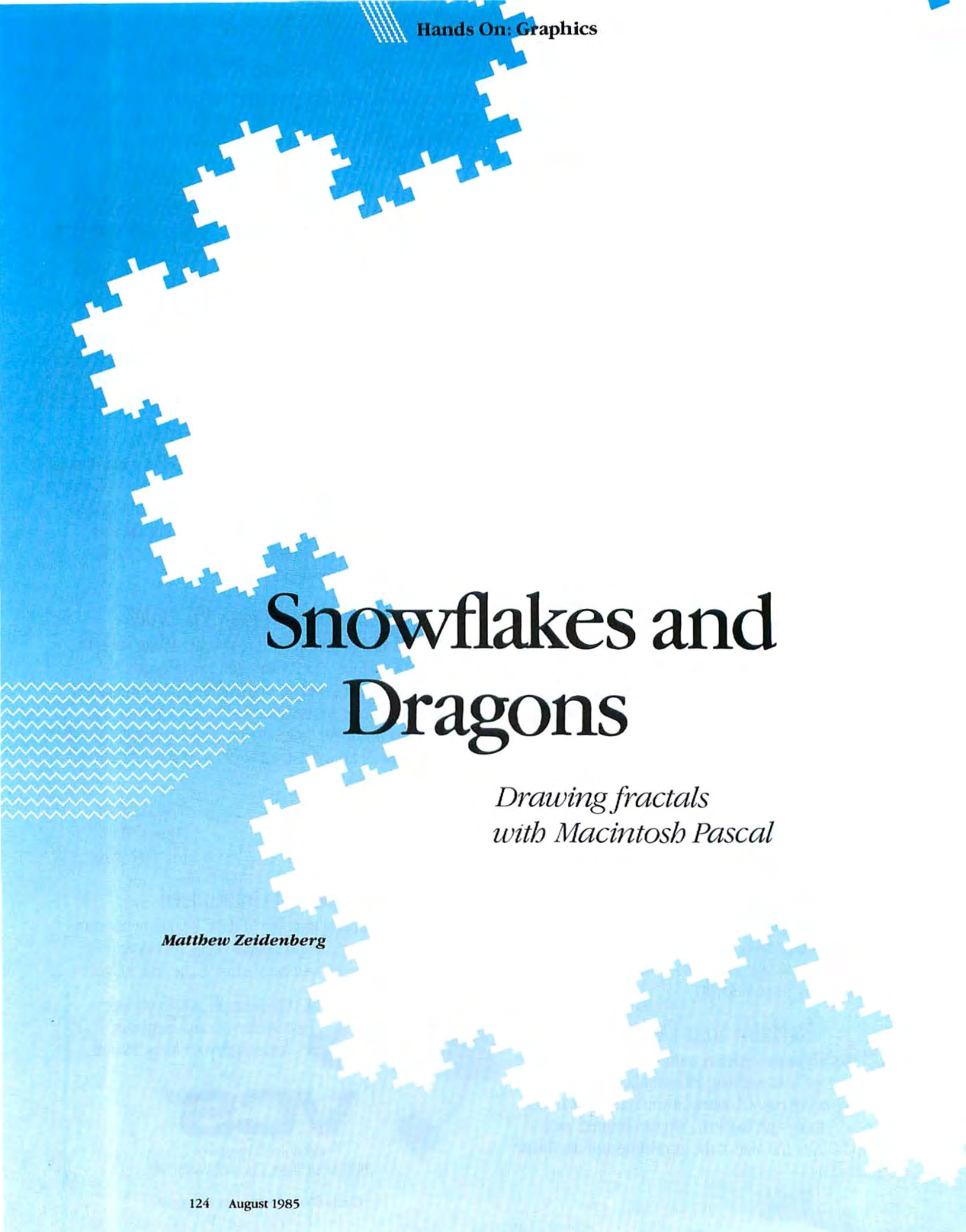


Snowflakes and Dragons

Drawing fractals with Macintosh Pascal

Matthew Zeidenberg

Looking at a snowflake from 6 feet away, you could describe it as a dot. Get a little closer, and the dot takes on detail. Zoom in to within a couple of inches, and its shape becomes difficult to describe. In the natural world you see shapes that have infinite detail, such as the edge of a leaf or the surface of a crystal. Describing detail at the level that occurs in nature is a problem that is addressed in the study of fractals.

