Introduction to Chaos, Fractals and Dynamical Systems

By Phil Laplante, PhD

Introduction to Chaos, Fractals and Dynamical Systems

Phil Laplante, PhD

June 2020

Forward for Reprinted Edition

Originally published as *Fractal Mania* in 1993, this book included a distribution disk with code samples and an iron on patch. This was my fourth book and I am still grateful to McGraw-Hill and publisher Roland Phelps for supporting me when I was still relatively unknown.

When *Fractal Mania*[1] was published, chaos theory and fractals were raging, even influencing popular literature and movies (e.g. Jurassic Park). Within a few years, however, these ideas largely faded from public attention. I also eventually lost interest in this line of research.

I had intended the book for self-study or use in appropriate middle or high school courses, but never learned if there was widespread adoption as such. The sales figures presented to me suggest that the book was mostly sold one or two copies at a time to individuals. It was successful enough, however, to be translated and reprinted in Japanese, Hebrew and Standard Chinese. After several printings, and many years, McGraw-Hill chose not to republish the book. Under the terms of our contract the copyrights reverted back to me upon request. But I wasn't sure what to do with the book.

Recent world events have made me reconsider the importance of understanding chaotic systems, second order effects and the unintended consequences of human actions – topics which were teased. I considered reconstructing *Fractal Mania* around those topics and releasing it with a different publisher. Instead, I am making it available for free.

The book is suitable for homeschooling or as a classroom supplement. Or, it could be used for informal family study and discussion. I hope adults will find some value in the information contained herein and their interest in dynamical and chaotic systems piqued for further exploration.

Writing and Code

In retrospect I can see the evolution and improvement in my writing over 30 years. *Fractal Mania* was written to reach a pre-College audience and the copy editor made sure that level was enforced, but my own writing was also less sophisticated then. But I think the writing is quite accessible and understandable to almost anyone, and that was my intention all along.

While all the code samples in the book are in Pascal, C code equivalents are available from my Website at https://phil.laplante.io/resources.php. The Pascal code is not offered in electronic form but could easily be extracted from the PDF file. Use these at your own risk.

Future Editions and Engagement

Depending on the response I get to this version, I may rewrite and expand the book significantly. I welcome your suggestions for future editions at phil.laplante.io.

[1] I never loved this title – it was determined by the publisher.

I am available for a customized, virtual talk to your company, group, class, etc. on Chaos Theory, Dynamical Systems and other topics discussed in this book and related items. Contact me for fee structure and availability at phil.laplante.io.

Other Books

My complete set of published books can be found here:

https://phil.laplante.io/books.php

One particular recommendation is:

Phillip A. Laplante, *Technical Writing: A Practical Guide for Scientists, Engineers and Nontechnical Professionals, Second Edition*, CRC Press/Taylor & Francis Publishing, 2019.

Copyright, Disclaimers and Permissions

To my mother, Edith.

Dedication

Contents

Appendices

Acknowledgments

I'd like to thank the following individuals for their critical reading of the manuscript at various stages. However, errors are inevitable, and I take full responsibility for them.

- Prof. Kathryn Douglas of Fairleigh Dickinson University.
- Dr. Charles Giardina of CUNY—Staten Island.
- Prof. Marvin Goldstein of Fairleigh Dickinson University.
- Prof. Diane Richton of Fairleigh Dickinson University.
- Prof. Michael Scanlon of Fairleigh Dickinson University.
- Dr. Richard Segers.
- Dr. Constantine Stivaros of Fairleigh Dickinson University.
- Dr. Paul Strauss of Fairleigh Dickinson University.

Many thanks to John Cannon of the Fairleigh Dickinson University Academic Computer Center for his assistance in the generation of the plates, and to my student, Frank D'Erasmo for generating some of the code, for proofreading, and for valuable discussions.

Finally, thanks to my wife, Nancy, for her patience and support on this project.

Introduction

The purpose of this book is to introduce fractals and chaos theory to those with no more formal mathematical training than basic algebra, geometry, and perhaps some trigonometry. The emphasis is on natural and human-made phenomena that can be modeled as fractals and on the applications of fractals to computer-generated graphics and image compression. Because I keep the mathematics to a minimum, I rely on intuitive descriptions, computer-generated graphics, and photographs of natural scenes to make my points. I also present a brief history of the evolution of fractal and chaos theory. For those with access to an IBM-compatible personal computer, the book includes a diskette with executable programs and source code illustrating most of the concepts described in the text.

Mathematical background

I assume that you have a basic knowledge of algebra and geometry. In particular, you should be familiar with functions of real numbers, and it would also be helpful if you were familiar with a little trigonometry such as sines and cosines. However, I develop most of the needed mathematical background along the way.

Organization & flexibility

While this text is primarily intended for self-study, it could be used to supplement several courses at the high-school level. For example, this book could be used to supplement the following courses:

- Pascal Programming
- Precalculus
- Geometry
- Computer Graphics

The text can also be used in mathematics courses for undergraduates who are not science majors.

How to use the programs on the disk

The disk included with this book contains all of the programs described in the text. These programs were written using Borland's Turbo Pascal 5.5 compiler, and they require an IBM-compatible PC with an Intel 80286 or superior processor and an EGA or VGA monitor. The code is written to take advantage of a numeric coprocessor if it's available.

For a quick demonstration of the programs, place the disk in drive `a:` of your system (or another appropriate drive), and type:

```
a:
```

Hit Enter to make `a:` the active drive. Then type:

```
demo
```

Hit Enter. Now sit back and enjoy the show. Depending on the speed of your computer, the demonstration could take 5–15 minutes. You can abort the demo by pressing the Ctrl-Break key combination.

To run the programs, create a directory called "FRACTAL" by entering the following command from the root directory of your hard disk. (Be sure your logged drive is the hard disk.)

```
mkdir FRACTAL
```

Change directories to this new one by entering the command:

```
cd fractal
```

Copy the programs from the enclosed disk by entering the command:

```
copy a:*.*
```

In the previous command, `a` is assumed to be the drive containing the distribution diskette included with the book. You'll have to be in the FRACTAL directory to run the programs, unless you want to put the FRACTAL directory in your path (see your DOS manual for directions). To run the executable program corresponding to the source of a particular program, say JULIA1.PAS, simply type:

```
JULIA1
```

Modifying the programs

If you have Turbo Pascal version 5.5 or greater, you can modify the programs and have even more fun. (The programs might work with older versions of Turbo Pascal, but the programs were not tested in these environments). In many cases, especially with the programs that display Julia and Mandelbrot sets, changing one line will lead to vastly different and fascinating results. I strongly encourage you to study the programs and play with them.

If you don't have a compatible version of Pascal, or if you have an incompatible monitor, it should be relatively easy to modify the programs to run in your particular environment. Furthermore, if you're using another structured programming language, such as C or Ada, most of these programs should easily translate. If you program in BASIC, the code should serve as an easy-to-follow specification so that you can rewrite the routines.

I didn't build extensive error checking into the programs, nor did I try to make them too clever. I wanted to keep them short but simple enough to encourage you to look at them critically and modify them as desired. You can add error checking if you like.

Finally, depending on your computer, many of these programs (especially those that produce Julia sets) will run very slowly and could possibly take

hours to produce the final image. Although I could have optimized these programs to run faster, the result would have been difficult to follow. I opted to trade fast performance for code that's clear and readable. Also, indeed, the slow performance of some of the programs aptly demonstrates the inherent trade-offs between the memory required to store a pixel image and the time needed to regenerate that image using compression techniques, which the fractal programs represent. I encourage you, however, to experiment with the programs and try to optimize them by taking advantage of symmetry and mathematical tricks. In doing so, you can gain a better understanding of the underlying phenomena.

FRACTINT

Although I've provided you with code to generate many different kinds of fractals, a free program is available that will generate many other fractal images. The program, FRACTINT, was written and is distributed by the STONE SOUP GROUP. The FRACTINT executable program, documentation, and even source code are available on CompuServe in the "fractals" library of the COMART forum. To get more information from CompuServe, call (800) 848-8199.

Disclaimer

I make no guarantees for the performance of the supplied program code on any given machine.

1 What is chaos? What are fractals?

"First there was Chaos, the vast immeasurable abyss,
Outrageous as a sea, dark, wasteful, wild."

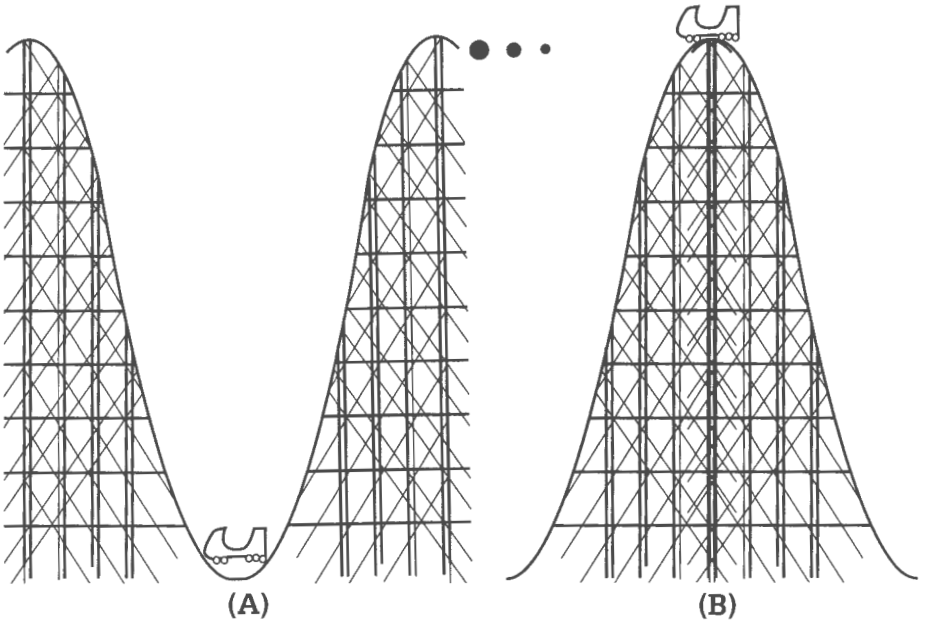
— John Milton, *Paradise Lost* (1674)

In this chapter I look at chaos, instability, stability, and other ideas that relate to fractals. I define the term fractal, show some early examples, and look at the history of fractals, chaos, and dynamical systems.

Stable/unstable systems

Consider the section of infinite roller coaster shown in FIG. 1-1A. The car is located in the trough and is still. If you shove the car gently in either the front or the back, it's clear that friction, gravity, and rotational kinetics will act to return the car to the trough. Within certain limits, it doesn't matter how far you push the car to the right or the left, the car consistently returns to the same place. This system is said to be in *stable equilibrium*.

Now consider the section of the same roller coaster shown in FIG. 1-1B. In this case, if you gently shove the roller coaster in the front or the back, the car begins a wild ride, and it's unclear where the car stops. This system is said to be in *unstable equilibrium*. The concepts of stable and unstable equilibrium, as well as sensitivity to initial conditions (in this case where the car starts), are crucial in the study of fractals and chaos.



1-1 *Two sections of the infinite roller coaster.*

What is chaos?

Chaos is derived from a Greek verb that means "to gape open," but in our society, chaos evokes visions of disorder. In a sense, chaotic systems are in unstable equilibrium—even the slightest change to the initial conditions of the system at time t leads the system to a very different outcome at some arbitrary later time. Such systems are said to have a *sensitive dependence* on initial conditions.

Some system models—such as that for the motion of planets within our solar system—contain many variables, yet still are accurate. With chaotic systems, however, even when there are hundreds of thousands of variables involved, no accurate prediction of their behavior can be made.

For example, the weather is known to be a chaotic system. Despite the best efforts of beleaguered meteorologists to forecast the weather, they very frequently err. There's a famous anecdote, which you might have heard, about the movement of a butterfly's wings in Tokyo affecting the weather in New York. This is typical of a chaotic system and illustrates, apocryphally, sensitive dependence on initial conditions.

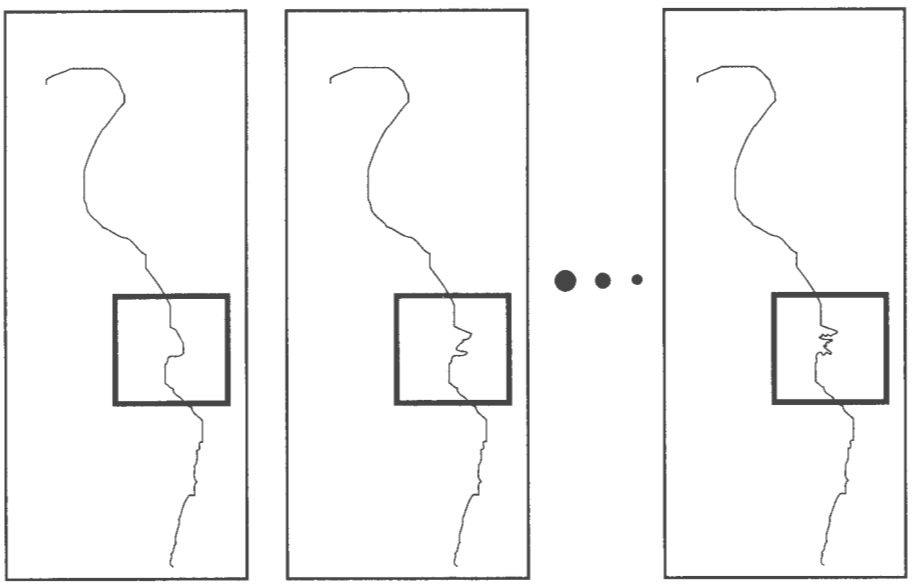
Chaotic systems appear in virtually every aspect of life. Traffic patterns tend to be chaotic—the errant maneuver of even one car can create an accident or traffic jam that can affect thousands of others. Many people feel that the stock market is a chaotic system because the behavior of one investor,

political situation, or corporation can alter prices and supply. Finally, those of you who enjoy science fiction are familiar with story lines where a time traveler goes back and alters a course of events, even slightly, with traumatic consequences.[1] Just as the ripples from a pebble tossed into a lake affect the farthest shore, our slightest actions can have far-reaching repercussions.[2]

What are fractals?

There's a rigorous and precise definition of a fractal, but that's beyond the scope of this text. For our purposes, a *fractal* is an image[3] with an infinite amount of self-similarity.

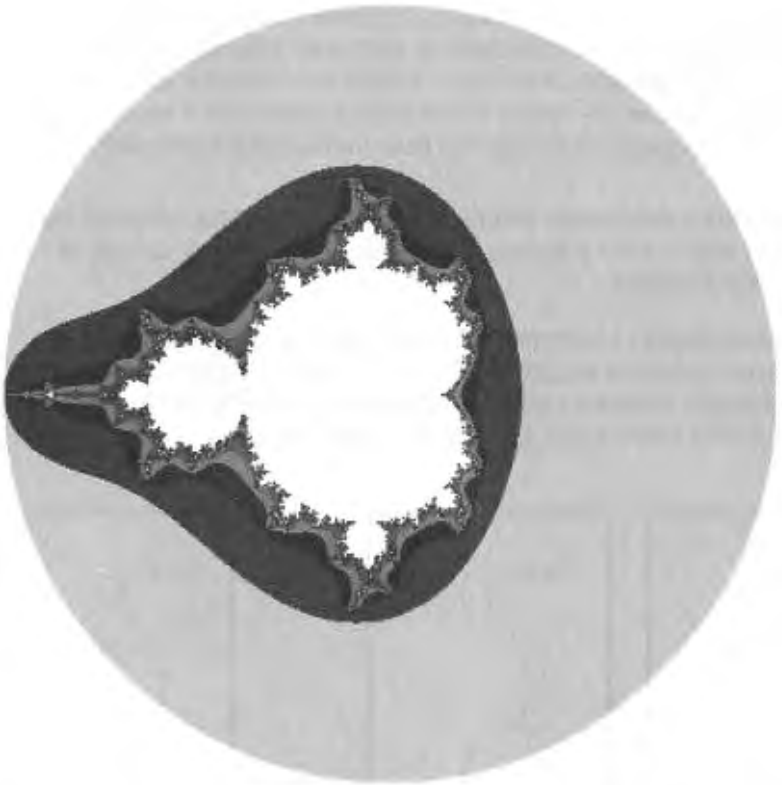
What is self-similarity? In natural and human-made phenomena, *self-similarity* means that the structure of the whole is often reflected in every part. For example, consider a section of coastline photographed from space. (See FIG. 1-2.) You can see that the view from space is similar to the one from



1-2
Increasingly nearer views of a section of coastline.

10 miles away, 1 mile away, 1 foot away, 1 inch away, and so on. This is exactly the kind of self-similarity that characterizes fractals.

Let's consider another example of self-similarity. Look at the image in FIG. 1-3. Notice that there are several globes. Look closely at them. Can you see that these globes are just a copy of the larger image? The globes also have a number of "pimples" on them. If you look at them closely too, you'll see that they're also a small reproduction of the larger image. If the image in the figure had an infinite level of detail, you could examine the picture to any magnification and still find a copy of the larger image.



1-3
The Mandelbrot set.

Fractals tend to be jagged and irregular. It's not surprising, then, that they were named by mathematician Benoit Mandelbrot after the Latin word *fractus*, meaning broken.

How are fractals created?

It's unlikely that you could simply "discover" an image that had the property of self-similarity. In the two previous examples, the coastline and the Mandelbrot set, the first was fabricated for illustration (although it could, in theory exist) and the second was created via a careful mathematical procedure. How do you find such mathematical procedures?

Before I answer this question, let's explore the place where the fractal lives, the realm of dynamical systems. The study of *dynamical systems* is a subfield of mathematics that's concerned with the repeated application of an algorithm.

What's an algorithm? An *algorithm* is simply a recipe or set of rules that describes some process. A cookbook contains many algorithms for baking cakes and other goodies. The assembly instructions for a bicycle represent an algorithm. A computer program is simply an encoded form of an algorithm and it's these types of algorithms that you see in this book.

Algorithms can be presented in many ways: in words, in flowcharts or other pictures, in pseudocode, or in mathematical notation. I'll be using combinations of words and mathematical notations throughout this text to describe the algorithms needed to generate fractals. These algorithms involve the application of some function defined on real or complex numbers (to be defined later), or the application of some graphical or geometric procedure. I'll show how repeated application of either type of algorithm can result in a fractal.

Attracting & escaping points

Let's begin developing the basic vocabulary needed to describe the algorithms needed for making fractals. Consider a function f, which is just a mapping or rule, from the real number line onto itself. Let's denote this:

$$f : \Re \rightarrow \Re$$

$$x \rightarrow f(x)$$

The symbol $\Re$ stands for the real number line, and the arrow, $\rightarrow$, denotes the fact that the function f is a rule that relates each real number x with another real number $f(x)$. At this point you might want to get a calculator and use it as you read the following discussion.

Consider the function $f(x) = x^2$. If you enter the number 2 and press the "X^2" key, you get 4. Press it again, and you get 16, and so on. This procedure is called *function composition*. The composition of $f(x)$ with itself is denoted $f(f(x))$ and it simply means apply the rule f to value x, then apply the rule f again to the result. If you compose the result with the function f again, denoted $f(f(f(x)))$, you've performed another *iteration* of the composition of f. You can continue composing f by itself many times, a procedure called *function iteration*. For simple functions, iteration is easily performed with a calculator.

Continuing with the example, if you compose the X^2 function enough times, your calculator will probably revert to exponential notation and display:

```
3.4028 E 38
```

That means 3,4028 times the number 1 followed by 38 zeros. Eventually, your calculator will give up and display something like

```
ERROR
```

That means that the number obtained was too large for your calculator to hold, even using exponential notation. You can then say that the point 2 iterated under the function $f(x) = x^2$ *escaped*, or it tended towards infinity. Functions that tend toward minus infinity at a point under iteration also escape.

For example, the point $x_0 = 2$ iterated under the function $f(x) = -x^2$ will tend towards negative infinity. (Try it on your calculator.) Points that escape under

iteration are also sometimes called *repelling* points or are said to be repelled under iteration.

Now consider iterating any point that's larger than 0 but less than 1, say $x_0 = .5$ under the function $f(x) = x^2$. Enter this number into your calculator and press the "X^2" key a few times. You'll notice that the product gets smaller and smaller. Eventually, the exponential notation will be displayed, but this time it will have a negative exponent. It might look like:

```
2.3283 E - 10
```

That means that the number is 22.3283 times 0.0000000001. This is a very small number indeed! If you keep pressing the "X^2" key, the ERROR indicator might be displayed. This doesn't mean that the point escaped, rather the number became so close to zero that the calculator could no longer calculate the function X^2 without making an error. In this case, the iterated function tended towards a single point, 0. You could say that 0 is an *attractor* of this function because the iterated function tended toward this point. If you iterate this function with a starting value between – 1 and 1 (noninclusive) the iterated result will always be 0.

Some points act as neither attractors nor repellers under iteration. Such points are said to be *indifferent*. You can use your calculator, or write programs, to determine attracting, repelling, or indifferent points for functions. For example, try to determine some attracting, repelling, indifferent points for the following functions:

1. $f(x) = x^3 - 1$
2. $f(x) = -2x(2 - x)$
3. $f(x) = sin(x)$
4. $f(x) = 1/x$
5. $f(x) = \frac{sin(x)}{x}$

Later on, I'll look at the collection of all attracting points for some iterated functions or iterated geometric procedures. When the attracting set of an iterated function or procedures is an infinitely self-similar set (a fractal) then the attracting set is called a *strange attractor*.

Bifurcation diagrams

Let's look at the set of attractors for some functions defined on real numbers. Suppose you're given the simple polynomial function:

$f(x) = x^2 + c$

for some real constant c, and you compose the function with itself many times. Let's try this by picking some c, say $c = -1.1$ and set $x = 0$. Using a calculator, you'll see that:

$f(0) = 0^2 - 1.1 = -1.1$

Iterating, you find $f(f(0))$, that is:

$f(f(0)) = f(-1.1) = (-1.1)^2 - 1.1 = .11$

Now finding $f(f(f(0)))$, you have:

$f(f(f(0))) = f(.11) = (.11)^2 - 1.1 = -1.0879$

Applying the rule f to this result again gives:

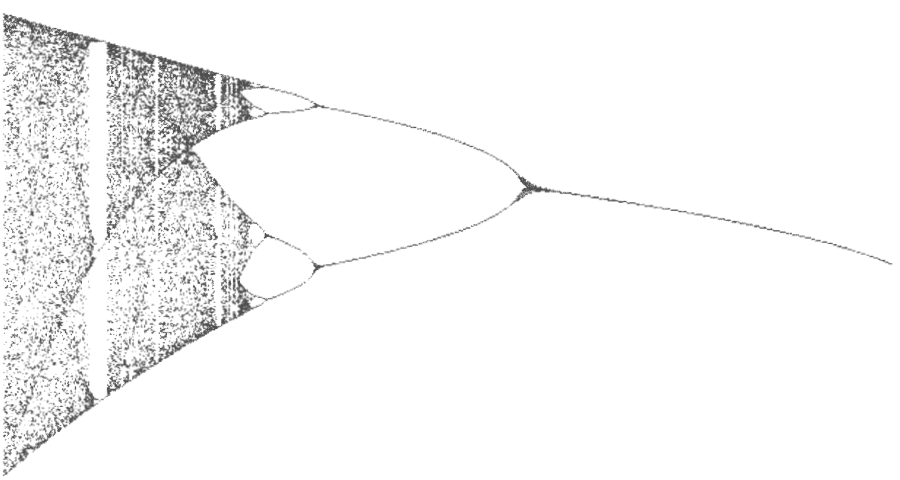
$f(f(f(f(0)))) = (-1.0879)^2 = .08$

You could continue this indefinitely, but you should see that the result of the composed functions seem to bounce back and forth between a number somewhere near – 1.0 and another number near .1. Suppose you compose f itself many times, say 200, and you do this for a range of values of c, for functions that look like:

$f(x) = x^2 + c$

A strange and beautiful thing happens.

If you compose this function many times, plotting points after each composition, and you do this for many values of c, the resulting image is called a *bifurcation diagram*, which is shown in FIG. 1-4. The term "bifurcation" is used because the image divides into two distinct bands of points. Like a fractal, it's also self-similar.



1-4
Bifurcation diagram for $f(x) = x^2 + c$ *with* $x = 0$ *and various values of* c *produced using BIFUR.PAS.*

To see this, let's regenerate FIG. 1-4 by running the program BIFUR.PAS. The program prompts you for a scale. You should enter a "1" in response. The program will then begin to display the bifurcation diagram as it sweeps values of *c*.

Notice the bands of stability (the "bald spots") where the function only takes on two values instead of infinitely many. The interpretation of these will become clearer later. Finally, by choosing different scaling factors, you should see that the bifurcation diagram is self-similar.

BIFUR.PAS is a straightforward program. You choose an appropriate scale factor based on the user input number `sf` and the maximum *x*-coordinate `MaxX`. Some experimentation shows that dividing by a factor of 8 yields a better picture. The code for the scaling looks like this:

```
scale : = sf*MaxX/8;       { calculate overall scale factor }
```

Based on experience, I chose a starting value for *c* as –2.0. You then sweep the value of *c*, adding enough for each increment in the *x*-axis, so that the final value of *c* is .25.

Within the loop, you initialize the value $x = 0$ and compose the function $f = x^2 + c$ by itself 200 times. Ignore the first 50 iterations to allow the composition to stabilize. Then plot the point. The salient code looks like:

```
c : = -2.0;                { set starting point }
for i : = 1 to MaxX do
 begin
  x : = 0.0;               { calculate orbit about x = 0 }
  c : = c + 2.25/MaxX;     { iterate c }
  for j : = 1 to 200 do    { calculate orbit after 200 iterations}
  begin
   x : = x*x + c;
   if j > 50 then          { skip first 50 iterations }
   begin
    putpixel(i,round(MaxY/2 + x*scale), j div MaxColor);
   end
 end
end
```

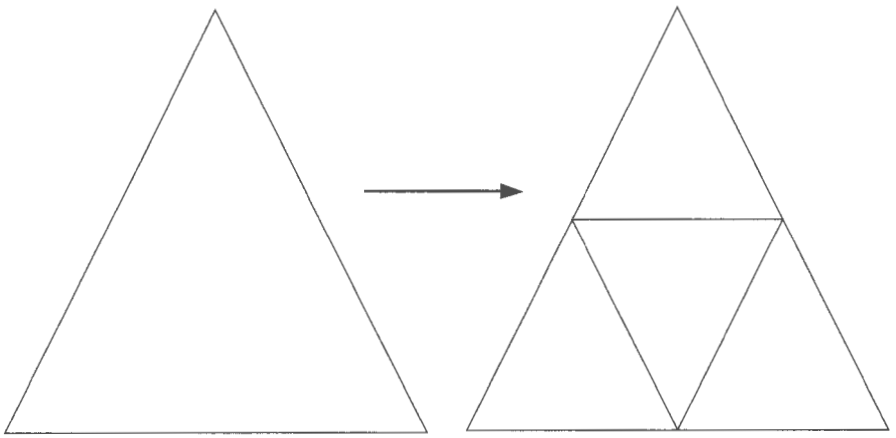
There is one subtlety. You output the pixel with an *x*-coordinate of `i`, which controlled the sweep of `c`, and a *y*-coordinate of the composed function value, suitably scaled and offset to the middle of the screen. To make the diagram pretty, I picked a different color based on the number of compositions of the function *f* that were applied.

Bifurcation diagrams are amazingly simple little fractals that have applications that you'll see later. You can experiment with BIFUR.PAS by playing with the function *f* and the sweep value for *c*. Be careful, though, because the function might give you integer overflow problems and "blow up."

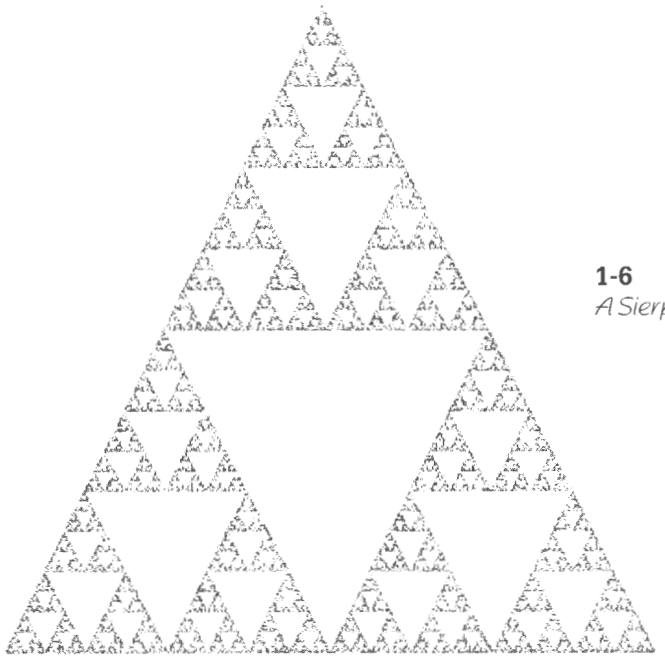
The Sierpinski triangle

Another way to generate fractals is by the repeated application of special geometric procedures. Such fractals are called *iterated function systems* (IFS). A nice two-dimensional fractal that can be generated this way is the *Sierpinski triangle*.

Consider a filled triangle. Suppose you remove a section from the middle so that the result is three copies of the original at ¼ size, as shown in FIG. 1-5. If you continue to apply this rule to the three triangles, and then the nine resulting triangles and so on, you obtain the fractal shown in FIG. 1-6. Let's look at how the Sierpinski triangle is created with a Pascal program.



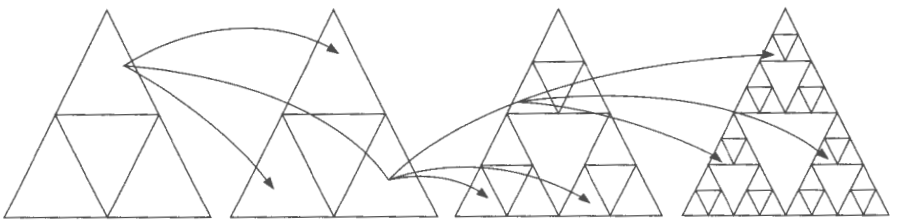
1-5
Rule for creation of a Sierpinski triangle.



1-6
A Sierpinski triangle.

The easiest way to generate such a picture is to generate *random orbits* and look for attracting points. To do this, you need to encode the graphical procedures in Pascal, select a random starting point, and apply the rule to it a fixed number of times. Repeated application of these rules will generate a strange attractor, that is, a fractal. Program SIERP.PAS on your disk applies the algorithm that I'll describe.

To make an equilateral Sierpinski triangle, map the random starting point into one of three randomly chosen rules corresponding to one of the three triangles in the large triangle with the center removed. Figure 1-7 shows how to map an arbitrary point into one of three possible sites.



1-7
Sierpinski triangle mapping procedure.

If you aren't already familiar with the Turbo Pascal screen, I suggest that you review appendix A, or simply note in the following discussion that Turbo Pascal assigns the coordinate (0,0) to the upper left-hand corner of the screen.

If the maximum x-coordinate is `MaxX` and the maximum y-coordinate is `MaxY`, then the large triangle has vertices:

V_1 = (0, MaxY)

V_2 = (MaxX/2,0)

V_3 = (MaxX,MaxY)

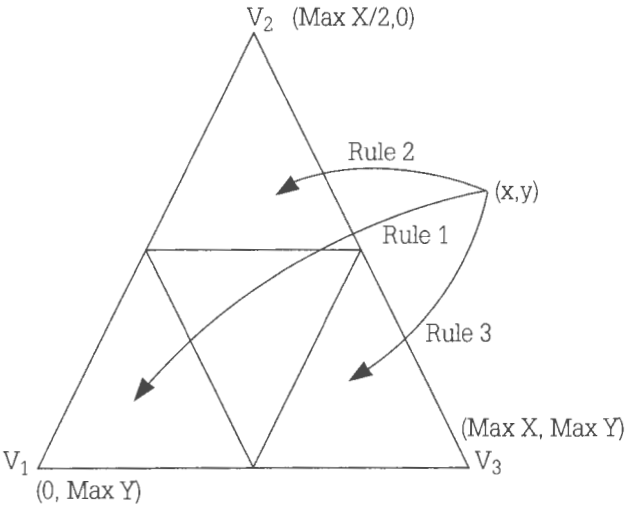
It turns out that if the random point is halfway to the outer vertex of one of the three triangles inside the larger triangle, then the random point is inside one of them. For the three triangles within it, this is also true, and so on. To find the halfway points to the outer vertices, use the rules illustrated in FIG. 1-8.

Rule one is:

$(x',y') = (x/2, (\text{MaxY} + y)/2)$

That will find the point halfway between point (x, y) and vertex V_1. The following is rule two:

$(x',y') = (\text{MaxY}/2 + x, y/2)$



1-8
Vertices for Sierpinski triangle mapping procedure.

Use rule two for the point halfway between point (x, y) and vertex V_2. Here's the third rule:

$(x', y') = ((\text{MaxY} + x)/2 + x, (\text{MaxY} + y)/2)$

The third rule finds the point halfway between point (x, y) and vertex V_3.

If you continually choose one of the three mapping rules and apply it to the coordinate just mapped, you generate points at finer and finer resolutions within the Sierpinski triangle.

The main code, which maps points into one of the three inner triangles, and found in SIERP.PAS is:

```
1:  begin
        x : = x div 2;          { find 1/2 way point to V1}
        y : = (MaxY + y) div 2
    end;
2:  begin
        x : = (MaxY div 2 + x) div 2; { find 1/2 way point to V2 }
        y : = y div 2
    end;
3:  begin
        x : = (MaxY + x) div 2;       { find 1/2 way point to V3}
        y : = (MaxY + y) div 2
    end
```

Note that you use `MaxX = MaxY` to ensure that no distortion is introduced (see appendix A for an explanation). Also note that in the Pascal routine you use `div 2` instead of `/2` to ensure an integer quotient. Finally, you perform 1000 iterations before plotting to be sure that the fractal has begun attracting. You can apply the procedure to any triangle—equilateral, right, or otherwise.

Another fractal that can be constructed in a similar way is a *Sierpinski gasket* or *Sierpinski carpet*. To make a Sierpinski carpet, start with a square, divide it into nine equal-sized squares, and remove the middle one. Proceed with the remaining eight squares, repeatedly applying the same procedure. You'll see a Sierpinski carpet shortly.

Iterated function system transformations

The geometric rules applied to create the Sierpinski triangles and other fractals can be represented mathematically as a set of operations including sliding, stretching, and rotating. These types of mathematical operations are called *affine transformations*. They can be easily coded using matrix operations.

A *matrix* consists of rows and columns that hold numbers. If the matrix shown is called "*d*," then the number in the first row, first column is denoted *d*[1,1]; in the second row first column it's denoted *d*[2,1], and so on. In general, the number in row *i* and column *j* is denoted *d*[*i*,*j*]. Special rules involving the multiplication and addition of the numbers in the matrix simplify the description of affine transformations. I won't review matrix rules here, but I encourage you to pick up a text on linear algebra.

Table 1-1 shows a matrix-encoded form of the rules that generate a Sierpinski triangle. Here, for example, the *d*[1,5] position contains the number 25. The last column has a special meaning in that it determines the chance or probability that the transformation described in that row will be used. For example, in the Sierpinski triangle, as in SIERP.PAS, the three transformations are equally likely.

Table 1-1
IFS transformation rule for the Sierpinski triangle.

	1	2	3	4	5	6	probability
1	0.5	0	0	0.5	25	1	0.33
2	0.5	0	0	0.5	1	50	0.33
3	0.5	0	0	0.5	50	50	0.33

Let's look at what the transformations in the rows of the matrix do. Each transformation maps a point (x,y) into a new point (x',y') by handling each coordinate separately. The transformation in row *i* obtains a new *x*-coordinate, x', by transforming the given *x*-coordinate of the point by the mapping:

$$x' = d[i, 1]x + d[i, 2]y + d[i, 5]$$

The transformation in row *i* transforms the *y*-coordinate by the rule:

$$y' = d[i, 3]x + d[i, 4]y + d[i, 6]$$

In his book *Fractals Everywhere* (Barnsley 1988), Michael Barnsley gives the matrix form codes to generate a variety of iterated function system fractals. (He calls programs used to generate fractals this way the "Chaos Game.") For example, if you look at SIERP2.PAS file on the distribution disk, you'll see a program that implements the IFS code to generate a Sierpinski triangle that's the same as the one we made before.

For example, row one specifies that the following transformations are to be applied with a probability of 0.33, or one-third of the time:

$x' = 0.5x + 0y + 25$

$y' = 0x + 0.5y + 1$

If you were to apply these transformations to the point (3,2) you'd get:

$x' = 0.5 \cdot 3 + 0 \cdot 2 + 25 = 26.5$

$y' = 0 \cdot 3 + 0.5 \cdot 2 + 1 = 2$

Thus, the transformed point is (26.5,2). Try applying the transformations to this point for four iterations.

You can take advantage of the simplicity of the matrix form of the IFS in many fractal programs. For instance, in program SIERP2.PAS, the code that holds the matrix data is:

```
{ initialize IFS data array }

  d[1,1]: = 0.5; d[1,2]: = 0; d[1,3]: = 0; d[1,4]: = 0.5; d[1,5]: = 25; d[1,6]: = 1;
  d[2,1]: = 0.5; d[2,2]: = 0; d[2,3]: = 0; d[2,4]: = 0.5; d[2,5]: = 1; d[2,6]: = 50;
  d[3,1]: = 0.5; d[3,2]: = 0; d[3,3]: = 0; d[3,4]: = 0.5; d[3,5]: = 50; d[3,6]: = 50;
```

The code to pick one of the three rows that hold the transformation and apply it is:

```
k : = random(3) + 1;                { pick random row }
x : = d[k,1]*x + d[k,2]*y + d[k,5]; { transform coordinates }
y : = d[k,3]*x + d[k,4]*y + d[k,6];
```

Isn't this code much more compact than that in SIERP.PAS? This code is also quite fast—much faster than the code that you'll see in the next chapter. Finally, it's an amazing fact that by simply changing the data in the IFS code table, you can generate vastly different fractal images.

For example, to make the Sierpinski carpet, simply change the data matrix in program SIERP2.PAS to that shown in TABLE 1-2.

Table 1-2
IFS transformation rule for the Sierpinski carpet.

