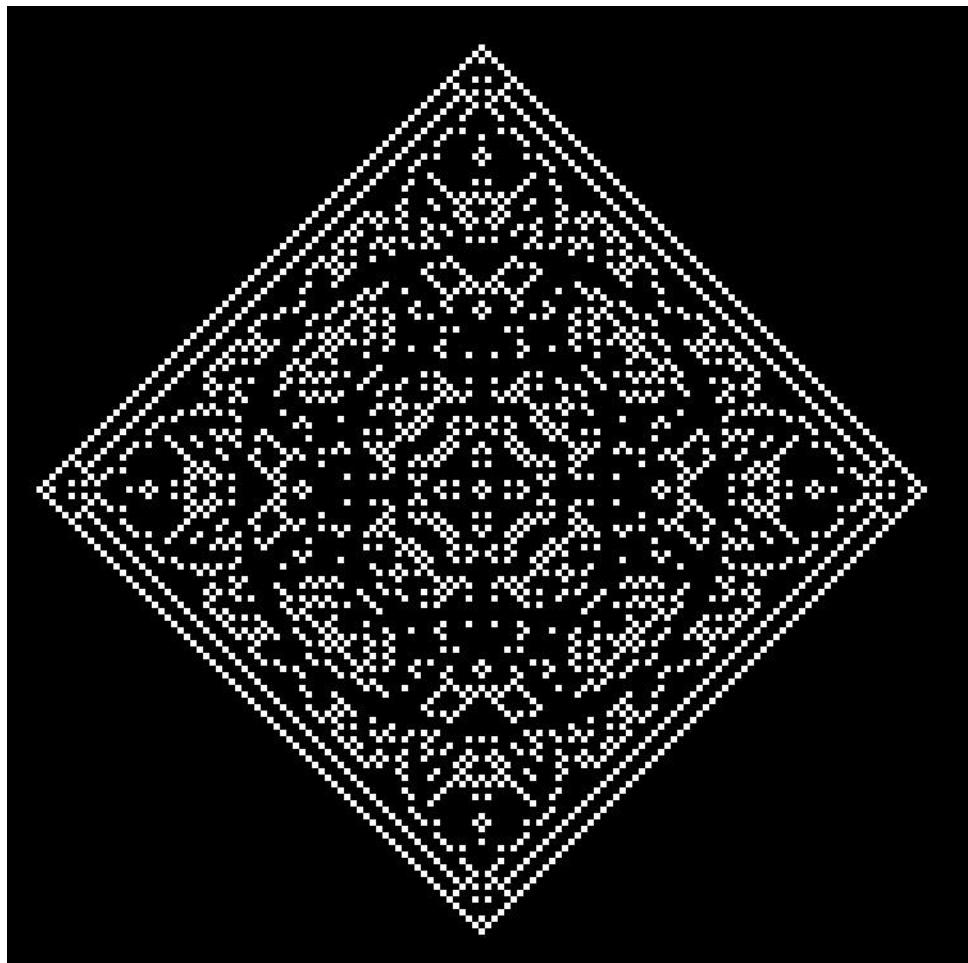


Figure 11.8. Rule 621.