If you use Macintosh Pascal, you have probably seen the program Sierpinski, which is on the Pascal disk. This program generates a beautifully complex pattern that is an example of a mathematical object called a fractal. The visual and conceptual appeal of fractals was made popular by mathematician Benoit B. Mandelbrot, whose book *The Fractal Geometry of Nature* (Freeman, 1983) is a collection of exotic mathematical objects and a detailed study of their nature and application in a variety of fields, including biology, physics, and economics.

Mandelbrot asks the question, How long is a coastline? The answer depends on how closely you look. If you measure a coastline on a map of the world, you get one answer, and if you measure each little inlet on a detailed coastal map, you get another. If you measure each grain of sand and every pebble on the beach, you get still another answer. Effectively, the length of a coastline is infinite; a coastline is an example of a fractal, an object of fractional dimension. It is the ultimate zigzag, in which the zigs and the zags are infinitely short.

Besides coastlines, the surfaces of many natural objects, such as forests and mountains, can be simulated with fractals. *Star Trek* fans may remember the planet Genesis in the film *Star Trek II: The Wrath of Khan.* It was created with fractals and computer animation.

Fractals in a plane have a dimension between a line (dimension one) and a surface (dimension two). Fractals in space have a dimension between a surface and a solid (dimension three).

One property common to many fractals is that a part resembles the whole. Each branch of a tree, for example, somewhat resembles the whole tree. Each secondary branch resembles the tree and the primary branch. If the branching in a tree continued ad infinitum, the tree would be a fractal.

Generating Fractals

Fractals can be constructed by repeatedly applying a set of rules to a given shape. Each fractal grows in generations from a seed called an *initiator.* The initiator can have the value $n = 0$ or $n = 1$, depending on where you choose to start counting generations. Each generation (n) is the product of the previous generation ($n - 1$). The rule or set of rules that you use to transform a given fractal from its initiator to each succeeding generation is called the *generator.* While the initiator and the generator completely describe how to construct a given fractal, its final shape is defined when the number of generations approaches infinity.

Since it's impossible to run a program for an infinite length of time, you can't create an actual fractal on a computer–or any other way, for that matter–but you can produce a good approximation.

Using Recursion

Since fractals are recursive objects, you can draw them on a computer using a recursive procedure or function. A recursive function is one that calls itself. The classic example of a recursive function computes the factorial of an integer n. The factorial of n is the product of the integers from 1 to n, expressed mathematically as $F(n) = n^* F(n - 1)$.

For a good introduction to the mysteries of recursion, see Douglas Hofstader's *Godel, Escher, Bach: The Eternal Golden Braid* (Basic Books, 1979).

Fractal Programs

The Macintosh is probably the first personal computer with adequate graphics for producing fractals. The fractals shown on the following pages are included in Mandelbrot's book and were created by programs in Macintosh Pascal using recursive Pascal procedures (see Listings 2, 3, and 4). If you type the programs in and run them, you can watch the fractals grow.

The fractal programs share the procedure Init, which asks you for the generation (n) of the fractal and creates the drawing window. If you plan to type more than one of these programs into the Mac, save Init in a separate file, which you can paste into each program through the Clipboard.

The statement **SaveDrawing('***Filename***')** right before the end of each program saves the contents of the drawing window in a file that can be read by *MacPaint.* You can also press ⌘-Shift-3 to take a snapshot of the screen.