	1	2	3	4	5	6	probability
1	0.33	0	0	0.33	1	1	0.125
2	0.33	0	0	0.33	MaxY	1	0.125
3	0.33	0	0	0.33	1	MaxY	0.125
4	0.33	0	0	0.33	MaxY	MaxY	0.125
5	0.33	0	0	0.33	MaxY/2	1	0.125
6	0.33	0	0	0.33	MaxY	MaxY/2	0.125
7	0.33	0	0	0.33	1	MaxY/2	0.125
8	0.33	0	0	0.33	MaxY/2	MaxY	0.125

This was done in program CARPET.PAS, which displays the Sierpinski carpet shown in FIG. 1-9. In this case, the new data IFS array is:

```
d[1,1]: = 0.33; d[1,2]: = 0; d[1,3]: = 0; d[1,4]: = 0.33; d[1,5]: = 1; d[1,6]: = 1;
d[2,1]: = 0.33; d[2,2]: = 0; d[2,3]: = 0; d[2,4]: = 0.33; d[2,5]: = MaxY; d[2,6]: = 1;
d[3,1]: = 0.33; d[3,2]: = 0; d[3,3]: = 0; d[3,4]: = 0.33; d[3,5]: = 1; d[3,6]: = MaxY;
d[4,1]: = 0.33; d[4,2]: = 0; d[4,3]: = 0; d[4,4]: = 0.33; d[4,5]: = MaxY; d[4,6]: = MaxY;
d[5,1]: = 0.33; d[5,2]: = 0; d[5,3]: = 0; d[5,4]: = 0.33; d[5,5]: = MaxY div 2; d[5,6]: = 1;
d[6,1]: = 0.33; d[6,2]: = 0; d[6,3]: = 0; d[6,4]: = 0.33; d[6,5]: = MaxY; d[6,6]: = MaxY div 2;
d[7,1]: = 0.33; d[7,2]: = 0; d[7,3]: = 0; d[7,4]: = 0.33; d[7,5]: = 1; d[7,6]: = MaxY div 2;
d[8,1]: = 0.33; d[8,2]: = 0; d[8,3]: = 0; d[8,4]: = 0.33; d[8,5]: = MaxY div 2; d[8,6]: = MaxY;
```

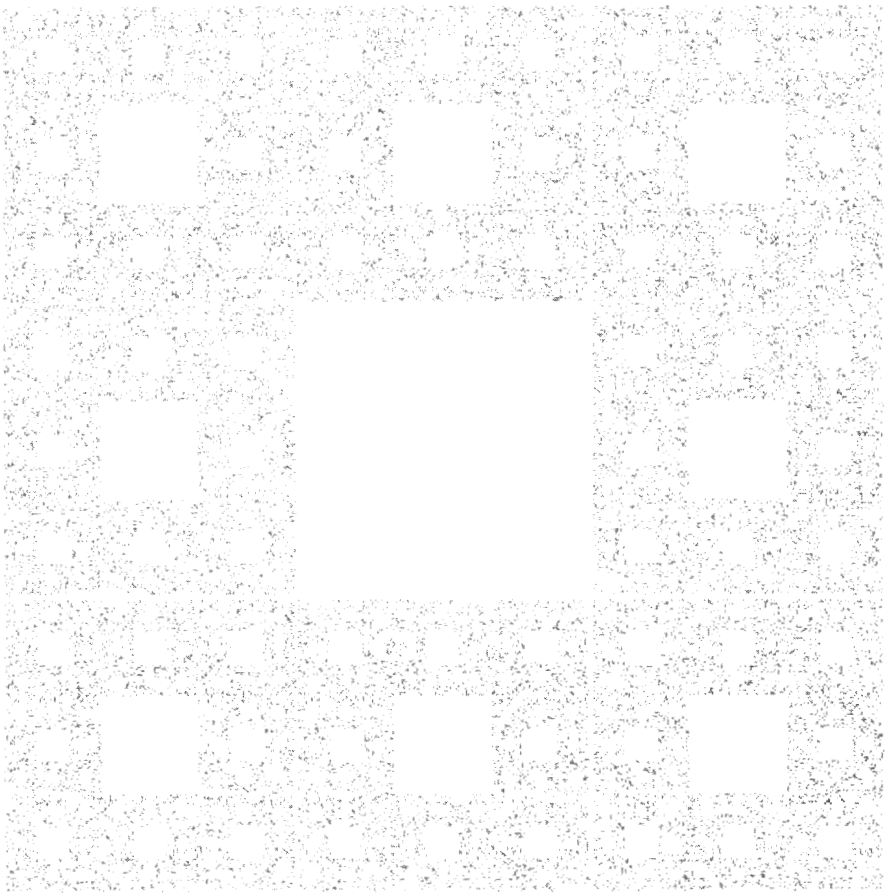
The code to pick the random row is:

```
k : = random(8) + 1;                { pick random row }
x : = d[k,1]*x + d[k,2]*y + d[k,5]; { transform coordinates }
y : = d[k,3]*x + d[k,4]*y + d[k,6];
```

You can experiment with SIERP2.PAS, CARPET.PAS, or the other programs using the IFS data algorithm that you'll see later. Experimenting can produce some amazing results, and you'll see several of these in chapters 3 and 4.

Recursive generation of fractals

Most fractals are generated by applying a certain procedure infinitely (or at least a very large, albeit finite number of times). I generated the first fractal, the bifurcation diagram, by repeated iteration of a function. In the last two fractals, I iterated random functions by applying repeated geometric procedures. Another way to apply geometric procedures, however, is by



1-9
Sierpinski carpet.

coding them so that they're self-referential, or *recursive*. A quote from Jonathan Swift[4] captures the spirit of recursion:

"So, naturalists observe, a flea
Hath smaller fleas that on him prey;
And these have smaller fleas to bite'em,
And so proceed *ad infinitum*.
Thus every poet, in his kind
Is bit by him that comes behind."

In mathematics, self-reference usually implies recursion in the sense that a function is defined in terms of itself. For example, consider the numbers in the famous *Fibonacci sequence*. Let the $f(0) = 0$ and let $f(1) = 1$ be the first two numbers in the sequence. Then the n^{th} number in the sequence $f(n)$ is given by:

$f(n) = f(n-1) + f(n-2)$

The n^{th} number in the sequence is just the sum of its two predecessors. Then the first few numbers in the sequence are:

0 1 1 2 3 5 8 ...

What would the twenty-third number in the sequence, denoted $f(23)$ be? Well, you say, it's just $f(22) + f(21)$. However, what are these? You'd have to perform a large number of calculations to find $f(23)$ this way. It's an amazing fact, and part of the allure of mathematics, that you can find an algebraic solution for $f(23)$ or $f(n)$ in general. For the Fibonacci sequence, with $n \geq 0$:

$$f(n) = \frac{\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

So, if you plug $n = 23$ into the formula, you get:

$f(23) = 28657$

Just for fun, try to find $f(22)$ and $f(21)$ using this formula, then show that:

$f(23) = f(22) + f(21)$

In computer science, certain programming languages, such as Pascal, support recursion in the sense that procedures can call themselves. For example, consider the program FIB.PAS, which finds the n^{th} number in the Fibonacci sequence. It makes use of the self-referential function, `fibo`, given in the following:

```
function fibo(i:integer) : integer;
{ a recursive function }
begin
   if i = 0 then
      fibo : = 0
   else
     if i = 1 then
       fibo : = 1
       else
         fibo : = fibo(i - 1) + fibo(i - 2)
end;
```

Notice how the function calls to itself. You should study the program and try running it with any number between 0 and 23 (any other numbers will cause overflow problems). You'll be seeing other recursive programs later in this book.

Recursion and self-reference can be found in visual art as well. For example, the work of celebrated artist M. C. Escher demonstrates an incredible insight into these concepts. For example, in the work *Fish and Scales* (woodcut, 1959), the scales of the fish are themselves fish at many levels. Also, in Escher's *Circle Limit I* (woodcut, 1958) and *Circle Limit II* (woodcut 1959), a high degree of self-reference is present. To see these and other beautiful Escher works, and read his own insightful narrative, see "Escher on Escher:

Exploring the Infinite." (Abrams 1989). To learn more about recursion in mathematics, music, art, and life, read *Gödel, Escher, Bach: An Eternal Golden Braid*, the Pulitzer Prize winning book by Douglas Hofstadter. (Hofstadter 1989).

The Cantor set

To illustrate mathematical recursion again, let's look at the fractal created when we recursively apply the following procedure to a section of the real line.

> remove the middle third of the real line, then remove the middle third of the remaining line segments, and so on.

This procedure, called the *Cantor middle third argument*, was introduced by mathematician Georg Cantor in the late nineteenth century and has a very powerful result. The effect of applying the procedure an infinite number of times is illustrated in FIG. 1-10. The resulting figure is sometimes called the *Cantor set*, and it's a fractal in one-dimensional Euclidian space. If you look at it closely, you should see that there are infinite levels of self-similarity.

1-10
Construction of the Cantor set.

When you perform the Cantor procedure an infinite number of times, a very strange thing happens—you never completely eliminate the line. In fact, there will be an infinite number of minuscule line segments. Yet, if you strung them all together, their length would be zero!

If you have trouble believing the first part of the previous statement, suppose I placed you at the end of a room that was exactly 20 feet long. I then ask you to halve your current distance to the other side of the room, and to repeat this procedure. Your distance in feet from the opposite wall would then be:

10 5 2.5 1.25 .625 ...

However, would you ever reach the other wall? The answer is no! Although you can get arbitrarily close to the wall, you can never actually arrive at it using the procedure outlined. The effect is similar when you apply the Cantor procedure. You never completely annihilate the line[5].

You'll find the program CANTOR.PAS on your disk, which applies the Cantor procedure recursively to a line segment on the screen. The heart of the code for generating the Cantor set is a recursive procedure called `Cantor`. Let's look at the code in `Cantor`.

```
begin
    line(x1, y1, x2, y1);
    y1 : = y1 + 16;                    {increment $y$-coordinate}
    if y1 < = 128 then                 {don't do to far }
    begin
    delta : = (x2 - x1) div 3;        {calculate third of line segment }
    Cantor(x1, y1, x1 + delta, y1); {draw first third}
    Cantor(x2 - delta, y1, x2, y1); {draw third third}
    end;
end;
```

Suppose that the starting line segment has the left-end point coordinate `(x1,y1)`. Then, if the line is level, it has the right-end point at `(x2,y1)`. The first line of the program invokes the graphics procedure `line`, which draws a line between these points.

You then calculate one-third of the distance between these points, which is done by the statement:

```
delta : = (x2 - x1) div 3
```

The `div` operator ensures that the result of the division will be an integer and not a floating point number, which Pascal will not allow here.

You add 16 pixels to the *y*-coordinate to move the next iteration down, and then perform procedure `Cantor` on the third of a line segment starting at point `(x1,y1)` and ending at `(x1 + delta,y1)`. You do the same on the other third of the line segment, starting from the point at `(x2 - delta, y1)` and ending at the point `(x2,y1)`.

The following code ensures that you don't perform recursion too far down the y-axis (no more than 128 pixels from the top).

```
if y1 < = 128 then
```

That's all there is to it. You should run the CANTOR program yourself and see that it works.

Repeated application of different geometric rules can generate other fractals. Although recursive geometric rules can be performed by an IFS system, which is easier to code, finding the IFS codes that are equivalent to the recursive geometric procedure is often quite difficult.

Fractal dimension

Suppose you were given a piece of tinfoil with a thickness of exactly zero. You could say that the foil has a Euclidian dimension of two; that is, it's two-dimensional (the Cartesian plane is two-dimensional). Suppose now that you took that piece of foil and crumpled it into a ball. Although the ball now exists in three-dimensional space, it's not quite three-dimensional because it really is not a solid (remember the foil has zero thickness) and can't be described precisely using Euclidian geometry. You would then say that the ball has a fractional or *fractal dimension*.

As you might expect, fractal images also have a fractal dimension. Consider, for example, the Cantor set. Normally, a line (with thickness of zero) is said to have a dimension of one (it's one-dimensional). However, the Cantor set is not quite a line, but rather a collection of an infinite number of disconnected line segments. What, then, is the fractal dimension of the Cantor set?

Consider next the Sierpinski triangle or Sierpinski carpet. Both appear to be two-dimensional, but because they aren't filled, their fractal dimension should be somewhat less than two. What, then, are their fractal dimensions?

There are precise answers to these questions, which are far beyond the scope of this text and most undergraduate mathematics courses. However, using an approximate technique, you can get a feel for the fractal dimension of some of the images described in this text. To do this, however, I need to introduce the concept of logarithms and exponentials.

First, recall the notation:

$$x^y \equiv x \cdot x \cdot \cdot \cdot x \tag{1.1}$$

You multiply x by itself y times, where y is said to be the *exponent*[6] and x is the base. You say this as "x raised to the power y." For example:

$4^3 = 4 \cdot 4 \cdot 4 \cdot = 64$

Now suppose you're given x^y the number x, and you want to find the number y. To do this, you apply a special operation called the base x *logarithm* to x^y (x>0). The result will be y and is denoted:

$$\log_x(x^y) = y \tag{1.2}$$

From the previous example, you can see that:

$\log_4(64) = 3$

So, how do you use logarithms to find fractal dimension? It turns out that a good approximation of fractal dimension D is:

$D = \log_{10}(\text{number of pieces})/\log_{10}(\text{magnification})$

That uses the logarithm.

To describe the formula for D in words:

> To find the fractal dimension, count the number of self-similar pieces, take the logarithm to the base 10, and divide by the base 10 logarithm of the magnification, where the amount of magnification is the amount of "closeness" needed to get the original image back.

For example, consider a solid line. At magnification n, you divide the line into n equal or self-similar pieces of length $1/n$. Thus, the solid line has fractal dimension:

$D = \log_{10}(n)/\log_{10}(n) = 1$

Now, look at the Cantor set. Since it was produced by removing the middle third from a line, a magnification by three yields two self-similar pieces. Using a calculator, you can then find the fractal dimension to be:

$D = \log_{10}(2)/\log_{10}(3) = 0.63...$

Next consider a square on a two-dimensional plane. At magnification n, the square is divided into n^2 self-similar squares, so it has fractal dimension:

$D = \log_{10}(n^2)/\log_{10}(n) = 2$

(To verify this, pick any number, $n > 1$, plug it into the preceding formula, and work it out on a calculator.) However, for the Sierpinski triangle, any magnification by two yields three self-similar pieces, so it has fractal dimension:

$D = \log_{10}(3)/\log_{10}(2) = 1.58...$

How are fractals & chaos related?

There's an intricate and often subtle relationship between fractals and chaos. One way of interpreting their relationship is to note that fractals are generated by detecting attractors and repellers. Thus, they represent, in a sense, a visual representation of chaotic behavior. You can create black-and-white fractal images by plotting points on the real line or Cartesian plane (attractors (stable points) in one color, repellers (chaotic points) in another color). By keeping track of the "speed" at which attractors attract and repellers repel, and by plotting bands of these rates in different colors, you can generate elegant fractals in color.

In addition, fractals are chaotic in that they're very sensitive to changes in initial conditions. For example, you'll see that if you change the function to be iterated even slightly, this results in a vastly different fractal image output. This sensitive dependence on initial conditions is a theme that unites unstable systems, fractals, and chaos.

Another way to see the relationship between fractals and chaos is to study *cellular automata*, a special mathematical abstraction from dynamical systems. Cellular automata can be stable or chaotic, and many generate fractals. I discuss cellular automata in chapter 4.

A brief history of fractals & chaos

It would be impossible to trace the pedigree of fractals or chaos precisely. To begin, you would have to study dynamical systems, nonlinear mathematics, functional analysis, and so forth. Listing the names of those who have contributed, at least in part, to the theory of dynamical systems is like reading a "Who's Who of Mathematics."

However, although the early threads of fractals and chaos theory are old, the science itself is very new. For example, shortly after World War I (1919) Gaston Julia began work on what would later be called attractive cycles of complex functions, but for the next fifty years most of his work lay dormant.

Much of the work on dynamical systems and cellular automata can trace its heritage to the great mathematician (and leading figure in the development of the digital computer), John von Neumann in the 1940s and 1950s. Also, in the early 1960s Edward Lorenz studied chaotic phenomena in weather.

However, it wasn't until Mandelbrot came along that natural and human-made phenomena were associated with self-similarity in such a clear way. In the last thirty years since Mandelbrot's first publications, scientists in diverse fields have linked fractals and chaos to their work. Since the 1970s many scientists like Michael Barnsley have extended the work of Julia, Mandelbrot, Lorenz, and others. For a complete treatise on the history of chaos and fractals, see James Gleick's book. (Gleick 1987).

2 Foundations of chaos & fractal theory

"From Nature's chain whatever link you strike,
Tenth, or ten thousandth, breaks the chain alike."

— Alexander Pope, *Essay on Man* (1733)

In this chapter I introduce the mathematical background that's needed to generate some of the more breathtaking fractal images. Don't be intimidated if you've never seen this material before. Most of the mathematics involves simple variations on algebraic concepts covered in high school. If you aren't comfortable with the mathematics, simply skip the formulas now, and, if you like, come back to them later. However, the formulas do help unlock some of the mysterious beauty of the fractal images, and the formulas are worth the effort to master. Finally, all of the complex operations described here have been coded for you, and you can find them in the Pascal unit COMPLEX.PAS on the disk that comes with this book. Looking at the code might help you understand the mathematics.

Complex numbers & functions

Consider the mapping signified by the symbol $\sqrt{}$, which is generally called the "principal square root." This mapping represents the inverse of the function $f(x) = x^2$ defined on the real line. For positive numbers and 0, this is well defined. However, for negative numbers, it's undefined. For example, what is $\sqrt{-5}$? What number multiplied by itself yields –5?

To get around this problem, mathematicians[1] have defined the abstract notion of the square root of –1, denoted i. That is:

$$i^2 = -1$$

You should realize that i is only the positive square root of –1, and that a negative square root, say $k = -i$ exists as well, since:

$$k^2 = (-1i)^2 = i^2 = -1$$

However, right now, you're only interested in the positive square root.

Notice how i takes care of the square root of all negative numbers, since, for example:

$$\sqrt{-4} = \sqrt{-1 \cdot 4} = \sqrt{4} \cdot i = 2i$$

Suppose now that you have a number, z of the following format:

$$z = a + bi$$

The z is called a *complex number* where a is called the *real part* and b is called the *imaginary part.* For example:

$$4 + 5i, -3.12 + .01i, 0 + 9i, 2.7 + 0i$$

All of those are complex numbers. *Complex variables* (placeholders for complex numbers) are usually denoted with some variation of the letter z. For example, z_1, z_2, and so on.

Plotting complex numbers

A complex number can be plotted as a point on the Cartesian plane by letting the real part represent the x-coordinate and the imaginary part represent the y-coordinate. For example, consider the complex numbers $z_1 = -1 + 2i$, $z_2 = 1.2 - 3i$, $z_3 = -2.1 - .1i$, and $z_4 = .5 + .5i$. These are plotted on a Cartesian plane, except that the y axis is labeled as the "iy" axis, as shown in FIG. 2-1. This map is called the *complex plane*, and it's where complex numbers "live." Functions of complex numbers, which you'll see shortly, can also be plotted on the complex plane.

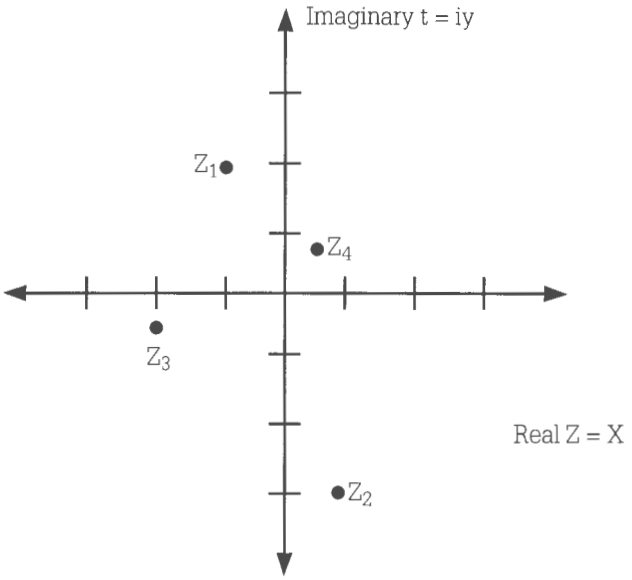
Arithmetic with complex numbers

Complex numbers can be added, subtracted, multiplied, and divided. Addition and subtraction are the easiest to perform—simply add or subtract the respective real and imaginary parts of the numbers. Officially, let $z_1 = a_1 + b_1i$, *and* $z_2 = a_2 + b_2i$ be any complex numbers. Then for addition:

$$z_1 + z_2 = a_1 + a_2 + (b_1 + b_2)i$$

For subtraction:

$$z_1 - z_2 = a_1 - a_2 + (b_1 - b_2)i$$



2-1
The complex plane and some points on it.

For example, let $z_1 = 4.1 + 3.1i$ and $z_2 = -1 + .1i$, then:

$$z_1 + z_2 = 4.1 + (-1) + (3.1 + .1)i$$

$$= 3.1 + 3.2i$$

and

$$z_1 - z_2 = 4.1 - (-1) + (3.1 - .1)i$$

$$= 5.1 + 3.0i$$

The following Pascal procedures, found in file COMPLEX.PAS, will perform addition and subtraction of two complex numbers. To find $z3 = z1 + z2$ use:

```
procedure add (x1, y1, x2, y2 : real; var x3, y3 : real);
begin
  x3 : = x1 + x2;
  y3 : = y1 + y2
end;
```

For $z3 = z1 - z2$ use:

```
procedure sub(x1, y1, x2, y2 : real; var x3, y3 : real);
begin
  x3 := x1 - x2;
  y3 := y1 - y2
end;
```

In both procedures, the real and imaginary parts of z1, z2, and z3 are x1, y1, x2, and x3, y3 respectively.

Notice how the procedures accept the input of two complex numbers by the real and imaginary parts separately, and the procedures also output the complex sum of these numbers in terms of the real and imaginary parts.

Multiplication and division of complex numbers is just a bit trickier. Let's consider multiplication first. Let $z_1 = a_1 + b_1 i$ and $z_2 = a_2 + b_2 i$, then by multiplication using the FOIL method (first, inner, outer, and last products) and simplifying:

$$z_1 \cdot z_2 = a_1 a_2 + a_1 b_2 i + a_2 b_1 i + b_1 b_2 i^2$$

$$= (a_1 a_2 - b_1 b_2) + (a_1 b_2 + a_2 b_1) i$$

For example, let $z_1 = 4.1 + 3.1i$ and $z_2 = -1 + .1i$, then:

$$z_1 \cdot z_2 = -4.41 - 2.69i$$

The following Pascal code, found in COMPLEX.PAS, will perform multiplication of two complex numbers. The input and output to the procedure is in the same manner as for the addition and subtraction procedures.

```
procedure mult(x1, y1, x2, y2 : real; var x3, y3 : real);
begin
  x3 := x1*x2 - y1*y2;
  y3 := y1*x2 + x1*y2
end;
```

The real and imaginary parts of z1, z2, and z3 are x1, y1, x2, y2, and x3, y3 respectively.

Next let's consider division. Let $z_1 = a_1 + b_1 i$ and $z_2 = a_2 + b_2 i$, then:

$$\frac{z_1}{z_2} = \frac{a_1 + b_1 i}{a_2 + b_2 i}$$

It's not known how to do this division directly. To put the ratio in a form that you can handle, multiply numerator and denominator by $a_2 - b_2 i$. This is like multiplying it by 1, which doesn't change the equality[2]. You then get:

$$\frac{(a_1 + b_1 i)}{(a_2 + b_2 i)} \frac{(a_2 - b_2 i)}{(a_2 - b_2 i)} = \frac{(a_1 a_2 + b_1 b_2) + (a_2 b_1 - a_1 b_2) i}{(a_2 a_2 - b_2 b_2) + (a_2 b_2 - a_2 b_2) i}$$

$$= \frac{(a_1 a_2 + b_1 b_2) + (a_2 b_1 - a_1 b_2) i}{a_2^2 + b_2^2}$$

$$= \frac{(a_1 a_2 + b_1 b_2)}{a_2^2 + b_2^2} + \frac{(a_2 b_1 - a_1 b_2)}{a_2^2 + b_2^2} i$$

Now you've expressed the ratio in terms of the real number division of the real and complex parts.

For example, let $z_1 = 4.1 + 3.1i$ and $z_2 = -1 + .1i$, then:

$$\frac{z_1}{z_2} = \frac{3.79 - 3.51i}{(-1)^2 + (.1)^2}$$

$$= \frac{-3.79}{1.01} - \frac{3.51}{1.01}i$$

$$= -3.752 - 3.475i$$

The following Pascal code, found in file COMPLEX.PAS, will perform division of two complex numbers where $z_3 = \frac{z_1}{z_2}$. The input and output to the procedure is in the same manner as for the addition and subtraction procedures.

```
procedure cdiv(x1, y1, x2, y2 : real; var x3, y3 : real);
var
  denom : real;                    { denominator }
begin
  denom := x2*x2 + y2*y2;
  x3 := (x1*x2 + y1*y2)/ denom;    { real part }
  y3 := (x2*y, - x1*y2)/ denom     { imaginary part }
end;
```

Again, the real and imaginary parts of z1, z2, and z3 are x1, y1, x2, y2, and x3, y3 respectively.

Functions of complex variables

In order to generate some really interesting-looking fractals, you need more than just addition, subtraction, multiplication, and division.[3] You need more powerful functions of both real and complex numbers.

The first of these more powerful functions are two simple functions of real variables: the *hyperbolic sine* and *hyperbolic cosine*, denoted *cosh* and *sinh* respectively and defined as follows[4]:

$$cosh(x) = \frac{e^x + e^{-x}}{2} \tag{2.1}$$

$$sinh(x) = \frac{e^x - e^{-x}}{2} \tag{2.2}$$

The two following Pascal procedures generate the hyperbolic cosine and sine respectively:

```
function cosh( x : real) : real;      { calculates cosh(x) }
begin
    cosh : = (exp(x) + exp(-x))/2.0
end;
```

```
function sinh( x : real) : real;      { calculates sinh(x) }
begin
    sinh : = (exp(x) - exp(-x))/2.0
end;
```

Using the hyperbolic cosine and sine, along with the cosine and sine function of real numbers, you can define the cosine and sine of a complex number *z*, denoted *cos(z)* and *sin(z)* respectively, as follows:

```
cos(z) = cos(x + iy) = cos(x) cosh(y) - isin(x) sinh(y)          (2.3)
```

```
sin(z) = sin(x + iy) = sin(x) cosh(y) - icos(x) sinh(y)          (2.4)
```

Procedures for these operations in Pascal are found in COMPLEX.PAS and are shown in the following, where the returned values `x1` and `y1` are the real and imaginary parts of the answers respectively. For example, to compute the complex cosine:

```
procedure ccos(x, y : real; var x1, y1 : real);
begin
    x1 : = cos(x)*cosh(y);
    y1 : = -sin(x)*sinh(y)
end;
```

To compute the complex sine:

```
procedure csin(x, y : real; var x1, y1 : real);
   begin
    x1 : = sin(x)*cosh(y);
    y1 : = cos(x)*sinh(y)
end;
```

You need a way of finding the exponential of a complex number. First note that Euler's equation[5] relates the exponential to the sine and cosine:

$$e^{ix} = cos(x) + isin(x) \qquad (2.5)$$

Then, to calculate the exponential of a complex number, use the equation:

$$e^z = e^{x+iy} = e^x \cdot e^{iy} = e^x(cosy + isiny) \qquad (2.6)$$

The distributive laws for multiplication over addition yields:

$$e^{x+iy} = e^x cosy + ie^x siny$$

A Pascal procedure to find this exponential, which is found in the COMPEX.PAS file, is:

```
procedure cexp(x, y : real; var x1, y1 : real);
begin
  x1 := exp(x)*cos(y);
  y1 := exp(x)*sin(y)
end;
```

The real part of the answer is returned as `x1`, and `y1` is the imaginary part of the answer.

Finally, it's interesting to note that, from Euler's equation, it's possible to show that the sine and cosine functions of a real number can be defined solely in terms of exponentials, namely:

$$cos(x) = \frac{e^{ix} + e^{-ix}}{2} \tag{2.7}$$

and

$$sin(x) = \frac{e^{ix} - e^{-ix}}{2i} \tag{2.8}$$

For practice, you might want to try to prove this.

Finding attractors of complex functions

In this section, you'll see some beautiful fractals that you can generate by finding the attracting (or escaping) points of iterated complex functions. It's a fascinating characteristic of chaotic systems that you can generate vastly different fractals by a slight alteration of the iterated complex function.

Julia sets

The *Julia set* of a complex function $f(z)$ is the boundary of the set of points that escape; points in the Julia set don't themselves escape, but points arbitrarily close by do. You couldn't possibly determine these points without infinite computer power. Instead, you can find the points that themselves escape and assume that the points that don't escape are arbitrarily close by.

There are three basic techniques for finding Julia sets of complex functions. The first is by computing escaping orbits. The second is called the Inverse Iteration Method (IIM), and the third is called the Boundary Scanning Method (BSM). The two latter ones are superior to first, but the first is easier to code and understand, so I use it throughout the text.

To find escaping orbits, you iterate the function $f(z)$ at each point on a portion of the complex plane centered at (0,0). You iterate the function until either the point attracts or escapes (indifferent points are treated as escaping).

However, finding escaping and attracting points for complex valued functions is a little more difficult than for real valued functions. You can't test complex valued functions to see if they're less than infinity (and greater than minus infinity) because they have an imaginary part. Instead, you need to find the modulus of the complex function at each iteration.

The *modulus* of a complex number z is equal to the square root of the sum of the squares of its real and imaginary parts (remember the Pythagorean theorem?). If $z = a + ib$, then its modulus, denoted $|z|$ is:

$$|z| = \sqrt{a^2 + b^2} \tag{2.9}$$

For example, let $z = 3 + 4i$, then:

$$|z| = \sqrt{3^2 + 4^2} = \sqrt{25} = 5$$

Now let's define attraction and repulsion of complex valued functions. A function $f(z)$ iterated at the point z_0 attracts if the square of its modulus (that's the sum of the squares of its real and complex parts) at any point in the iteration is less than some threshold, which is called the *attractor sensitivity*. The threshold is generally set to be much less than 1. In some cases, minor variations in the attractor sensitivity can result in wild variations in the image produced.

If a function is iterated at a point, and after a certain number of iterations it has not attracted, or if its modulus exceeds some number, then the point has escaped. The number of times that a function is iterated before you decide that it has escaped depends on a couple of factors. First, if the modulus of the iterated function is less then the sensitivity, then the point attracts. If the modulus is less than 100, and the number of iterations is less than the maximum, continue iterating. Finally, if the modulus of the iterated function exceeds 100 or doesn't attract after the maximum allowed iterations, the point is considered as escaping[6]. The maximum number of iterations is actually controlled by the number of allowable colors your screen can display. Most EGA and VGA screens can display 16, and you can use twice this as the maximum number of iterations. If you have a screen capable of displaying 256 colors or more, beware. Many of these programs will then take hours to run. You might want to change the variable `MaxColor` to 16 in this case by changing the line:

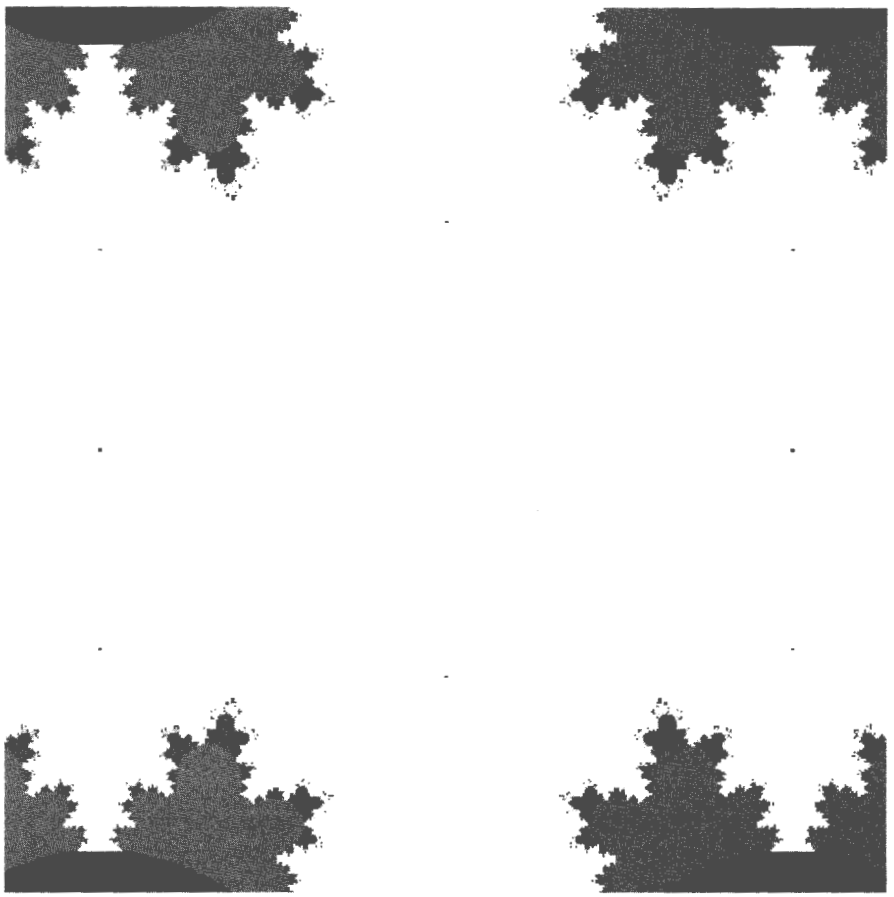
```
MaxColor : = GetMaxColor;
```

to:

```
MaxColor := 16;
```

If you have a very fast computer, however, and you want to leave the code unchanged, you'll be rewarded with images that are incredibly beautiful.

You can start by running some of the programs and generating Julia sets for yourself. Begin by generating the Julia set for $f(z)=cos(z)$ contained in program JULIA1.PAS. Run the program by typing JULIA1 at the DOS prompt. The lovely output is shown in FIG. 2-2. Let's talk about how the program works.



2-2
Julia set for $f(z) = \cos(z)$.

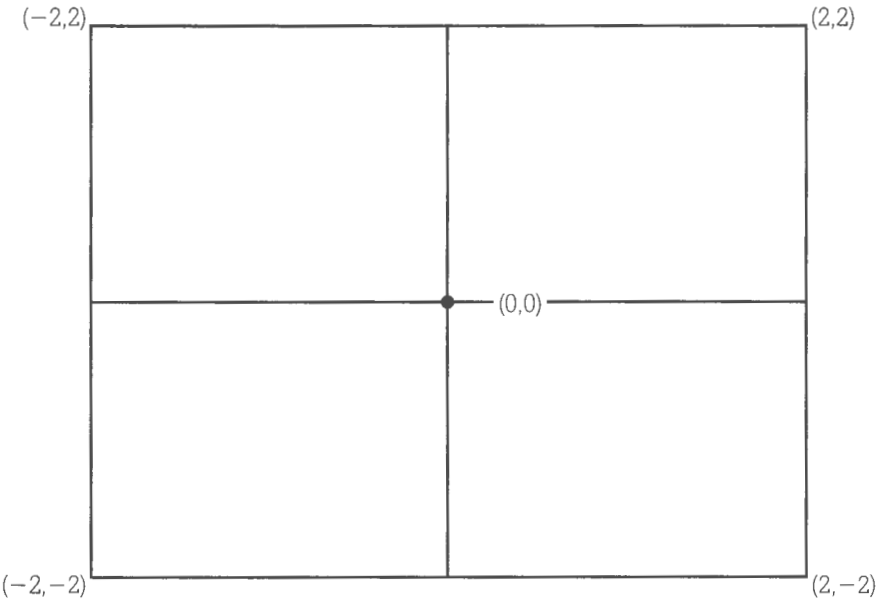
First, you'll notice two constants defined in the beginning of the program:

```
zoom = 2.0;                { create 4 by 4 window }
attract = 0.0001;          { attractor sensitivity }
```

Described fully in appendix A, the first constant establishes a window of the complex plane to be displayed. In this example, both the real and imaginary axis of the complex plane will range from –2 to 2, giving a window of width and height of four units (see FIG. 2-3). The second parameter sets the attractor sensitivity. In this case, an iterated function is assumed to attract at a point if, after a suitable number of iterations (`MaxColor * 2`), the square of the modulus of the number (`x*x + y*y`) is less than 0.0001.

You can play with the attractor sensitivity and zoom factor to obtain different effects. The program, however, does take a long time to run. To see why, consider an ordinary VGA screen with 640 × 480 pixels and 16 colors. Then:

640 • 480 = 307200



2-3
The plotting window for Julia and Mandelbrot programs.

That's how many pixels are on the screen. Since there are 16 colors, you'll have to iterate at each point as many as 32 times. That's 307,200 pixels times 32 iterations, which is 9,830,400 total iterations. Each iteration requires the calculation of a complex sine, which on many PCs takes about a thousandth of a second. The total calculation time is about 9,830 seconds or 2.73 hours!. The actual time is probably better; this analysis was a worst-case one and assumed that all points had to be iterated 32 times, whereas many will attract (or escape) before then. Furthermore, if you have a coprocessor, the code will be at least 50 percent faster. On the other hand, if you have a high-quality screen with many more pixels, the program might take many more hours to run. If this is the case, try using larger zoom factors or modifying the attractor sensitivity so that it's closer to unity.

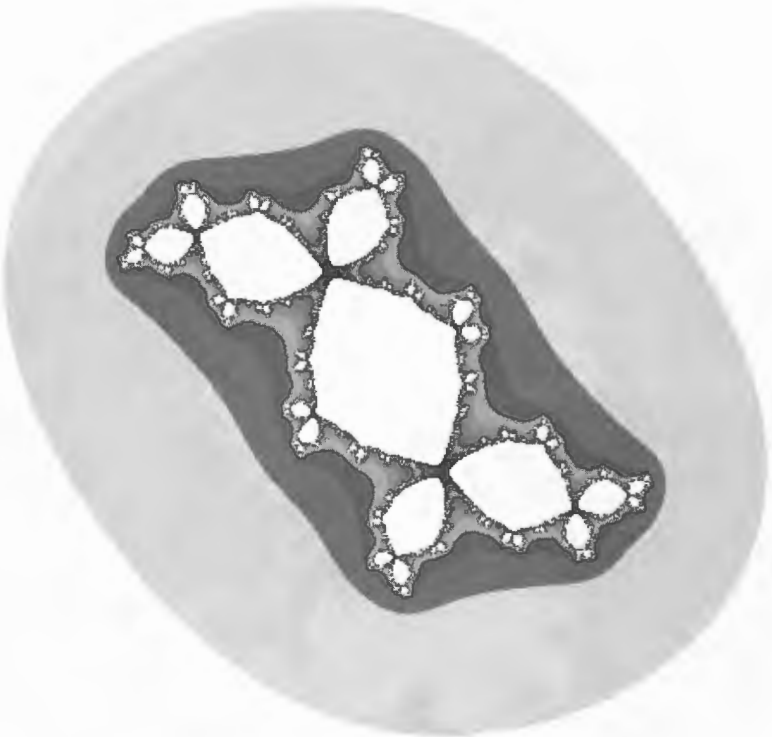
You can generate many other interesting Julia sets with only modest variation of the iterated function $f(z)$. For example, FIG. 2-4 shows what's known as Duoady's rabbit. It's generated by finding the Julia set of the function:

$$f(z) = z^2 + -0.122 + 0.745i$$

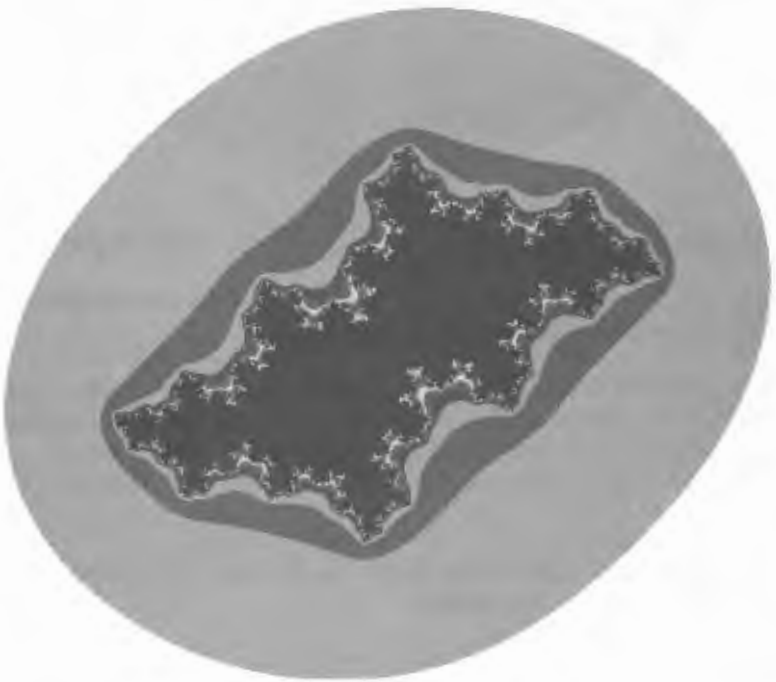
You can generate Duoady's rabbit by running the program RABBIT.PAS, and the "Siegel disk," shown in FIG. 2-5, is a Julia set generated by running the program SIEGEL.PAS. Examine the file to determine which complex function is being iterated in this program.

