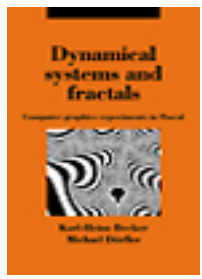



Dynamical Systems and Fractals

Computer Graphics Experiments with Pascal

Karl-Heinz Becker, Michael Dörfler, Translated by I. Stewart

Book DOI: http://dx.doi.org/10.1017/CBO9780511663031

Online ISBN: 9780511663031

Hardback ISBN: 9780521360258

Paperback ISBN: 9780521369107

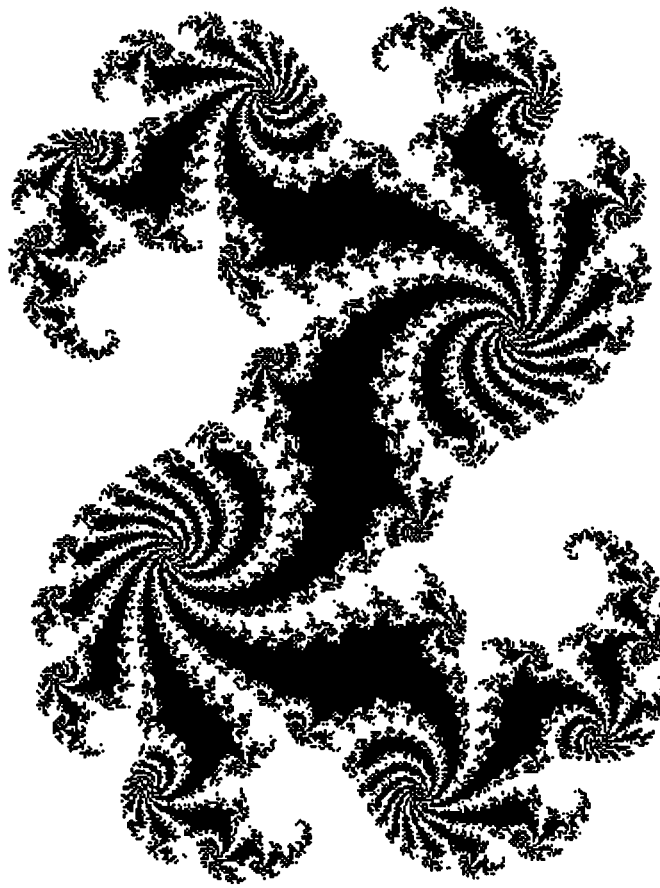
Chapter

8 - Fractal Computer Graphics pp. 203-230

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511663031.010

Cambridge University Press

# 8 Fractal Computer Graphics



## 8.1 All Kinds of Fractal Curves

We have already encountered fractal geometric forms, such as Julia sets and the Gingerbread Man. We will develop other aspects of their interesting and aesthetically appealing structure in this chapter. We gradually leave the world of dynamical systems, which until now has played a leading role in the formation of Feigenbaum diagrams, Julia sets, and the Gingerbread Man. There exist other mathematical functions with fractal properties. In particular we can imagine quite different functions, which have absolutely nothing to do with the previous background of dynamical systems. In this chapter we look at purely geometric forms, which have only one purpose - to produce interesting computer graphics. Whether they are more beautiful than the Gingerbread Man is a matter of personal taste.

Perhaps you already know about the two most common fractal curves. The typical structure of the Hilbert and Sierpiński curves is shown in Figures 8.1-1 and 8.1-2. The curves are here superimposed several times.