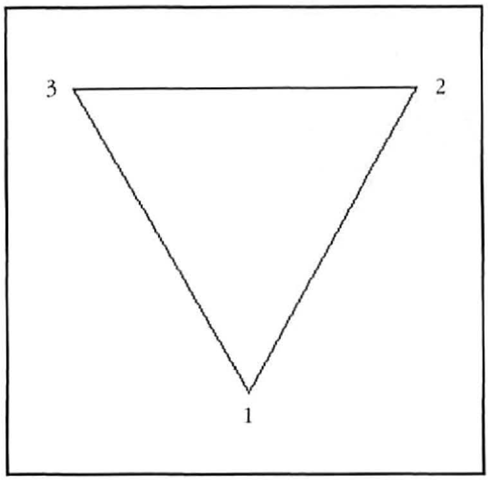
Another interesting modification you can make is to allow the initiator and the generator to be entered with the mouse and to generate a fractal based on the input.

The next six pages contain three examples of the infinite variety of fractals and the Pascal programs that produced them. You can generate many of the others in Mandelbrot's book, or you can invent your own. Many beautiful fractals have yet to see the light of the Macintosh screen.

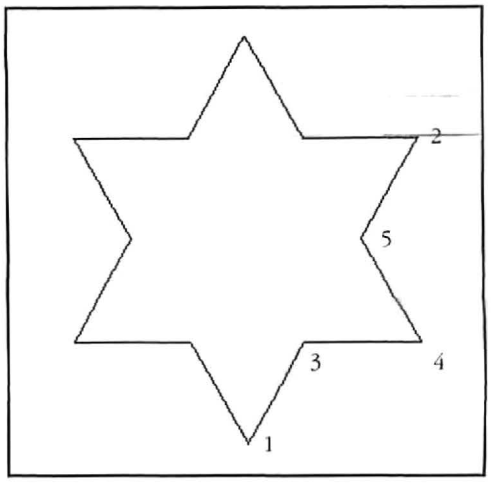
Matthew Zeidenberg is a graduate student in computer science at the University of Wisconsin, Madison.

Koch
Generated by the Koch program (see Listing 2), this fractal begins with an equilateral triangle. The generator replaces the middle third of each side with two sides of an equilateral triangle erected from the endpoints of the removed segment, forming a Star of David. The procedure is repeated for each generation.

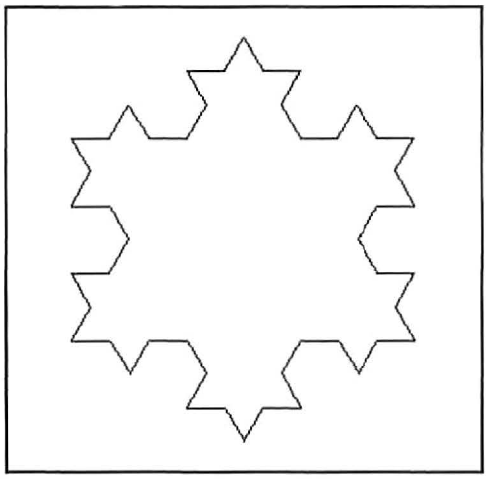
Since the initiator consists of three line segments, the main program calls the procedure kochr three times, once for each segment. The intermediate points necessary to create the second generation define four new segments that become initiators in recursive calls to kochr.