Next, you can create the dragon-like image shown in FIG. 2-6 by finding the Julia set of:

$$f(z) = z^2 + 0.360284 + 0.100376i$$



2-4
Douady's rabbit.



2-5
A Julia set generated by running the program SIEGEL.PAS.

2-6
A dragon.

You can generate this image by running the program DRAGON.PAS.

Notice how just a slight change in the complex constant being added to the function z^2 dramatically changes the image.

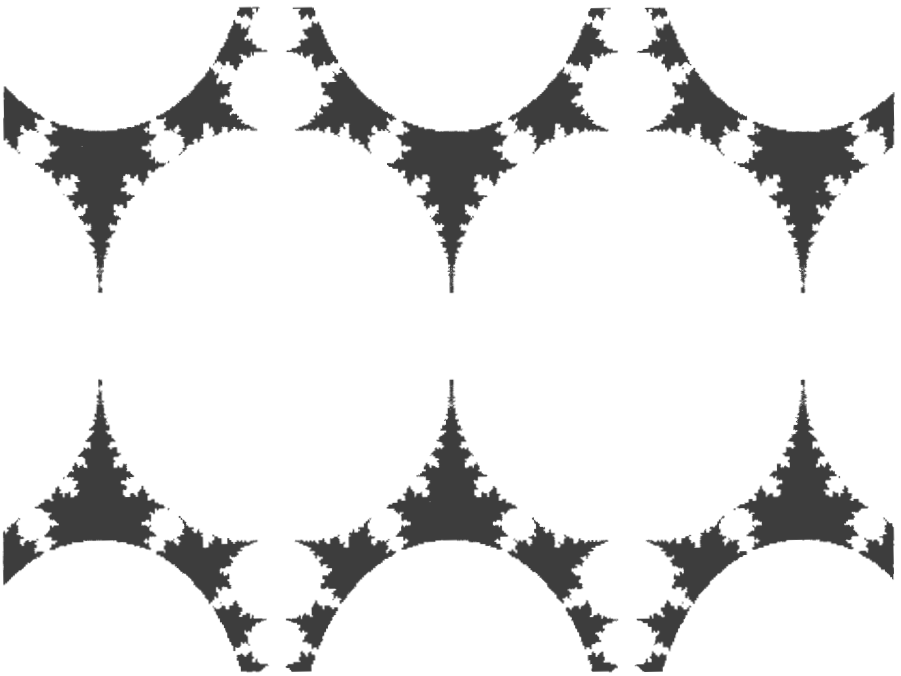
Finally, let's look at the Julia set of $f(z) = sin(z)$. Its image looks like Christmas ornaments and is shown in FIG. 2-7. The image was generated by the program JULIA2.PAS. Because many iterations of the complex sine are being evaluated, this image takes several hours to generate on most home computers.

Incidentally, for fun you should try modifying program JULIA1.PAS to find the Julia sets of the following functions:

1. $f(z) = \pi i e^z$

2. $f(z) = (1 + 0.1i)sinz$

3. $f(z) = 2.965cosz$



2-7
The Julia set of sin(z).

Be sure to use the complex functions found in the Turbo Pascal unit COMPLEX.PAS.

The Mandelbrot set

A different kind of fractal that can be generated by finding escaping and attracting points of a complex function is the Mandelbrot set. A *Mandelbrot set* is the set of complex constants c_i for which the orbits[7] of the function $f(z)=g(z)+c_i$, evaluated at the initial condition of $z_0 = 0$, do not escape. You might have noticed that the Mandelbrot set is somewhat similar to a Julia set, but it's not exactly a graph in the complex plane. Rather, it's a graph of the parameter space determined by the c_i, where the real part of c_i is plotted on the x-axis, and the imaginary part is plotted on the y-axis.

Normally, the Mandelbrot set is the set of points whose orbits do not escape for the function:

$$f(z) = z^2 + c_i \tag{2.11}$$

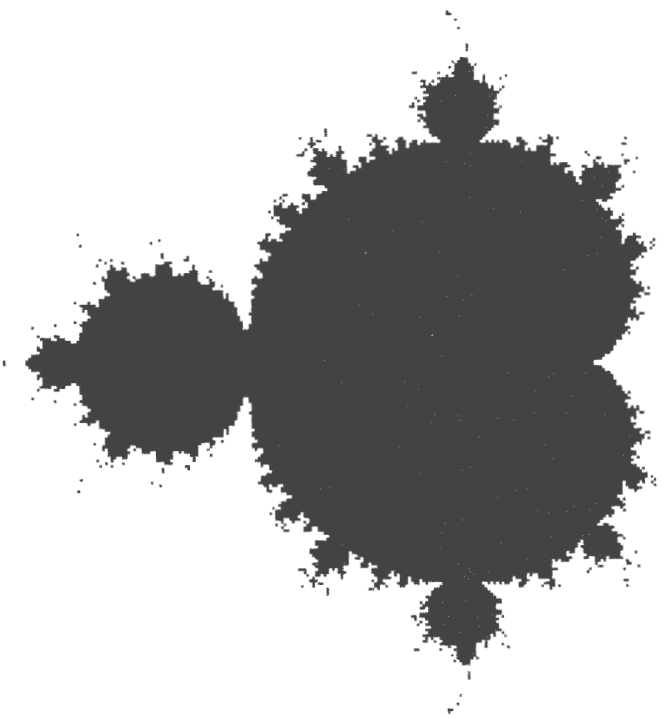
In this case, $g(z) = z^2$ in equation 2.10. However, the "Mandelbrot set," which is named after its discoverer, Benoit Mandelbrot, can be found for other functions of z.

To write a program that generates Mandelbrot sets, you only have to modify the JULIA1.PAS slightly. Whereas with Julia sets you sweep the value of z over some range, here you fix $z = 0$ and sweep the complex constant c. You then check for attracting points and color-escaping points in a similar manner. Let's look at the program that does this, MANDEL.PAS.

In the program, you color the attracting points blue, but you don't color the escaping points. The pertinent code is:

```
begin
    while (iter < 30) and (mag < escape)  do
    begin
      mult(x,y,x,y,x,y);    { square z }
      add(x,y,cx,cy,x,y);   { add c    }
      mag : = x*x+y*y;      { calculate square of modulus }
      iter : = iter + 1;    { increment counter }
    end;
    if mag < escape then    { output blue for non-escapees}
       putpixel(i,j, BLUE)
end  { while loop}
```

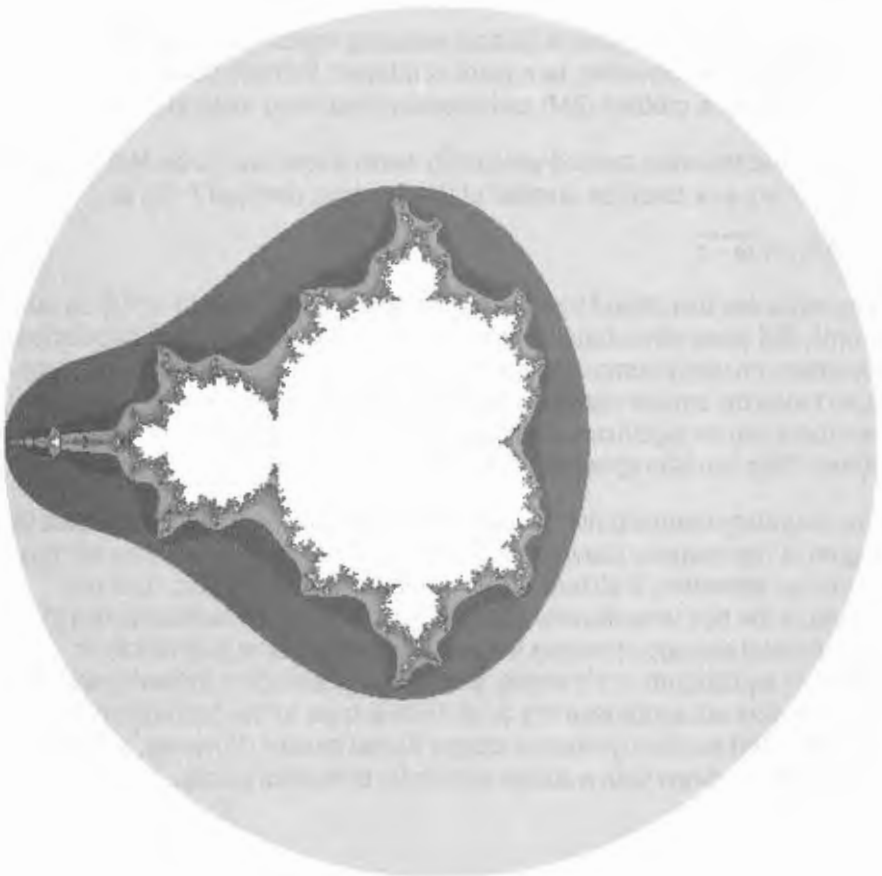
Notice how you test the square of the modulus, variable `mag`, to see if it's less than the escape threshold. If it is, then the point attracts, and you output a blue pixel at the point. Otherwise, you keep iterating the function up to 30 times. The result, called the *filled Mandelbrot set* because it uses only one color, is shown in FIG. 2-8.



2-8
A filled Mandelbrot set.

If, however, you color the escaping points in terms of the number of iterations it takes them to escape, and you don't color the attracting points, as in program MANDEL2.PAS, then you get the beautiful and well-known image of FIG. 2-9, which is generally called "the Mandelbrot set." An important piece of code in MANDEL2.PAS looks like:

```
while (iter < MaxColor * 2) and (mag < escape)  do
begin
  mult(x,y,x,y,x,y);        { square z }
  add(x,y,cx,cy,x,y);       { add c    }
  mag : = x*x+y*y;          { calculate square of modulus }
  iter : = iter + 1;        { increment counter }
end;
if mag > escape then        { color escaping points}
   begin
     putpixel(i,j, iter div 2);
     continue : = FALSE
   end
```



2-9
The Mandelbrot set.

Notice how you output the pixel color in terms of the number of iterations. Also notice that you have a flag, `continue`, which is used to break out of the outer loop. You might want to examine program MANDEL2.PAS more closely to get a better feel for the algorithm.

A note on the images

By making tiny adjustments to the initial conditions in the Mandelbrot and Julia sets, you can generate an amazing variety of different types of fractals. These dynamical systems are very sensitive to minor variations in initial conditions, the accuracy of the computer, and the number of iterations used to determine escape, and all of these can affect the appearance of the final image. For example, some of the images generated here, which you might have seen elsewhere, might differ slightly in levels of detail. Remember that the images in this book have been generated with a simple personal computer with ordinary graphics and not on a supercomputer with high-resolution graphics. The details might be different, but the overall morphology or shape is the same. Finally, the choice of colors and their assignment is arbitrary.

Inverse iteration & boundary scanning methods

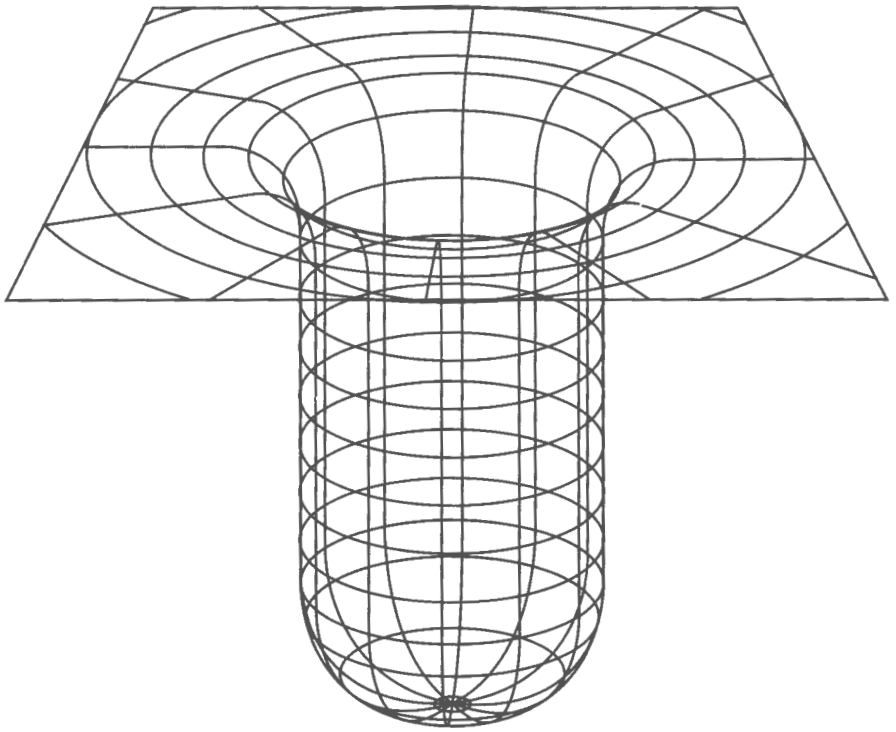
This book uses the method of finding escaping orbits to generate Julia and Mandelbrot sets. However, as a point of interest, I should briefly discuss the inverse iteration method (IIM) and boundary scanning method (BSM).

The inverse iteration method essentially takes advantage of the fact that if $f(z) = z^2 + c = \omega$, then the *inverse*[8] of the function, denoted $f^{-1}(z)$, is:

$$f^{-1}(z) = \pm\sqrt{\omega - c}$$

The orbits are then found by iterating randomly on $+f^{-1}(z)$ and $-f^{-1}(z)$. In other words, IIM takes advantage of symmetry to halve the number of calculations. However, on many computers the square root operation takes much longer than twice the square operation (so the savings might not always be there), but there can be significant savings in memory use if tables of values are stored. This can also speed execution.

The boundary scanning method works by sliding a box or window across the region of the complex plane to be plotted. The points at the corners are then tested for attraction. If all four points attract to the same point, then the center of the box is an attractor; otherwise it repels. The technique is a three-dimensional analogy of testing if a car on a roller coaster is at stable or unstable equilibrium. If it's stable, then the cuplike region formed is said to be a *basin of attraction* (see FIG. 2-10). BSM is truer to the definition of the Julia set, and so often generates crisper fractal images. However, it takes far more computations than a simple search for attracting points.



2-10
Basin of attraction used in the boundary scanning method.

Three-dimensional fractals

You can generate fractals in three dimensions, but instead of plotting points on a plane, you need a cube or three-dimensional space. For this purpose, complex numbers aren't sufficient; instead, you need more sophisticated mathematical tools called *quaternions*. These are hyper-complex numbers or, in essence, a pair of complex numbers.

A quaternion q is written as:

$$q = \omega + ix + jy + kz \qquad (2.12)$$

The numbers i, j, and k are all the positive square root of –1. That is:

$$i^2 = j^2 = k^2 = -1$$

Note that if y and z are equal to 0, q is just a complex number. In three-dimensional fractals, if you set $w = 0$, the x, y and z terms are used to select the x, y *and* z coordinates of a pixel in three-dimensional space.

The manipulation of quaternions is much more complicated than for complex numbers because quaternions and their associated operations form what's termed a *noncommutative algebra*. What this means is very simple. Whereas $x \cdot y = y \cdot x$ for any two real or complex numbers, with quaternions this doesn't hold!

For example, for quaternions, the following is true:

$i \bullet j = k$

However:

$j \bullet i = -k$

In general, quaternions satisfy the multiplicative rules:

$i^2 = j^2 = k^2 = -1$

$i \bullet j = -j \bullet i = k$

$j \bullet k = -k \bullet \mathrm{j} = i$

$k \bullet i = -i \bullet k = j$

Quaternions are used in the generation of three-dimensional fractals and in three-dimensional rotational *kinematics* (the study of motion). A further study of quaternions is beyond the scope of this book, but it's interesting to note that three-dimensional fractals are likely to appear in many applications.

3 Chaos & fractals in nature

"And Chaos, ancestors of Nature, hold
Eternal anarchy, amidst the noise"

— John Milton, *Paradise Lost*

This chapter looks at natural phenomena that are chaotic in nature and discusses how you can model natural phenomena in terms of fractals. You might be especially interested in writing programs that can simulate natural beauty.

Population dynamics

You can see the chaos of nature in the study of population dynamics, particularly in the relationship between predator and prey. Although the models used are necessarily simplistic, they provide significant insight into the interrelationship between animals in a small part of the food chain.

For example, suppose an ecologist is studying the population of caribou on an island in Canada. The population is unstable because of crowding, disease, and lack of food. The ecologist proposes to introduce some wolves on the island to help stabilize the population.

Let's model this system and see why it's highly unstable. Let *caribou(t)*, *wolf(t)* be the number of caribou and wolves at time *t*, respectively, and let $caribou_b$ be the rate of birth of the caribou. If there were unlimited resources of food, space, and so on, then the excess of the birth rate over the death rate

for the caribou is positive. In the absence of predators, then the population of caribou grows at a rate of:

$$\text{growth}(t) = \text{caribou}_b \cdot \text{caribou}(t) \tag{3.1}$$

The death rate from wolves depends on the number of encounters between wolves and caribou, K, and is assumed to be proportional to the number of caribou:

$$\text{death}(t) = K \cdot \text{caribou}(t) \cdot \text{wolf}(t) \tag{3.2}$$

Then the equation controlling the caribou population is:

$$\text{caribou}(t+1) = \text{caribou}(t) + \text{growth}(t) - \text{death}(t)$$

or

$$\text{caribou}(t+1) = \text{caribou}(t) + \text{caribou}_b \cdot \text{caribou}(t) - K \cdot \text{caribou}(t) \cdot \text{wolf}(t)$$

To simplify the model, assume that the death of each caribou results in the birth of one wolf. This is the only means by which the wolf population can grow. However, it's subject to a death rate of $wolf_d$. Thus, the wolf population is:

$$\text{wolf}(t+1) = \text{wolf}(t) + K \cdot \text{caribou}(t) \cdot \text{wolf}(t) - \text{wolf}_d \cdot \text{wolf}(t) \tag{3.3}$$

I've set up this simulation as a *discrete simulation*, which means that I've used a finite difference equation to model it[1]. I could have created a *continuous simulation*, but this would have involved a mathematical tool called a *differential equation* and very sophisticated software to solve the equation.

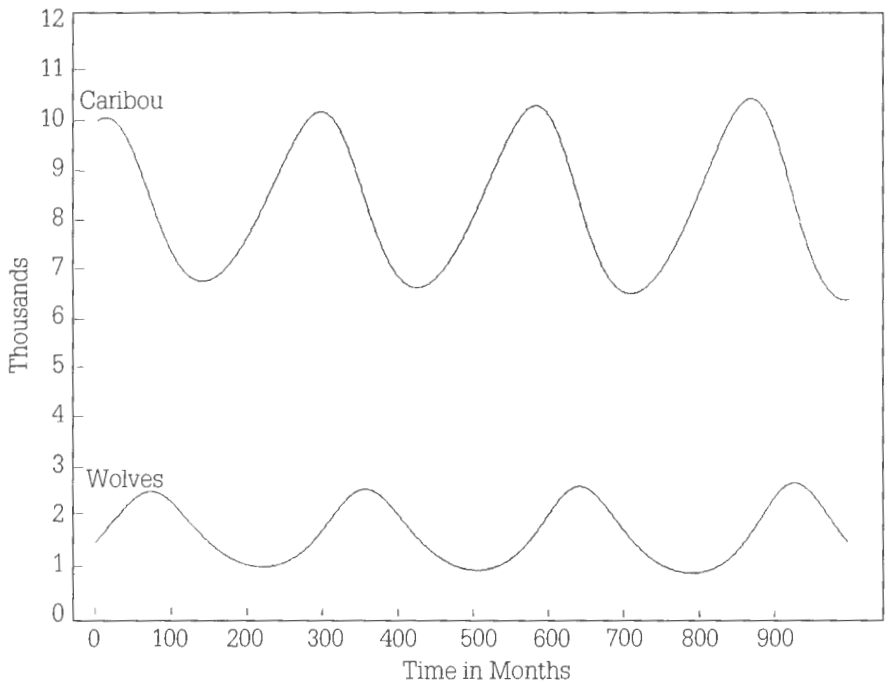
By correct selection of the predator and prey populations, the birth rates, and the death rates, the system should show stable oscillations of both species. Otherwise, the system will become unstable (chaotic), resulting in the extinction of the wolf or both populations.

To illustrate, a Pascal program to calculate the caribou and wolf populations is given in the PREY.PAS file on your disk. You should run this program as you follow this discussion. You can also use a common spreadsheet application program such as Lotus 1-2-3 to generate graphs that give a visual representation of the population dynamics. Included on your disk is a Lotus version 3.0 file, WOLVES.WK3, which simulates the caribou-wolf system. You need to copy row 6 down to row 1000 to simulate 1000 months of activity. For discussion, I've included some of the graphs generated using Lotus 1-2-3.

You'll find that our little predator-prey system is not very sensitive to the initial populations. However, it's extremely sensitive to the death rate to contact ratio K.

For example, FIG. 3-1 depicts the populations over 1000 months with the following parameters:

- Initial population of caribou, *caribou*(0) = 10,000
- Initial population of wolves, *wolf*(0) = 1500
- Birth rate for caribou, $caribou_b$ = .01
- Death rate for wolves, $wolf_d$ = .05
- Death rate contact ratio K = .000006

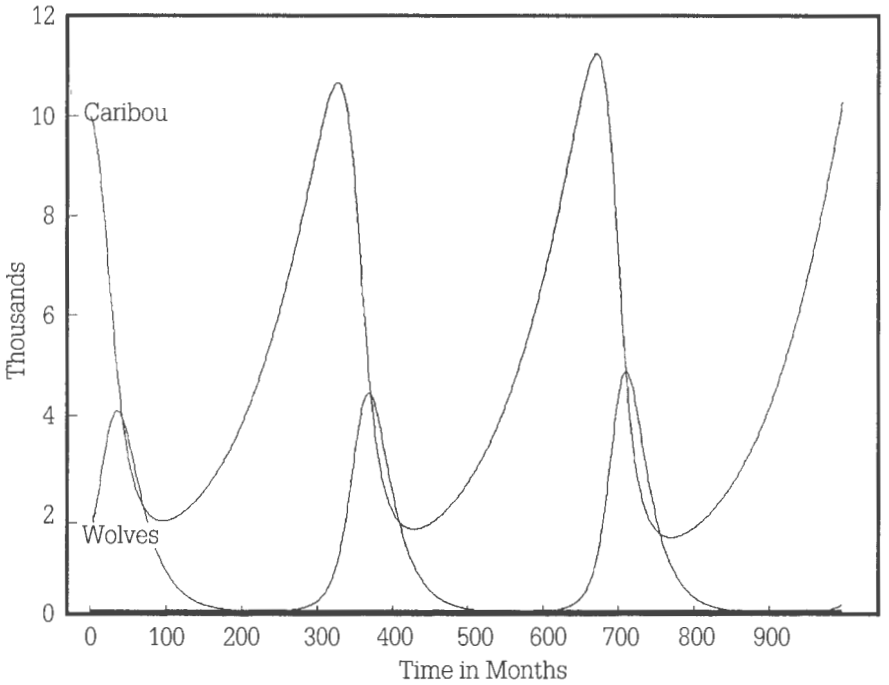


3-1
Population dynamics of the caribou-wolf system with K = .000006.

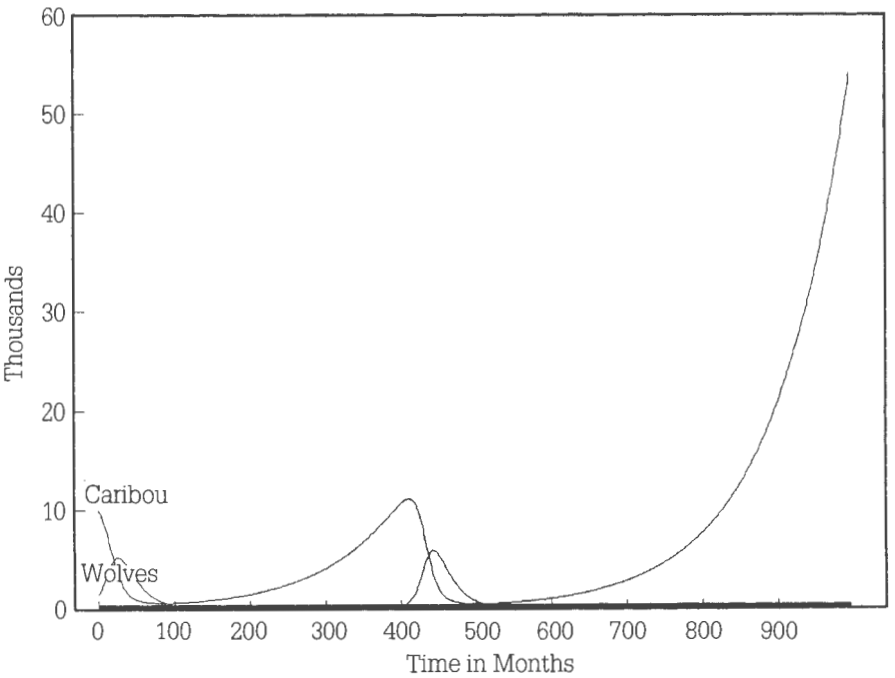
Try running PREY.PAS with these values. Notice how this is a nicely stable system. When the caribou population gets too high, the wolf population increases shortly thereafter to keep it in check. When the caribou population drops, the wolf population falls soon after.

However, if you change the parameter K, the death contact ratio, even slightly to K = .00001, you generate the population profile shown in FIG. 3-2. Try running PREY.PAS with these values. Notice how there are wild swings in both populations, and at times, the wolf population is dangerously close to extinction.

Finally, when parameter K = .000014, the system is completely unstable, as depicted in FIG. 3-3. The initial population of caribou is quickly decimated, leading to the eventual drop in wolves. Both populations make a weak recovery, but when the caribou population drops, the wolves are eventually



3-2
Population dynamics of the caribou-wolf system with K = .00001.



3-3
Population dynamics of the caribou-wolf system with K = .000014.

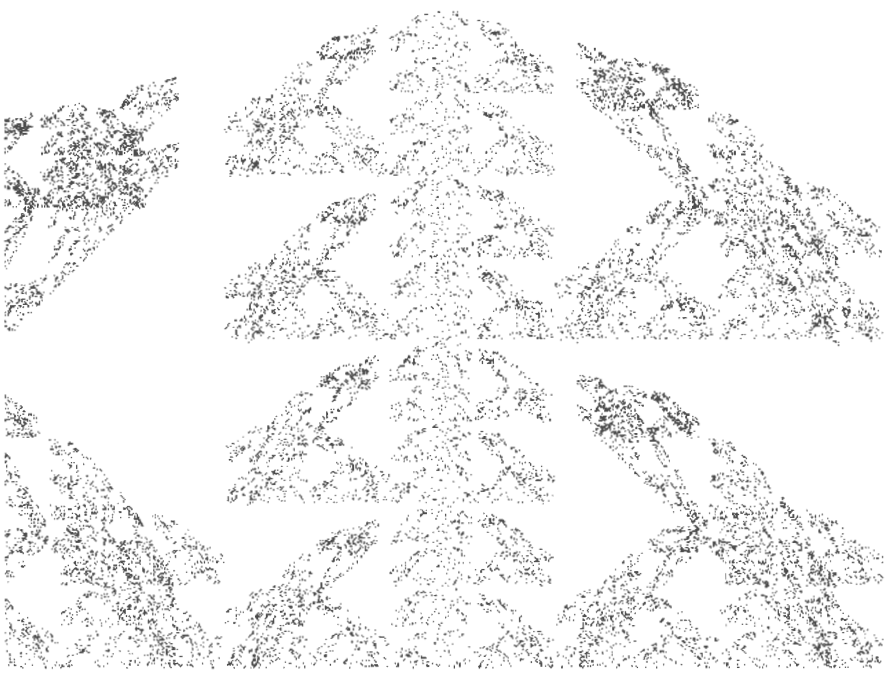
extinct. This leads to an explosion in caribou, which would probably drop dramatically due to lack of food, space, disease, and so on, although this is not captured by the model. Using the spreadsheet model, or program PREY.PAS, you should experiment with initial conditions, and the various parameters to determine which of them lead to instability.

Animals

You can use fractals to generate images that resemble many types of animals. For example, you can see an infinite number of self-similar seals or dolphins frolicking in FIG. 3-4. I generated this image by running program SEAL.PAS with the IFS codes given in TABLE 3-1.

Table 3-1
IFS transformation rule for seals.

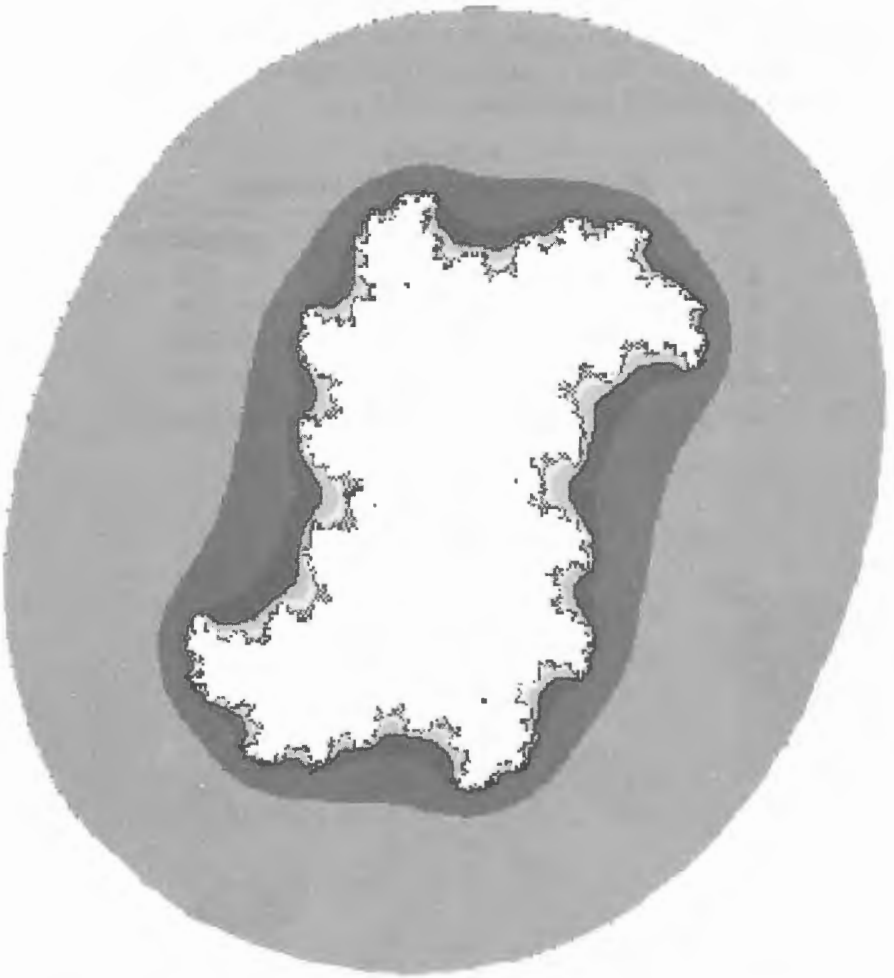
	1	2	3	4	5	6	probability
1	–0.5	0	0	0.5	0	0	0.25
2	–0.5	0	0	0.5	2	0	0.25
3	–0.4	0	1	0.4	0	1	0.25
4	–0.5	0	0	0.5	2	1	0.25



3-4
Seals.

Genetics

It's no wonder that some genetic researchers theorize that the mutation of genes—previously thought to be random—is not, but rather chaotic with various strange attractors. Moreover, you can conjecture about the morphology of cells. Is the shape of a cell random or chaotic? Figure 3-5 shows an amoeba-like image generated from the filled Julia set of $f(z) = z^2 + .3 - 4i$.



3-5
An amoeba-like image generated from the filled Julia set of $f(z) = z^2 + .3 - 4i$.

Weather

The weather is widely known to be a chaotic system. Storms and calm weather often appear without explanation. Embarrassed weather-people are constantly trying to decide where a certain prediction went awry. Edward Lorenz, one of the fathers of meteorology, and the first to recognize chaos in climatic systems, noted that in theory the flapping of a butterfly's wings in Tokyo might cause a storm over New York.

In this sense, measuring the weather can also affect it. Certainly the devices measuring wind speed have a more profound influence than the flapping of a butterfly's wings. This reminds me of the well-known principle in physics called Heisenberg Uncertainty. The principle states that an observer can't know precisely the position and velocity of a particle at the same instant. An interpretation of this is that in measuring the position or velocity of the particle, the measuring instrumentation changes one or both. Could this mean that, by measuring the forces that determine weather, you're doomed to affect it, thus rendering your predictions hopelessly inaccurate?

Scenes from nature

In this section you'll see many beautiful computer-generated images, most of which were created by playing with the data in the transformation matrix of Barnsley's iterated function system algorithm.

Trees, leaves, & flowers

You can generate many beautiful trees, leaves, and flowers using both IFS and Julia set fractals. For example, one of the most commonly seen fractals is the black spleenworth fern leaf shown in FIG. 3-6. This was generated by iterating a well-known mapping rule, encoded in the program FERN.PAS, which you can run. Table 3-2 shows a matrix-encoded form of the mapping for the fern.

Table 3-2
IFS transformation rule for fern.

	1	2	3	4	5	6	probability
1	0.5	0	0	0.16	0	0	0.01
2	0.85	0.04	−0.04	0.85	0	1.6	0.85
3	0.2	0.26	0.23	0.22	0	1.6	0.07
4	0.15	0.28	0.26	0.24	0	0.44	0.07

By changing the parameters in the IFS matrix, you can generate a tree using code similar to the one used for the fern. In this case, the program is called TREE.PAS, and it uses the mapping rule described in TABLE 3-3. The output of the program is shown in FIG. 3-7.

Table 3-3
IFS transformation rule for tree.

	1	2	3	4	5	6	probability
1	0	0	0	0.5	0	0	0.05
2	0.42	−0.42	0.42	0.42	0	0.2	0.40
3	0.42	0.42	−0.42	0.42	0	0.2	0.40
4	0.1	0.0	0	0.1	0	0.2	0.15

3-6
A fern leaf.

3-7
IFS representation of a tree.

By outputting many trees of different size, color, and position, you can create a forest. Program FOREST.PAS does just that, and its output is shown in FIG. 3-8. When you run the program, trees start popping up all over the place, like some primeval forest. It really is quite a wonderful effect.

The program is essentially the same as the other IFS programs except that the following code has been added:

```
xpos : = random(MaxX);              { pick tree position }
ypos : = random(MaxX);
scale : = random(3) + 1;            { pick tree scale }
crand : = random(10) + 1;           { pick tree color }
case  crand of
   0,1,2,3,4,5,6,7,8:
        color : = GREEN;            { most trees are green }
   9  : color : = YELLOW;           { some trees are yellow }
   10 : color : = BROWN;            { some trees die }
end;
```

The first two lines select a starting *x* and *y*-coordinate for the root of the tree between one-fourth and eleven-twelfths of the way from the edges of the screen. The scale of the tree is selected from between one and three, so that the largest trees are three times larger than the smallest, and some are in between. The fourth line selects a random number between one and ten, so that a color can be assigned. The program assumes that 80 percent of the

3-8
A forest of randomly generated fractal trees.

trees are green, while 10 percent are dead (brown) and another 10 percent are yellow. With these criteria, the program then proceeds to generate 100 trees. You'll use this approach to generate other scenes.

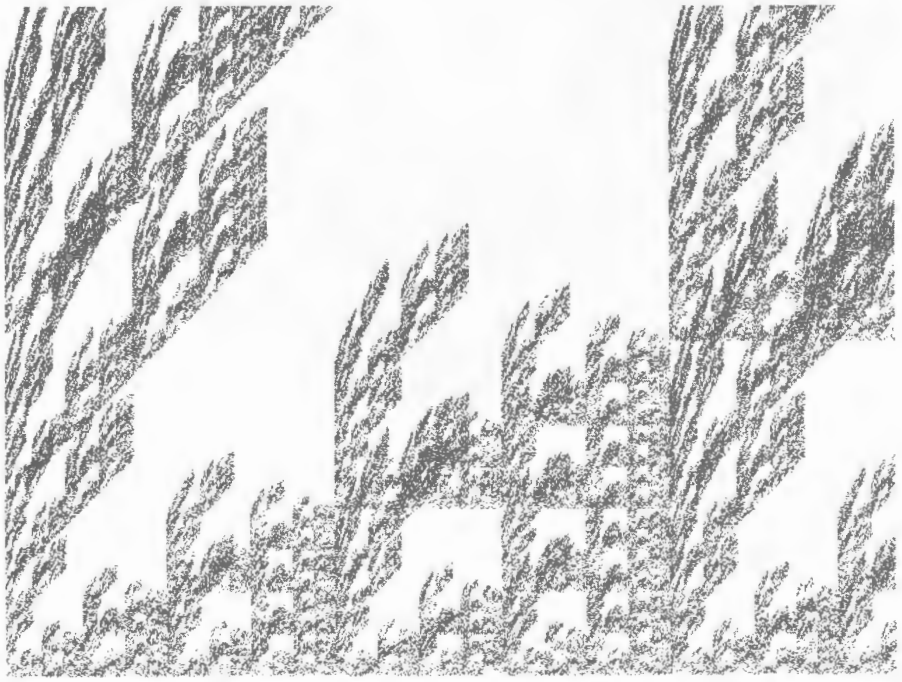
Another type of forest is illustrated in FIG. 3-9 and can be generated by running program REDMOSCL.PAS. Here you see a view of a redwood forest with a lush green floor and huge trees, whose tops are obscured by a mist. Again, the effect was achieved with iterated function systems. The floor of the forest is simply composed of trees again, while the redwoods and mist were generated with other IFSs with different parameters.

Finally, I produced green seaweed using the program SEAWEED.PAS. Figure 3-10 shows the output. Table 3-4 gives the IFS codes for the seaweed.

Table 3-4
IFS transformation rule for seaweed.

	1	2	3	4	5	6	probability
1	0.5	0	0	0.5	0	0	0.25
2	0.5	0	0	0.5	2	0	0.25
3	0.4	0	1	0.4	0	1	0.25
4	0.5	0	0	0.5	2	1	0.25

3-9
A redwood forest.