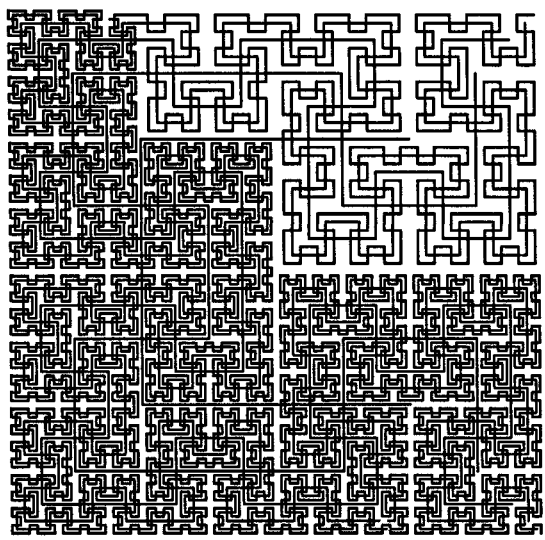


**Figure 8.1-1** 'Genesis' of the Hilbert curve.

As with all computer graphics that we have so far drawn, the pictures are 'static' representations of a single situation at some moment. This is conveyed by the word 'genesis'. Depending on the parameter $n$, the number of wiggles, the two 'space-filling' curves become ever more dense. These figures are so well known that in many computer science books they are used as examples of recursive functions. The formulas for computing them, or even the programs for drawing them, are written down there: see for

example Wirth (1983). We therefore do not include a detailed description of these two curves. Of course, we encourage you to draw the Hilbert and Sierpiński curves, even though they are well known.

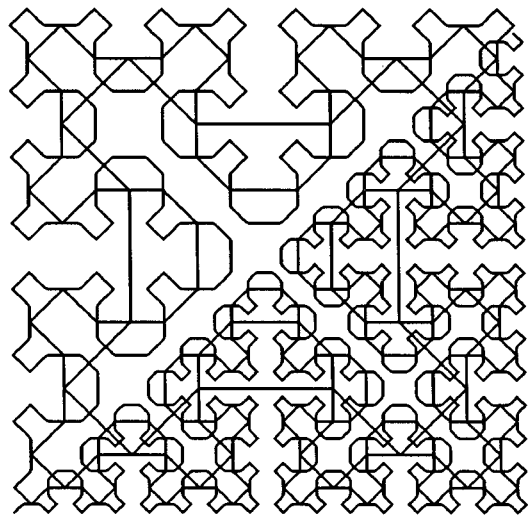


**Figure 8.1-2** 'Genesis' of the Sierpiński curve.

Recursive functions are such that their definition is in some sense 'part of itself'. Either the procedure calls itself (direct recursion), or it calls another, in which it is itself required (indirect recursion). We refer to recursion in a graphical representation as self-similarity.

We obtain further interesting graphics if we experiment with *dragon curves* or *C-curves*. Figures 8.1-3 to 8.1-5 show the basic form of these figures.

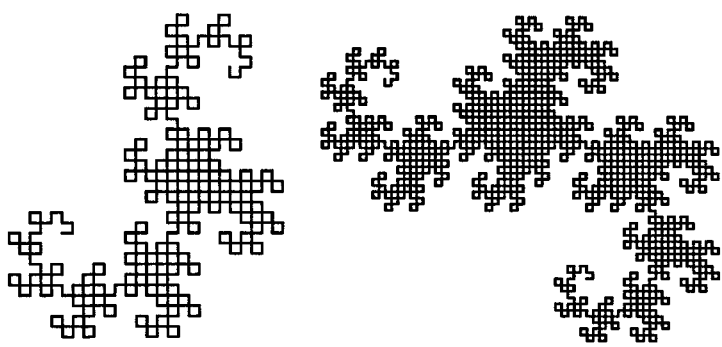


**Figure 8.1-3** Different dragon curves.

Before you proceed to your own experiments, with the aid of the exercises, we will provide you with some drawing instructions in the form of a program fragment. Having set up this scheme you can easily experiment with fractal figures.

It is simplest if we start the drawing-pen at a fixed position on the screen and begin drawing from there. We can move forwards or backwards, thereby drawing a line.