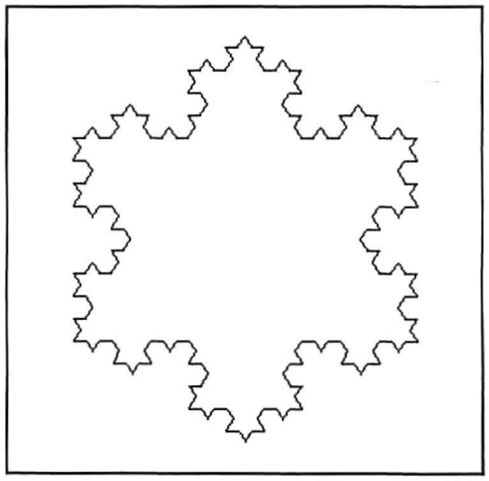
n = 1



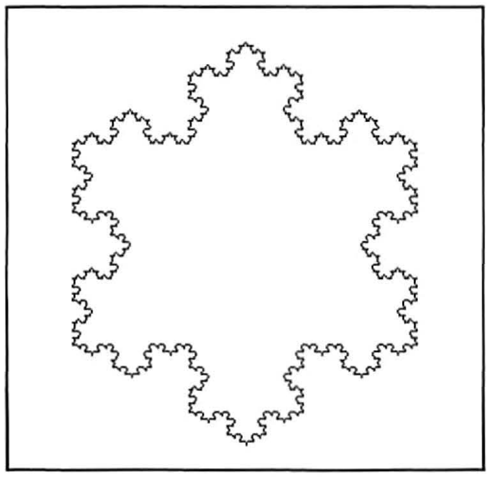
n = 2



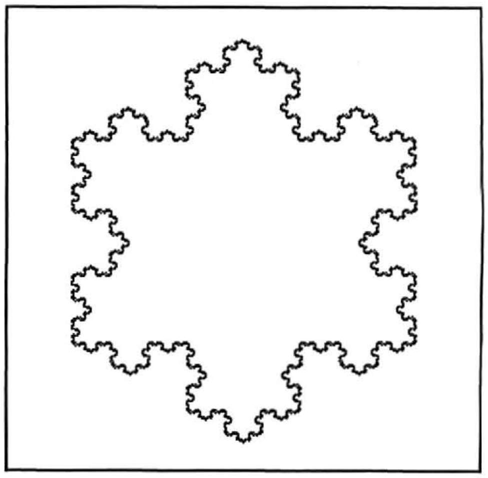
n = 3



n = 4



n = 5



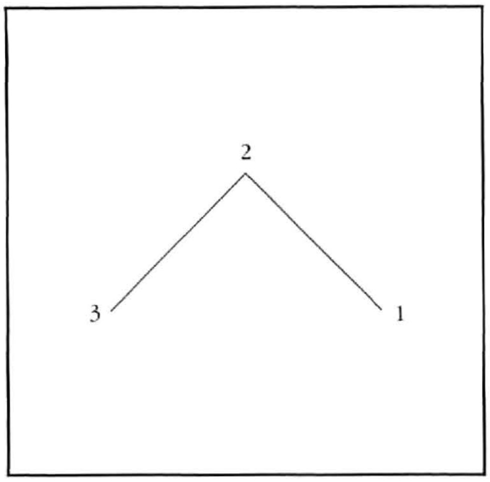
n = 6

```
program koch;
 const
   sin60 = 0.8660254;
   scaling = 100.0;
{xorig, yorig is center point of koch}
   xorig = 250;
   yorig = 150;
 var
   x1, y1, x2, y2, n, x3, y3 : integer;

{Procedure to create nth generation curve}
{given two points that define the initiator}
 procedure kochr (x1, y1, x2, y2, n : integer);
   var
    xdiff, ydiff, x3, y3, x4, y4, x5, y5 : integer;
 begin
{if n=1, draw the curve}
   if (n = 1) then
    drawline(x1 + xorig, y1 + yorig, x2 + xorig, y2 + yorig)
   else
{otherwise create the next generation}
    begin
{calculate the points of the next generation}
     xdiff := x2 - x1;
     ydiff := y2 - y1;
     x3 := x1 + round(xdiff / 3);
     y3 := y1 + round(ydiff / 3);
     x4 := x1 + round(xdiff / 2 - ydiff * sin60 / 3);
     y4 := y1 + round(ydiff / 2 + xdiff * sin60 / 3);
     x5 := x1 + round(xdiff * 2 / 3);
     y5 := y1 + round(ydiff * 2 / 3);
{recursive calls to kochr for each new segment}
     kochr(x1, y1, x3, y3, n - 1);
     kochr(x3, y3, x4, y4, n - 1);
     kochr(x4, y4, x5, y5, n - 1);
     kochr(x5, y5, x2, y2, n - 1)
    end;
 end;

{Draw text window, prompt for n}
{Display drawing window}
 procedure init (var n : integer);
  var
   drawwin : rect;
 begin
  hideall;
  showtext;
  write(' Input order of curve:');
  readln(n);
  hideall;
  drawwin.top := 40;
  drawwin.left := 0;
  drawwin.right := 512;
  drawwin.bottom := 512;
  setdrawingrect(drawwin);
  showdrawing;
 end;

{main program to draw Koch curve}
begin
 init(n);
{define initiator}
 x1 := 0;
 y1 := round(scaling * sin60);
 x2 := round(scaling);
 y2 := -round(scaling * sin60);
 x3 := -round(scaling);
 y3 := -round(scaling * sin60);
{call kochr for each segment}
 kochr(x1, y1, x2, y2, n);
 kochr(x2, y2, x3, y3, n);
 kochr(x3, y3, x1, y1, n);
end.
```