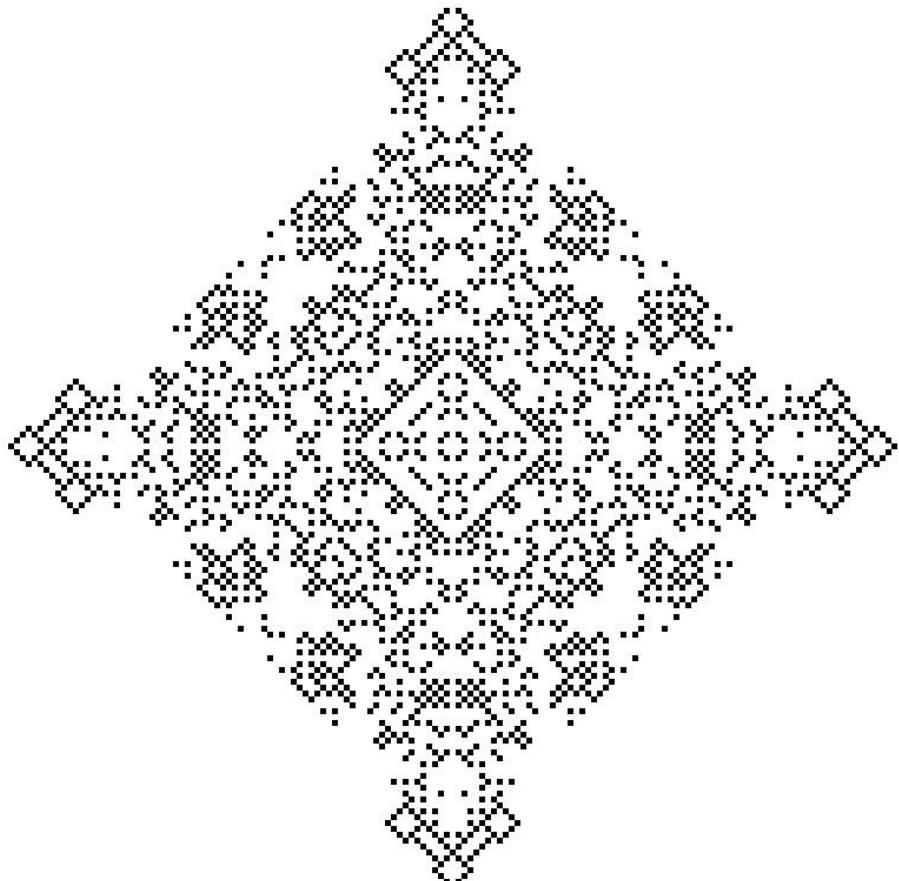


Figure 11.9. Rule 210.

Of course, not all of the images you get are necessarily fractals. I'm not even sure exactly how you would classify which ones are and are not fractals, but I'd say some of them obviously have some self-similarity going on. Even if they aren't strictly fractals, they are interesting images to explore.

Now that you know the lay of the land, you might want to convert the last example so that it uses Moore rather than von Neumann neighborhoods. Mainly, you'd have to change the `getState` function to count all eight Moore neighbors instead of just the four von Neumann neighbors.

If you're very ambitious, you might try creating a system that has more than two states. If so, remember to start small and simple. These things can get out of hand quickly.

Summary

In this chapter, you learned about different types of cellular automata and how they are classified. And you have seen that they can sometimes lead to fractal forms. As I said earlier, this chapter barely scratches the surface of the subject, which is vast and extends far beyond fractals. I hope you find it interesting enough to explore further.

Chapter 12: Fractals in the Real World, and Resources

Fractals in the Real World

CLOUDS ARE NOT SPHERES, COASTLINES ARE NOT CIRCLES, BARK IS NOT SMOOTH, NOR DOES LIGHTNING TRAVEL IN STRAIGHT LINES.

- Benoît Mandelbrot, THE FRACTAL GEOMETRY OF NATURE

This will be a short chapter to wrap up what I've covered in the rest of the book and send you off to explore some more. Throughout the earlier chapters, you learned many techniques for creating fractal images. I've given you a few suggestions here and there for how to expand or change the programs, other areas to explore, techniques to try, etc. But one of the best sources of inspiration for fractals is the world around you.

As Benoît Mandelbrot pointed out, fractals are the geometry of nature. With occasional exceptions, it's easy to spot a man-made object in nature. It's the thing that has the straight edges or the perfectly rounded corners. It's symmetrical, regular and smooth.

Natural shapes have more complexity. At first, this complexity may look like chaos in the sense of uncontrolled randomness. When you look closer, though, you start to see the self-similarity on different scales. The boulders look like the rocks that look like the pebbles that look like the grains of sand that probably look like the particles of dust.

Of course, living things often have even more obvious patterns. I don't need to bombard you with pictures of trees, plants and ferns to demonstrate to you that these things have a fractal nature. If you want to have fun though, search for images of Romanesco broccoli, perhaps one of the most striking fractal plant forms you can find.

Another way to explore fractal images is Google Earth, or any map program that shows satellite images. Figure 12.1 is an image from Egypt. You could almost be convinced that this was generated with a Mandelbrot rendering program.



Figure 12.1. Egypt or Mandelbrot?

You also have multiple fractal systems in your own body. Your veins and arteries are one example. They branch out much like the fractal trees you created in this book, from the large vessels leading in and out of your heart to the many tiny capillaries connecting to every part of your body. It's estimated that this fractal branching results in around 100,000 miles of blood vessels in an adult human.

Your lungs are another fractal system. In order to transfer gases, such as oxygen and carbon dioxide, in and out of your body, your lungs need as much surface area as possible. Their fractal structure results in 1,500 miles of airways and 750 square feet of surface area. Not bad for a five pound organ.