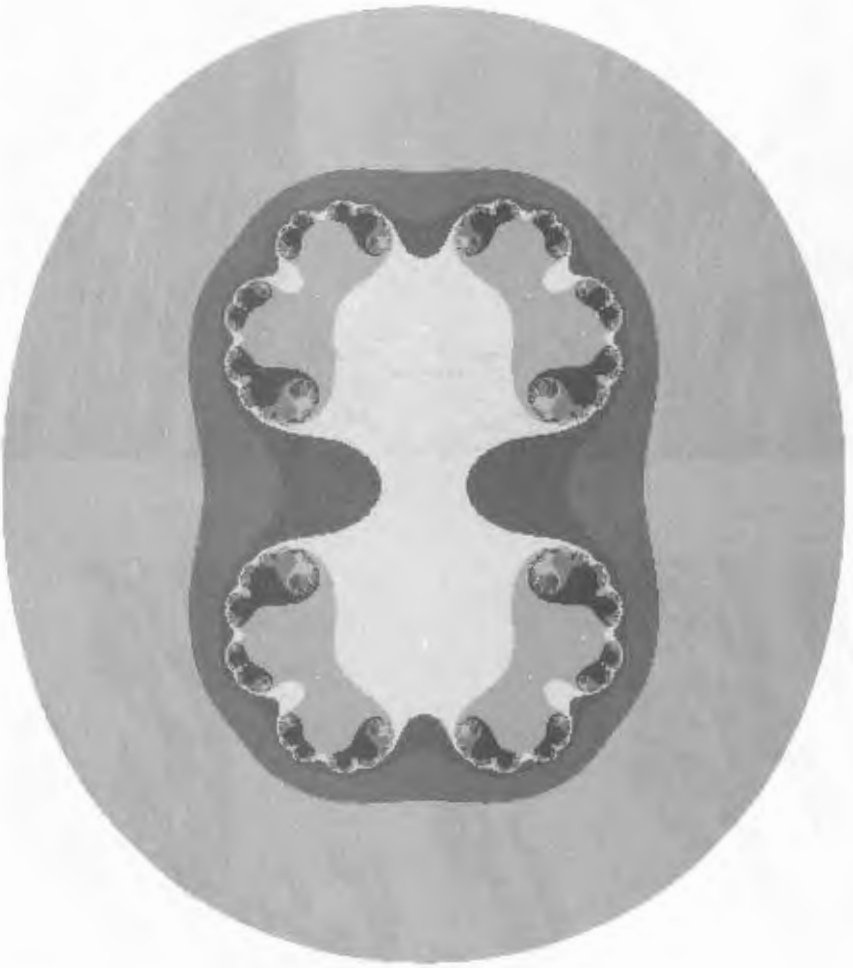


3-10
Green seaweed.

Using Julia sets, you can create beautiful flowers. For example, the program FLOWER1.PAS implements the Julia set of:

$f(z) = z^2 + 0.384$

This generates the lovely four-petaled rose shown in FIG. 3-11.

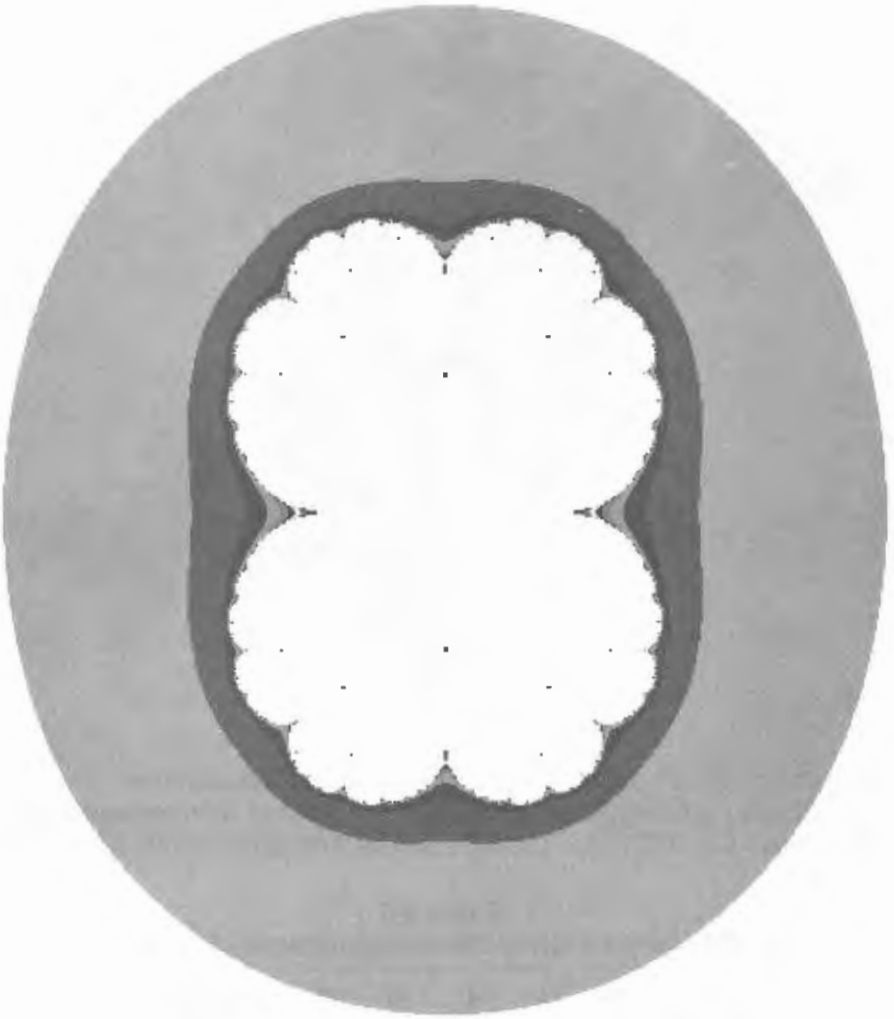


3-11
Four-petaled flower from the Julia set of $f(z) = z^2 + 0.384$.

You can generate a chrysanthemum by changing the constant term slightly to 0.2541. That is:

$f(z) = z^2 + 0.2541$

The program FLOWER2.PAS implements this, and the output is shown in FIG. 3-12.



3-12
Another four-petaled flower from the Julia set of $f(z) = z^2 + 0.2541$.

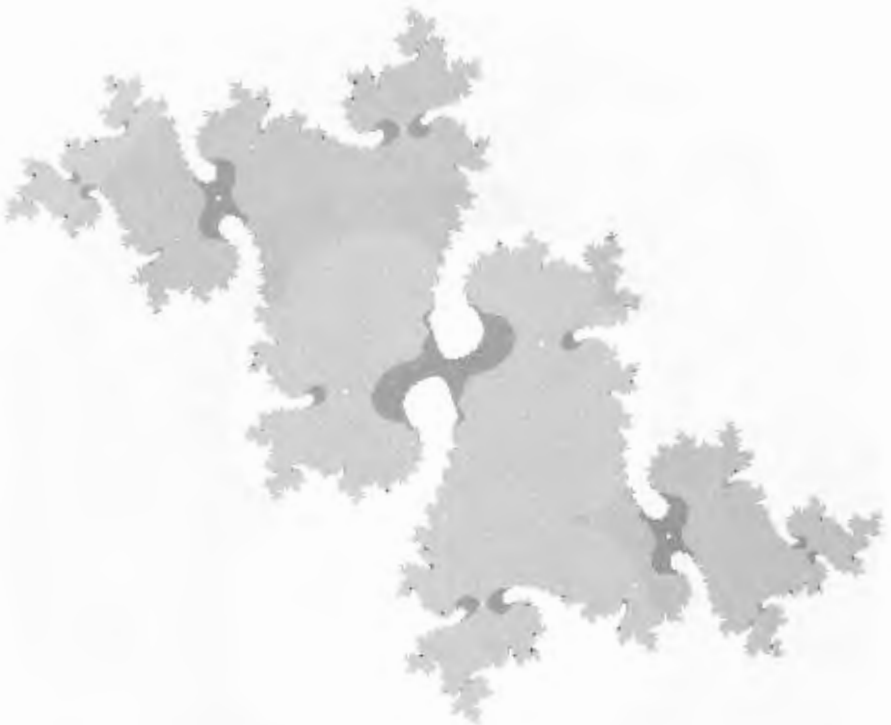
Clouds

You can exploit the billowy appearance of some fractals to generate cloudlike pictures. For example, FIG. 3-13 shows a threatening storm cloud generated by program CLOUD.PAS. The program finds the Julia set of:

$f(z) = z^2 - 0.194 + 0.6557i$

It suppresses all colors except white and yellow, which are mapped into the colors light gray and dark gray using the case statement:

```
case iter div 2 of
  WHITE: putpixel(i,j, LightGray);
  YELLOW:putpixel(i,j, DarkGray)
end;
```



3-13
A fractal storm cloud.

You can also use IFS systems to generate clouds that appear three-dimensional. For example, the clouds shown in FIG. 3-14 were generated with the program CLOUDS2.PAS. The IFS codes for it are given in TABLE 3-5.

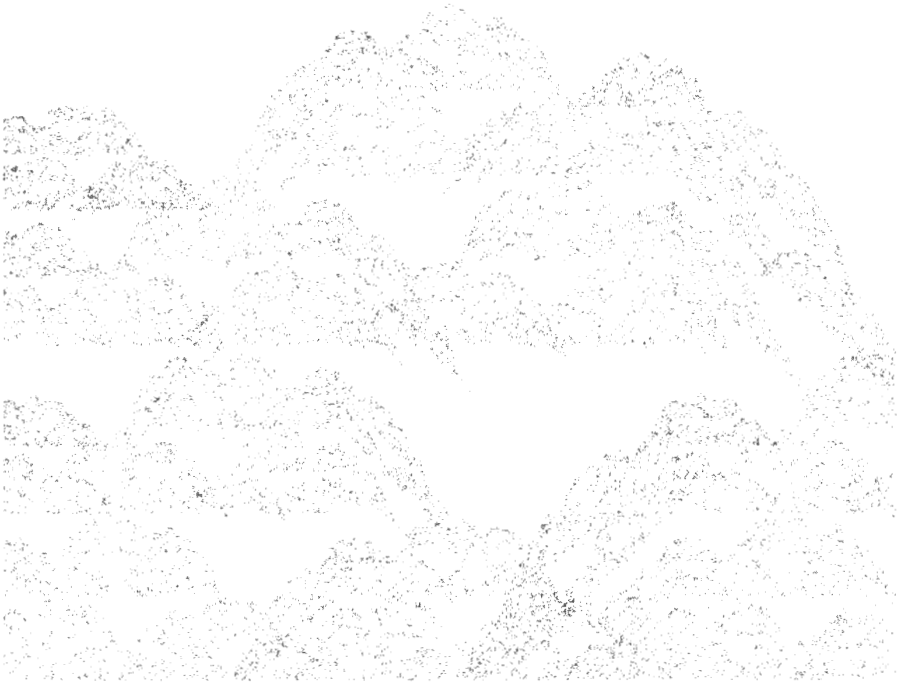
Table 3-5
IFS codes for three-dimensional fractal clouds.

	1	2	3	4	5	6	probability
1	0.5	0	0	0.5	0	0	0.25
2	0.5	0	0	0.5	2	0	0.25
3	−0.4	0	1	0.4	0	1	0.25
4	−0.5	0	0	0.5	2	1	0.25

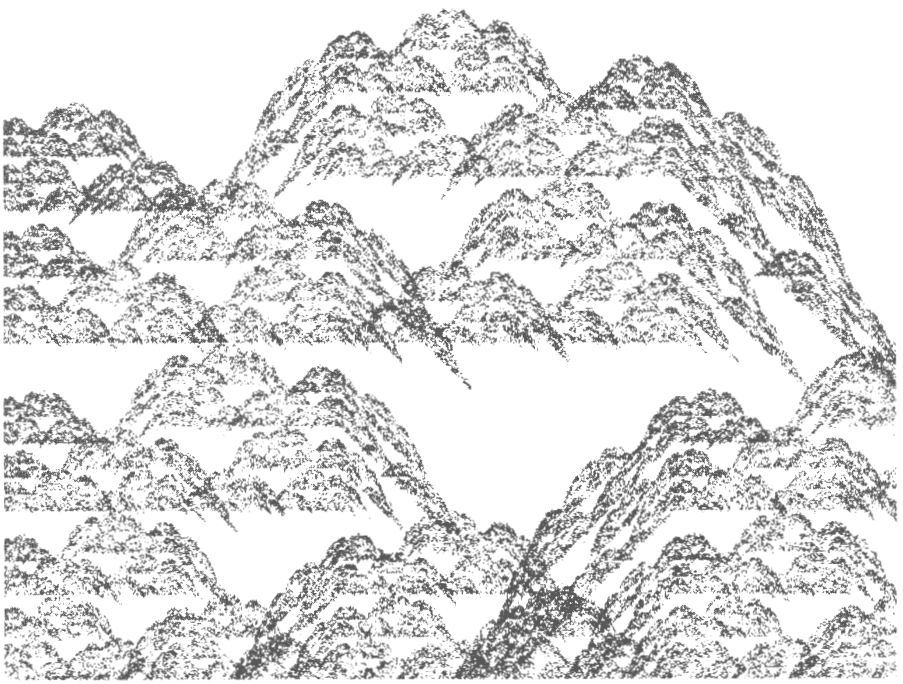
Rocks

The generation of rocks and clouds can be handled similarly by changing colors. For example, try changing colors in program CLOUD.PAS by mapping white into brown, and yellow into red to generate a rock-like formation.

Another way to generate a rock formation is with iterated function systems. For example, consider the IFS code table for program CLOUDS2.PAS. By changing the color to brown, you can generate the rocks shown in FIG. 3-15.



3-14
"Three-dimensional" fractal clouds.

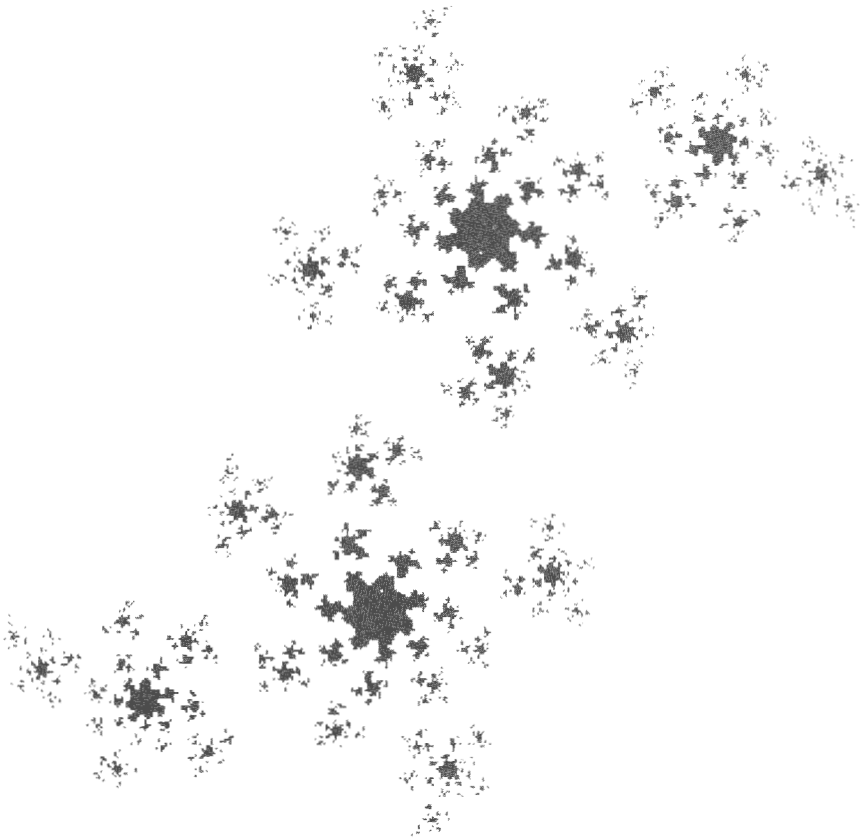


3-15
Fractal rocks using the same IFS codes as fractal clouds.

Snowflakes

Using fractal techniques, you can easily generate images that look like snowflakes. One type of snowflake, shown in FIG. 3-16, was generated from the Julia set of:

$$f(z) = z^2 + 0.11031 - 0.67037i$$

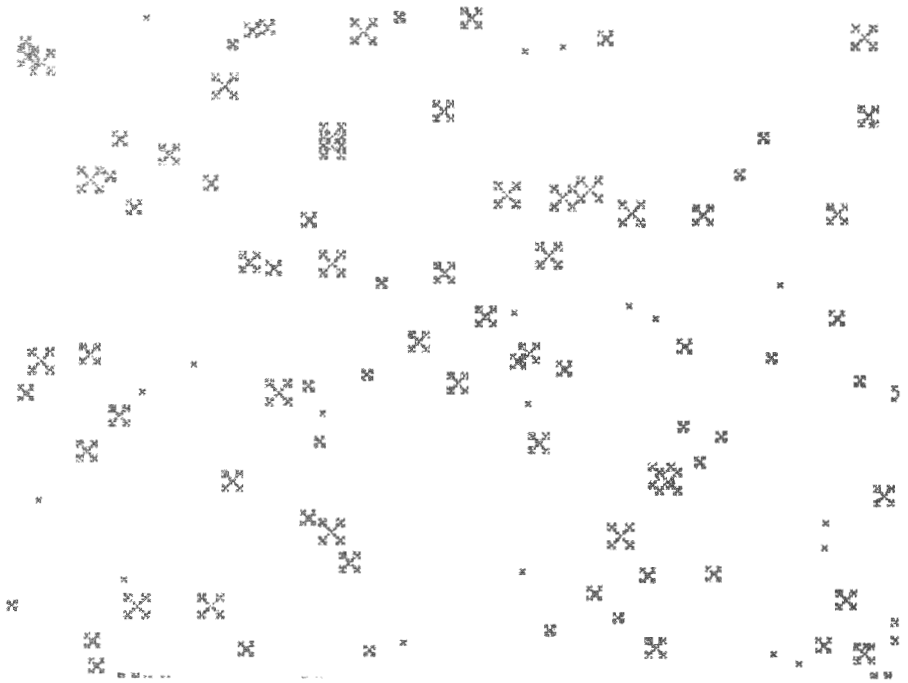


3-16
Snowflakes generated by SNOW.PAS.

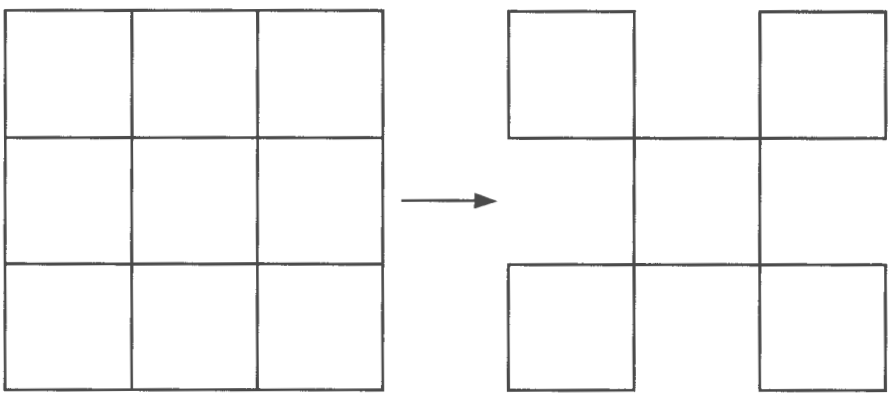
This marvelous effect was achieved by running the program SNOW.PAS and coloring only the white pixels, with the code:

```
if (iter div 2 = WHITE)
   putpixel(i,j, WHITE);
```

Finally, you can generate a lovely snowfall, as shown in FIG. 3-17, by repeated random generation of the "cross fractal," much in the same way the forest was generated. Look at one of the snowflakes. It's just a square divided into nine equal parts with the four outer-middle boxes removed, as shown in FIG. 3-18. The cross fractal is generated with the IFS codes shown in TABLE 3-6. Many of these little fractals are generated in different scales and positions to achieve the effect. You can produce the snowfall by running the FALL.PAS program. The resulting image appears to progress from a flurry to a blizzard.



3-17
A snowfall generated by FALL.PAS.



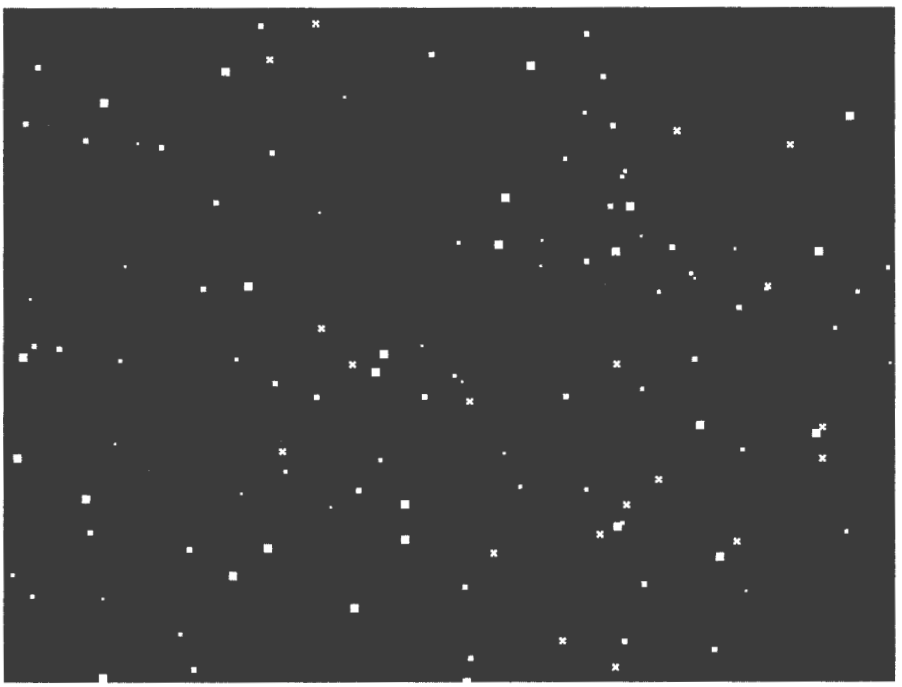
3-18
Cross fractal used to generate snowflakes.

Table 3-6
IFS codes for cross fractal.

	1	2	3	4	5	6	probability
1	0.33	0	0	0.33	1	1	0.20
2	0.33	0	0	0.33	MaxX	1	0.20
3	0.33	0	0	0.33	1	MaxX	0.20
4	0.33	0	0	0.33	MaxX	MaxX	0.20
5	0.33	0	0	0.33	MaxX div 2	MaxX div 2	0.20

Galaxies

Slight modification of the FALL.PAS program yields what appears to be the view of some unknown region of space shown in FIG. 3-19. By changing the snowflake scaling factor so that it's very small, the flakes become stars. You can see this by running the GALAX1.PAS program. Finally, by looking again at FIG. 3-16, you can see that it resembles twin swirling galaxies.



3-19
A randomly generated view of space.

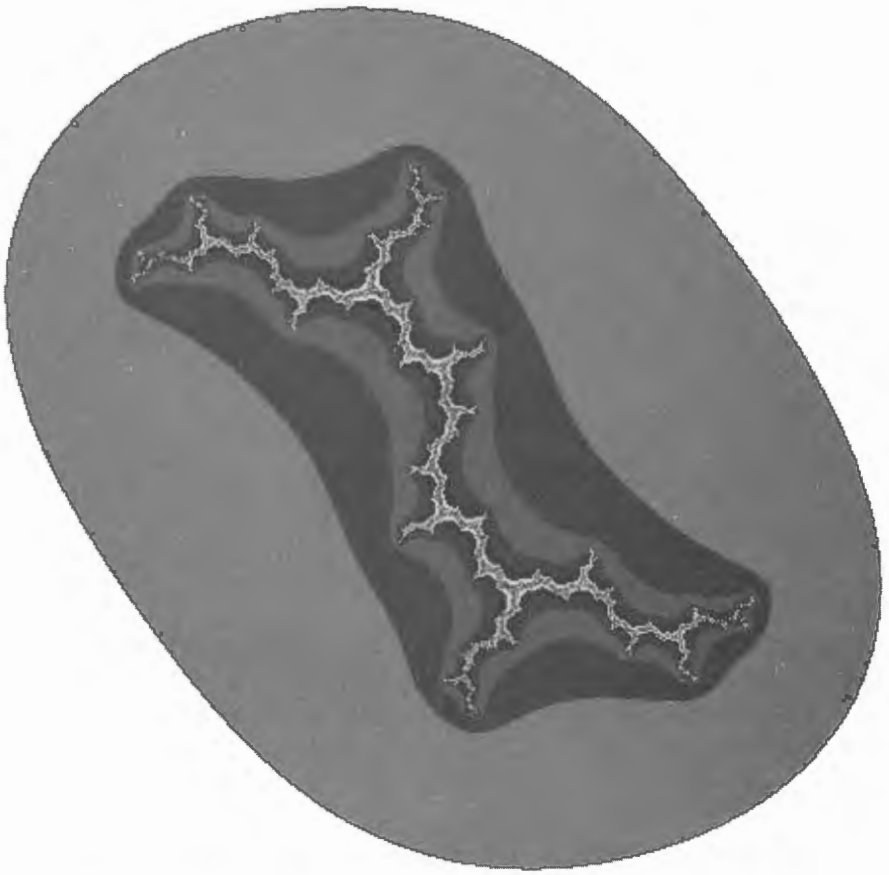
Coastlines

Any one of the fractals generated using Julia sets (for example, FIGS. 3-13 and 3-20) could represent the coastline of some mythical country viewed from above. Because the fractal is self-similar, it doesn't matter whether the viewer is one-thousand miles or one foot away.

In addition, any section of these fractals has the property of infinite length. This might defy intuition, but if you tried to use a piece of string to trace the coastal outline, you'd run out of string. Perhaps the coast of England is infinitely long, as Mandelbrot has said!

Fractals in the human body

Many structures within the human body suggest a complex interrelation between biological development, form, and function. Scientists have wondered if underlying physical constraints lead, through scaling, to the ultimate form of plants and animals. For example, does the shape of a DNA molecule have a direct relationship to the shape of the organism it describes? Let's look at some instances where the human body might harbor fractals.



3-20
Dendrite structure generated by $f(z) = z^2 + i$.

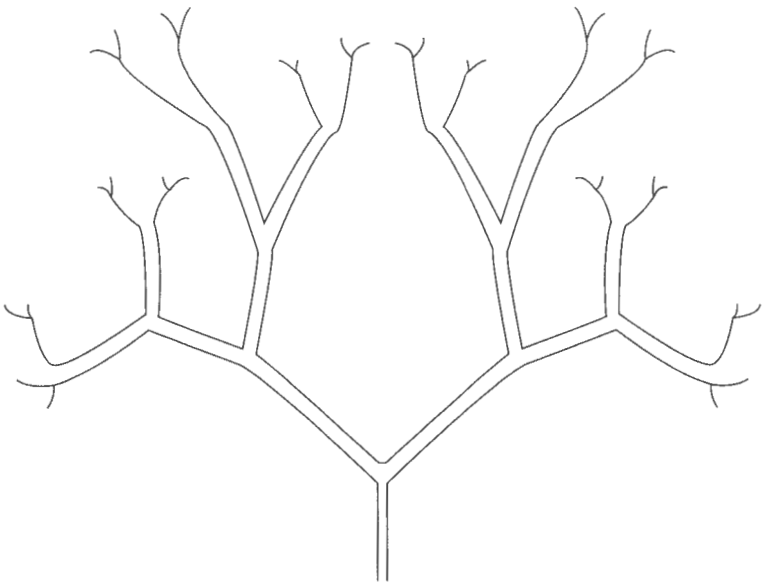
Bronchial growth

Our lungs contain millions of air sacs called *alveoli*, which provide a mechanism for the exchange of gases. These are connected via increasingly larger *bronchial tubes* to the trachea in a structure shown in FIG. 3-21. That structure is very similar to the tree fractal previously shown in FIG. 3-7.

As the bronchial tree branches out, its tubes decrease in size. From one branching to the next, the diameter decreases at about the same ratio until there's a change in the mechanism of flow, from minimum resistance near the beginning to molecular diffusion within the alveoli. This structure might also be similar to neural connections in the brain.

Neuron growth

Some researchers have suggested modeling the wiring of the brain and neuron growth using fractal bifurcation patterns (DeAngelis 1993). For example, the tree fractal has been cited as one mechanism for neuron wiring.[2] In addition, neural activity tends to be fractal-like and chaotic—more so when the brain is involved in active problem solving (DeAngelis 1993).



3-21
The bronchial tree structure in our lungs resembles a fractal.

With fractals, you can generate a structure that resembles *dendrites*, the main connectors between the neural processing elements of the brain. For example, look again at FIG. 3-20, which shows the filled Julia set generated by the function:

$f(z) = z^2 + i$

Could the function f be the underlying mathematics behind the dendrite? Nobody knows.

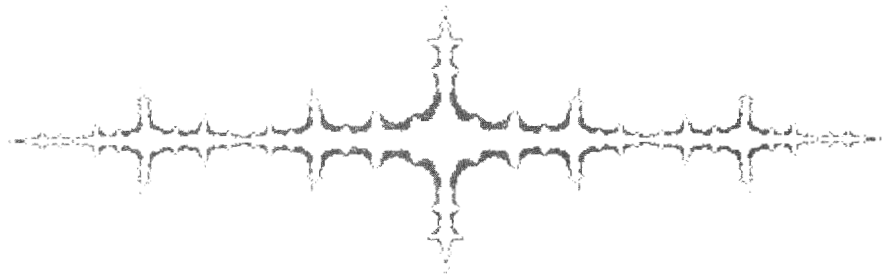
Physiological processes

In addition to physical structure, physiological processes might be subject to the scaling properties that characterize fractals. Fractal processes within organisms can't be characterized by a single scale of time, but instead, have components at many frequencies. For example, some researchers have related the geometry of the nerves in the heart to the associated electrocardiogram[3] output. Similar findings have been reported for the electrical activity of a neuron and variability in heart rate. Some scientists even speculate that diseases are caused by a disruption of the normal fractal scaling (West and Goldberger 1987).

The image in FIG. 3-22 was generated by the program EKG.PAS using the Julia set for:

$f(x) = z^2 - 1.5$

Could this be related to an actual EKG?



3-22
An EKG output?

Chaos of the mind?

Recent theories in psychology (DeAngelis 1993) conjecture that behavior might be determined by chaotic phenomena (some lay persons believe this already!). For example, viewing the mind as a complex dynamical system, psychologists contend that everyday, "normal" behavior represents attracting states. However, the chaotic and unstable nature of the mind often leads to drastic, random behavior shifts that, when harmless, are considered impulsiveness, but, when harmful, are considered dangerous psychosis. Some theorists believe that "crisis-prone" families aren't behaving randomly but simply according to a different norm. Practitioners believe that because the behavior patterns are highly chaotic, they can be changed from an abnormal pattern to a normal one with only a slight nudge.

4 Simulated fractals & chaos

Chaos umpire sits,
And by decision more embroils the fray
By which he reigns; next him high arbiter
Chance governs all

— John Milton, *Paradise Lost*

In this chapter, you'll see how human-made systems can exhibit chaotic behavior, how this chaotic behavior can be harnessed, and how it can be harmful. You'll also see how fractal images can be used in modeling and predicting the performance of systems created by humans.

Turbulent flow

Fluid flow (of liquids and air) is a major application area in dynamical systems. Anyone who has ever flown in an airplane is familiar with turbulent flow, or *turbulence*. Turbulence is characterized by disorder on all scales, with backward eddy currents and circular waves. In most systems, turbulence is undesirable because it creates drag and loss of energy through increased friction. Many human-made systems can exhibit chaotic turbulent flow, from the output of jet engines to the flow of oil through a pipeline. Automobile, airplane, and boat manufacturers use wind tunnels to design vehicle profiles that don't promote turbulence.[1]

You can readily find situations that exhibit turbulent behavior in everyday life. For example, boil a pot of water over a stove. As the water slowly boils, steam begins to escape from the surface. Next, the water slowly and

rhythmically begins to ripple until finally a turbulent, rolling boil is reached. This turbulence is chaotic and random-appearing, yet there's some semblance of regularity as well. It's almost as if some pattern wants to emerge.

A second demonstration of turbulence, suggested by Moon (1992), also involves water. Take a dinner plate and place it under a tap. Fill the dish with water to overflowing, and continue to gently run the water. Place a ping-pong ball in the dish and adjust the water until the ball bounces around merrily, performing chaotic oscillations.

Structures

The artist M. C. Escher was famous for his sketches and woodcuts that depicted impossibly beautiful buildings and other architectural marvels. The architect I. M. Pei was also noted for his impossible designs, with bizarre acute angles that defied conventional technique. Find a book on architecture to see the mathematical intuition that Pei possessed. Using fractals, and in particular IFS fractals, you can create "human-made" structures that appear to be functional as well as beautiful.

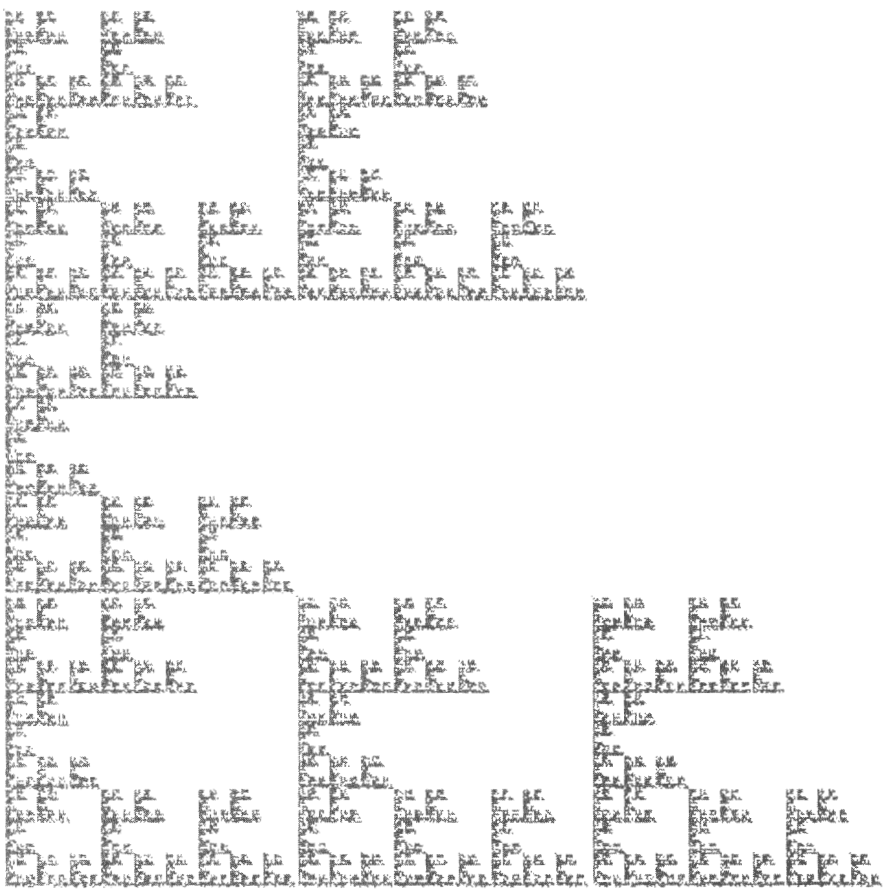
For example, try running the program CASTLE.PAS, which is shown in FIG. 4-1. It appears to be the walled ramparts of some medieval castle. The IFS

4-1
Castle.

codes for the program are shown in TABLE 4-1. The beautiful, mazelike structure shown in FIG. 4-2 can be generated by running the MAZE1.PAS program. Table 4-2 shows the IFS codes for this program.

Table 4-1
IFS codes for castle.

	1	2	3	4	5	6	probability
1	0.5	0	0	0.5	0	0	0.25
2	0.5	0	0	0.5	2	0	0.25
3	0.4	0	0	0.4	0	1	0.25
4	0.5	0	0	0.5	2	1	0.25



4-2
A fractal maze.

Table 4-2
IFS codes for maze.

	1	2	3	4	5	6	probability
1	0.33	0	0	0.33	1	1	0.166
2	0.33	0	0	0.33	MaxY div 2	1	0.166
3	0.33	0	0	0.33	1	MaxY div 2	0.166
4	0.33	0	0	0.33	MaxY div 2	MaxY	0.166
5	0.33	0	0	0.33	MaxY	MaxY	0.166
6	0.33	0	0	0.33	1	MaxY	0.166

Computer scene analysis

Scene analysis is the process of extracting specific features from a larger picture or scene. Many applications exist for scene analysis. For example, mobile robots typically use scene analysis to navigate over terrain. Also, target acquisition programs used in civilian and defense systems use scene analysis to locate a designated object inside an image. For example, medical diagnosis software uses feature analysis to locate specific cell configurations such as tumors or fractures.

Fractal models are useful tools in certain types of three-dimensional scene analysis of two-dimensional images because fractals can model an object's surface roughness. Fractals are especially well suited for this because surface roughness is known to be scale invariant within the effective resolution of most imaging devices. Fractal models have been used to very accurately classify natural textures such as skin, rock, cloth, and grass.

Image compression

Image compression is the process of reducing the amount of stored information needed to reproduce an image. In fractal compression techniques, the bit-by-bit storage of the image is replaced by its representation by an iterated function that requires significantly less storage. The disadvantage, of course, is that it often requires significant time to regenerate the image from the iterated function rather than simply displaying the image pixel by pixel.

The quality or level of compression is expressed in terms of a compression ratio. The *compression ratio* is the ratio of the bytes required to store an uncompressed image to those needed to store the compressed equivalent. Compression rates for fractal compression seem to be in the range of from 20:1 to 60:1, but with questionable quality.

To illustrate the power of fractal compression, consider the forest that was shown in FIG. 3-8. If the computer screen that displays the image is 640 by 480 pixels and requires 16 bits or two bytes per pixel, then the following equation shows the bytes of storage needed:

$$640 \times 480 \times 2 = 614400$$

Whereas the program used to generate the image required only the data contained in the IFS matrix. Assuming that each number in the matrix required four bytes and four additional four-byte numbers were needed for the probability that the transformation in a row was applied, then you only needed to store the following number of bytes:

$$24 \times 4 + 4 \times 4 = 112$$

Then the compression ratio for the forest image is:

$$\frac{\text{number of bytes for screen image}}{\text{number of bytes for IFS codes}}$$

That's 614400/112, which is a compression ratio of 5485:1. This is an amazing savings in storage, which is due to the very low quality of the image rendered. A higher-quality image would necessitate a much lower compression ratio. However, if the image were to be transmitted by a satellite to the earth, you would realize an incredible savings in the time needed to transmit the image.

According to Barnsley (1988), fractal compression is facilitated by a measure of deviation between a given image and its approximation by an iterated function system (IFS). The Collage Theorem (1988) essentially states that to find an IFS attractor that's close to the desired image, you have to find a set of mappings and transformations such that their union or collage is close to the desired image. Unfortunately, this process of finding the transformations is agonizingly slow, even on the most powerful supercomputers.

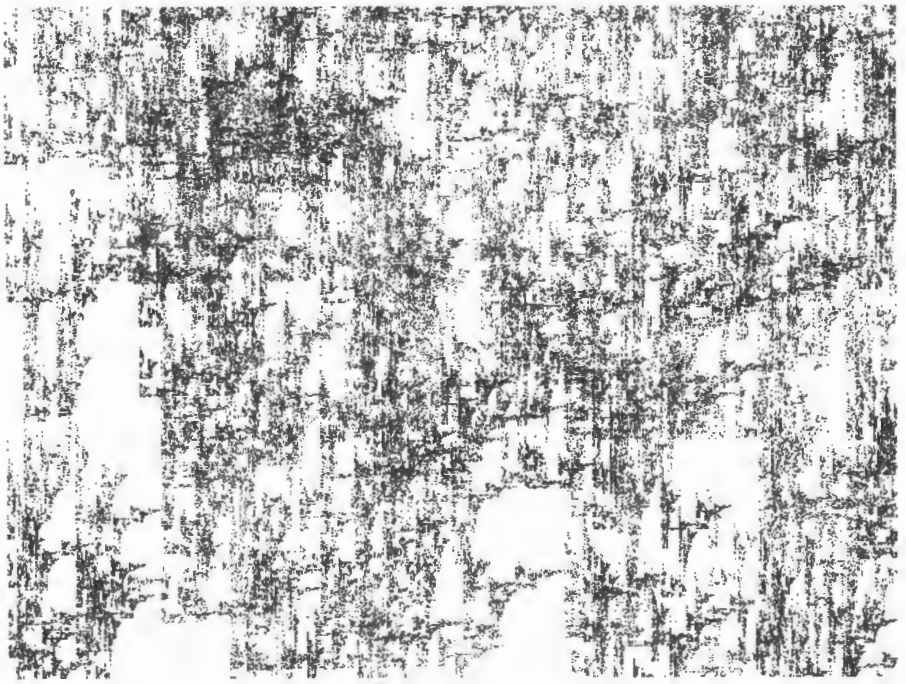
Problems with fractal compression

One of the greatest problems with fractal compression of images is that it's difficult (if not impossible) to find a fractal that will generate a given image. At this writing, it's generally done by trial and error. There are certain fractals that look like trees, mountains, clouds, and so on. These can be tuned to imitate a picture that needs to be compressed.