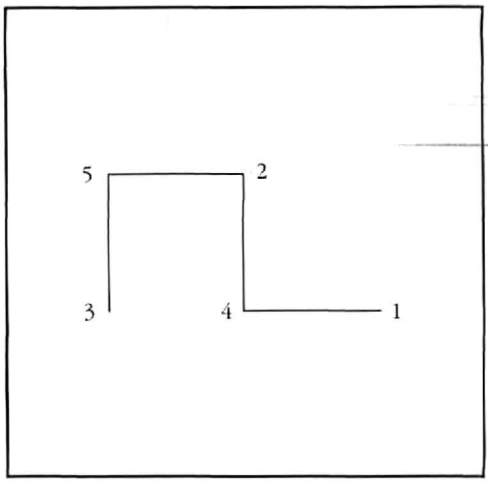
Listing 2

Dragon
The Dragon program produces this fractal (see Listing 3). Each generation replaces each line segment in the previous generation with two line segments at a right angle to each other. Like the snowflake fractal, this fractal is an example of how complex and apparently inexplicable patterns in nature can be generated with relatively simple rules.

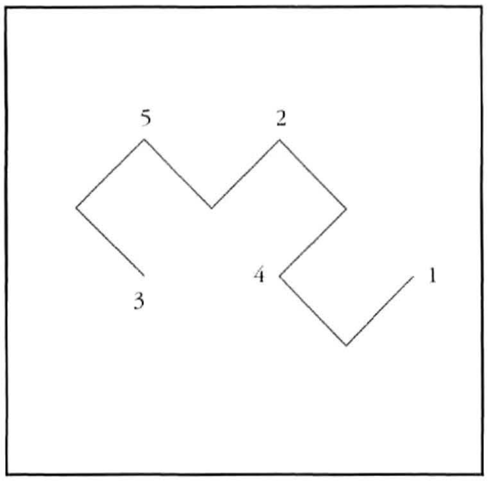
The procedure dragonr takes the initiator, which is defined by three points, and calculates two intermediate points. The resulting five points define two new initiators, which are used in recursive calls to dragonr.