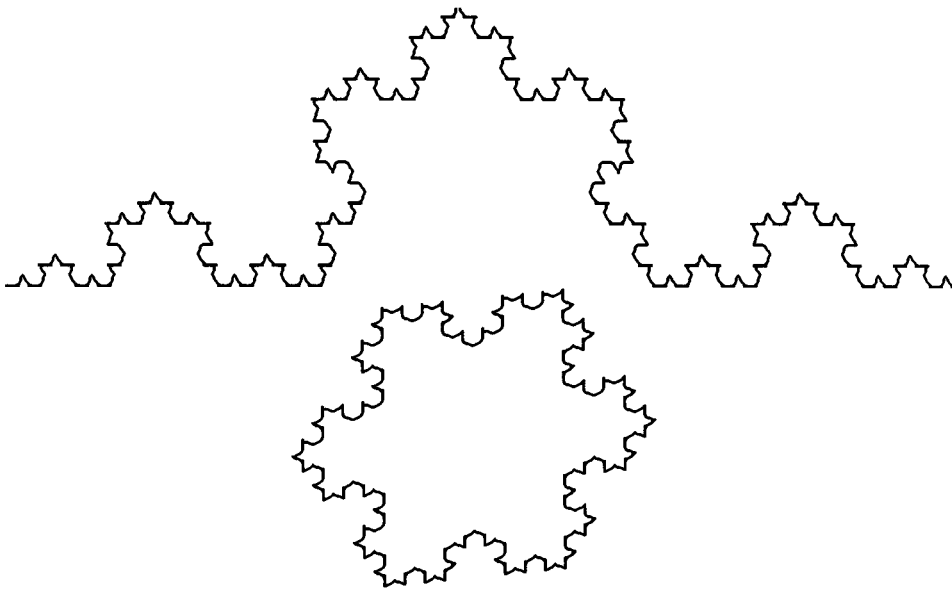


**Figure 8.1-4** Different Koch curves.

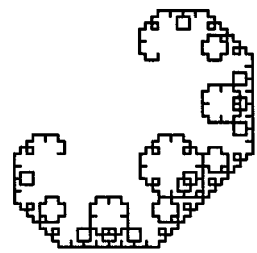


**Figure 8.1-5** A C-curve.

Seymour Papert's famous *turtle* can be used in the same way. This creature has given its name to a type of geometric drawing procedure: *turtle geometry*: see Abelson and diSessa (1982). The turtle can of course turn as well, changing the direction in which it moves. If the turtle is initially placed facing the $x$-direction, the command

```
forward (10)
```

means that it takes 10 steps in the direction of the positive $x$-axis. The command

```
turn (90)
```

rotates it clockwise through 90°. A repeat of the command

```
forward (10)
```

makes the turtle move 10 steps in the new direction. Basically, that is everything that the turtle can do. On some computers, turtle graphics is already implemented. Find out more from your handbook.

We have also formulated the main commands in Pascal, so that you can easily carry out this process on your computer. This scheme, which works with 'relative' coordinates, is as follows:

**Program Fragment 8.1-1** (Procedures for turtle graphics)

```
PROCEDURE forward (step: integer);
   CONST
      pi = 3.1415;
   VAR
      xstep, ystep: real;
BEGIN
   xstep := step * cos ((turtleangle*pi)/180);
   ystep := step * sin ((turtleangle*pi)/180);
   turtlex := turtlex + trunc(xstep);
   turtley := turtley + trunc(ystep);
   drawLine (turtlex, turtley);
END;

PROCEDURE backward (step: integer);
BEGIN
   forward (-step);
END;

PROCEDURE  turn (alpha: integer);
BEGIN
   turtleangle := (turtleangle + alpha) MOD 360;
END;

PROCEDURE startTurtle;
BEGIN
   turtleangle := 90; turtlex := startx; turtley := starty;
   setPoint (startx, starty);
END;
```

With this scheme you are in a position to carry out the following exercises. We hope you have a lot of fun experimenting.

## Computer Graphics Experiments and Exercises for §8.1

### Exercise 8.1-1

Set up a program to draw the Hilbert graphic. Experiment with the parameters. Draw pictures showing overlapping Hilbert curves. Draw the Hilbert curve tilted to different inclinations. Because the task is easier with a recursive procedure, we will describe the important features now. The drawing instructions for the Hilbert curve are:

```
PROCEDURE hilbert (depth, side, direction : integer);
BEGIN
   IF depth > 0 THEN
   BEGIN
      turn (-direction * 90);
      hilbert (depth-1, side, -direction);
      forward (side);
      turn (direction * 90);
      hilbert (depth-1, side, direction);
      forward (side);
      hilbert (depth-1, side, direction);
      turn (direction * 90);
      forward (side);
      hilbert (depth-1, side, direction);
      turn (direction * 90);
   END;
END;
```

As you have already discovered in earlier pictures, you can produce different computer-graphical effects by varying the depth of recursion or the length of the sides. The value for the direction of the Hilbert curve is either 1 or -1.