For example, FIG. 4-3 depicts a swamp pond near my home. This certainly appears to be fractal-like. By tinkering with various IFS parameters, I came up with the suggestive image shown in FIG. 4-4.

To generate this image, I used the IFS codes given in TABLE 4-3. By generating this fractal in many positions, sizes, and colors, I made it similar to the forest image. The code for this is in SWAMP.PAS, and I encourage you to examine it to see that it's very close to FOREST.PAS. Is the computer-generated image faithful to the original? I'll leave it to you to decide if it's close enough.

4-3
Photograph of a swamp pond.



4-4
Computer-generated equivalent of a swamp pond.

Table 4-3
IFS codes for one clump in a swamp.

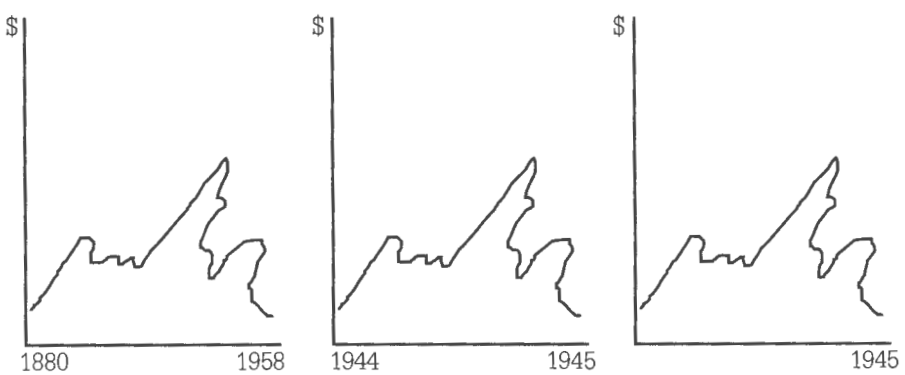
	1	2	3	4	5	6	probability
1	0.5	0	0	0.25	1	1	0.25
2	0.25	0	0	0.7	50	1	0.25
3	0.25	0	0	0.7	1	50	0.25
4	0.5	0	0	0.25	50	50	0.25

Apparently, scientist Michael Barnsley has developed an algorithm that can generate the IFS codes for any image (although there are probably restrictions). However, at this writing, his technique is proprietary.

Another problem associated with the compression of images using fractals or otherwise is that the compression and decompression can take substantial time, and this penalty can sometimes be prohibitive. Consider, for example, the program JULIA1.PAS, which generates the image associated with the Julia set of $f(z) = cos(z)$. (Refer to FIG. 2-2.) Remember how long it took to generate this picture? In some cases, it took much more than an hour. That's the drawback of image compression. In many cases involving real-time image processing, for the human eye to perceive continuous motion the screen must be updated at 33.3 times per second; this time delay is wholly unacceptable.

Economic systems

Mandelbrot was one of the first to recognize that scaling is an important feature of pricing in economics. He analyzed the price of cotton (based on Department of Agriculture figures) during the period from 1880 to 1958. An interesting pattern emerged. When Mandelbrot plotted a suitable function of the price of cotton during the period from 1900–1905, it resembled the same plot for the period from 1880-1940 and for the period from 1944–1958. In other words, he identified a self-similarity in price across scale. (See FIG. 4-5.)



4-5
Self-similarity of cotton prices during a century, a year, and a month.

It's still unknown if Mandelbrot discovered fundamental truth in pricing of commodities or if it was simply a coincidence. Certainly, if you could guarantee the fractal nature of the price of cotton with certainty, you could make a killing on the Chicago Mercantile Exchange!

To see, however, that this might be possible, consider a simple economic system that features a single product, say widgets, with price P and a market of buyers and sellers. If the amount of widgets produced stayed fixed, then basic economic theory says the price should be a linear function of the demand, and thus would rise or fall by a factor of a. The price of widgets at time t would then be:

$$P(t + 1) = aP(t)$$

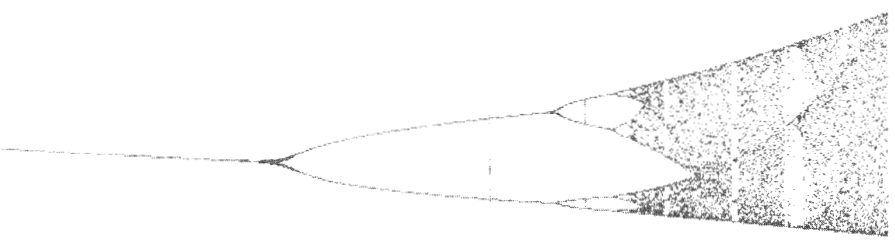
The price at any time $t + 1$ is just the price at the previous time times the factor a.

Suppose that, in order not to appear greedy, the sellers decide to lower their price by the quantity $aP(t)^2$. (The sellers know that what they lose in profit margin they make up in volume anyway.) The price, then, at time $t + 1$ is:

$$P(t + 1) = aP(t) - aP(t)^2 \qquad (4.1)$$

This equation is known as the *logistics* equation, and it was first proposed as a model for population growth by P. F. Verhulst in 1845.

Let's simulate the system by assuming that the starting price for the widget is 90 cents and sweeping the demand factor a from 2.5 to 4.0. You then iterate the price over time (the time scale could be days, months, years, or otherwise), skipping the first 50 iterations to allow the price to "stabilize." You then plot the price $P(t)$ on the y-axis against the demand factor a on the x-axis. You can see this by running the PRICE.PAS program. Its output is shown in FIG. 4-6. Notice that the figure is similar to the bifurcation fractal shown in FIG. 1-4, just as PRICE.PAS is similar to BIFUR.PAS.



4-6
Bifurcation diagram for a model economic system.

Note that for any particular value of a (plotted on the x-axis), there are many prices associated with it. The price is unstable. However, there are a few values for which the price seems to flip-flop between two numbers. These bands of stability are characterized by bald spots in the plot.

One final note is that equation 4.1 is almost identical to equation 2.11 (used to generate the Mandelbrot set) except that the former involves real numbers instead of complex ones. It should be no wonder, then, that they both generate self-similar images.

Cellular automata

A type of mathematical abstraction, called *cellular automata*, has a profound relationship to both fractals and chaos. Cellular automata have been studied as a model for biological cell behavior and massively parallel computers.

Cellular automata, which were originally investigated by John von Neumann and others, consist of a space of unit cells. These cells are initialized with some value, generally a "1" representing a "live" cell, and a "0" for a dead or unoccupied cell. Different characters can be displayed to represent these states, but the idea is that some rule describing the evolution of the system is defined. This rule describes the contents of a unit cell at time t in terms of the contents of the cell and its neighbors at time $t - 1$.

An important feature of cellular automata is the ability to self-organize, or in the terms of chaos theory, find attractors. In addition, many types of cellular automata will eventually attract to stable, fractal-like formations. This attraction occurs with relative indifference to the initial state of the cell field.

Steven Wolfram, a leading expert on cellular automata, has classified cellular automata in a way that helps to reveal their relationship to chaos.

- Class I: evolution to a homogeneous state (an attractor)
- Class II: evolution to isolated periodic segments
- Class III: evolution that is always chaotic
- Class IV: evolution to isolated chaotic segments

You'll be seeing some cellular automata that fit each of these categories. As you read along, try to decide for yourself what stage of evolution the automata are in.

One-dimensional cellular automata

In a *one-dimensional cellular automaton*, the cells are organized in rows, and a cell's contents at time t are based only on the contents of the cell and its neighbors on either side at time $t - 1$. In one-dimensional cellular automata, you trace the evolution of the system by observing the row at time t followed by the row at time $t + 1$, and so on. In many cases, the result is chaotic or unstable, but in some cases a strange attractor is found.

For example, on your disk you'll find a program called CELL1.PAS, which is a Pascal implementation of a cellular automaton that follows the cell rule:

$$a_0' = (a_{-1}a_0a_1) + (\bar{a}_{-1}a_1) + (a_0a_1)$$

Let's see what this rule means.

The symbol a_0' represents the contents of a given cell at time t. Similarly, a_0 is the contents of the cell at the previous time, $t-1$. Finally, a_{-1} is the contents of the cell on the left at time $t-1$, and a_1 is the contents of the cell on the right at time $t-1$.

Multiplication represents the *Boolean AND operation*, which produces a one only if both operands are one[2]. The addition symbol, +, represents the *Boolean OR operation*, which produces a one if one or both operands are one. Finally, the bar over a cell's contents, for example $\overline{a_0}$, indicates that its *Boolean complement* is to be taken, which is simply a one if the cell contains a zero, and vice versa. Thus, the cell rule says in words that

> A cell is alive if both its neighbors and it are alive, or if its left neighbor is dead and its right neighbor alive, or if it and its right neighbor are alive.

Isn't the mathematical notation more compact? In fact, it can be simplified, by logical inference, to be:

> A cell is alive if its left neighbor is dead and its right neighbor alive, or if it and its right neighbor are alive.

That statement has the following symbolic equivalent:

$$a_0' = (a_{-1}a_1) + (a_0a_1)$$

I didn't simplify the rule in CELL1.PAS, but you should, just for practice.

When you run program CELL1.PAS, it will prompt you for the starting configuration of the cell system. You're then to enter a line of asterisks ("*") corresponding to the live cells. Try entering a line of spaces followed by an asterisk in column 40 like this:

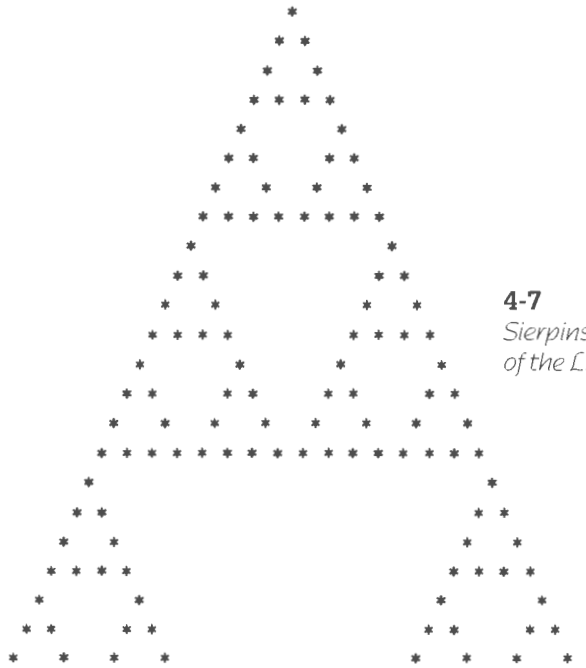
```
                                        *
```

The program will prompt you for the number of iterations to run, that is, the number of time ticks for the system. Respond with 22. The output of the program should look like a Sierpinski triangle, as shown in FIG. 4-7. It's amazing that, like a seed crystal, a single cell site results in the strange attractor of the Sierpinski triangle. (If you had written the program so that single pixel activations represented cells, then you could see this even more dramatically. Try it as project!)

You can have fun experimenting with function `rule` in program CELL1. For example, try coding these rules:

1. $a_0' = (a_{-1}\overline{a_0}) + (a_{-1}\overline{a_1}) + (\overline{a_{-1}}a_1)$

2. $a_0' = (a_{-1}a_0\overline{a_1}) + (\overline{a_{-1}}a_0) + (\overline{a_{-1}}a_1) + (a_0a_1)$



4-7
Sierpinski triangle output of the LIFE.PAS program.

Then run the program and use the same input as before; that is, seed it with a single cell site.

Some one-dimensional cellular automata, for instance the one you just saw, generate interesting fractal-like patterns based on an initial configuration using only one cell. These systems can be models of crystal growth in certain types of structures that are generated from a single seed crystal. However, other cellular automata take a random, multiple cell input and organize it to find attractors after many iterations. The organization can be in the form of a regular or fractal-like pattern, or it can result in some form of oscillating structure.

Let's look at one rule that organizes a random initial cell configuration into a fractal-like state. Examine file CELL2.PAS. It's essentially the same program as CELL1.PAS, except with the evolution rule:

$$a_0' = (a_{-1}a_0a_1 + \overline{a}_{-1}a_1)$$

Run it, giving it any random (nonempty) initial cell configuration, such as:

*** * ** * **** * * *** *** * * *** * * *** ** ** *** ** ** **

Be sure to populate the input line with many live cells. Remember, this is supposed to be a random starting configuration; asterisks or live cells are just as likely to occur as dead or blank cells.

The program will then prompt you for the number of iterations (generations) to run. Respond with at least 50 iterations and watch the dynamic results. The output will resemble FIG. 4-8 but will be constantly changing. Notice that, although it's random, it appears to be organizing or at least oscillating through different configurations. Try running this automaton for several hundred iterations.

Other rules that will organize a random initial configuration after a large number of iterations are:

1. $a_0' = (a_{-1}a_0\,\overline{a_1}) + (\overline{a_{-1}}\,\overline{a_0}a_1)$

2. $a_0' = (a_{-1}\,\overline{a_0}a_1)$

You can test these by modifying CELL1.PAS or CELL2.PAS.

Two-dimensional cellular automata

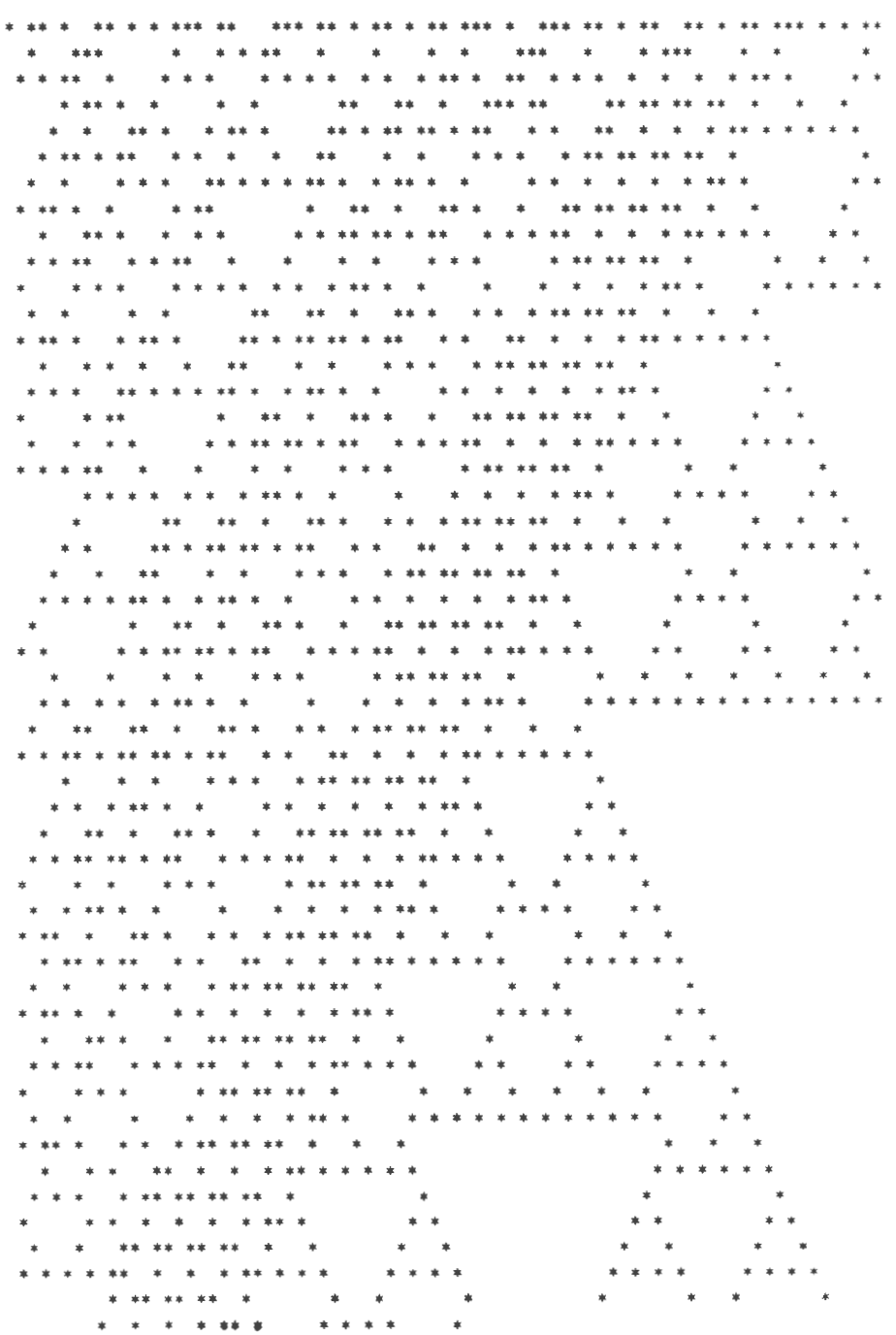
A cellular automaton that's organized as a two-dimensional matrix or array of cells is called a *two-dimensional cellular automaton.* Here, a cell's contents at time t is based on its own contents and the contents of all its immediate neighbors at time $t-1$. One two-dimensional cellular automaton that has been studied extensively is the "Game of Life," developed by John Conway. In the Game of Life, the local rule states that a cell "dies" (gets a value of zero) unless two or three of its neighbors are alive (have a value of one). If two neighbors are alive, then the value of the cell site is unchanged. If three neighbors are alive, the site always takes on the value one.

Depending on the initial configuration, various static equilibrium states, such as squares or hexagons, have been found. Oscillating or periodic segments can exist, as can "traveling"or "glider gun" states where cell configuration moves across the cell field and can be regenerated indefinitely.

The program LIFE.PAS on your disk is an implementation of the Game of Life. I have made the traditional assumption that if the number of live cells around a given cell is greater than 3, then the cell dies of overcrowding. The program was not difficult to write, but would be tedious to describe. If you're interested, I encourage you to look at the code and modify procedure `rule` to create an organism that behaves according to another rule.

Run the program. It will ask you for a file containing the initial configuration of the cell system. File TEST on your disk contains a sample configuration file. To change initial configurations, modify this file using an ASCII-based editor[3]. When prompted, input a large number of iterations (say 1000) and watch the little universe evolve.

A second implementation of Life is also on your disk. This program LIFE2.PAS generates a random starting configuration. The program requests the number of iterations to run. Be generous; several hundred is good. Once



4-8
Sample output from the CELL2.PAS program.

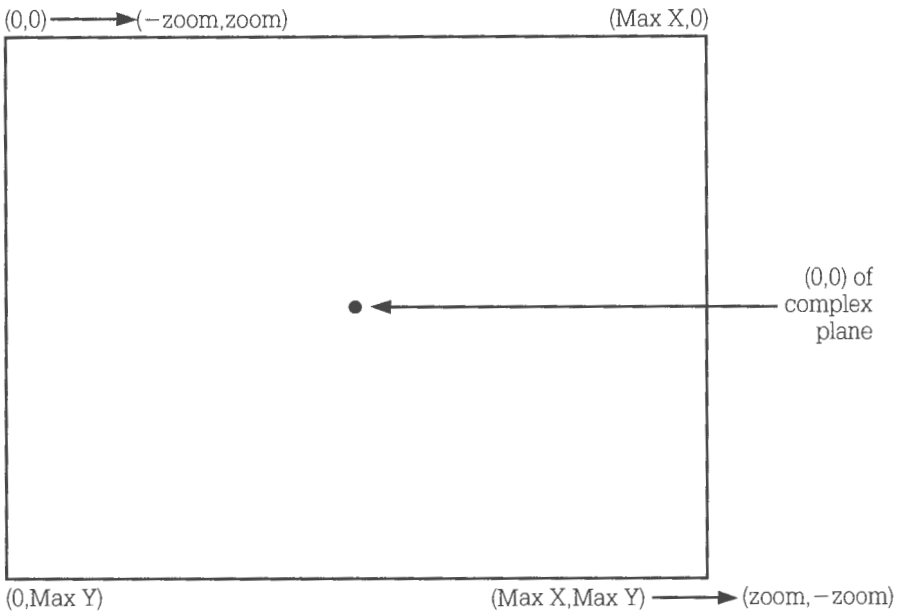
input, the cell world will mutate and evolve. Look for stable configurations such as squares and circles, and bi-stable configurations called "blinkers," which alternate between two states. Other configurations of cells will appear to walk across the screen. Still others will split or join. It really is a fascinating program to watch.

A Turbo Pascal graphics

For those of you who are interested in modifying or writing fractal programs using Turbo Pascal graphics, the following discussion is about the graphics screen of your computer. A typical color graphics screen is organized in an array of picture elements or *pixels* capable of displaying one or more colors. The quality (and cost) of your monitor is dependent upon, among many things, the density of the pixels (called the *resolution*) and the number of colors that each pixel is capable of displaying.

For example, most low-cost enhanced graphics adapter (EGA) color monitors consist of a 640 × 480 array of pixels that can display in 16 different colors. The first number, 640, represents the number of columns of pixels, whereas 480 is the number of rows. Low-cost video graphics adapter (VGA) monitors have a similar configuration. Super VGA monitors often have 1280 × 1024 pixel arrays that can display in 256 colors. Higher-quality and more expensive monitors also exist.

Regardless of the resolution and number of colors your monitor supports, the upper left-most pixel has coordinate (0,0), and the lower right-most pixel has coordinate (`MaxX,MaxY`) where `MaxX` is the number of pixel columns and `MaxY` is the number of rows. Since you're interested in mapping this screen into the complex plane so that coordinate (0,0) is at the center of the screen (see FIG. A-1), you need to devise an algorithm to do this.



A-1
Mapping the screen coordinate system onto the complex plane.

Suppose you want to map the screen with the upper left-hand coordinate (0,0) and the lower right-hand coordinate (`MaxX`,`MaxY`) onto a square section of the complex plane. The plane has the center coordinate (0,0), the upper left-hand corner coordinate (`-zoom`,`zoom`), and the lower right-hand corner coordinate (`zoom`,`-zoom`), were `zoom` is some arbitrary constant. To do this, use the transformation for the x-coordinate:

$$f(x) = x\frac{2 \cdot \text{zoom}}{\texttt{MaxX}} - \text{zoom} \qquad \text{(A.1)}$$

The transformation for the *y*-coordinate is:

$$g(y) = -y\frac{2 \cdot \text{zoom}}{\texttt{MaxY}} + \text{zoom} \qquad \text{(A.2)}$$

Can you see that these transformations map the point (0,0) into the point (f(0), g(0)) = (–zoom,zoom) and the point (`MaxX`,`MaxY`) into the point (f(`MaxX`), g(`MaxY`)) = (zoom,–zoom) as desired?

Since on most screens `MaxX` and `MaxY` are unequal, stretching a picture along these directions in these proportions would result in distortion (in technical jargon, the *aspect ration* would have changed). To prevent distortion, then, you pull in both directions equally, generally choosing `MaxX` as the zoom factor because it's larger than `MaxY`. This will result in a loss of information on the right-hand side of the screen (an important behavior known as clipping).

Stretching in both directions by MaxY will prevent clipping but will result in an image that doesn't completely fill out the horizontal space.

The code supplied with this book is designed to run on any EGA or VGA type monitor. (Some other monitors, such as those with the older, CGA technology might also work.) The code takes into account MaxX and MaxY and the number of colors that can be displayed by these monitors.

Graphics in the programs

Let's look at a piece of code that appears in virtually all of the programs and is used to set up the Turbo Graphics environment. The first line in almost every program that uses graphics has the code:

```
uses
   Crt,Complex,Graph;
```

This is a Pascal statement that declares that the units Crt, Complex, and Graph are going to be used. Unit Crt is a standard Turbo Pascal unit that provides nongraphics screen utilities such as ClrScr, which clears the screen. Unit Complex is a unit that I wrote. It contains the complex function routines you need to generate Julia and Mandelbrot sets. Unit Graph is a standard Turbo Graphics unit that provides the routines to display graphics.

The next code is common to all graphics routines and contains the variable declarations that are needed to call the Graph unit and to do the overall scaling. These variables are:

```
GraphDriver : integer;    { Stores graphics driver number}
GraphMode : integer;      { Stores graphics mode for driver}
ErrorCode : integer;      { Reports any error condition}
MaxY      : integer;      { Maximum Y screen coordinate}
zoom      : real;         { overall zoom factor }
scale     : real;         { scale factor }
x,y       : real;         { intermediate variables }
MaxColor  : integer;      { maximum number of colors on graphics card }
```

I've included the comments associated with these variables, as they are self-descriptive.

The next important phase in using graphics is to detect the type of monitor that the code is going to run on. To this effect, you'll see the code:

```
GraphDriver : = Detect;     {try to detect graphics card}
InitGraph(GraphDriver,GraphMode,"");  {initialize graphics}
ErrorCode : = GraphResult;
if ErrorCode <> grOk then     {check for error}
begin
     Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
     Writeln('Graphics card not found');
     Writeln('Program aborted');
     Halt(1)
end;
```

The first line detects the type of graphics driver (for example, EGA or VGA) and stores it in variable `GraphDriver`. The second line initializes the graphics mode so that it's compatible with the graphics driver. The next seven lines take care of any errors during graphics initialization. If there's a problem, such as no graphics capability, the program will immediately halt. This indicates that either your monitor doesn't have graphics capability or it's damaged in some way. In this case, refer to your computer owner's manual.

Later in the code, you'll find:

```
MaxColor := GetMaxColor; { find maximum number of colors }
MaxY := GetMaxY;         { find maximum Y screen coordinate }
scale:= 2.0*zoom/MaxX;   { calculate zoom factor}
```

The first line reads the number of colors that your system is capable of displaying. Remember that this will also be used as the iteration limit for calculating Julia and Mandelbrot sets.

The second line senses the maximum y-coordinate. Since, in general, the y-coordinate is always less than the x-coordinate, you use the y-coordinate as both the maximum x- and y-coordinate to be displayed. While this chops the screen off to the right, it prevents distortion of the images. See the previous discussion on aspect ratio for more details.

The last line is the scale factor needed to perform the calculations in equations A.1} and A.2. The scale factor `scale` is used in conjunction with variable `zoom`, which is a user input, to complete the coordinate transformation. That is:

```
x := scale*i - zoom;
y := zoom - scale*j;
```

That will implement equations A.1 and A.2.

The final statement that appears in the programs using graphics is:

```
putpixel(i,j, color);
```

That statement activates the pixel at row i and column j, with color given by the variable `color`. The variable `color` will either be a 1 (which is generally dark blue) if a filled image was chosen, or will be based on the number of iterations before escape (variable `iter`) if an unfilled image was selected by the user.

Program listings

```
program amoeba;
{ compute and display an "amoeba" from the Julia set of

    f(z) = z^2 + .3 - .4i

  12 - 21 - 92  Phil Laplante                    }
uses
   Complex, Graph;           { include graphics and complex routines}

const
   zoom = 2.0;               { create 4 by 4 window }
   attract = 0.0001;         { attractor sensitivity }

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   i, j      : integer;      { loop variables}
   MaxY      : integer;      { Maximum Y screen coordinate}
   scale     : real;         { scale factor }
   mag       : real;         { square of magnitude of complex number }
   iter      : integer;      { escape iteration counter }
   continue  : boolean;      { continue iteration counter }
   x,y       : real;         { real and complex parts of z }
   MaxColor  : integer;      { maximum number of colors on graphics card }
begin
```

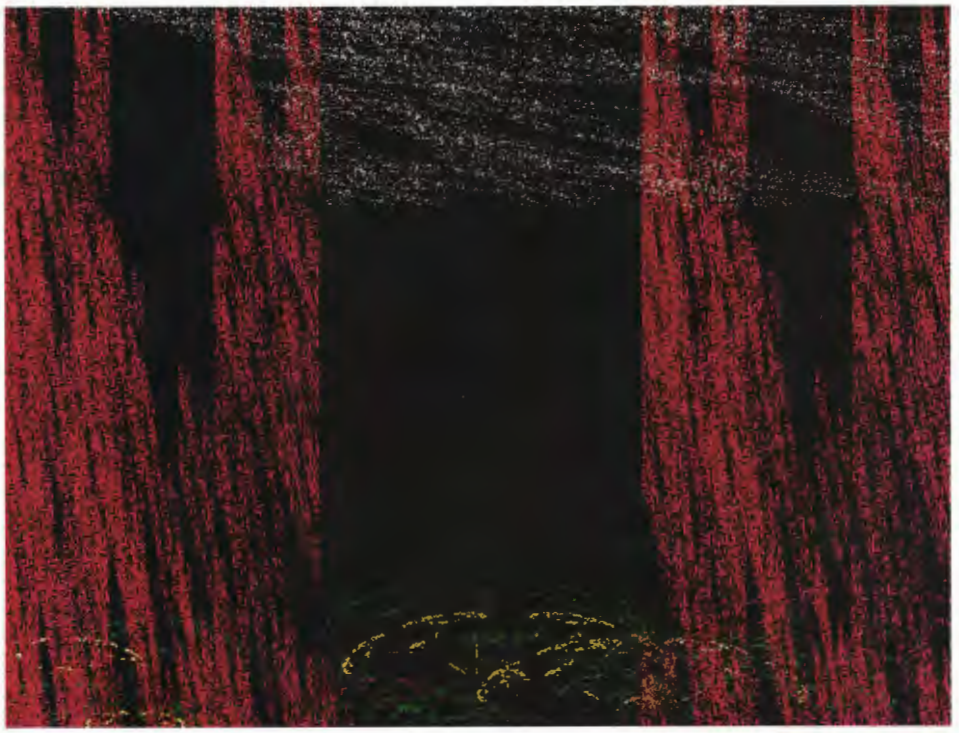
```
    { initialize graphics }

    GraphDriver : = Detect;   {try to detect graphics card}
    InitGraph(GraphDriver,GraphMode,"");  {initialize graphics}
    ErrorCode : = GraphResult;
    if ErrorCode <> grOk then    {check for error}
    begin
         Writeln('Graphics error:', GraphErrorMsg(ErrorCode));
         Writeln('Graphics card not found');
         Writeln('Program aborted');
         Halt(1)
    end;
    MaxColor : = GetMaxColor; { find maximum number of colors }
    MaxY : = GetMaxY;         { find maximum Y screen coordinate }
    scale: = 2.0*zoom/MaxY;   { calculate zoom factor}

    for i : = 0 to MaxY do    { MaxY is usually smaller than MaxX }
    begin
        for j : = 0 to MaxY do
        begin
             x : = scale*i - zoom;           { set starting value of real(z) }
             y : = zoom - scale*j;           { set starting value of imag(z) }
             continue : = TRUE;              { assume point does not escape  }
             iter : = 0;
             while continue = TRUE do
             begin
                 mult(x,y,x,y,x,y);          { square z }
                 add(x,y,0.3,-0.4,x,y);      { add 0.3 - 0.4i }
                 mag : = x*x + y*y;          { calculate square of magnitude }
                 if mag < attract then
                   continue : = FALSE        { point is an attractor }
                 else
                   if (mag < 100) AND (iter < MaxColor*2) then { keep iterating function }
                      iter : = iter + 1
                   else                      { point escapes, plot it }
                      begin
                        putpixel(i,j, iter div 2);
                        continue : = FALSE        { get out of loop }
                      end
             end  { while loop}
        end  {j loop}
    end{ i loop}
end.
```

```
program bifur;
{ compute and display bifurcation diagram for

  f(x) = x^2 + c

  12 - 21 - 93   Phil Laplante
```

```
uses
   Crt,Graph;                { include CRT and graphics routines}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   i, j      : integer;      { loop variables}
   MaxX      : integer;      { Maximum X screen coordinate}
   MaxY      : integer;      { Maximum Y screen coordinate}
   x         : real;         { iterated value }
   c         : real;         { constant of iteration  }
   MaxColor  : integer;      { maximum number of colors on graphics card }
   scale     : real;         { plotting scale factor }
   sf        : real;         { user input scale factor }

begin
                                          { get user input scale factor }
   ClrScr;
   write(' input scale factor (1 - 10) ');
   readln(sf);

   { initialize graphics }

   GraphDriver := Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,"; {initialize graphics}
   ErrorCode := GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxColor := GetMaxColor;  { find maximum number of colors }
   MaxX := GetMaxX;          { find maximum X screen coordinate }
   MaxY := GetMaxY;          { find maximum Y screen coordinate }

   scale := sf*MaxX/8;       { calculate overall scale factor }

   c := - 2.0;               { set starting point }
   for i := 1 to MaxX do
    begin
     x := 0.0;               { calculate orbit about x=0 }
     c := c + 2.25/MaxX;     { iterate c }
     for j := 1 to 200 do    { calculate orbit after 200 iterations}
     begin
       x := x*x + c;
       if j> 50 then         { skip first 50 iterations }
       begin
         putpixel(i,round(MaxY/2 + x*scale), j div MaxColor);
       end
```

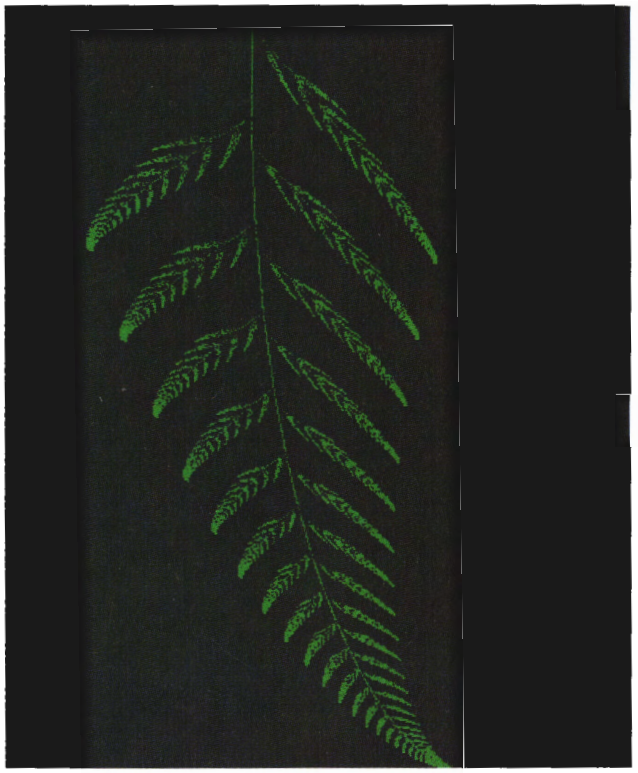
```
     end
    end
end.
```

```
program Cantor_set;
{ Produce Cantor set using recursion
  12 - 20 - 92   Phil Laplante            }

uses
   Graph;                    {include graphics package}
var
   GraphDriver : integer;    {Stores graphics driver number}
   GraphMode : integer;      {Stores graphics mode for driver}
   ErrorCode : integer;      {Reports any error condition}
   MaxX      : integer;      {Maximum X coordinate }

procedure Cantor(x1, y1, x2, y2 : integer);
{ Applies middle third algorithm to segment of real line }
var
   delta : integer;                    {one third of line segment }
begin
    line(x1, y1, x2, y1);
    y1 := y1 + 16;                     {increment y coordinate}
    if y1 <=128 then                   {don't do to far }
    begin
      delta := (x2 - x1) div 3;        {calculate quarter of line segment }
      Cantor(x1, y1, x1 + delta, y1);  {draw first third}
      Cantor(x2 - delta, y1, x2, y1);  {draw third third}
    end;
end;

begin
   GraphDriver : = Detect;               {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,"");  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then             {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;

   MaxX :=  GetMaxX;

   Cantor(0,8,MaxX,8);          { start Cantor set at row 8 }
end.
```

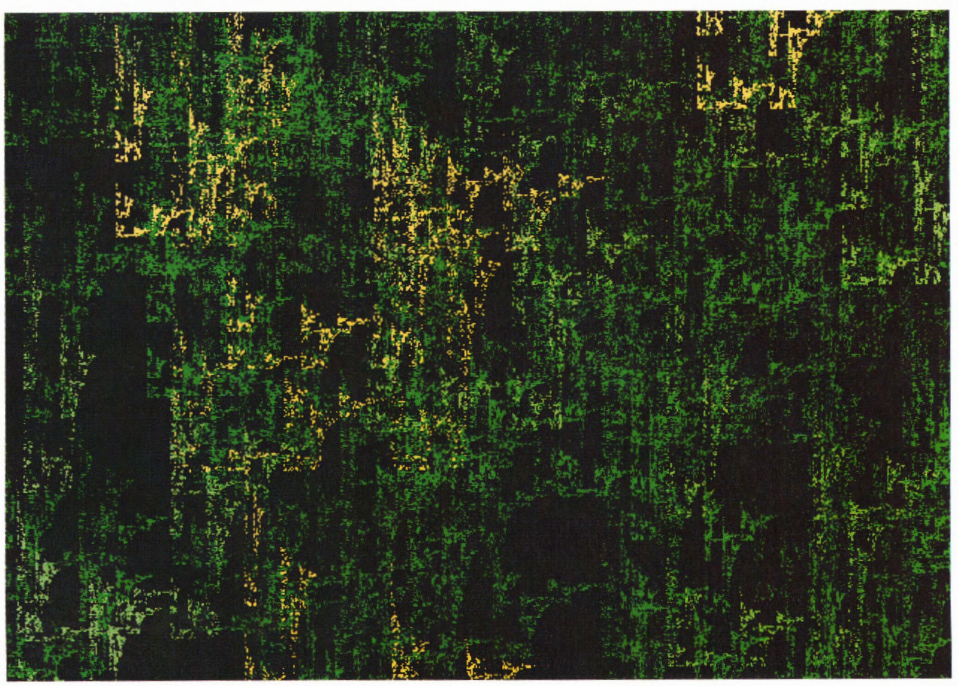


Redwood forest.

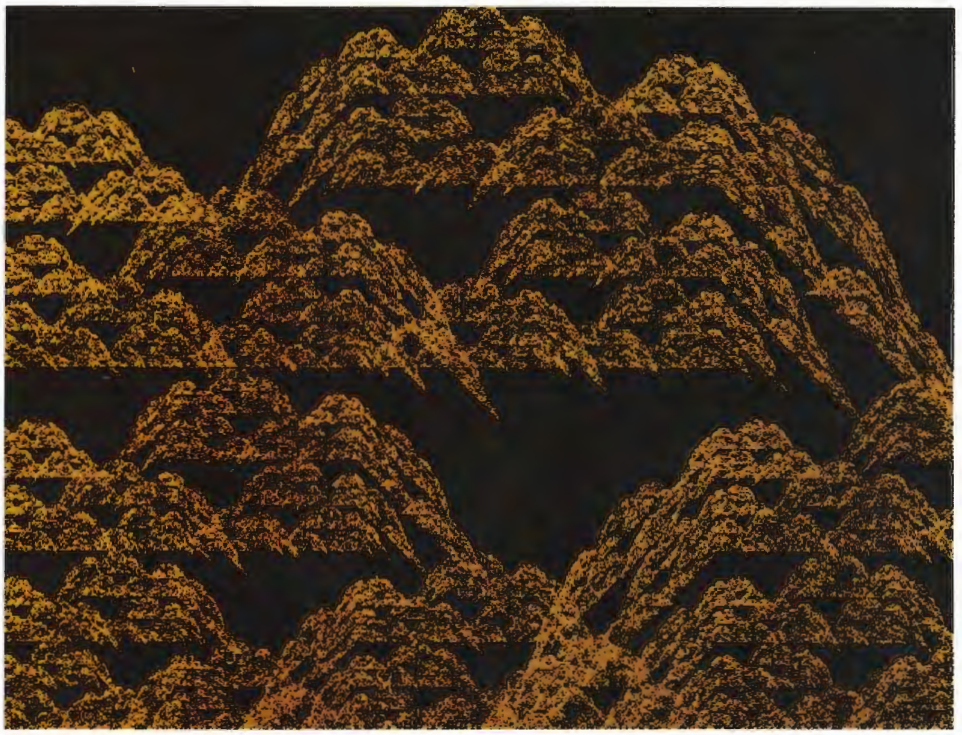
A forest of randomly generated fractal trees.