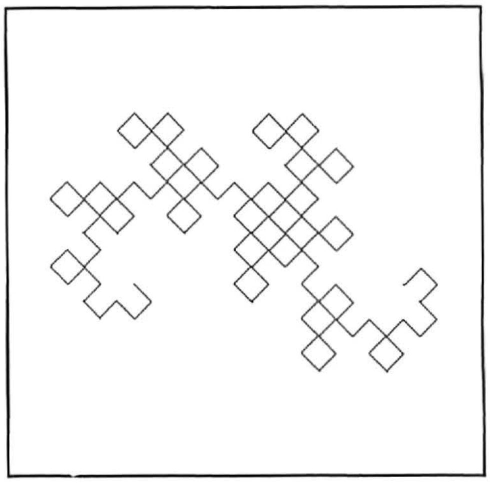
n = 1



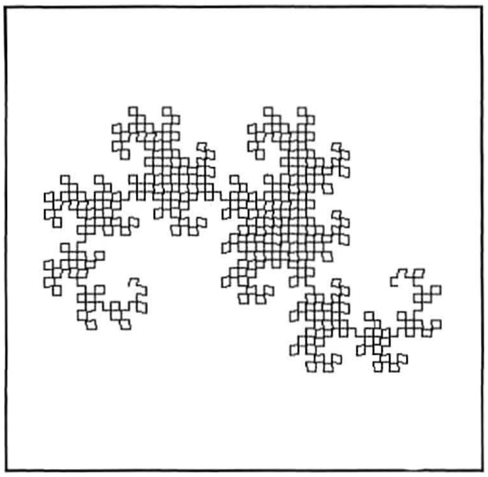
n = 2



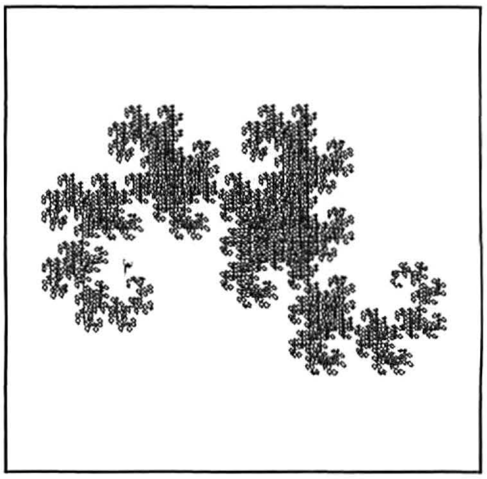
n = 3



n = 7



n = 10



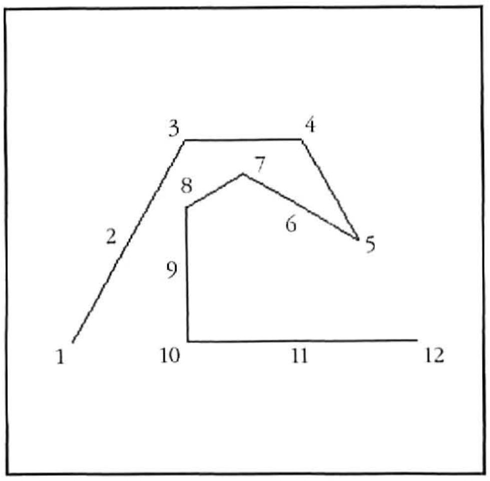
n = 13

```
program dragon;
 const
{Center point of dragon}
  xorig = 250;
  yorig = 180;
  scaling = 100;
 var
  x1, y1, x2, y2, x3, y3, n : integer;

{Procedure to create nth generation}
{Given three points for the initiator}
 procedure dragonr (x1, y1, x2, y2, x3, y3, n : integer);
  var
   x4, y4, x5, y5, ydiff, xdiff : integer;
 begin
{If n=1, draw the curve}
  if (n = 1) then
   begin
    drawline(x1, y1, x2, y2);
    drawline(x2, y2, x3, y3);
   end
{Otherwise construct next generation}
  else
   begin
{Calculate next generation points}
    x4 := ((x1 + x3) div 2);
    y4 := ((y1 + y3) div 2);
    x5 := x3 + (x2 - x4);
    y5 := y3 + (y2 - y4);
{Recursive calls to dragonr}
{create succeeding genrations }
    dragonr(x2, y2, x4, y4, x1, y1, n - 1);
    dragonr(x2, y2, x5, y5, x3, y3, n - 1);
   end;
 end;

{Draw text window, prompt for n}
{Display drawing window}
 procedure init (var n : integer);
  var
   drawwin : rect;
 begin
  hideall;
  showtext;
  write(' Input order of curve:');
  readln(n);
  hideall;
  drawwin.top := 40;
  drawwin.left := 0;
  drawwin.right := 512;
  drawwin.bottom := 512;
  setdrawingrect(drawwin);
  showdrawing;
 end;

{Main program to draw dragon}
begin
 init(n);
{Define the initiator}
 x1 := xorig + scaling;
 y1 := yorig;
 x2 := xorig;
 y2 := yorig - scaling;
 x3 := xorig - scaling;
 y3 := yorig;
{Call dragonr to create nth generation}
 dragonr(x1, y1, x2, y2, x3, y3, n);
end.
```