### Exercise 8.1-2

Experiment with the following curve. The data are, e.g.:

| depth | side | startx | starty |
|---|---|---|---|
| 9 | 5 | 50 | 50 |
| 10 | 4 | 40 | 40 |
| 11 | 3 | 150 | 30 |
| 12 | 2 | 150 | 50 |
| 13 | 1 | 160 | 90 |
| 15 | 1 | 90 | 150 |

The drawing instructions for the dragon curve are:

```
PROCEDURE dragon (depth, side: integer);
BEGIN
   IF depth > 0 THEN
      foward (side)
   ELSE IF depth > 0 THEN
      BEGIN
         dragon (depth-1, trunc (side));
         turn (90);
         dragon (-(depth-1), trunc (side));
      END
   ELSE
      BEGIN
         dragon (-(depth+1),trunc (side));
         turn (270);
         dragon (depth+1,trunc (side);
      END;
END;
```

**Exercise 8.1-3**

Experiment in the same way with the Koch curve. The data are, e.g.:

| depth | side | startx | starty |
|---|---|---|---|
| 4 | 500 | 1 | 180 |
| 5 | 500 | 1 | 180 |
| 6 | 1000 | 1 | 180 |

The drawing procedure is:

```
PROCEDURE koch (depth, side: integer);
BEGIN
   IF depth = 0 then forward (side)
   ELSE BEGIN
      koch (depth-1, trunc (side/3)); turn (-60);
      koch (depth-1, trunc (side/3)); turn (120);
      koch (depth-1, trunc (side/3)); turn (-60);
      koch (depth-1, trunc (side/3));
   END;
END;
```

**Exercise 8.1-4**

In the lower part of Figure 8.1-4 you will find the snowflake curve. Snowflakes can be built from Koch curves. Develop a program for snowflakes and experiment with it:

```
PROCEDURE snowflake;
BEGIN
   koch (depth, side); turn (120);
   koch (depth, side); turn (120);
   koch (depth, side); turn (120);
END;
```

**Exercise 8.1-5**

Experiment with right-angled Koch curves. The generating method and the data for pictures are, e.g.:

```
PROCEDURE rightkoch (depth, side: integer);
BEGIN (* depth = 5, side = 500, startx = 1, starty = 180 *)
   IF depth = 0 THEN forward (side)
   ELSE
   BEGIN
      rightkoch (depth-1, trunc (side/3)); turn (-90);
      rightkoch (depth-1, trunc (side/3)); turn (90);
      rightkoch (depth-1, trunc (side/3)); turn (90);
      rightkoch (depth-1, trunc (side/3)); turn (-90);
      rightkoch (depth-1, trunc (side/3));
   END;
END;
```

**Exercise 8.1-6**

Experiment with different angles, recursion depths, and side lengths. It is easy to change the angle data in the procedural description of C-curves:

```
PROCEDURE cCurve (depth, side : integer);
BEGIN (* depth = 9,12; side = 3; startx = 50, 150;
                  starty = 50, 45 *)
   IF depth = 0 THEN forward (side)
   ELSE
   BEGIN
      cCurve (depth-1, trunc (side)); turn (90);
      cCurve (depth-1, trunc (side)); turn (-90);
   END;
END;
```

**Exercise 8.1-7**

At the end of the C-curve procedure, after the `turn(-90)`, add yet another procedure call `cCurve (depth-1, side)`. Experiment with this new program. Insert the new statement after the procedure call `turn(90)`.

Experiment also with C-curves by changing the side length inside the procedure,

for example `forward (side/2)`, etc.