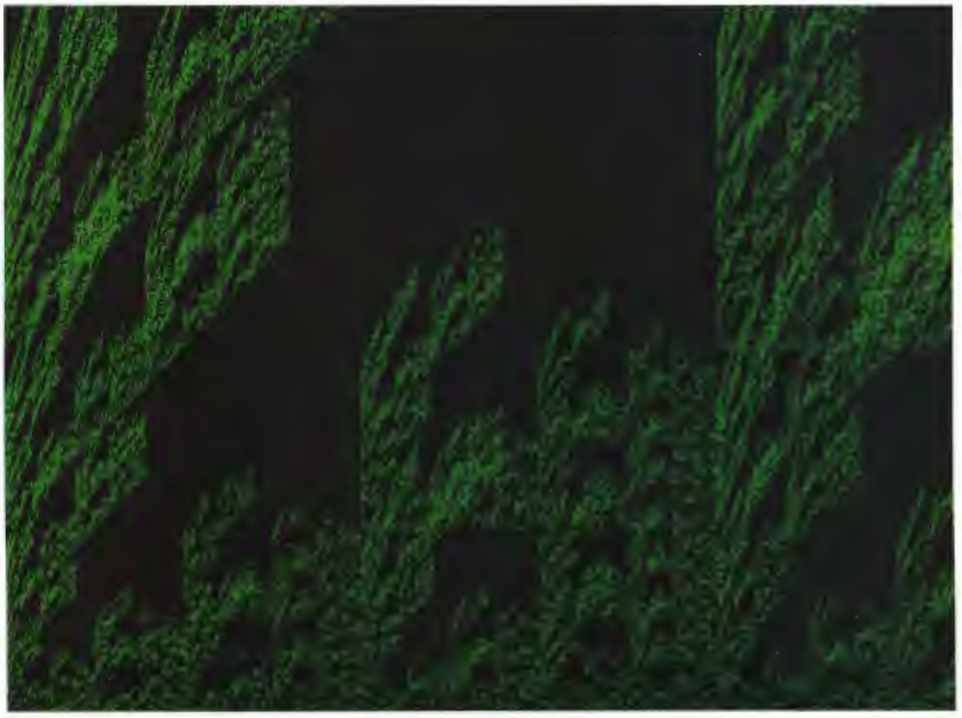
Fern leaf.



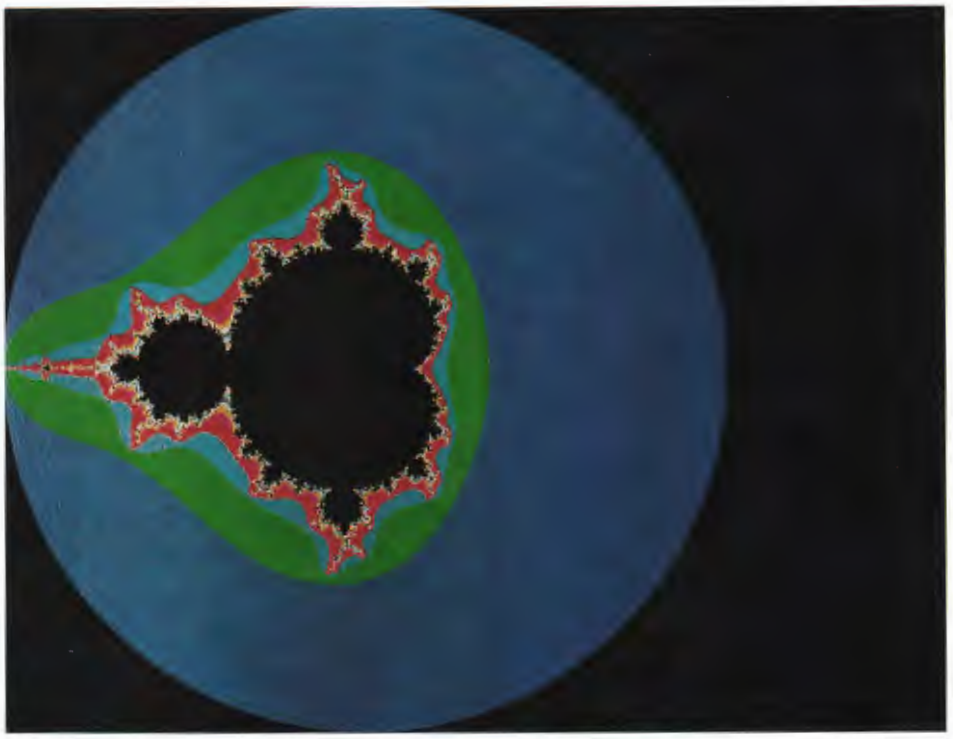
Computer-generated equivalent of a swamp pond.



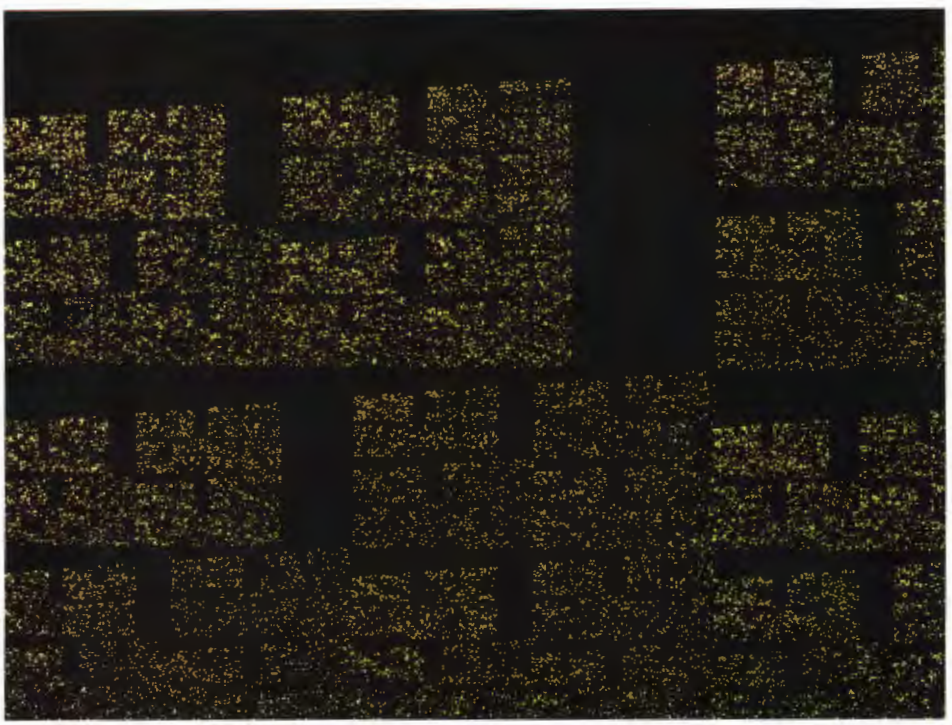
Fractal rocks using the same IFS codes as fractal clouds.



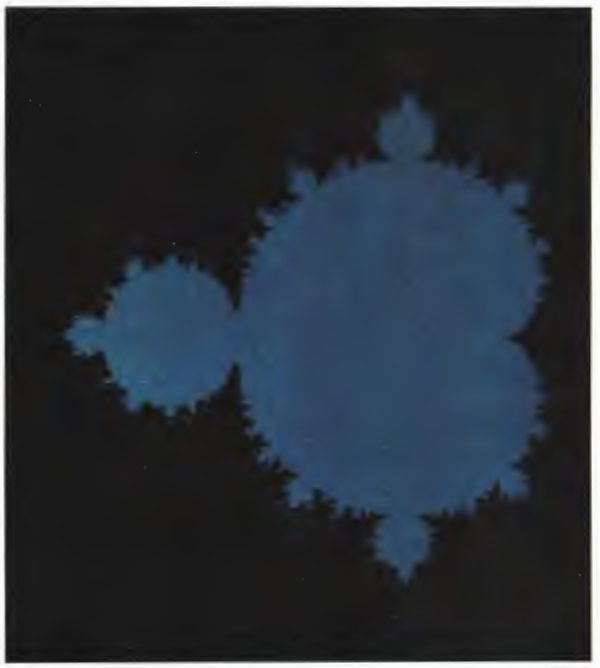
Green seaweed.



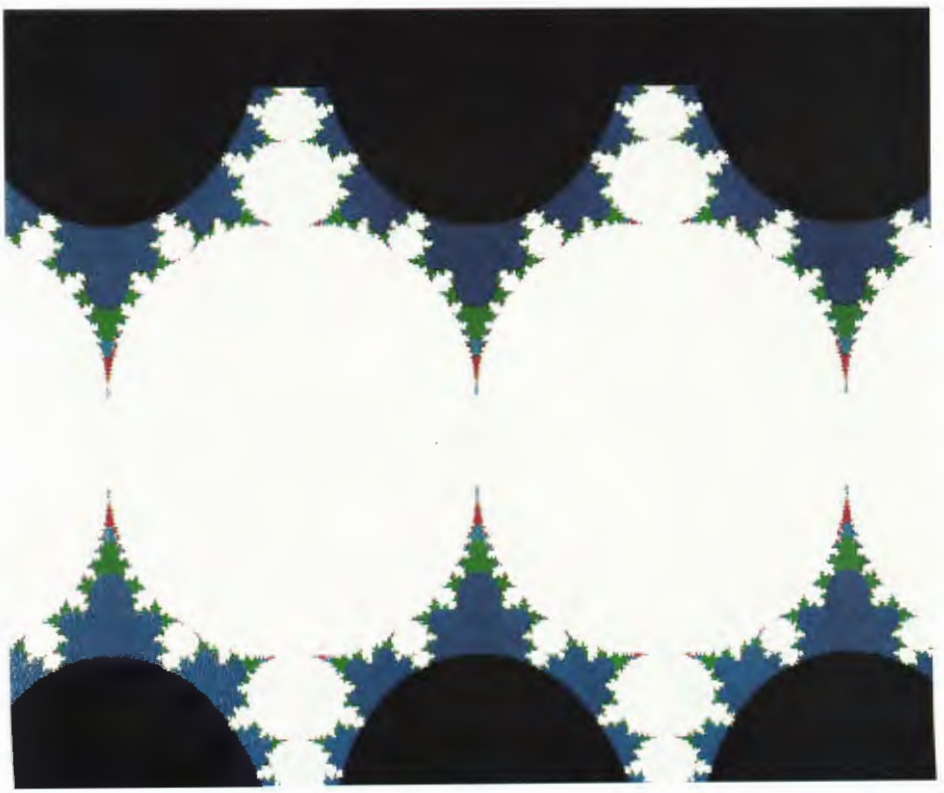
The Mandelbrot set.



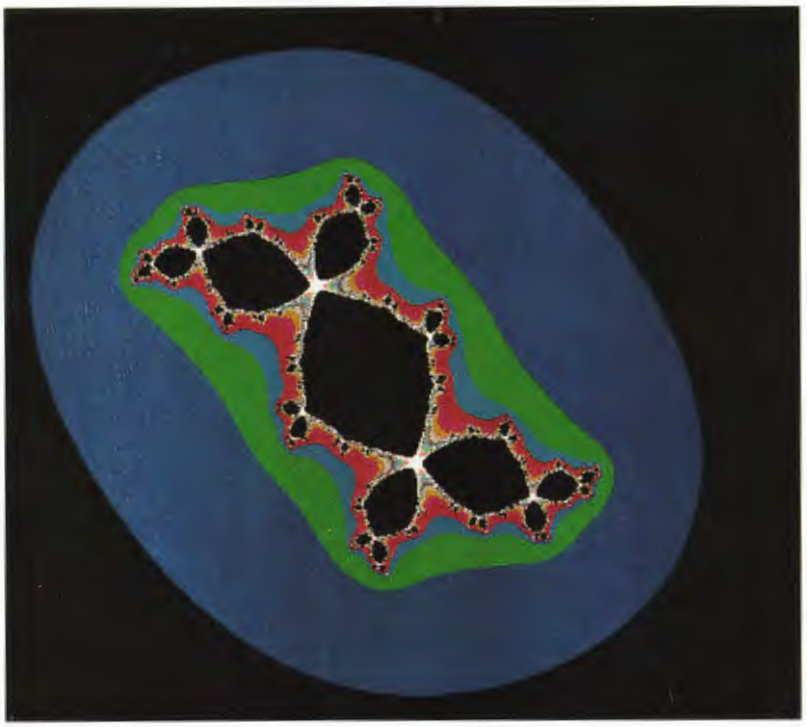
Castle.



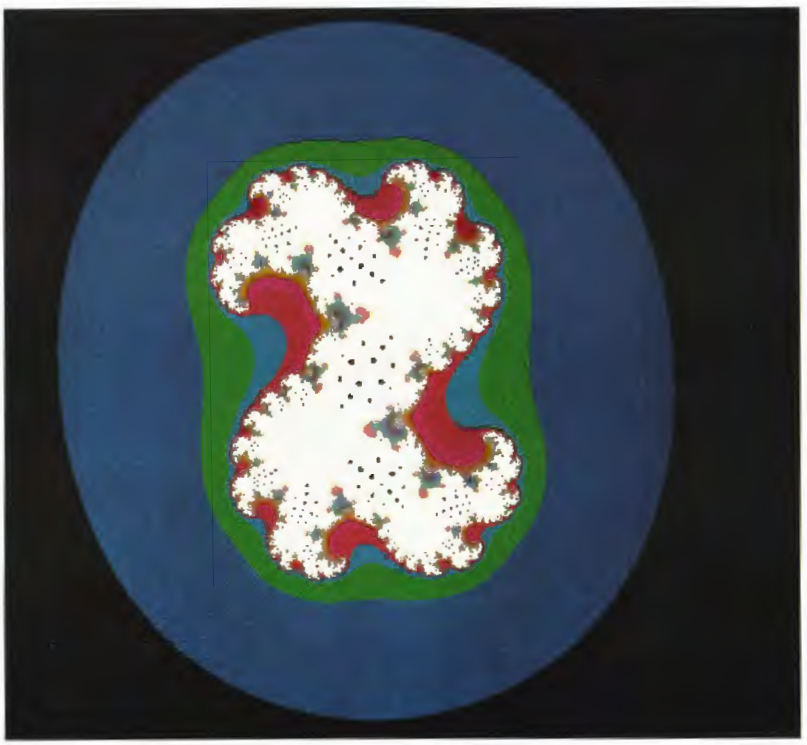
A filled Mandelbrot set.



The Julia set of $\sin(z)$.

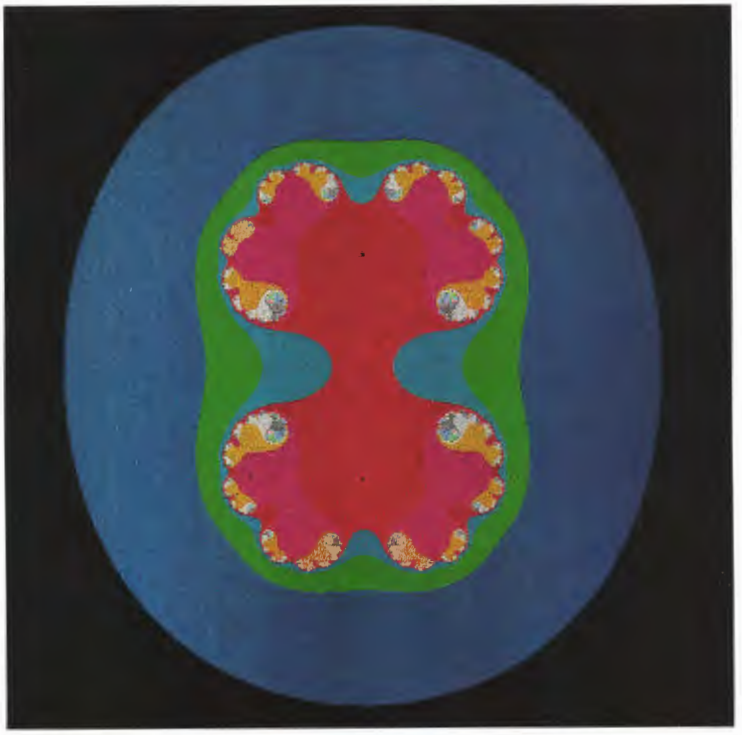


Douady's rabbit.

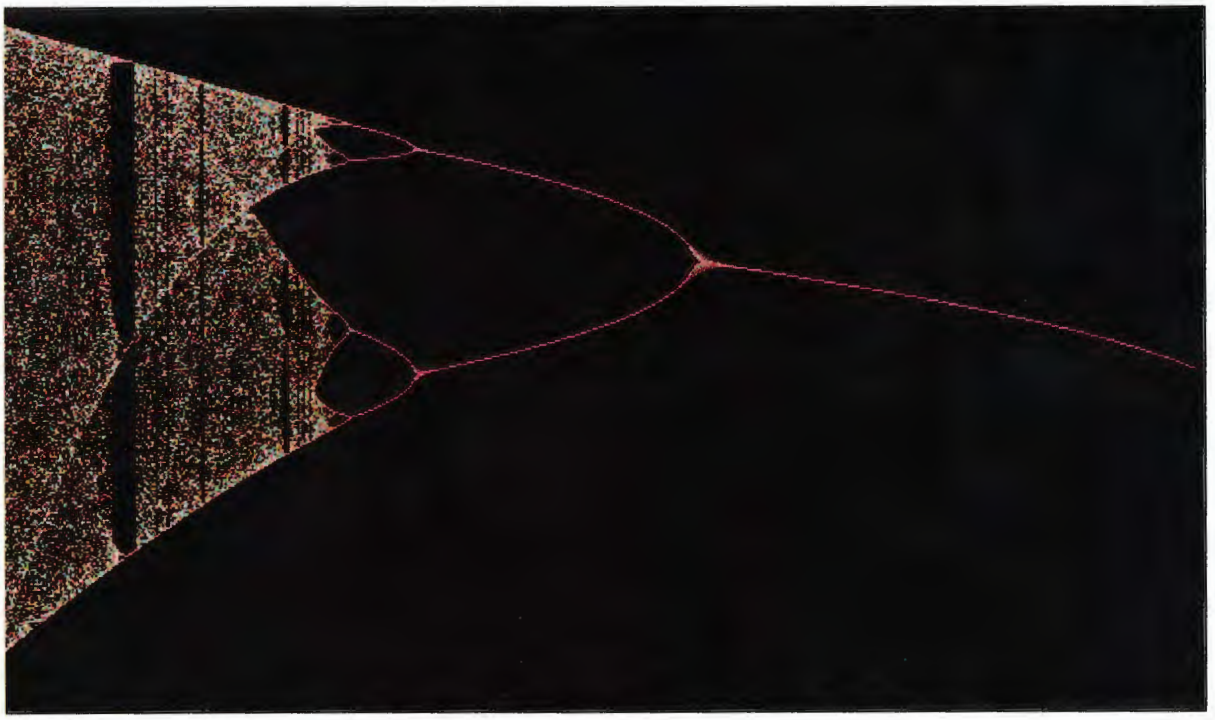


Dragon.

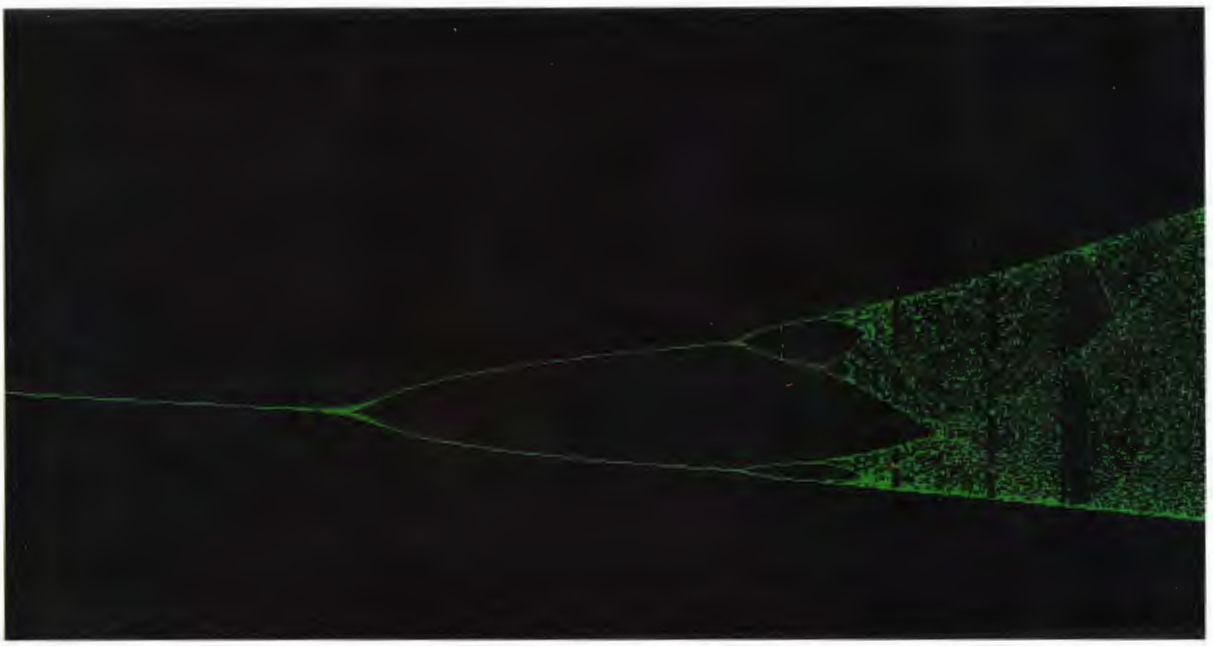
Dendrite structure generated by $f(z) = z^2 + i$.



Four-petaled flower from the Julia set of $f(z) = z^2 + 0.384$.



Bifurcation diagram for $f(x) = x^2 + c$ *with* $x = 0$ *and various values of* c *produced using BIFUR.PAS.*