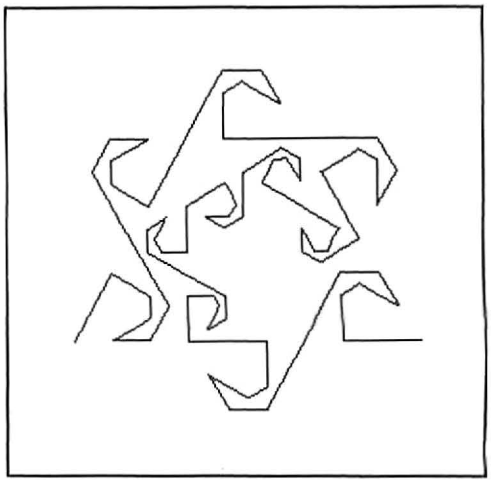
Listing 3

Snowflake
Produced by the Snowflake program (see Listing 4), this fractal gives you some insight into how snowflakes grow in nature, although it doesn't have true sixfold symmetry. The initiator is a line segment. The generator replaces each line segment with seven line segments defined by 12 numbered points, as shown in the first generation (n = 1).

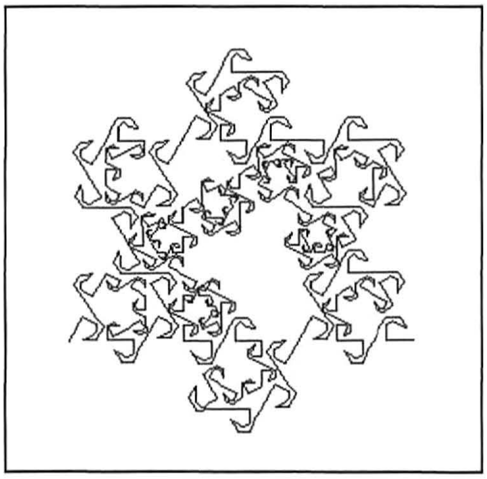
The procedure snowr takes the initiator and defines 12 intermediate points that define 11 segments. Each line segment is used as input in a recursive call to snowr.