If you have ever considered the incredibly complex structures of the human body (or any other living thing for that matter) and wondered how all that complexity could have been encoded into the couple of cells that a body starts out with, fractals just might provide one answer. You've already seen how the single-line formula for the Mandelbrot set can result in an infinitely zoomable image. Or a few initial symbols and rules in an L-system can grow into a large, complex, ordered structure. It doesn't seem like too much of a stretch to consider similar things at work in the growth of a human body.

You can find fractal patterns in surprising places, such as the shells of sea creatures. The conus textile, a type of extremely venomous sea snail, has a pattern of triangles on its shell that bears a surprising likeness to the cellular automata patterns you created in Chapter 11. In fact, some have likened this pattern specifically to Wolfram's Rule 30. You can see this pattern in Figure 12.2.



Figure 12.2. Sea snail or Rule 30?

Electricity can also demonstrate fractal properties. The jagged branching of lightning is an example, but it can be difficult to get a good picture of this. A Lichtenberg figure is a fascinating way to view the fractal branching of electricity. This is the name given to the pattern sometimes formed when high-voltage electricity, such as lightning, hits some solid object. The German physicist Georg Lichtenberg studied these figures in the late 1700s. Lichtenberg figures can appear where lightning hits sand, grass or other surfaces. They can even appear on the skin of people who have been struck or nearly struck by lightning.

Rather than waiting around for lightning to strike something (or someone) interesting, people now create such figures by shooting high-voltage electricity into blocks of acrylic or other materials. You can buy these online in all shapes and sizes. Many of them are quite beautiful. Figure 12.3 shows a photograph of a Lichtenberg figure I own.



Figure 12.3. Lichtenberg figure.

Electricity consists of negatively charged electrons. As the charge flows into the acrylic, it melts the acrylic, forming the pattern. The electrons also repel each other and seek areas where there is a lesser negative charge. This causes the branching action. It is thought that this fractal branching continues down to an atomic level. In Figure 12.4, you can see a macro shot of my Lichtenberg figure.



Figure 12.4. Lichtenberg figure magnified.

Can you see the resemblance to a fractal tree? You might even consider that it resembles the diffusion-limited aggregation example.

I could go on for pages with additional examples, but once you've programmed a few fractals, understand the principles and learned to see with a "fractal eye," I'm sure you'll start seeing them all over the place yourself.

Uses for Fractals

In addition to naturally occurring fractals, man-made fractals are being used for all sorts of purposes.

One of the most obvious applications is the use of fractal images and forms in graphic design and in special effects in video games and movies. There are many landscape generation programs on the market. These almost always use fractal landscape-generation and populate the land with fractal foliage and trees. Textures that are mapped onto 3D objects are often generated with fractal programs as well.

You became acquainted with the work of Michael Barnsley in Chapter 5. As I mentioned there, Barnsley was also a pioneer in the field of fractal image compression. This is a method of reducing the size of images using fractals. A complex image may have many similar parts. It's possible to use this fact to reduce the picture to a smaller data set that can later be reassembled fractally to the original image. This method has even been used to reduce the size of video.

Another use of fractals is radio antennas. The length of an antenna in relation to the wavelength of signal it is designed to receive is very important. But it is not always practical to make an antenna the size that it needs to be. By using a fractal space-filling curve, like some of the shapes you created with L-systems in Chapter 10, you can fit quite a large length of antenna into a small space, such as the smartphone in your pocket right now. Figure 12.5 shows an example.