Bifurcation diagram for a model economic system.

```
program carpet;
{ compute and display Sierpinski carpet
  using Michael Barnsley's IFS algorithm

  12 - 5 - 93  Phil Laplante                    }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
   i         : integer;      { loop counters}
   k         : integer;      { row selector }
   MaxY      : integer;      { maximum X and Y coordinates}
   d         : array[1..8,1..6] of real; { holds data of IFS attractor }

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,"");  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxY : = GetMaxY;          { get screen limits }

{ initialize IFS data array }

   d[1,1]: =0.33; d[1,2]: =0; d[1,3]: =0; d[1,4]: =0.33; d[1,5]: =1; d[1,6]: =1;
   d[2,1]: =0.33; d[2,2]: =0; d[2,3]: =0; d[2,4]: =0.33; d[2,5]: =MaxY; d[2,6]: =1;
   d[3,1]: =0.33; d[3,2]: =0; d[3,3]: =0; d[3,4]: =0.33; d[3,5]: =1; d[3,6]: =MaxY;
   d[4,1]: =0.33; d[4,2]: =0; d[4,3]: =0; d[4,4]: =0.33; d[4,5]: =MaxY; d[4,6]: =MaxY;
   d[5,1]: =0.33; d[5,2]: =0; d[5,3]: =0; d[5,4]: =0.33; d[5,5]: =MaxY div 2; d[5,6]: =1;
   d[6,1]: =0.33; d[6,2]: =0; d[6,3]: =0; d[6,4]: =0.33; d[6,5]: =MaxY; d[6,6]: =MaxY div 2;
   d[7,1]: =0.33; d[7,2]: =0; d[7,3]: =0; d[7,4]: =0.33; d[7,5]: =1; d[7,6]: =MaxY div 2;
   d[8,1]: =0.33; d[8,2]: =0; d[8,3]: =0; d[8,4]: =0.33; d[8,5]: =MaxY div 2; d[8,6]: =MaxY;

   MaxY : = GetMaxY;

   randomize;           {initialize random number generator}

   x : = 0;             {set starting coordinates}
   y : = 0;

   for i : = 1 to 30000 do
     begin
        k : = random(8) + 1;                  { pick random row }
```

```
        x : = d[k,1]*x + d[k,2]*y + d[k,5];   { transform coordinates }
        y : = d[k,3]*x + d[k,4]*y + d[k,6];
        if i > 10 then                        { skip first 10 iterations }
           putpixel(round(2*x/3),round(2*y/3),WHITE)
     end
end.
```

```
program castle;
{ compute and display "castle" fractal
  using Michael Barnsley's IFS algorithm

  12 - 5 - 93  Phil Laplante                  }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
   i         : integer;      { loop counters}
   k         : integer;      { row selector }
   MaxY      : integer;      { Maximum Y screen coordinate}
   d         : array[1..4,1..6] of real; { holds data of IFS attractor }

begin
   GraphDriver := Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,"");  {initialize graphics}
   ErrorCode := GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;

   MaxY := GetMaxY;

{ initialize IFS data array }

   d[1,1]:=0.5;  d[1,2]:=0;  d[1,3]:=0;  d[1,4]:=0.5;  d[1,5]:=0; d[1,6]:=0;
   d[2,1]:=0.5;  d[2,2]:=0;  d[2,3]:=0;  d[2,4]:=0.5;  d[2,5]:=2; d[2,6]:=0;
   d[3,1]:=0.4;  d[3,2]:=0;  d[3,3]:=0;  d[3,4]:=0.4;  d[3,5]:=0; d[3,6]:=1;
   d[4,1]:=0.5;  d[4,2]:=0;  d[4,3]:=0;  d[4,4]:=0.5;  d[4,5]:=2; d[4,6]:=1;


   randomize;           {initialize random number generator}

   x := 0;              {set starting coordinates}
```

```
   y := 0;

   for i := 1 to 32000 do
     begin
        k := random(4) + 1;                    { pick random number from 1-4}
        x := d[k,1]*x + d[k,2]*y + d[k,5];     { transform coordinates }
        y := d[k,3]*x + d[k,4]*y + d[k,6];
        if i > 10 then                         { skip first 10 iterations }
             putpixel(round(MaxY*x/2),MaxY - round(MaxY*y/2),YELLOW)
          end                                  { scale for screen }
end.
```

```
program cell1;
{ Simulate One - dimensional cellular automata
  1/29/93   Phil Laplante       }

uses
     Crt;                    { unit for CRT driver }

const
     columns  = 80;          { number of columns on screen }
     rows     = 24;          { number of rows on screen }
     cell     = '*';         { cell symbol }

type
     cell_field = array[1..columns] of boolean; { "playing field" }

var
   cells     : cell_field;       { playing field for experiment }
   initcell  : string;           { input cell configuration }

{ ------------------------------------------------------------ }

procedure init;
{ initializes the cell configuration space (the "playing field") }
var
  i,j : integer;
begin
        for j : = 1 to columns do
           cells[j] : = FALSE;   { initialize cell space }
end;

{ ------------------------------------------------------------ }

procedure display;
{ displays line of cells to screen }
var
   i,j : integer;

begin
```

```
    begin
        for j : = 1 to columns do
            if cells[j] = TRUE then
                write(cell)          { cell in space }
            else
                write(' ');          { no cell in space }
    end
end;

{ ------------------------------------------------------------ }

procedure load;
{ load initial cell file into array of cells }
var
   i,j : integer;

begin
    ClrScr;                                  { clear screen  - }
    writeln('Input initial cell configuration ');
    readln(initcell);
    init;                                    { initialize cell space }
    for j : = 1 to length(initcell) do
        if initcell[j] = cell then
            cells[j] : = TRUE                { live cell in space }
end;

{ ------------------------------------------------------------ }

procedure rule;
{ apply one dimensional cellular automata rule for one iteration.
  note that the first and last cells in a row are omitted
}
var
   oldcells : cell_field;            { holds copy of old cell field }
   i,j : integer;
   am1,a0,a1 : boolean;              { left, current and right cells }
begin
     for j : = 1 to columns do
       oldcells[j] : = cells[j];     { remember old cell configuration }

    init;                            { initialize next configuration  }

    for j : = 2 to columns - 1 do    { omit boundary cells }
        begin
           am1 : = oldcells[j - 1];
           a0  : = oldcells[j];
           a1  : = oldcells[j + 1];
           cells[j] : = (am1 AND NOT a0 AND NOT a1) OR (NOT am1 AND a1) OR (a0 AND a1)
        end
end;

{ ------------------------ begin program ----------------------- }
```

```
var
   i : integer;
   iter      : integer;       { number of iterations for simulation}

begin
     load;
     display;
     write('Enter number of iterations for simulation ');
     readln(iter);
     write('Press Enter to Begin Simulation ');
     readln;
     ClrScr;                  { clear screen }
     display;                 { display starting configuration}
     for i : = 1 to iter do
     begin
        rule;                 { apply rule to cell field }
        display               { display updated universe }
     end
end.
```

```
program cell2;
{ Simulate a self-organizing one-dimensional cellular automata
  1/29/93   Phil Laplante       }

uses
     Crt;                     { unit for CRT driver }

const
     columns  = 80;           { number of columns on screen }
     rows     = 24;           { number of rows on screen }
     cell     = '*';          { cell symbol }

type
     cell_field = array[1..columns] of boolean; { "playing field" }

var
   cells     : cell_field;       { playing field for experiment }
   initcell  : string;           { input cell configuration }

{------------------------------------------------------------------ }

procedure init;
{ initializes the cell configuration space (the "playing field") }
var
  i,j : integer;
begin
        for j : = 1 to columns do
           cells[j] : = FALSE;   { initialize cell space }
end;
```

```
{ ------------------------------------------------------------- }

procedure display;
{ displays line of cells to screen }
var
   i,j : integer;

begin
     begin
          for j : = 1 to columns do
              if cells[j] = TRUE then
                  write(cell)          { cell in space }
              else
                  write(' ');          { no cell in space }
    end
end;

{ ------------------------------------------------------------- }

procedure load;
{ load initial cell file into array of cells }
var
   i,j : integer;

begin
     ClrScr;                                  { clear screen         }
     writeln('Input initial cell configuration ');
     readln(initcell);
     init;                                    { initialize cell space }
     for j : = 1 to length(initcell) do
         if initcell[j] = cell then
              cells[j] : = TRUE               { live cell in space }
end;

{ ------------------------------------------------------------- }

procedure rule;
{ apply one dimensional cellular automata rule for one iteration.
  note that the first and last cells in a row are omitted
}
var
   oldcells : cell_field;            { holds copy of old cell field }
   i,j : integer;
   am1,a0,a1 : boolean;              { left, current and right cells }

begin
      for j : = 1 to columns do
        oldcells[j] : = cells[j];    { remember old cell configuration }

     init;                           { initialize next configuraion  }

     for j : = 2 to columns - 1 do   { omit boundary cells }
```

```
        begin
           am1 : = oldcells[j - 1];
           a0  : = oldcells[j];
           a1  : = oldcells[j + 1];
           cells[j] : = (am1 AND NOT a0 and NOT a1) OR (NOT am1 AND a1)
        end
end;

{ ------------------------ begin program ------------------------- }

var
   i : integer;
   iter      : integer;       { number of iterations for simulation}

begin
     load;
     display;
     write('Enter number of iterations for simulation ');
     readln(iter);
     write('Press Enter to Begin Simulation ');
     readln;
     ClrScr;                  { clear screen }
     display;                 { display starting configuration}
     for i : = 1 to iter do
     begin
        rule;                 { apply rule to cell field }
        display               { display updated universe }
     end
end.
```

```
program cloud;
{ compute and display Julia set of function
  f(z) = z^2 - 0.194 + 0.6557i

  1 - 2 - 92  Phil Laplante                    }
uses
   Complex, Graph;           { include graphics and complex routines}

const
   zoom=1.5;                 { create 3 by 3 window }
   attract=0.0001;           { attractor sensitivity }

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   i, j      : integer;      { loop variables}
   MaxY      : integer;      { Maximum Y screen coordinate}
   scale     : real;         { scale factor }
   mag       : real;         { square of magnitude of complex number }
```

```
   iter      : integer;      { escape iteration counter }
   continue  : boolean;      { continue iteration counter }
   x,y       : real;         { real and complex parts of z }
   MaxColor  : integer;      { maximum number of colors on graphics card }

begin

   { initialize graphics }

   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxColor : = GetMaxColor; { find maximum number of colors }
   MaxY : = GetMaxY;         { find maximum Y screen coordinate }
   scale: = 2.0*zoom/MaxY;   { calculate zoom factor}

   for i : = 0 to MaxY do    { MaxY is usually smaller than MaxY }
   begin
       for j : = 0 to MaxY do
       begin
            x : = scale*i - zoom;            { set starting value of real(z) }
            y : = zoom - scale*j;            { set starting value of imag(z) }
            continue : = TRUE;               { assume point does not escape  }
            iter : =0;
            while continue = TRUE do
            begin
                mult(x,y,x,y,x,y);         { square z }
                add(x,y,-0.194,0.6557,x,y); { add constant }
                mag : = x*x + y*y;          { calculate square of magnitude }
                if mag < attract then
                  continue : = FALSE       { point is an attractor }
                else
                  if (mag < 100) AND (iter < MaxColor*2) then { keep iterating function }
                     iter : = iter + 1
                  else                     { point escapes, plot it }
                     begin
                       case iter div 2 of
                        WHITE: putpixel(i,j, LightGray);
                        YELLOW: putpixel(i,j, DarkGray)
                     end;
                       continue : = FALSE        { get out of loop }
                     end
            end  { while loop}
       end  {j loop}
```

```
   end{ i loop}
end.
```

```
program clouds2;
{ compute and display clouds
  using Michael Barnsley's IFS algorithm

  12 - 5 - 93  Frank D'Erasmo }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
   i         : integer;      { loop counters}
   k         : integer;      { row selector }
   MaxY      : integer;      { Maximum Y screen coordinate}
   d         : array[1..4,1..6] of real; { holds data of IFS attractor }

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;

   MaxY : = GetMaxY;

{ initialize IFS data array }

   d[1,1]: = 0.5; d[1,2]: = 0; d[1,3]: = 0; d[1,4]: = 0.5; d[1,5]: = 0; d[1,6]: = 0;
   d[2,1]: = 0.5; d[2,2]: = 0; d[2,3]: = 0; d[2,4]: = 0.5; d[2,5]: = 2; d[2,6]: = 0;
   d[3,1]: = -0.4; d[3,2]: = 0; d[3,3]: = 1; d[3,4]: = 0.4; d[3,5]: = 0; d[3,6]: = 1;
   d[4,1]: = -0.5; d[4,2]: = 0; d[4,3]: = 0; d[4,4]: = 0.5; d[4,5]: = 2; d[4,6]: = 1;
   randomize;           {initialize random number generator}

   x : = 0;              {set starting coordinates}
   y : = 0;

   for i : = 1 to 32000 do
     begin
        k : = random(4) + 1;                  { pick random number from 1-4}
        x : = d[k,1]*x + d[k,2]*y + d[k,5];    { transform coordinates }
```

```
        y : = d[k,3]*x + d[k,4]*y + d[k,6];
        if i > 10 then                          { skip first 10 iterations }
           putpixel(round(MaxY*x/2),MaxY - round(MaxY*y/2),WHITE)
           end;                                 { scale for screen }
end.
  unit complex;

interface
procedure add (x1, y1, x2, y2 : real; var x3, y3 : real); { complex addition       }
procedure sub (x1, y1, x2, y2 : real; var x3, y3 : real); { complex subtraction    }
procedure mult(x1, y1, x2, y2 : real; var x3, y3 : real); { complex multiplication }
procedure cdiv(x1, y1, x2, y2  : real; var x3, y3 : real); { complex division       }
function cosh(x : real) : real;                  { hyperbolic cosine      }
function sinh(x : real) : real;                  { hyperbolic sine        }
procedure csin(x,y : real; var x1, y1 :real);    { complex sine           }
procedure ccos(x,y : real; var x1, y1 :real);    { complex cosine         }
procedure cexp(x,y : real; var x1, y1 :real);    { complex exponentiation }

implementation

procedure add (x1, y1, x2, y2 : real; var x3, y3 : real);
{ calculates z3 = z1 + z2 where:
  z1 = x1 + iy1;
  z2 = x2 + iy2;
  z3 = x3 + iy3;
}
begin
  x3 := x1 + x2;
  y3 := y1 + y2
end;

procedure sub (x1, y1, x2, y2 : real; var x3, y3 : real);
{ calculates z3 = z1-  z2 where:
  z1 = x1 + iy1
  z2 = x2 + iy2
  z3 = x3 + iy3
}
begin
  x3 := x1 - x2;
  y3 := y1 - y2
end;


procedure mult(x1, y1, x2, y2 : real; var x3, y3 : real);
{ calculates z3 = z1 * z2 where:
  z1 = x1 + iy1
  z2 = x2 + iy2
  z3 = x3 + iy3
}
begin
  x3 := x1*x2 - y1*y2;
  y3 := y1*x2 + x1*y2
```

```
end;

procedure cdiv(x1, y1, x2, y2 : real; var x3, y3 : real);
{ calculates z3 = z1 / z2 where:
  z1 = x1 + iy1
  z2 = x2 + iy2
  z3 = x3 + iy3
}
var
  denom : real;                    { denominator }
begin
  denom : = x2*x2 + y2*y2;
  x3 : = (x1*x2 + y1*y2)/ denom;   { real part }
  y3 : = (x2*y1-  x1*y2)/ denom    { imaginary part }
end;


function cosh( x : real) : real;     { calculates cosh(x) }
begin
    cosh : = (exp(x) + exp(-x))/2.0
end;

function sinh( x : real) : real;     { calculates sinh(x) }
begin
    sinh : = (exp(x) - exp(-x))/2.0
end;

procedure ccos(x, y : real; var x1, y1 : real);
{ calculates z = cos(x + iy) where:
  x1 is real part of z
  y1 is imaginary part of z }
begin
    x1 : = cos(x)*cosh(y);
    y1 : = -sin(x)*sinh(y)
end;

procedure csin(x, y : real; var x1, y1 : real);
{ calculates z = sin(x + iy) where:
  x1 is real part of z
  y1 is imaginary part of z }
   begin
    x1 : = sin(x)*cosh(y);
    y1 : = cos(x)*sinh(y)
end;

procedure cexp(x, y : real; var x1, y1 : real);
{ calculates z = e^(x + iy) where
  x1 is real part of z
  y1 is imaginary part of z
}
begin
  x1 : = exp(x)*cos(y);
```

```
  y1 : = exp(x)*sin(y)
end;
end.  {unit complex}
```

```
program dendrite;
{ compute and display Julia set of function
  f(z) = z^2 + i

  1 - 2 - 92  Phil Laplante                    }
uses
   Complex, Graph;           { include graphics and complex routines}


const
   zoom=2.0;                 { create 4 by 4 window }
   attract=0.0001;           { attractor sensitivity }

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   i, j      : integer;      { loop variables}
   MaxY      : integer;      { Maximum Y screen coordinate}
   scale     : real;         { scale factor }
   mag       : real;         { square of magnitude of complex number }
   iter      : integer;      { escape iteration counter }
   continue  : boolean;      { continue iteration counter }
   x,y       : real;         { real and complex parts of z }
   MaxColor  : integer;      { maximum number of colors on graphics card }

begin

   { initialize graphics }

   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxColor : = GetMaxColor; { find maximum number of colors }
   MaxY : = GetMaxY;         { find maximum Y screen coordinate }
   scale: = 2.0*zoom/MaxY;   { calculate zoom factor}

   for i : = 0 to MaxY do    { MaxY is usually smaller than MaxY }
   begin
```

```
      for j : = 0 to MaxY do
      begin
           x : = scale*i - zoom;              { set starting value of real(z) }
           y : = zoom - scale*j;              { set starting value of imag(z) }
           continue : = TRUE;                 { assume point does not escape  }
           iter : =0;
           while continue = TRUE do
           begin
               mult(x,y,x,y,x,y);             { square z }
               add(x,y,0.0,1.0,x,y);          { add 0 + i }
               mag : = x*x + y*y;             { calculate square of magnitude }
               if mag < attract then
                 continue : = FALSE           { point is an attractor }
               else
                 if (mag < 100) AND (iter < MaxColor*2) then { keep iterating function }
                    iter : = iter + 1
                 else                         { point escapes, plot it }
                    begin
                      putpixel(i,j, iter div 2);
                      continue : = FALSE        { get out of loop }
                    end
           end  { while loop}
      end  {j loop}
   end{ i loop}
end.
```

```
program dragon;
{ compute and display a "dragon" from the Julia set of

    f(z) = z^2 + 0.360284 + 0.100376i

  12 - 21 - 92  Phil Laplante                    }

uses
   Complex, Graph;           { include graphics and complex routines}

const
   zoom=2.0;                 { create 4 by 4 window }
   attract=0.0001;           { attractor sensitivity }

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   i, j      : integer;      { loop variables}
   MaxY      : integer;      { Maximum Y screen coordinate}
   scale     : real;         { scale factor }
   mag       : real;         { square of magnitude of complex number }
   iter      : integer;      { escape iteration counter }
   continue  : boolean;      { continue iteration counter }
```

```
   x, y      : real;          { real and complex parts of z }
   MaxColor  : integer;       { maximum number of colors on graphics card }

   { initialize graphics }

begin

   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,''); {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxColor : = GetMaxColor; { find maximum number of colors }
   MaxY : = GetMaxY;         { find maximum Y screen coordinate }
   scale: = 2.0*zoom/MaxY;   { calculate zoom factor}

   for i : = 0 to MaxY do    { MaxY is usually smaller than MaxX }
   begin
       for j : = 0 to MaxY do
       begin
            x : = scale*i - zoom;            { set starting value of real(z) }
            y : = zoom - scale*j;            { set starting value of imag(z) }
            continue : = TRUE;               { assume point does not escape  }
            iter : =0;
            while continue = TRUE do
            begin
                mult(x,y,x,y,x,y);        { square z }
                add(x,y,0.360284,0.100376,x,y);    { add constant }
                mag : = x*x + y*y;       { calculate square of magnitude }
                if mag < attract then
                  continue : = FALSE       { point is an attractor }
                else
                  if (mag < 100) AND (iter < MaxColor*2) then { keep iterating function }
                     iter : = iter + 1
                  else                     { point escapes, plot it }
                     begin
                       putpixel(i,j, iter div 2);
                       continue : = FALSE        { get out of loop }
                     end
            end  { while loop}
       end  {j loop}
   end{ i loop}
end.
```

```
program EKG;
{ compute and display a simulated "EKG" from the Julia set of

    f(z) = z^2 + -1.5

  12 - 21 - 92  Phil Laplante                    }
uses
   Complex, Graph;           { include graphics and complex routines}

const
   zoom = 2.0;               { create 4 by 4 window }
   attract = 0.0001;         { attractor sensitivity }

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   i, j      : integer;      { loop variables}
   MaxY      : integer;      { Maximum Y screen coordinate}
   scale     : real;         { scale factor }
   mag       : real;         { square of magnitude of complex number }
   iter      : integer;      { escape iteration counter }
   continue  : boolean;      { continue iteration counter }
   x,y       : real;         { real and complex parts of z }
   MaxColor  : integer;      { maximum number of colors on graphics card }

begin

   { initialize graphics }

   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxColor : = GetMaxColor; { find maximum number of colors }
   MaxY : = GetMaxY;         { find maximum X screen coordinate }
   scale: = 2.0*zoom/MaxY;   { calculate zoom factor}

   for i : = 0 to MaxY do    { MaxY is usually smaller than MaxX }
   begin
       for j : = 0 to MaxY do
       begin
            x : = scale*i - zoom;          { set starting value of real(z) }
            y : = zoom - scale*j;          { set starting value of imag(z) }
            continue : = TRUE;             { assume point does not escape  }
            iter : =0;
```

```
            while continue = TRUE do
            begin
                mult(x,y,x,y,x,y);          { square z }
                x: =x - 1.5;                {add- 1.5}
                mag : = x*x + y*y;          { calculate square of magnitude }
                if mag < attract then
                  continue : = FALSE        { point is an attractor }
                else
                  if (mag < 100) AND (iter < MaxColor*2) then { keep iterating function }
                     iter : = iter + 1
                  else                      { point escapes, plot it }
                     begin
                       if iter div 2 = MAGENTA then
                           putpixel(i,j, WHITE);
                       continue : = FALSE       { get out of loop }
                     end
            end  { while loop}
       end  {j loop}
   end{ i loop}
end.
```

```
program fall;
{ compute and display cross fractal
  using Michael Barnsley's IFS algorithm.  Then generate
  snow fall by generating many of them.

  12 - 5 - 93  Phil Laplante                }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
   i, j      : integer;      { loop counters}
   k         : integer;      { row selector }
   MaxX      : integer;      { maximum X screen coordinate}
   d         : array[1..5,1..6] of real; { holds data of IFS attractor }
   scale     : real;         { random scale factor}
   xpos,ypos : integer;      { tree position }

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,"");  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
```

```
        Writeln('Program aborted');
        Halt(1)
    end;
    MaxX : = GetMaxX;          { get screen limits }

{ initialize IFS data array }

    d[1,1]: =0.33; d[1,2]: =0; d[1,3]: =0; d[1,4]: =0.33; d[1,5]: =1;  d[1,6]: =1;
    d[2,1]: =0.33; d[2,2]: =0; d[2,3]: =0; d[2,4]: =0.33; d[2,5]: =MaxX; d[2,6]: =1;
    d[3,1]: =0.33; d[3,2]: =0; d[3,3]: =0; d[3,4]: =0.33; d[3,5]: =1;  d[3,6]: =MaxX;
    d[4,1]: =0.33; d[4,2]: =0; d[4,3]: =0; d[4,4]: =0.33; d[4,5]: =MaxX; d[4,6]: =MaxX;
    d[5,1]: =0.33; d[5,2]: =0; d[5,3]: =0; d[5,4]: =0.33; d[5,5]: =MaxX div 2; d[5,6]: =MaxX div 2;

    MaxX : = GetMaxX;

    randomize;           {initialize random number generator}

    x : = 0;             {set starting coordinates}
    y : = 0;

    for j : = 1 to 150 do                    { make snow flakes }
     begin
      xpos : = random(MaxX);                 { pick flake position }
      ypos : = random(MaxX);
      scale : = (random(5) + 1)/250;         { pick flake scale }

      for i : = 1 to 800 do
      begin
         k : = random(5) + 1;                { pick random row }
         x : = d[k,1]*x + d[k,2]*y + d[k,5]; { transform coordinates }
         y : = d[k,3]*x + d[k,4]*y + d[k,6];
         if i > 10 then                      { skip first 10 iterations }
            putpixel(round(scale*x + xpos),round(scale*y + ypos),WHITE)
      end
     end
end.
```

```
program fern;
{ compute and display fern
  using Michael Barnsley's IFS algorithm

  12 - 5 - 93  Phil Laplante                 }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
```

```
   i         : integer;      { loop counter}
   q         : integer;      { random number }
   k         : integer;      { row selector }
   MaxY      : integer;      { Maximum Y screen coordinate}
   d         : array[1..4,1..6] of real; { holds data of IFS attractor }

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,''); {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;

   MaxY : = GetMaxY;

{ initialize IFS data array }

   d[1,1]: = 0; d[1,2]: = 0; d[1,3]: = 0; d[1,4]: =0.16; d[1,5]: = 0; d[1,6]: = 0;
   d[2,1]: = 0.85; d[2,2]: = 0.04; d[2,3]: = -0.04; d[2,4]: = 0.85; d[2,5]: = 0; d[2,6]: =1 .6;
   d[3,1]: = 0.2; d[3,2]: = -0.26; d[3,3]: = 0.23; d[3,4]: = 0.22; d[3,5]: = 0; d[3,6]: = 1.6;
   d[4,1]: = -0.15; d[4,2]: = 0.28; d[4,3]: = 0.26; d[4,4]: = 0.24; d[4,5]: = 0; d[4,6]: = 0.44;


   randomize;           {initialize random number generator}

   x : = 0;             {set starting coordinates}
   y : = 0;

   for i : = 1 to 30000 do
     begin
        q : = random(100) + 1;              { pick random number from 1-100}
        if q < = 85 then                    { assign row according to }
           k : = 2;                         { probability }
        if q = 86 then
           k : = 1;
        if (q > 86) AND (q < 94) then
           k : = 3;
        if (q > = 94 ) then
           k : = 4;

        x : = d[k,1]*x + d[k,2]*y + d[k,5];  { transform coordinates }
        y : = d[k,3]*x + d[k,4]*y + d[k,6];
        if i > 10 then                      { skip first 10 iterations }
             putpixel(round(MaxY/2 + MaxY*x/10),round(MaxY*y/10),GREEN)
           end                              { scale for screen }
end.
```

```
Program fib;

{ finds the nth fibonacci number
  11 - 24 - 92  Phil Laplante          }

function fibo(i:integer) : integer;
{ a recursive function }
begin
   if i = 0 then
      fibo : = 0
   else
     if i = 1 then
       fibo : = 1
       else
         fibo : = fibo(i - 1) + fibo(i - 2)
end;

var
  n : integer;           { nth number in sequence }

begin
  write('Enter n > 0 and less than 24 --> ');
  readln(n);
  writeln('f(',n,') = ',fibo(n))
end.
```

```
program flower1;
{ compute and display a "rose" from the Julia set of

    f(z) = z^2 + .384

  12 - 21 - 92  Phil Laplante                    }
uses
   Complex, Graph;           { include graphics and complex routines}

const
   zoom = 2.0;               { create 4 by 4 window }
   attract = 0.0001;         { attractor sensitivity }

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   i, j      : integer;      { loop variables}
   MaxY      : integer;      { Maximum Y screen coordinate}
   scale     : real;         { scale factor }
   mag       : real;         { square of magnitude of complex number }
   iter      : integer;      { escape iteration counter }
   continue  : boolean;      { continue iteration counter }
   x,y       : real;         { real and complex parts of z }
```

```
   MaxColor  : integer;      { maximum number of colors on graphics card }

begin

   { initialize graphics }

   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxColor : = GetMaxColor; { find maximum number of colors }
   MaxY : = GetMaxY;         { find maximum X screen coordinate }
   scale: = 2.0*zoom/MaxY;   { calculate zoom factor}

   for i : = 0 to MaxY do    { MaxY is usually smaller than MaxX }
   begin
       for j : = 0 to MaxY do
       begin
            x : = scale*i - zoom;          { set starting value of real(z) }
            y : = zoom - scale*j;          { set starting value of imag(z) }
            continue : = TRUE;             { assume point does not escape  }
            iter : =0;
            while continue = TRUE do
            begin
                mult(x,y,x,y,x,y);         { square z }
                x: =x + 0.384;             { add 0.384 }
                mag : = x*x + y*y;         { calculate square of magnitude }
                if mag < attract then
                  continue : = FALSE       { point is an attractor }
                else
                  if (mag < 100) AND (iter < MaxColor*2) then { keep iterating function }
                     iter : = iter + 1
                  else                     { point escapes, plot it }
                     begin
                       putpixel(i,j, iter div 2);
                       continue : = FALSE        { get out of loop }
                     end
            end  { while loop}
       end  {j loop}
   end{ i loop}
end.
```

```
program flower2;
{ compute and display a "chrysanthemum" from the Julia set of
```

```
    f(z) = z^2 + .2541

  12 - 21 - 92  Phil Laplante                    }
uses
   Complex, Graph;           { include graphics and complex routines}

const
   zoom = 2.0;               { create 4 by 4 window }
   attract = 0.0001;         { attractor sensitivity }

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   i, j      : integer;      { loop variables}
   MaxY      : integer;      { Maximum Y screen coordinate}
   scale     : real;         { scale factor }
   mag       : real;         { square of magnitude of complex number }
   iter      : integer;      { escape iteration counter }
   continue  : boolean;      { continue iteration counter }
   x,y       : real;         { real and complex parts of z }
   MaxColor  : integer;      { maximum number of colors on graphics card }

begin

   { initialize graphics }

   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxColor : = GetMaxColor; { find maximum number of colors }
   MaxY : = GetMaxY;         { find maximum X screen coordinate }
   scale: = 2.0*zoom/MaxY;   { calculate zoom factor}

   for i : = 0 to MaxY do    { MaxY is usually smaller than MaxX }
   begin
       for j : = 0 to MaxY do
       begin
            x : = scale*i - zoom;          { set starting value of real(z) }
            y : = zoom - scale*j;          { set starting value of imag(z) }
            continue : = TRUE;             { assume point does not escape  }
            iter : = 0;
            while continue = TRUE do
            begin
                mult(x,y,x,y,x,y);         { square z }
```

```
                x: = x + 0.2541;          {add 0.2541}
                mag : = x*x + y*y;        { calculate square of magnitude }
                if mag < attract then
                  continue : = FALSE      { point is an attractor }
                else
                  if (mag < 100) AND (iter < MaxColor*2) then { keep iterating function }
                     iter : = iter + 1
                  else                    { point escapes, plot it }
                     begin
                       putpixel(i,j, iter div 2);
                       continue : = FALSE       { get out of loop }
                     end
            end  { while loop}
        end  {j loop}
    end{ i loop}
end.
```

```
program forest;
{ compute and display forest of trees
  using Michael Barnsley's IFS algorithm

  12 - 5 - 93  Phil Laplante                }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
   i, j      : integer;      { loop counters}
   q         : integer;      { random number }
   k         : integer;      { row selector }
   MaxX      : integer;      { Maximum X screen coordinate}
   d         : array[1..4,1..6] of real; { holds data of IFS attractor }
   scale     : real;         { random scale factor}
   xpos,ypos : integer;      { tree position }
   crand     : integer;      { pick random color (green, blue, yellow) }
   color     : integer;      { random color value }

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
```

```
   end;

   MaxX : = GetMaxX;

{ initialize IFS data array }

   d[1,1]: = 0;     d[1,2]: = 0;       d[1,3]: = 0;       d[1,4]: = 0.5;  d[1,5]: = 0; d[1,6]: = 0;
   d[2,1]: = 0.42; d[2,2]: = - 0.42; d[2,3]: = 0.42;   d[2,4]: = 0.42; d[2,5]: = 0; d[2,6]: =0.2;
   d[3,1]: = 0.42; d[3,2]: = 0.42;   d[3,3]: = - 0.42; d[3,4]: = 0.42; d[3,5]: = 0; d[3,6]: =0.2;
   d[4,1]: = 0.1;  d[4,2]: = 0;       d[4,3]: = 0;       d[4,4]: = 0.1;  d[4,5]: = 0; d[4,6]: =0.2;

   randomize;           {initialize random number generator}

   x : = 0;             {set starting coordinates}
   y : = 0;

   for j : = 1 to 150 do                        { make 150 trees }
     begin
     xpos : = random(MaxX);                     { pick tree position }
     ypos : = random(MaxX);
     scale : = random(3) + 1;                   { pick tree scale }
     crand : = random(10) + 1;                  { pick tree color }
     case  crand of
        0,1,2,3,4,5,6,7,8:
             color : = GREEN;                   { most trees are green }
        9  : color : = YELLOW;                  { some trees are yellow }
        10 : color : = BROWN;                   { some trees die }
     end;

     for i : = 1 to 800 do                     { 800 pixels per tree }
      begin
        q : = random(100) + 1;                 { pick random number from 1 - 100}
        if q < = 40 then                       { assign row according to }
           k : = 2;                            { probability }
        if (q> 40) AND (q < 81) then
           k : = 3;
        if (q > = 81) AND (q < 95) then
           k : = 4;
        if (q > = 95 ) then
           k : = 1;

        x : = d[k,1]*x + d[k,2]*y + d[k,5];    { transform coordinates }
        y : = d[k,3]*x + d[k,4]*y + d[k,6];

        if i > 10 then                        { skip first 10 iterations }
             putpixel(xpos + round(x*MaxX/scale),round(ypos - y*MaxX/scale),color)
      end
   end
end.
```

```
program galax1;
{ compute and display view of space using Michael Barnsley's
  IFS algorithm.

  12 - 5 - 93  Phil Laplante                    }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
   i, j      : integer;      { loop counters}
   k         : integer;      { row selector }
   MaxX      : integer;      { maximum X and Y coordinates}
   d         : array[1..5,1..6] of real; { holds data of IFS attractor }
   scale     : real;         { random scale factor}
   xpos,ypos : integer;      { tree position }

begin
   GraphDriver : = Detect;   {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxX : = GetMaxX;          { get screen limits }

{ initialize IFS data array }

   d[1,1]: = 0.33; d[1,2]: = 0; d[1,3]: = 0; d[1,4]: = 0.33; d[1,5]: = 1; d[1,6]: = 1;
   d[2,1]: = 0.33; d[2,2]: = 0; d[2,3]: = 0; d[2,4]: = 0.33; d[2,5]: = MaxX; d[2,6]: = 1;
   d[3,1]: = 0.33; d[3,2]: = 0; d[3,3]: = 0; d[3,4]: = 0.33; d[3,5]: = 1; d[3,6]: = MaxX;
   d[4,1]: = 0.33; d[4,2]: = 0; d[4,3]: = 0; d[4,4]: = 0.33; d[4,5]: = MaxX; d[4,6]: = MaxX;
   d[5,1]: = 0.33; d[5,2]: = 0; d[5,3]: = 0; d[5,4]: = 0.33; d[5,5]: = MaxX div 2; d[5,6]: = MaxX div

   MaxX : = GetMaxX;

   randomize;           {initialize random number generator}

   x : = 0;             {set starting coordinates}
   y : = 0;

   for j : = 1 to 150 do                      { make stars }
    begin
     xpos : = random(MaxX);                   { pick star position }
     ypos : = random(MaxX);
```

```
     scale : = (random(5) + 1)/1000;          { pick star scale }

     for i : = 1 to 800 do
     begin
        k : = random(5) + 1;                  { pick random row }
        x : = d[k,1]*x + d[k,2]*y + d[k,5];   { transform coordinates }
        y : = d[k,3]*x + d[k,4]*y + d[k,6];
        if i > 10 then                        { skip first 10 iterations }
           putpixel(round(scale*x + xpos),round(scale*y + ypos),WHITE)
     end
    end
end.
```

```
program julial;
{ compute and display Julia set of cos z
  12 - 21 - 93   Phil Laplante                    }

uses
   Complex, Graph;           { include graphics and complex routines}

const
   zoom = 2.0;               { create 4 by 4 window }
   attract = 0.0001;         { attractor sensitivity }

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   i, j      : integer;      { loop variables}
   MaxY      : integer;      { Maximum Y screen coordinate}
   scale     : real;         { scale factor }
   mag       : real;         { square of magnitude of complex number }
   iter      : integer;      { escape iteration counter }
   continue  : boolean;      { continue iteration counter }
   x, y      : real;         { real and imaginary parts of z }
   MaxColor  : integer;      { maximum number of colors on graphics card }

begin

   { initialize graphics }

   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
```

```
    end;
    MaxColor : = GetMaxColor; { find maximum number of colors }
    MaxY : = GetMaxY;         { find maximum X screen coordinate }
    scale: = 2.0*zoom/MaxY;   { calculate zoom factor}

    for i : = 0 to MaxY do    { MaxY is usually smaller than MaxX }
    begin
        for j : = 0 to MaxY do
        begin
            x : = scale*i - zoom;          { set starting value of real(z) }
            y : = zoom - scale*j;          { set starting value of imag(z) }
            continue : = TRUE;             { assume point does not escape  }
            iter : =0;
            while continue = TRUE do
            begin
                ccos(x,y,x,y);             { calculate complex cosine }
                mag : = x*x + y*y;         { calculate square of magnitude }
                if mag < attract then
                  continue : = FALSE       { point is an attractor }
                else
                  if (mag < 100) AND (iter < MaxColor*2) then { keep iterating function }
                     iter : = iter + 1
                  else                     { point escapes, plot it }
                     begin
                       putpixel(i,j, iter div 2);
                       continue : = FALSE        { get out of loop }
                     end
            end  { while loop}
        end  {j loop}
    end{ i loop}
end.
```

```
program julia2;
{ compute and display Julia set of sine z
  12 - 21 - 92  Phil Laplante                    }

uses
    Complex, Graph;           { include graphics and complex routines}

const
    zoom = 2.0;               { create 4 by 4 window }
    attract = 0.0001;         { attractor sensitivity }

var
    GraphDriver : integer;    { Stores graphics driver number}
    GraphMode : integer;      { Stores graphics mode for driver}
    ErrorCode : integer;      { Reports any error condition}
    i, j      : integer;      { loop variables}
    MaxY      : integer;      { Maximum Y screen coordinate}
    scale     : real;         { scale factor }
```

```
   mag       : real;          { square of magnitude of complex number }
   iter      : integer;       { escape iteration counter }
   continue  : boolean;       { continue iteration counter }
   x, y      : real;          { real and imaginary parts of z }
   MaxColor  : integer;       { maximum number of colors on graphics card }

begin
   { initialize graphics }

   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,''); {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxColor : = GetMaxColor; { find maximum number of colors }
   MaxY : = GetMaxY;         { find maximum Y screen coordinate }
   scale: = 2.0*zoom/MaxY;   { calculate zoom factor}

   for i : = 0 to MaxY do    { MaxY is usually smaller than MaxX }
   begin
       for j : = 0 to MaxY do
       begin
            x : = scale*i - zoom;          { set starting value of real(z) }
            y : = zoom - scale*j;          { set starting value of imag(z) }
            continue : = TRUE;             { assume point does not escape  }
            iter : = 0;
            while continue = TRUE do
            begin
                csin(x,y,x,y);             { calculate complex sine }
                mag : = x*x + y*y;         { calculate square of magnitude }
                if mag < attract then
                  continue : = FALSE       { point is an attractor }
                else
                  if (mag < 100) AND (iter < MaxColor*2) then { keep iterating function }
                     iter : = iter + 1
                  else                     { point escapes, plot it }
                     begin
                       putpixel(i,j, iter div 2);
                       continue : = FALSE        { get out of loop }
                     end
            end  { while loop}
       end  {j loop}
   end{ i loop}
end.
```

```
program life;
{ Simulate Conway's Game of Life
   1/29/93   Phil Laplante       }

uses
     Crt;                    { unit for CRT driver }

const
     columns  = 80;          { number of columns on screen }
     rows     = 24;          { number of rows on screen }
     cell     = '*';         { cell symbol }

type
     cell_field = array[1..rows,1..columns] of boolean; { "playing field" }

var
   Field     : text;             { file containing initial universe}
   cells     : cell_field;       { playing field for experiment }
   oldcells  : cell_field;       { holds copy of old cell field }
   FileName  : string;           { name of initial universe }
   cellrow   : string[columns];  { row of cells             }

{ ------------------------------------------------------------ }

procedure init;
{ initializes the cell configuration space (the "playing field") }
var
  i,j : integer;
begin
     for i : = 1 to rows do
        for j : = 1 to columns do
           cells[i,j] : = FALSE;                  { initialize cell space }
end;

{ ------------------------------------------------------------- }

procedure display;
{ displays array of cells to screen }
var
   i,j : integer;

begin
     ClrScr;

     for i : = 1 to rows do
         begin
              for j : = 1 to columns do
                  if cells[i,j] = TRUE then
                     write(cell)          { cell in space }
                  else
                     write(' ');          { no cell in space }
         end
```

```
end;

{ ------------------------------------------------------------ }

procedure load;
{ load initial cell file into array of cells }
var
   i,j : integer;

begin
     write('Input life file name ');
     readln(FileName);
     assign(Field,FileName);

     reset(Field);                          { open file for reading }
     init;                                  { initialize cell space }
     for i : = 1 to rows do
         begin
             readln(Field,cellrow);
             for j : = 1 to length(cellrow) do
                 if cellrow[j] = cell then
                    cells[i,j] : = TRUE          { live cell in space
}
         end;
     close(Field)
end;

{ ------------------------------------------------------------ }

function count(i,j:integer) : integer;
{ count number of cells in neighborhood }
var
   l,m : integer;          { loop counters }
   sum: integer;           { running sum }
   srow, erow : integer;   { starting and ending row locations }
   scol, ecol : integer;   { starting and ending column locations }

begin
     sum : = 0;            { initialize running sums }

     if i = 1 then         { find starting row }
        srow : = 1
     else
        srow : = i - 1;

     if i = rows then      { find ending row }
        erow : = rows
     else
        erow : = i + 1;

     if j = 1 then         { find starting column }
        scol : = 1
```

```
     else
        scol : = j - 1;

     if j = columns  then  { find ending column }
        ecol : = columns
     else
        ecol : = j + 1;

     for l : = srow to erow do            { count nearby cells }
       for m : = scol to ecol do
        if (i<>l) OR (j<>m) then       { don't count self }
          if oldcells[l,m] = TRUE then
             sum: = sum + 1;
     count : = sum;                       { return sum }
end;

{ ------------------------------------------------------------ }

procedure rule;
{ apply Game of Life rule
  note that the first and last cells in a row are treated specially }
var
   i,j : integer;                  { local counters }

begin
     for i : = 1 to rows  do
      for j : = 1 to columns do
        oldcells[i,j] : = cells[i,j];    { remember old cell configuration }

     init;                               { initialize next configuration  }

     for i : = 1 to rows  do
          for j : = 1 to columns do
             case count(i,j) of         { count nearby cells }
             0,1 :       cells[i,j] : = FALSE;
             2:          cells[i,j] : = oldcells[i,j];
             3:          cells[i,j] : = TRUE;
             4,5,6,7,8 : cells[i,j] : = FALSE          { overcrowding }
             end
end;

{ ----------------------  begin program---------------------- }

var
   i : integer;
   iter      : integer;       { number of iterations for simulation}

begin
     load;
     write('File Loaded');
     display;
     write('Enter number of iterations for simulation ');
```

```
     readln(iter);
     write('Press Enter to Begin Simulation ');
     readln;
     for i : = 1 to iter do
     begin
        rule;                { apply rule to cell field }
        display              { display updated universe }
     end
end.
```

```
program life2;
{ Simulate Conway's Game of Life - generates random starting configuration
   1/29/93   Phil Laplante       }

uses
     Crt;                    { unit for CRT driver }

const
     columns  = 80;          { number of columns on screen }
     rows     = 24;          { number of rows on screen }
     cell     = '*';         { cell symbol }

type
     cell_field = array[1..rows,1..columns] of boolean; { "playing field" }

var
   cells     : cell_field;       { playing field for experiment }
   oldcells  : cell_field;       { holds copy of old cell field }
   cellrow   : string[columns];  { row of cells                 }

{ ---------------------------------------------------- }

procedure init;
{ initializes the cell configuration space (the "playing field") }
var
  i,j : integer;
begin
     for i : = 1 to rows do
        for j : = 1 to columns do
           cells[i,j] : = FALSE;                  { initialize cell space }
end;

{ ---------------------------------------------------- }

procedure display;
{ displays array of cells to screen }
var
   i,j : integer;

begin
```

```
     ClrScr;

     for i : = 1 to rows do
         begin
              for j : = 1 to columns do
                  if cells[i,j] = TRUE then
                     write(cell)          { cell in space }
                  else
                     write(' ');          { no cell in space }
         end
end;

{ ------------------------------------------------------------- }

procedure load;
{ load initial cell file into array of cells }
var
   i,j : integer;
   temp: integer;              { hold random number }
begin
     randomize;                { initialize random number generator }
     for i : = 1 to rows do
       for j : = 1 to columns do
         begin
            temp : = random(2);   { select random number between 1 and 2 }
            if temp = 1 then
                cells[i,j] : = TRUE
            else
                cells[i,j] : = FALSE
         end;
end;

{ ------------------------------------------------------------- }

function count(i,j:integer) : integer;
{ count number of cells in neighborhood }
var
   l,m : integer;          { loop counters }
   sum: integer;           { running sum }
   srow, erow : integer;   { starting and ending row locations }
   scol, ecol : integer;   { starting and ending column locations }

begin
     sum : = 0;            { initialize running sums }

     if i = 1 then         { find starting row }
        srow : = 1
     else
        srow : = i - 1;

     if i = rows then      { find ending row }
        erow : = rows
```

```
      else
         erow : = i + 1;

      if j = 1 then        { find starting column }
         scol : = 1
      else
         scol : = j - 1;

      if j = columns  then  { find ending column }
         ecol : = columns
      else
         ecol : = j + 1;

      for l : = srow to erow do            { count nearby cells }
        for m : = scol to ecol do
          if (i<>l) OR (j<>m) then       { don't count self }
            if oldcells[l,m] = TRUE then
               sum: = sum + 1;
      count : = sum;                       { return sum }
end;

{ ------------------------------------------------------------ }

procedure rule;
{ apply Game of Life rule
  note that the first and last cells in a row are treated specially
}
var
   i,j : integer;                  { local counters }

begin
      for i : = 1 to rows  do
       for j : = 1 to columns do
         oldcells[i,j] : = cells[i,j];    { remember old cell configuration }

      init;                               { initialize next configuration  }

      for i : = 1 to rows  do
           for j : = 1 to columns do
              case count(i,j) of         { count nearby cells }
              0,1 :       cells[i,j] : = FALSE;
              2:          cells[i,j] : = oldcells[i,j];
              3:          cells[i,j] : = TRUE;
              4,5,6,7,8 : cells[i,j] : = FALSE          { overcrowding }
              end
end;

{ ----------------------- begin program ---------------------- }

var
   i : integer;
   iter      : integer;       { number of iterations for simulation}
```

```
begin
     load;
     write('File Loaded');
     display;
     write('Enter number of iterations for simulation ');
     readln(iter);
     write('Press Enter to Begin Simulation ');
     readln;
     for i : = 1 to iter do
     begin
        rule;                { apply rule to cell field }
        display              { display updated universe }
     end
end.
```

```
program Mandel;
{ compute and display Mandelbrot set
  12 - 21 - 93   Phil Laplante                    }

uses
   Complex, Graph;           { include graphics and complex routines}

const
   zoom = 2.0;               { create 2 by 2 window }
   escape = 4.0;             { escape value }

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   i, j      : integer;      { loop variables}
   MaxY      : integer;      { Maximum Y screen coordinate}
   scale     : real;         { scale factor }
   mag       : real;         { square of magnitude of complex number }
   iter      : integer;      { escape iteration counter }
   cx,cy     : real;         { x and y components of c }
   x, y      : real;         { coordinate values in complex plane }

begin

   { initialize graphics }

   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
```

```
        Writeln('Program aborted');
        Halt(1)
    end;
    MaxY : = GetMaxY;          { find maximum X screen coordinate }
    scale: = 2.0*zoom/MaxY;    { calculate zoom factor}

    x : = 0;                   { zet z0 = 0 + 0i }
    y : = 0;

    for i : = 0 to MaxY do     { MaxY is usually smaller than MaxX }
    begin
        for j : = 0 to MaxY do
        begin
            x : = 0;                    { zet z0 = 0 + 0i }
            y : = 0;
            cx: = scale*i - zoom;       { sweep value of c }
            cy: = zoom-  scale*j;
            mag : = 0;                  { initial loop guards }
            iter : = 0;
            begin
                while (iter < 30) and (mag < escape)  do
                begin
                  mult(x,y,x,y,x,y);        { square z }
                  add(x,y,cx,cy,x,y);       { add c }
                  mag : = x*x + y*y;        { calculate square of magnitude }
                  iter : = iter + 1;        { increment counter }
                end;
                if mag < escape then        { output blue for non-escapees}
                   putpixel(i,j, BLUE)
            end  { while loop}
        end  {j loop}
    end{ i loop}
end.
```

```
program Mandel2;
{ compute and display unfilled Mandelbrot set
  12 - 21 - 93   Phil Laplante                    }

uses
   Complex, Graph;          { include graphics and complex routines}

const
   zoom = 2.0;              { create 2 by 2 window }
   escape = 4.0;            { escape value }

var
   GraphDriver : integer;   { Stores graphics driver number}
   GraphMode : integer;     { Stores graphics mode for driver}
   ErrorCode : integer;     { Reports any error condition}
   i, j      : integer;     { loop variables}
```

```
   MaxY      : integer;      { Maximum Y screen coordinate}
   MaxColor  : integer;      { Maximum number of colors}
   scale     : real;         { scale factor }
   mag       : real;         { square of magnitude of complex number }
   iter      : integer;      { escape iteration counter }
   continue  : boolean;      { keep iterating? }
   cx,cy     : real;         { x and y components of c }
   x, y      : real;         { coordinate values in complex plane }

begin

   { initialize graphics }

   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,''); {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxColor : = GetMaxColor; { get maximum number of colors }
   MaxY : = GetMaxY;         { find maximum X screen coordinate }
   scale: = 2.0*zoom/MaxY;   { calculate zoom factor}

   x : = 0;                  { zet z0 = 0 + 0i }
   y : = 0;

   for i : = 0 to MaxY do    { MaxY is usually smaller than MaxX }
   begin
       for j : = 0 to MaxY do
       begin
            x : = 0;                  { zet z0 = 0 + 0i }
            y : = 0;
            cx: = scale*i - zoom;           { sweep value of c }
            cy: = zoom - scale*j;
            mag : = 0;                      { initial loop guards }
            iter : = 0;
            continue : = true;
            begin
                while (iter < MaxColor*2) and (mag < escape)  do
                begin
                  mult(x,y,x,y,x,y);          { square z }
                  add(x,y,cx,cy,x,y);         { add c }
                  mag : = x*x + y*y;          { calculate square of magnitude }
                  iter : = iter + 1;          { increment counter }
                end;
                if mag > escape then          { color escaping points}
                   begin
                     putpixel(i,j, iter div 2);
```

```
                    continue : = FALSE
                  end
            end  { while loop}
       end  {j loop}
   end{ i loop}
end.
```

```
program mazel;
{ compute and display a "maze"
  using Michael Barnsley's IFS algorithm

  12 - 5 - 93  Phil Laplante                  }
uses
   Graph;                    { include graphics package}
var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
   i         : longint;      { loop counters}
   k         : integer;      { row selector }
   MaxY      : integer;      { maximum Y screen coordinate}
   d         : array[1..6,1..6] of real; { holds data of IFS attractor }

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxY : = GetMaxY;          { get screen limits }

{ initialize IFS data array }

   d[1,1]: = 0.33; d[1,2]: = 0; d[1,3]: = 0; d[1,4]: = 0.33; d[1,5]: = 1; d[1,6]: = 1;
   d[2,1]: = 0.33; d[2,2]: = 0; d[2,3]: = 0; d[2,4]: = 0.33; d[2,5]: = MaxY div 2; d[2,6]: = 1;
   d[3,1]: = 0.33; d[3,2]: = 0; d[3,3]: = 0; d[3,4]: = 0.33; d[3,5]: = 1; d[3,6]: = MaxY div 2;
   d[4,1]: = 0.33; d[4,2]: = 0; d[4,3]: = 0; d[4,4]: = 0.33; d[4,5]: = MaxY div 2; d[4,6]: = MaxY;
   d[5,1]: = 0.33; d[5,2]: = 0; d[5,3]: = 0; d[5,4]: = 0.33; d[5,5]: = MaxY; d[5,6]: = MaxY;
   d[6,1]: =0.33; d[6,2]: = 0; d[6,3]: = 0; d[6,4]: = 0.33; d[6,5]: = 1; d[6,6]: = MaxY;

   MaxY : = GetMaxY;

   randomize;           {initialize random number generator}
```

```
   x : = 0;              {set starting coordinates}
   y : = 0;

    for i : = 1 to 320000 do
     begin
        k : = random(8) + 1;                  { pick random row }
        x : = d[k,1]*x + d[k,2]*y + d[k,5];   { transform coordinates }
        y : = d[k,3]*x + d[k,4]*y + d[k,6];
        if i > 10 then                        { skip first 10 iterations }
           putpixel(round(2*x/3),round(2*y/3),DARKGRAY )
    end
end.
```

```
program prey;
{ Simulate Wolf - Caribou populations
  1/29/93   Phil Laplante       }

uses
     Crt;                    { unit for CRT driver }

var
     caribou_0 : real;       { initial caribou population }
     wolf_0    : real;       { initial wolf population  }

     caribou   : real;       { current caribou population }
     wolf      : real;       { current wolf population  }

     caribou_p : real;       { previous caribou population }
     wolf_p    : real;       { previous wolf population  }

     caribou_b : real;       { caribou birth rate }
     wolf_d    : real;       { wolf death rate  }
     K         : real;       { contact-death ratio }

     iter      : integer;    { number of iterations to track }

     i         : integer;

begin
     ClrScr;   { clear screen }
     write(' Enter initial caribou population ');
     readln(caribou_0);
     write(' Enter initial wolf population ');
     readln(wolf_0);
     write(' Enter caribou birth rate ');
     readln(caribou_b);
     write(' Enter wolf death rate ');
     readln(wolf_d);

     write(' Enter contact-death rate ');
```

```
      readln(K);

      write(' Enter number of iterations (months) for simulation ');
      readln(iter);

      caribou_p : = caribou_0;
      wolf_p    : = wolf_0;

      writeln('press enter to begin simulation ');
      readln;

      writeln('           wolves       caribou ');
      for i : = 1 to iter do
         begin
            if i mod 22 = 0 then
            begin
                writeln('press enter to continue ');
                readln;
                ClrScr;
                writeln('           wolves       caribou')
            end;

     { calculate and output current populations, truncate to nearest integer }
        caribou : = caribou_p + caribou_b * caribou_p-  K * caribou_p * wolf_p;
        wolf    : = wolf_p + k * caribou_p * wolf_p - wolf_d * wolf_p;
        writeln('month ',i:4,
        wolf:6:0,'       ',caribou:6:0);

        caribou_p : = caribou;        { reset previous generation counters }
        wolf_p    : = wolf
     end

end.
```

```
program price;
{ compute and display bifurcation diagram for
  mini-economic system given by

  P(t + 1) = at - t^2

  12 - 21 - 93   Phil Laplante                    }

uses
   Graph;                   { include graphics routines}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   i, j      : integer;      { loop variables}
```

```
   MaxX      : integer;      { Maximum X screen coordinate}
   MaxY      : integer;      { Maximum Y screen coordinate}
   t         : real;         { iterated value }
   a         : real;         { constant of iteration  }
   MaxColor  : integer;      { maximum number of colors on graphics card }
   scale     : real;         { plotting scale factor }

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,"");  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxColor : = GetMaxColor;   { find maximum number of colors }
   MaxX : = GetMaxX;           { find maximum X screen coordinate }
   MaxY : = GetMaxY;           { find maximum Y screen coordinate }

   scale : = MaxX/4;           { calculate overall scale factor }

   a : =  2.50;                { set starting point }
   for i : = 1 to MaxX do
    begin
     t : = 0.9;                { calculate orbit about t=1 }
     a : = a + 1.50/(MaxX);    { iterate a }
     for j : = 1 to 100 do     { calculate orbit after 200 iterations}
     begin
       t : = a*t- a*t*t;       { calculate new price }
       if j > 50 then          { skip first 50 iterations }
       begin
         putpixel(i, round(MaxY/2 + t*scale ), GREEN);
       end
     end
    end
end.
```

```
program rabbit;
{ compute and display Douady's Rabbit from the Julia set of

    f(z) = z^2 + -0.122 + 0.745i

  12 - 21 - 92  Phil Laplante                    }

uses
   Complex, Graph;           { include graphics and complex routines}
```

```
const
   zoom = 2.0;                { creates 4 by 4 window }
   attract = 0.0001;          { attractor sensitivity }

var
   GraphDriver : integer;     { Stores graphics driver number}
   GraphMode : integer;       { Stores graphics mode for driver}
   ErrorCode : integer;       { Reports any error condition}
   i, j      : integer;       { loop variables}
   MaxY      : integer;       { Maximum Y screen coordinate}
   scale     : real;          { scale factor }
   mag       : real;          { square of magnitude of complex number }
   iter      : integer;       { escape iteration counter }
   continue  : boolean;       { continue iteration counter }
   x, y      : real;          { real and imaginary parts of z }
   MaxColor  : integer;       { maximum number of colors on graphics card }

begin

   { initialize graphics }

   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,''); {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxColor : = GetMaxColor; { find maximum number of colors }
   MaxY : = GetMaxY;         { find maximum Y screen coordinate }
   scale: = 2.0*zoom/MaxY;   { calculate zoom factor}

   for i : = 0 to MaxY do    { MaxY is usually smaller than MaxX }
   begin
       for j : = 0 to MaxY do
       begin
            x : = scale*i - zoom;            { set starting value of real(z) }
            y : = zoom - scale*j;            { set starting value of imag(z) }
            continue : = TRUE;               { assume point does not escape  }
            iter : =0;
            while continue = TRUE do
            begin
                mult(x,y,x,y,x,y);     { square z }
                add(x,y, - 0.122,0.745,x,y);    { add constant }
                mag : = x*x + y*y;        { calculate square of magnitude }
                if mag < attract then
                  continue : = FALSE       { point is an attractor }
                else
                  if (mag < 100) AND (iter < MaxColor*2) then { keep iterating function }
```

```
                    iter : = iter + 1
                  else                        { point escapes, plot it }
                    begin
                      putpixel(i,j, iter div 2);
                      continue : = FALSE        { get out of loop }
                    end
            end  { while loop}
        end  {j loop}
    end{ i loop}
end.
```

```
program forest_scene;
{ compute and display forest of trees
  using Michael Barnsley's IFS algorithm

  12 - 5 - 93  Frank D'Erasmo                  }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
   i, j      : longint;      { loop counters}
   q         : integer;      { random number }
   k         : integer;      { row selector }
   MaxX      : integer;      { Maximum X screen coordinate}
   MaxY      : integer;      { Maximum Y screen coordinate}
   d         : array[1..8,1..6] of real; { holds data of IFS attractor }
   scale     : real;         { random scale factor}
   xpos,ypos : integer;      { tree position }
   crand     : integer;      { pick random color (green, blue, yellow) }
   color     : integer;      { random color value }

procedure trees;
begin
{ initialize IFS data array }

  d[1,1]: = 0;    d[1,2]: = 0;     d[1,3]: = 0;     d[1,4]: = 0.5;  d[1,5]: = 0; d[1,6]: = 0;
  d[2,1]: =  0.42; d[2,2]: = - 0.42; d[2,3]: = 0.42;  d[2,4]: = 0.42; d[2,5]: = 0; d[2,6]: = 0.2;
  d[3,1]: = 0.42; d[3,2]: = 0.42;  d[3,3]: = - 0.42; d[3,4]: = 0.42; d[3,5]: = 0; d[3,6]: =0.2;
  d[4,1]: = 0.1;  d[4,2]: = 0;     d[4,3]: = 0;     d[4,4]: = 0.1;  d[4,5]: = 0; d[4,6]: = 0.2;

  randomize;           {initialize random number generator}

  x : = 0;             {set starting coordinates}
  y : = 0;

   for j : = 1 to 200 do                       { make 200 trees }
```

```
    begin
      xpos : = random(MaxX);                    { pick tree position }
      ypos : = random(MaxX);
      scale : = random(3) + 1;                  { pick tree scale }
      crand : = random(10) + 1;                 { pick tree color }
      case  crand of
         0,1,2,3,4,5,6,7,8:
              color : = GREEN;                  { most trees are green }
         9  : color : = YELLOW;                 { some trees are yellow }
         10 : color : = RED;                    { some trees die }
      end; {case}

      for i : = 1 to 800 do              { 800 pixels per tree }
       begin
         q : = random(100) + 1;          { pick random number from 1 - 100}
         if q < = 40 then                { assign column according to }
            k : = 2;                     { probability }
         if (q > 40) AND (q < 81) then
            k : = 3;
         if (q > = 81) AND (q < 95) then
            k : = 4;
         if (q > = 95 ) then
           k : = 1;

        x : = d[k,1]*x + d[k,2]*y + d[k,5];   { transform coordinates }
        y : = d[k,3]*x + d[k,4]*y + d[k,6];
        if (ypos- y*MaxY/scale > MaxY - 50)  then {clears area on screen}
           putpixel(xpos + round(x*MaxX/scale),round(ypos - y*MaxX/scale),color)
       end;
    end;
end;

procedure redwds;     {generates redwood trees}

begin
{ initialize IFS data array }

   d[1,1]: = 0.33; d[1,2]: = 0; d[1,3]: = 0; d[1,4]: = 0.33; d[1,5]: = 1; d[1,6]: = 0;
   d[2,1]: = 0.33; d[2,2]: = 0; d[2,3]: = 0; d[2,4]: = 0.33; d[2,5]: = MaxX; d[2,6]: = 0;
   d[3,1]: = 0.33; d[3,2]: = 0; d[3,3]: = 0; d[3,4]: = 0.33; d[3,5]: =1; d[3,6]: = MaxX;
   d[4,1]: = 0.33; d[4,2]: = 0; d[4,3]: = 0; d[4,4]: = 0.33; d[4,5]: = MaxX; d[4,6]: = MaxX;
   d[5,1]: = 0.33; d[5,2]: = 0; d[5,3]: =2; d[5,4]: = 0.33; d[5,5]: = MaxX div 2; d[5,6]: = 1;
   d[6,1]: = 0.33; d[6,2]: = 0; d[6,3]: =2; d[6,4]: = 0.33; d[6,5]: = MaxX; d[6,6]: = MaxX div 2;
   d[7,1]: = 0.33; d[7,2]: = 0; d[7,3]: =2; d[7,4]: = 0.33; d[7,5]: = 1; d[7,6]: = MaxX div 2;
   d[8,1]: = 0.33; d[8,2]: = 0; d[8,3]: =2; d[8,4]: = 0.33; d[8,5]: = MaxX div 2; d[8,6]: = MaxX;

   randomize;           {initialize random number generator}

   x : = 0;             {set starting coordinates}
   y : = 0;

   for i : = 1 to 300000 do                 {300,000 pixels}
```

```
    begin
       k : = random(8) + 1;                { pick random column }
       x : = d[k,1]*x + d[k,2]*y + d[k,5];   { transform coordinates }
       y : = d[k,3]*x + d[k,4]*y + d[k,6];
       putpixel(round(2*x/3),round(2*y/3),RED);
       putpixel(round(2*(x + 1)/3),round(2*(y + 1)/2),RED)
    end;
end;

procedure clouds;     {generates mist/clouds}
begin

{ initialize IFS data array }

   d[1,1]: = 0.33; d[1,2]: = 1; d[1,3]: = 0; d[1,4]: = 0.33; d[1,5]: = 0; d[1,6]: = 0;
   d[2,1]: = 0.33; d[2,2]: = 1; d[2,3]: = 0; d[2,4]: = 0.33; d[2,5]: = MaxY; d[2,6]: = 0;
   d[3,1]: = 0.33; d[3,2]: = 1; d[3,3]: = 0; d[3,4]: = 0.33; d[3,5]: = 1; d[3,6]: = MaxY;
   d[4,1]: = 0.33; d[4,2]: = 1; d[4,3]: = 0; d[4,4]: = 0.33; d[4,5]: = MaxY; d[4,6]: = MaxY;
   d[5,1]: = 0.33; d[5,2]: = 0; d[5,3]: = 0; d[5,4]: = 0.33; d[5,5]: = MaxY div 2; d[5,6]: =1;
   d[6,1]: = 0.33; d[6,2]: = 0; d[6,3]: = 0; d[6,4]: = 0.33; d[6,5]: = MaxY; d[6,6]: = MaxY div 2;
   d[7,1]: = 0.33; d[7,2]: = 0; d[7,3]: = 0; d[7,4]: = 0.33; d[7,5]: = 1; d[7,6]: = MaxY div 2;
   d[8,1]: = 0.33; d[8,2]: = 0; d[8,3]: = 0; d[8,4]: = 0.33; d[8,5]: = MaxY div 2; d[8,6]: = MaxY;

   randomize;           {initialize random number generator}

   x : = 1;             {set starting coordinates}
   y : = 50;

   for i : = 1 to 30000 do                  { 30,000 pixels }
     begin
       k : = random(8) + 1;                 { pick random column }
       x : = d[k,1]*x + d[k,2]*y + d[k,5];    { transform coordinates }
       y : = d[k,3]*x + d[k,4]*y + d[k,6];
       if y < 200  then
          putpixel(round(2*x/3),round(2*y/3),LIGHTGRAY)
     end;
end;

begin  {main}
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,''); {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxY: = GetMaxY;
   MaxX: = GetMaxX;
   trees;
```

```
   redwds;
   clouds
end.
```

```
program rocks;
{ compute and display "clouds" fractal
  using Michael Barnsley's IFS algorithm

  12 - 5 - 93  Frank D'Erasmo }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
   i         : longint;      { loop counters}
   k         : integer;      { row selector }
   MaxY      : integer;      { Maximum X screen coordinate}
   d         : array[1..4,1..6] of real; { holds data of IFS attractor }

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,''); {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;

   MaxY : = GetMaxY;

{ initialize IFS data array }

   d[1,1]: = 0.5;  d[1,2]: = 0;  d[1,3]: = 0;  d[1,4]: = 0.5;  d[1,5]: = 0; d[1,6]: = 0;
   d[2,1]: = 0.5;  d[2,2]: = 0;  d[2,3]: = 0;  d[2,4]: = 0.5;  d[2,5]: = 2; d[2,6]: = 0;
   d[3,1]: = - 0.4;  d[3,2]: = 0;  d[3,3]: = 1;  d[3,4]: = 0.4;  d[3,5]: = 0; d[3,6]: = 1;
   d[4,1]: = - 0.5;  d[4,2]: = 0;  d[4,3]: = 0;  d[4,4]: = 0.5;  d[4,5]: = 2; d[4,6]: = 1;

   randomize;           {initialize random number generator}

   x : = 0;             {set starting coordinates}
   y : = 0;

   for i : = 1 to 320000 do
     begin
```

```
        k : = random(4) + 1;                    { pick random number from 1 - 4}
        x : = d[k,1]*x + d[k,2]*y + d[k,5];     { transform coordinates }
        y : = d[k,3]*x + d[k,4]*y + d[k,6];
        if i > 10 then                          { skip first 10 iterations }
           putpixel(round(MaxY*x/2),MaxY - round(MaxY*y/2),BROWN)
     end                                  { scale for screen }
end.
```

```
program seals;
{ compute and display seals or dolphins
  using Michael Barnsley's IFS algorithm

  12 - 5 - 93     Frank D'Erasmo             }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
   i         : integer;      { loop counters}
   k         : integer;      { row selector }
   MaxY      : integer;      { Maximum Y screen coordinate}
   d         : array[1..4,1..6] of real; { holds data of IFS attractor }

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;

   MaxY : = GetMaxY;

{ initialize IFS data array }

   d[1,1]: = - 0.5;  d[1,2]: = 0;  d[1,3]: = 0;  d[1,4]: = 0.5;  d[1,5]: = 0; d[1,6]: = 0;
   d[2,1]: = - 0.5;  d[2,2]: = 0;  d[2,3]: = 0;  d[2,4]: = 0.5;  d[2,5]: = 2; d[2,6]: = 0;
   d[3,1]: = - 0.4;  d[3,2]: = 0;  d[3,3]: = 1;  d[3,4]: = 0.4;  d[3,5]: = 0; d[3,6]: = 1;
   d[4,1]: = - 0.5;  d[4,2]: = 0;  d[4,3]: = 0;  d[4,4]: = 0.5;  d[4,5]: = 2; d[4,6]: = 1;

   randomize;           {initialize random number generator}

   x : = 0;             {set starting coordinates}
```

```
   y : = 0;

   for i : = 1 to 32000 do
     begin
        k : = random(4) + 1;                  { pick random number from 1 - 4}
        x : = d[k,1]*x + d[k,2]*y + d[k,5];   { transform coordinates }
        y : = d[k,3]*x + d[k,4]*y + d[k,6];
        if i > 10 then                        { skip first 10 iterations }
             putpixel(round(MaxY*x/2),MaxY - round(MaxY*y/2),WHITE)
     end;                                  { scale for screen }
end.
```

```
program seaweed;
{ compute and display seaweed
  using Michael Barnsley's IFS algorithm

  12 - 5 - 93  Frank D'Erasmo                 }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
   i         : longint;      { loop counters}
   k         : integer;      { row selector }
   MaxY      : integer;      { Maximum Y screen coordinate}
   d         : array[1..4,1..6] of real; { holds data of IFS attractor }

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode  grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;

   MaxY : = GetMaxY;

{ initialize IFS data array }

   d[1,1]: = 0.5;  d[1,2]: = 0;  d[1,3]: = 0;  d[1,4]: = 0.5;  d[1,5]: = 0; d[1,6]: = 0;
   d[2,1]: = 0.5;  d[2,2]: = 0;  d[2,3]: = 0;  d[2,4]: = 0.5;  d[2,5]: = 2; d[2,6]: = 0;
   d[3,1]: = 0.4;  d[3,2]: = 0;  d[3,3]: = 1;  d[3,4]: = 0.4;  d[3,5]: = 0; d[3,6]: = 1;
   d[4,1]: = 0.5;  d[4,2]: = 0;  d[4,3]: = 0;  d[4,4]: = 0.5;  d[4,5]: = 2; d[4,6]: = 1;
```

```
   randomize;           {initialize random number generator}

   x : = 0;             {set starting coordinates}
   y : = 0;

   for i : = 1 to 320000 do                    {320,000 pixels}
     begin
        k : = random(4) + 1;                  {pick random number from 1-4}
        x : = d[k,1]*x + d[k,2]*y + d[k,5]; { transform coordinates }
        y : = d[k,3]*x + d[k,4]*y + d[k,6];
        if i > 10 then                         { skip first 10 iterations }
             putpixel(round(MaxY*x/2),MaxY - round(MaxY*y/2),GREEN)
          end;                                  { scale for screen }
end.
```

```
program siegel;
{ compute and display a Siegel disk - the Julia set of
 f(z) = z^2 - 0.39054 - 0.58679i

  1 - 2 - 92  Phil Laplante                      }
uses
   Complex, Graph;          { include graphics and complex routines}

const
   zoom = 2.0;              { create 4 by 4 window }
   attract = 0.0001;        { attractor sensitivity }

var
   GraphDriver : integer;   { Stores graphics driver number}
   GraphMode : integer;     { Stores graphics mode for driver}
   ErrorCode : integer;     { Reports any error condition}
   i, j      : integer;     { loop variables}
   MaxY      : integer;     { Maximum Y screen coordinate}
   scale     : real;        { scale factor }
   mag       : real;        { square of magnitude of complex number }
   iter      : integer;     { escape iteration counter }
   continue  : boolean;     { continue iteration counter }
   x, y      : real;        { real and imaginary parts of z }
   MaxColor  : integer;     { maximum number of colors on graphics card }

begin

   { initialize graphics }

   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
```

```
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
    end;
    MaxColor : = GetMaxColor; { find maximum number of colors }
    MaxY : = GetMaxY;         { find maximum Y screen coordinate }
    scale: = 2.0*zoom/MaxY;   { calculate zoom factor}

    for i : = 0 to MaxY do    { MaxY is usually smaller than MaxY }
    begin
        for j : = 0 to MaxY do
        begin
            x : = scale*i - zoom;            { set starting value of real(z) }
            y : = zoom - scale*j;            { set starting value of imag(z) }
            continue : = TRUE;               { assume point does not escape  }
            iter : = 0;
            while continue = TRUE do
            begin
                mult(x,y,x,y,x,y);        { square z }
                add(x,y, - 0.390540, - 0.58679,x,y); { add constant }
                mag : = x*x + y*y;        { calculate square of magnitude }
                if mag < attract then
                  continue : = FALSE      { point is an attractor }
                else
                  if (mag < 100) AND (iter < MaxColor*2) then { keep iterating function }
                     iter : = iter + 1
                  else                    { point escapes, plot it }
                     begin
                       putpixel(i,j, iter div 2);
                       continue : = FALSE       { get out of loop }
                     end
            end  { while loop}
        end  {j loop}
    end{ i loop}
end.
```

```
program sierp;
{ compute and display Sierpinski triangle via random orbits
  12 - 5 - 93  Phil Laplante                    }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : integer;      { pixel coordinates }
   triangle  : integer;      { select random triangle}
   i, j, k   : integer;      { loop counters}
```

```
   MaxY      : integer;      { maximum X and Y coordinates}

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,''); {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;

   MaxY : = GetMaxY;

   randomize;        {initialize random number generator}

   x : = random(MaxY);  {select random starting point}
   y : = random(MaxY);  {use MaxX=MaxY }

   for i : = 1 to  20000 do
       if i > 1000 then           { skip first few points }
       begin
        triangle : = random(3) + 1;  { select random number between 1 and 3}
         case triangle of         { select which triangle to measure from }
         1:  begin
                   x : = x div 2;            { find 1/2 way point to  A}
                   y : = (MaxY + y) div 2
             end;
         2:  begin
                  x : = (MaxY div 2 + x) div 2; { find 1/2 way point to B }
                  y : = y div 2
             end;
         3:  begin
                  x : = (MaxY + x) div 2;       { find 1/2 way point to C}
                  y : = (MaxY + y) div 2
             end
         end;
            putpixel(x,y,WHITE)                { output pixel }
    end  { i loop}
end.
```

```
program sierp2;
{ compute and display Sierpinski triangle
  using Michael Barnsley's IFS algorithm

  12 - 5 - 93  Phil Laplante                  }
uses
   Graph;                    {include graphics package}
```

```
var
   GraphDriver : integer;     { Stores graphics driver number}
   GraphMode : integer;       { Stores graphics mode for driver}
   ErrorCode : integer;       { Reports any error condition}
   x, y      : real;          { pixel coordinates }
   i         : integer;       { loop counter }
   MaxY      : integer;       { maximum X and Y coordinates}
   k         : integer;       { select random row }
   d         : array[1..3,1..6] of real; { holds data of IFS attractor }

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,''); {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;

{ initialize IFS data array }

   d[1,1]: = 0.5; d[1,2]: = 0; d[1,3]: = 0; d[1,4]: = 0.5; d[1,5]: = 25; d[1,6]: = 1;
   d[2,1]: = 0.5; d[2,2]: = 0; d[2,3]: = 0; d[2,4]: = 0.5; d[2,5]: =  1; d[2,6]: = 50;
   d[3,1]: = 0.5; d[3,2]: = 0; d[3,3]: = 0; d[3,4]: = 0.5; d[3,5]: = 50; d[3,6]: = 50;

   MaxY : = GetMaxY;

   randomize;           {initialize random number generator}

   x : = 0.0;  {set starting coordinates}
   y : = 0.0;

   for i : = 1 to 30000 do
     begin
        k : = random(3) + 1;                { pick random row }
        x : = d[k,1]*x + d[k,2]*y + d[k,5];   { transform coordinates }
        y : = d[k,3]*x + d[k,4]*y + d[k,6];
        if i > 10 then                      { skip first 10 iterations }
           putpixel(round(x*MaxY/100),round(y*MaxY/100),WHITE) { convert to screen}
     end
end.
```

```
program snow;
{ compute and display "snow" from the Julia set of

 f(z) = z^2 0.11031 - 0.67037i
```

```
  1 - 2 - 92  Phil Laplante                    }
uses
   Complex, Graph;           { include graphics and complex routines}

const
   zoom = 1.5;               { create 3 by 3 window }
   attract = 0.0001;         { attractor sensitivity }

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   i, j      : integer;      { loop variables}
   MaxY      : integer;      { Maximum Y screen coordinate}
   scale     : real;         { scale factor }
   mag       : real;         { square of magnitude of complex number }
   iter      : integer;      { escape iteration counter }
   continue  : boolean;      { continue iteration counter }
   x, y      : real;         { real and imaginary parts of z }
   MaxColor  : integer;      { maximum number of colors on graphics card }

begin

   { initialize graphics }

   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;
   MaxColor : = GetMaxColor; { find maximum number of colors }
   MaxY : = GetMaxY;         { find maximum Y screen coordinate }
   scale: = 2.0*zoom/MaxY;   { calculate zoom factor}

   for i : = 0 to MaxY do    { MaxY is usually smaller than MaxY }
   begin
       for j : = 0 to MaxY do
       begin
            x : = scale*i - zoom;           { set starting value of real(z) }
            y : = zoom - scale*j;           { set starting value of imag(z) }
            continue : = TRUE;              { assume point does not escape  }
            iter : = 0;
            while continue = TRUE do
            begin
                mult(x,y,x,y,x,y);        { square z }
                add(x,y,0.11031, - 0.67037,x,y); { add constant }
                mag : = x*x + y*y;          { calculate square of magnitude }
                if mag < attract then
```

```
                continue : = FALSE        { point is an attractor }
              else
                if (mag < 100) AND (iter < MaxColor*2) then { keep iterating function }
                   iter : = iter + 1
                else                      { point escapes, plot it }
                   begin
                     if(iter div 2 = WHITE) then {output white only }
                       putpixel(i,j, WHITE);
                     continue : = FALSE        { get out of loop }
                   end
           end  { while loop}
       end  {j loop}
   end{ i loop}
end.
```

```
program swamp;
{ compute and display a swamp
  using Michael Barnsley's IFS algorithm

  12 - 5 - 93  Phil Laplante                  }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
   i, j      : integer;      { loop counters}
   q         : integer;      { random number }
   k         : integer;      { row selector }
   MaxX      : integer;      { Maximum X screen coordinate}
   d         : array[1..4,1..6] of real; { holds data of IFS attractor }
   scale     : real;         { random scale factor}
   xpos,ypos : integer;      { plant position }
   crand     : integer;      { pick random color (green, yellow, lightgreen)}
   color     : integer;      { random color value }

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,'');  {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;

   MaxX : = GetMaxX;
```

```
{ initialize IFS data array }

   d[1,1]: = 0.5; d[1,2]: = 0; d[1,3]: = 0; d[1,4]: = 0.25; d[1,5]: = 1; d[1,6]: = 1;
   d[2,1]: = 0.25; d[2,2]: = 0; d[2,3]: = 0; d[2,4]: = 0.7; d[2,5]: = 50; d[2,6]: = 1;
   d[3,1]: = 0.25; d[3,2]: = 0; d[3,3]: = 0; d[3,4]: = 0.7; d[3,5]: = 1; d[3,6]: = 50;
   d[4,1]: = 0.5; d[4,2]: = 0; d[4,3]: = 0; d[4,4]: = 0.25; d[4,5]: = 50; d[4,6]: = 50;

   randomize;           {initialize random number generator}

   x : = 0;             {set starting coordinates}
   y : = 0;

   for j : = 1 to 70 do                          { make 70 plants }
     begin
     xpos : = random(MaxX);                      { pick plant position }
     ypos : = random(MaxX);
     scale : = random(3) + 1;                    { pick plant scale }
     crand : = random(10) + 1;                   { pick plant color }
     case  crand of
        0,1,2,3,4,5,6,7,8:
             color : = GREEN;                    { most plants are green }
        9  : color : = YELLOW;                   { some plants are yellow }
        10 : color : = LIGHTGREEN;               { some plants are lightgreen}
     end;

     for i : = 1 to 3000 do
     begin
        k : = random(4) + 1;                    { pick random row }

        x : = d[k,1]*x + d[k,2]*y + d[k,5];     { transform coordinates }
        y : = d[k,3]*x + d[k,4]*y + d[k,6];

        if i > 10 then                          { skip first 10 iterations }
             putpixel(xpos + round(x*scale),round(ypos-  y*scale),color)
     end
   end
end.

-----------------------------------------

program tree;
{ compute and display a tree
  using Michael Barnsley's IFS algorithm

  12 - 5 - 93  Phil Laplante                    }
uses
   Graph;                    {include graphics package}

var
   GraphDriver : integer;    { Stores graphics driver number}
   GraphMode : integer;      { Stores graphics mode for driver}
   ErrorCode : integer;      { Reports any error condition}
   x, y      : real;         { pixel coordinates }
```

```
   i         : integer;      { loop counters}
   q         : integer;      { random number }
   k         : integer;      { row selector }
   MaxY      : integer;      { Maximum Y screen coordinate}
   d         : array[1..4,1..6] of real; { holds data of IFS attractor }

begin
   GraphDriver : = Detect;    {try to detect graphics card}
   InitGraph(GraphDriver,GraphMode,''); {initialize graphics}
   ErrorCode : = GraphResult;
   if ErrorCode <> grOk then    {check for error}
   begin
        Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
        Writeln('Graphics card not found');
        Writeln('Program aborted');
        Halt(1)
   end;

   MaxY : = GetMaxY;

{ initialize IFS data array }

   d[1,1]: = 0;     d[1,2]: = 0;     d[1,3]: = 0;     d[1,4]: = 0.5;  d[1,5]: = 0; d[1,6]: = 0;
   d[2,1]: = 0.42;  d[2,2]: = - 0.42; d[2,3]: = 0.42;  d[2,4]: = 0.42; d[2,5]: = 0; d[2,6]: = 0.2;
   d[3,1]: = 0.42;  d[3,2]: = 0.42;  d[3,3]: = - 0.42; d[3,4]: = 0.42; d[3,5]: = 0; d[3,6]: = 0.2;
   d[4,1]: = 0.1;   d[4,2]: = 0;     d[4,3]: = 0;     d[4,4]: = 0.1;  d[4,5]: = 0; d[4,6]: = 0.2;

   randomize;           {initialize random number generator}

   x : = 0;             {set starting coordinates}
   y : = 0;

   for i : = 1 to 30000 do
     begin
        q : = random(100) + 1;              { pick random number from 1 - 100}
        if q <= 40 then                     { assign row according to }
           k : = 2;                         { probability }
        if (q > 40) AND (q < 81) then
           k : = 3;
        if (q > = 81) AND (q < 95) then
           k : = 4;
        if (q > = 95 ) then
           k : = 1;

        x : = d[k,1]*x + d[k,2]*y + d[k,5];   { transform coordinates }
        y : = d[k,3]*x + d[k,4]*y + d[k,6];
        if i > 10 then                      { skip first 10 iterations }
             putpixel(round(MaxY/2 + 3*MaxY*x),round(MaxY-  3*MaxY*y),GREEN)
           end                              { scale for screen }
end.
```