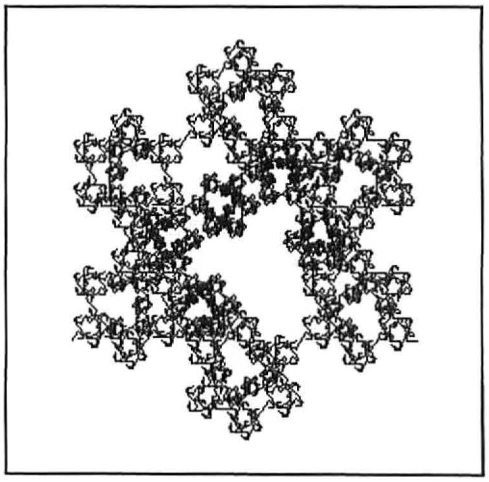
n=1



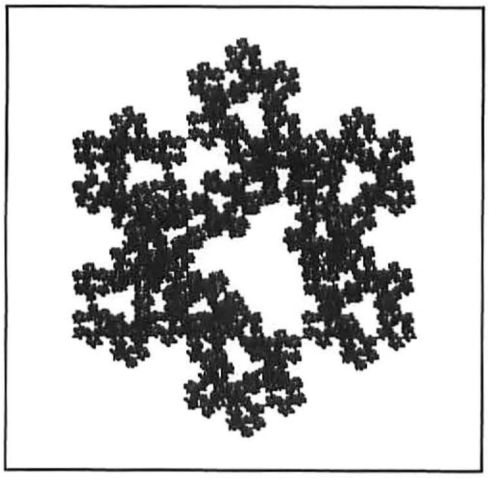
n=2



n=3



n=4



n=5

```
program snowflake;
 const
{Center point of snowflake}
  xorig = 250;
  yorig = 230;
  scaling = 100;
  s = 0.5773;
 var
  x1, y1, x2, y2, n : integer;

{Procedure to create nth generation curve}
{Given two points that define the initiator}
 procedure snowr (x1, y1, x12, y12, n : integer);
  var
   ydiff, xdiff, i : integer;
{arrays store 12 points of next generation}
   x : array[2..12] of integer;
   y : array[2..12] of integer;
 begin
{Calculate the points of the next generation}
  x[12] := x12;
  y[12] := y12;
  xdiff := x12 - x1;
  ydiff := y12 - y1;
  x[10] := x1 + xdiff div 3;
  y[10] := y1 + ydiff div 3;
  x[11] := x1 + (xdiff * 2) div 3;
  y[11] := y1 + (ydiff * 2) div 3;
  x[3] := x[10] + round(ydiff * s);
  y[3] := y[10] - round(xdiff * s);
  x[9] := x[10] + round(ydiff * s / 3.0);
  y[9] := y[10] - round(xdiff * s / 3.0);
  x[8] := x[10] + round(ydiff * s * 2.0 / 3.0);
  y[8] := y[10] - round(xdiff * s * 2.0 / 3.0);
  x[2] := (x1 + x[3]) div 2;
  y[2] := (y1 + y[3]) div 2;
  x[4] := x[11] + round(ydiff * s);
  y[4] := y[11] - round(xdiff * s);
  x[6] := x[11] + round(ydiff * s * 2.0 / 3.0);
  y[6] := y[11] - round(xdiff * s * 2.0 / 3.0);
  x[5] := (x12 + x[4]) div 2;
  y[5] := (y12 + y[4]) div 2;
  x[7] := (x[8] + x[4]) div 2;
  y[7] := (y[8] + y[4]) div 2;

 {if n=1, draw the curve}
   if (n = 1) then
    begin
     moveto(x1, y1);
     for i := 2 to 11 do
      begin
       lineto(x[i], y[i]);
      end;
     lineto(x12, y12);
    end
{otherwise calculate the next generation}
  else
   begin
{recursive calls to snowr for each of the 11 segments}
    snowr(x[2], y[2], x1, y1, n - 1);
    snowr(x[2], y[2], x[3], y[3], n - 1);
    snowr(x[3], y[3], x[4], y[4], n - 1);
    snowr(x[4], y[4], x[5], y[5], n - 1);
    snowr(x[5], y[5], x[6], y[6], n - 1);
    snowr(x[7], y[7], x[6], y[6], n - 1);
    snowr(x[7], y[7], x[8], y[8], n - 1);
    snowr(x[9], y[9], x[8], y[8], n - 1);
    snowr(x[9], y[9], x[10], y[10], n - 1);
    snowr(x[11], y[11], x[10], y[10], n - 1);
    snowr(x[11], y[11], x[12], y[12], n - 1);
   end;
 end;

{Draw text window, prompt for n}
{Display the drawing window}
 procedure init (var n : integer);
  var
   drawwin : rect;
 begin
  hideall;
  showtext;
  write(' Input order of curve:');
  readln(n);
  hideall;
  drawwin.top := 40;
  drawwin.left := 0;
  drawwin.right := 512;
  drawwin.bottom := 512;
  setdrawingrect(drawwin);
  showdrawing;
 end;

{main program to draw snowflake}
begin
 init(n);
{define the initiator}
 x1 := xorig - scaling;
 y1 := yorig;
 x2 := xorig + scaling;
 y2 := yorig;
{call snowr to create nth generation curve}
 snowr(x1, y1, x2, y2, n);
end.
```

Listing 4