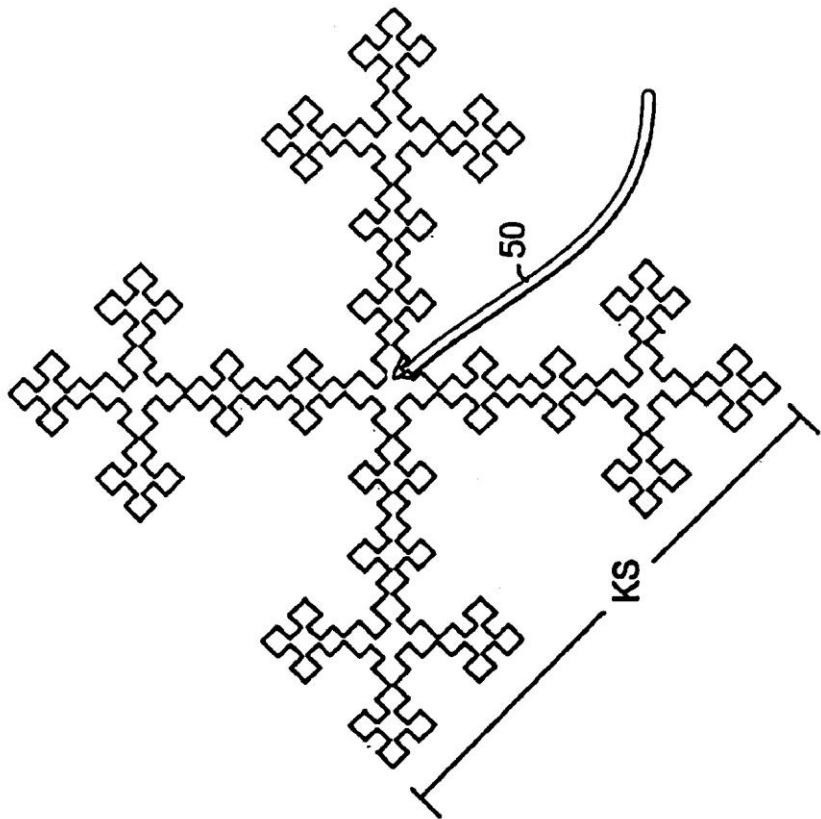


Figure 12.5. Fractal antenna.

In the fields of science, medicine and engineering, there are countless uses for fractals. Many of these involve creating fractal models of systems for analysis or testing.

Resources

If you've read this whole book and have coded and run all or most of the examples, you now have a strong general

overview of the subject of fractals and chaos. But this book should just be your jumping-off point.

Below are a few resources that I found very helpful while writing this book. I think you will find them equally as useful in your further exploration of the subjects.

Books

THE FRACTAL GEOMETRY OF NATURE by Benoît Mandelbrot

One of the books that started it all, by the father of fractals himself.

CHAOS AND FRACTALS: NEW FRONTIERS OF SCIENCE by Peitgen, Jürgens and Saupe

A massive book, full of images, diagrams and formulas, covering just about every aspect of fractals and chaos. This is definitely one to have around if you're serious about the subject.

CHAOS: MAKING A NEW SCIENCE by James Gleick

One of the more popular books on the subject, CHAOS explores the history and theory of chaos and fractals. You won't learn how to create fractals with this book, but if you want to understand more of the background and concepts of the subject, this is a must-read.

STRANGE ATTRACTORS: CREATING PATTERNS IN CHAOS by Julien C. Sprott

This book is now out of print, but you can get a free electronic copy on the author's website: <http://sprott.physics.wisc.edu/sa.htm> If Chapter

9 caught your attention, this is the book for you. It's the ultimate resource on generating strange attractors. In addition to all the theory and images, the book contains full source code for the programs that will generate the images. The code is in BASIC, but it shouldn't be too hard to port to the language of your choice.

A NEW KIND OF SCIENCE by Stephen Wolfram

Of course, this is the book that the first half of Chapter 11 is based on. If you don't have the room for such a large book in your house, you can also read it free online at <http://www.wolframsience.com/>

Websites

CLIFFORD PICKOVER'S

SITE <http://sprott.physics.wisc.edu/pickover/home.htm>

There's some far out stuff here, but plenty of fractal references as well. Definitely worth digging around in this site.

PAUL BOURKE'S FRACTAL

PAGE <http://paulbourke.net/fractals/>

This site was mentioned more than once in the book. If you haven't been there yet, go now.

SPROTT'S GATEWAY <http://sprott.physics.wisc.edu/>

Sprott is the author of the book on strange attractors mentioned just previously. There are many other fractal resources listed on his page.

YALE'S FRACTAL

COURSE <http://classes.yale.edu/fractals/>

A fantastic resource for just about all aspects of chaos theory and fractals. This site is very clearly written and understandable. I found myself coming back to it again and again while writing this book.

WOLFRAM

MATHWORLD <http://mathworld.wolfram.com/>

One of my go-to sites for anything math-related, including fractals. A huge wealth of information on fractals or anything else involving numbers.

WIKIPEDIA <http://www.wikipedia.org/>

This may seem like a no-brainer, but Wikipedia is usually my first stop when researching any subject. For fractals and chaos, the explanations are often very academic and usually go a bit over my head, but it's always a good starting point, and usually articles contain links to other useful resources.

Summary

Well, now we part ways. I hope you've had fun reading this book and have learned a few things. Most of all I hope it has sparked an area or two of interest in you and that you will continue to learn more on these subjects. Enjoy!