Glossary

affine transformations Mathematical operations involving sliding, stretching, and rotating.

algorithm A recipe or set of rules that describes some process.

aspect ration In computer screens, the ratio of the length of the x-coordinate range to the y-coordinate range.

attractor The point to which an iterated function tends toward if it doesn't escape and is not indifferent.

attractor sensitivity The threshold to which a function $f(z)$ is iterated at the point z_0. If the square of its modulus at any point is less than the attractor sensitivity, then the point attracts.

basin of attraction The set of all points that, when iterated by a function f, attract to the same point.

bifurcation diagram A diagram of an iterated function against the value of a swept constant. In many cases, this generates a fractal that tends to have two zones of activity.

binary variables and constants Variables and constants that can only take on the values 0 or 1.

Boolean AND operation A logical operation on binary variables and constants that produces a one only if both operands are one. It's usually denoted as •.

Boolean complement A logical operation on a binary variable or constant that produces a one if the operand is a zero, and vice versa. It's usually denoted by a bar over the operand.

Boolean OR operation A logical operation on binary variables and constants that produces a one if one or both operands are one. It's usually denoted as +.

Cantor middle third argument A recursive mathematical procedure involving the removal of the middle third of a line segment.

Cantor set The result of applying the Cantor middle third.

cellular automata A type of dynamical system involving matrices or cells.

chaos A state of disorder.

chaotic system Those that, when they are in equilibrium, are in unstable equilibrium.

complex conjugate If $z = a + bi$ is a complex number, then its complex conjugate, denoted $\overline{z}$, is $z = a - bi$.

complex number A number that has both real and imaginary parts. For example, in the complex number $3 + 4i$, 3 is the real part and 4 is the imaginary part.

complex plane A map where complex numbers are plotted. It's similar to the Cartesian plane, except that the y-axis is labelled as "iy."

complex variables Placeholders for complex variables. Usually denoted by some variant of the letter z.

continuous simulation A model involving differential equations.

compression ratio The ratio of the bytes required to store an uncompressed image to those needed to store the compressed equivalent.

discrete simulation A computer model using finite difference equations.

differential equations An equation involving a function and its derivatives.

dynamical systems A subfield of mathematics that's concerned with the repeated application of an algorithm.

escape When the result of iterating a function at a point tends toward infinity or minus infinity.

Fibonacci sequence A sequence of numbers that begins with 0, 1, then proceeds by adding the preceding two numbers in a sequence to get the next.

filled Mandelbrot set A Mandelbrot set in which colors aren't used.

finite difference equation A recursive equation that describes a function at time t in terms of the function at previous time samples, $t - 1$, $t - 2$, and so on.

fractal An image with an infinite amount of self-similarity.

fractal dimension Fractional dimension of geometric images. Defined as the logarithm of the number of self-similar pieces divided by the logarithm of the magnification needed to obtain them.

function In mathematics, a mapping or rule.

function composition The process of applying a function to the result of another function or itself.

function iteration Repeated composition of a function.

hyperbolic cosine A cosine function defined on real numbers. In particular, if x is a real number, then:

$$\cosh(x) = \frac{e^x + e^{-x}}{2}$$

hyperbolic sine A sine function defined on real numbers. In particular, if x is a real number, then:

$$sinh(x) = \frac{e^x - e^{-x}}{2}$$

image compression The process of reducing the amount of stored information needed to reproduce an image.

imaginary part In a complex number, the component that consists of a real number (a number found on the number line) times the positive square root of –1, denoted i.

indifferent A point that, under iteration, acts as neither an attractor nor a repelling point.

inverse For a real function f, if its inverse is f^{-1}, then $f^{-1}(f(x)) = x$.

iteration Repeated composition of a function or procedure.

iterated function systems A way to generate fractals by the repeated application of special geometric procedures. Abbreviated as IFS.

Julia set A complex function, $f(z)$, is the boundary of the set of points that escape.

logistics equation An equation first proposed as a model for population growth. It's given by:

$$P(t + 1) = aP(t) - aP(t)^2$$

In this equation, $aP(t)$ is the previous year's population plus newborns, and $aP(t)^2$ is the death rate. Plotting the values of $P(t)$ over many years and over many values of a yields a bifurcation diagram.

matrix A mathematical construct that consists of rows and columns that hold numbers.

Mandelbrot set The set of complex constants c_i for which the orbits of the function $f(z) = g(z) + c_i$ evaluated at the initial condition of $z_0 = 0$, don't escape. The Mandelbrot set is usually found for the function $g(z) = z^2$.

modulus The modulus of a complex number, z, is equal to the square root of the sum of the squares of its real and imaginary parts.

noncommutative algebra An algebraic system where the commutative laws don't hold. For example, $x \cdot y = y \cdot x$ does not hold for quaternions.

one-dimensional cellular automaton A cellular automaton in which you trace the evolution of the system by observing a row of cells at time t followed by the row at time $t + 1$, and so on.

orbits See function iteration.

pixels Screen picture elements capable of displaying one or more colors.

quaternions Hyper-complex numbers (complex number pair) used in the generation of three-dimensional fractals.

random orbits Attracting points determined by iteration of random starting points by an appropriate geometric procedure.

real number Any number that can be found on the real number line.

real part In a complex number, the component that consists of a real number, that is, a number that can be found on the number line.

recursive Mathematical, graphical, or geometric procedures that are self-referential.

repelling point A point that escapes.

resolution On a computer screen, the density of the pixels.

scene analysis The process of extracting specific features from a larger picture or scene.

self-similar In an image, when the structure of the whole is reflected in every part.

sensitive dependence On initial conditions, a system that's subject to great variance in later states due to only slight variance in the initial conditions.

Sierpinski gasket A fractal created by repeatedly dividing a square into nine equal-sized squares and removing the middle one. Also known as Sierpinski carpet.

Sierpinski triangle A fractal generated by repeatedly dividing a triangle into four self-similar ones and removing the inner fourth one.

stable equilibrium A system that can't easily be moved to a chaotic state.

strange attractor When the attracting set of an iterated function or procedures is a fractal.

turbulence A chaotic system condition characterized by disorder on all scales, with backward eddy currents and circular waves.

two-dimensional cellular automaton A cellular automaton in which a cell's contents at time t is based on its own contents and the contents of all its immediate neighbors at time $t - 1$.

unstable equilibrium A system that can easily move into a chaotic state.

Bibliography

Barnsley, M. 1988. *Fractals everywhere.* New York: Academic Press, Inc.

DeAngelis, Tori. January, 1993. "Chaos, chaos everywhere is what the theorists think." *The american psychological association monitor* 24 (1): 1, 41.

Devaney, R, and L. Keen, eds. 1989. *Chaos and fractals: The mathematics behind the computer graphics.* Providence, R.I.: American Mathematical Society.

Devaney, R. L. 1989. *An introduction to chaotic dynamical systems.* New York: Addison-Wesley.

_____. 1990. *Chaos, fractals, and dynamics.* New York: Addison-Wesley.

Devaney, R. 1992. *A first course in chaotic dynamical systems, theory and experiment.* New York: Addison Wesley.

Edgar, G. A. 1990. *Measure topology and fractal geometry.* New York: Springer-Verlag, Inc.

Escher on Escher: Exploring the infinite. 1989. New York: Harry N. Abrams, Inc.

Falconer, K. 1990. *Fractal geometry: Mathematical foundations and applications.* New York: Wiley.

Feder, J. 1988. *Fractals.* New York: Plenum Press.

Gleick, J. 1987. *Chaos: making a new science.* New York: Penguin.

Gutowitz, H., editor. 1991. *Cellular automata: theory and expeniment.* London.

Hofstadter, Douglas R. 1989. *Gödel, Escher, Bach: An eternal golden braid.* New York: Vintage Books.

Levy, S. 1992. *Artificial life : the quest for a new creation.* New York: Pantheon Books.

H. B. Lin, ed. 1984. *Chaos, world scientific.* Singapore.

Mandlebrot, B. *The fractal geometry of nature.* New York: W. H. Freeman and Co.

McGuire, Michael. 1991. *An eye for fractals.* New York: Addison-Wesley.

Moon, Francis C. 1992 *Chaotic and fractal dynamics*, New York: John Wiley & Sons.

Peitgen, H. O. and P. H. Richter. 1986. *The beauty of fractals.* New York: Springer-Verlag Inc.

Peitgen, H. and D. Saupe, eds. 1988. *The science of fractal images.* New York: Springer-Verlag Inc.

Peitgen, H., H. Jurgens and D. Saupe. 1992. *Fractals for the classroom.* New York: Springer-Verlag.

Peters, Edgar. *Chaos and order in the capital markets.*

Pickover, C. 1990. *Computers, pattern, chaos, and beauty: Graphics from an unseen world.* New York: St. Martin's Press.

Pritchard, J. 1992. *The chaos cookbook: A practical programming guide.* Oxford: Butterworth-Heinemann.

Prusinkiewicz, P. and A. Lindenmayer. 1990. *The algorithmic beauty of plants.* New York: Springer-Verlag.

Schroeder, M. 1991. *Fractals, chaos, and power laws: Minutes from an infinite paradise.* New York: W. H. Freeman.

Stein, D., ed. 1988. *Proceedings of the Santa Fe Institute's Complex Systems Summer School.* Redwood City, Ca.: Addison-Wesley.

Stewart, I. 1990. *Does God play dice?: The mathematics of chaos.* New York: B. Blackwell.

Wegner, T. and M. Peterson. 1991. *Fractal creations.* Corte Mader, Ca.: The Waite Group.

West, Bruce J. and Ary L. Goldberger. July-August, 1987. "Physiology in fractal dimensions." *American Scientist* 75: 354–365.

Wolfram, S., ed. 1986. *Theory and applications of cellular automata.* Singapore: World Scientific Publishing Co.

References

Chapter 1

[1]For example see the "Back to the Future" movies.

[2]For example, in George Herbert's *Jacula Prudentum*, referring to the tragic Richard III, "For want of a nail a shoe is lost, for want of a shoe a horse is lost, for want of a horse the rider is lost, [for want of a king, England was lost]." In *Poor Richard's Almanac* Ben Franklin prefaced the quote with, "A little neglect may breed great mischief."

[3]There might be musical fractals. For example, many canons and fugues have a recursive self-similarity. In addition, a school of musical thought called *minimalism* tends to produce sounds that are inherently self-similar.

[4]From "On Poetry.A Rhapsody" (1733).

[5]There is a paradox here, however. If you rigorously set this up as an equation with limits and solve it for an infinite number of iterations, the answer is that you do reach the wall. This is counter-intuitive and contradictory. In fact, this is well known by mathematicians as Zeno's "Achilles Paradox." But so as not to spoil the fun, let's assume that you can't reach the wall this way. Who has the time to do it infinitely anyway?

[6]Let's assume that y is a positive integer.

Chapter 2

[1]Engineers denote the positive square root of -1 as j.

[2]The term $a_2 - b_2 i$ is called the *complex conjugate* of $a_2 + b_2 i$

[3]If you are unfamiliar with trigonometry, you may wish to skip this section.

[4]Notice that these functions are defined in terms of the special constant e, which is roughly equal to 2.718. e has a special importance to mathematicians, scientists, and engineers, which is similar to that of π and i.

[5]Note that $e^{i\pi} = 1$, thus harmoniously uniting four important constants.

[6]These numbers are entirely arbitrary.

[7]Orbit is another term for iterated function values.

[8]Recall that for a real function f, with inverse f^{-1}, then $f^{-1}(f(x))=x$.

Chapter 3

[1]A finite difference equation is a recursive equation that describes a function at time t in terms of the function at previous time samples, $t-1$, $t-2$, and so on.

[2]The same mechanism has been investigated as a model for wiring local phone systems.

[3]An electrocardiogram (ECG or EKG) measures electrical activity in the heart.

Chapter 4

[1]Turbulent flow is found in many natural settings. For example, waterfalls and crashing waves are clearly turbulent, but here, I'll discuss turbulent flow in the context of human-made situations.

[2]Boolean operations are intended to be applied to binary variables and constants. *Binary variables and binary constants* can only take on the values 0 or 1.

[3]Do not use non-ASCII editors such as WordPerfect

Index

*__Boldface__ page numbers refer to art

D

E

F

*Boldface page numbers refer to art

G

H

I

J

*__Boldface__ page numbers refer to art

*__Boldface__ page numbers refer to art

Q

R

S

*__Boldface__ page numbers refer to art

T

U

V

W

*__Boldface__ page numbers refer to art