**Exercise 8.1-8**

Draw a tree-graphic. Use the following general scheme:

```
PROCEDURE tree (depth, side: integer);
BEGIN (* e.g. depth = 5, side = 50 *)
   IF  depth > 0 THEN
   BEGIN
      turn (-45); forward (side);
      tree (depth-1, trunc (side/2)); backward (side);
                  turn (90);
      forward (side);
      tree (depth-1, trunc (side/2)); backward (side);
                  turn (-45);
   END;
END;
```

We have collected together the solutions to these exercises in §11.2 in a complete program. Look there if you do not wish to develop the programs for yourself.

If you have done Exercise 8.1-10, the picture will doubtless have reminded you of structures in our natural surroundings - hence the name. We thereby open up an entire new chapter of computer graphics. Computer graphics experts throughout the world have been trying to construct convincing natural forms. The pictures that emerge from the computer are landscapes with trees, grass, mountains, clouds, and lakes. Of course you need rather fancy programs to draw really convincing graphics.

But even with small computers we can produce nice things.

## 8.2 Landscapes: Trees, Grass, Clouds, Mountains, and Lakes

Since 1980, when the discovery of the Mandelbrot set opened a new chapter in fundamental mathematical research, new discoveries in the general area of fractal structure have occurred almost daily. Examples include fractal models of cloud formation and rainfall in meteorology. Several international conferences on computer graphics have been devoted to this; see SIGGRAPH (1985).

Likewise, such procedures are part of the computer-graphical cookery used for special effects in films. The latest research objective is the convincingly natural representation of landscapes, trees, grass, and clouds. Some results are already available in the products of the American company LucasFilms, which in recent years has made several well-known science fiction films. It has its own team for basic scientific research into computer graphics. Not surprisingly, conference proceedings sometimes bear the address of the LucasFilms studios, e.g.Smith (1984). It is now only a matter of time and

computing power before films contain lengthy sequences calculated by a computer.

Some examples show how easy it is to construct grasses and clouds on a personal computer.

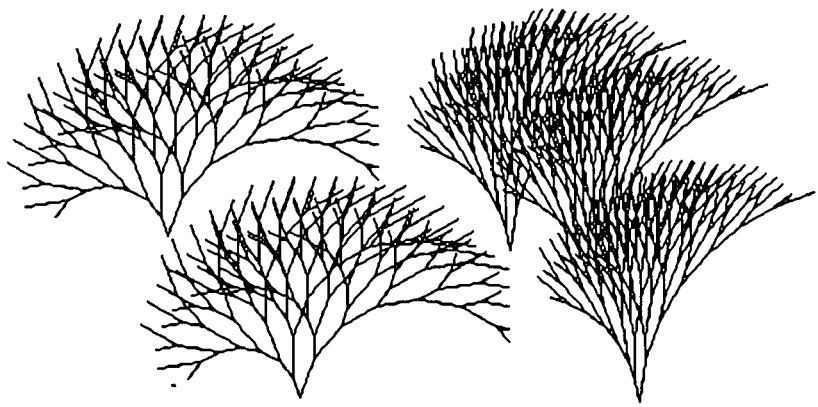


**Figure 8.2-1** Grass and twigs.

Figure 8.2-1 shows a tree curve (see exercises for §8.1). The generating procedure for Figure 8.2-2 is already known. Namely, it is a dragon curve of high recursion depth and side length 1. In fact, with all fractal figures, you can experiment, combining the recursive generating procedures together or generalising them. Changing the depth, side length, or angles according to parameters can produce baffling results, which it is quite likely that nobody has ever seen before. To simulate natural structures such as grass, trees, mountains, and clouds, you should start with geometrical figures whose basic form resembles them, and fit them together.

A new possibility arises if we change the parameters using a random number generator.

For example we can make the angle change randomly between 10 and 20 degrees, or the length. You can of course apply random numbers to all of our other figures. We recommend that you first become very familiar with the structure, to get the best possible grasp of the effect of parameter changes. Then you can introduce random numbers effectively.

Figure 8.2-3 shows examples of such experiments. For example, the command `forward (side)` may be changed by a random factor. Thus the expression `side` is replaced by `side * random(10,20)`. This is what we have done to get the left and middle pictures in Figure 8.2-3. More natural effects are obtained if we make only small changes. Thus in the right-hand picture the expression `side` is replaced by `side*random (side, side+5)`. The `random` function always produces values

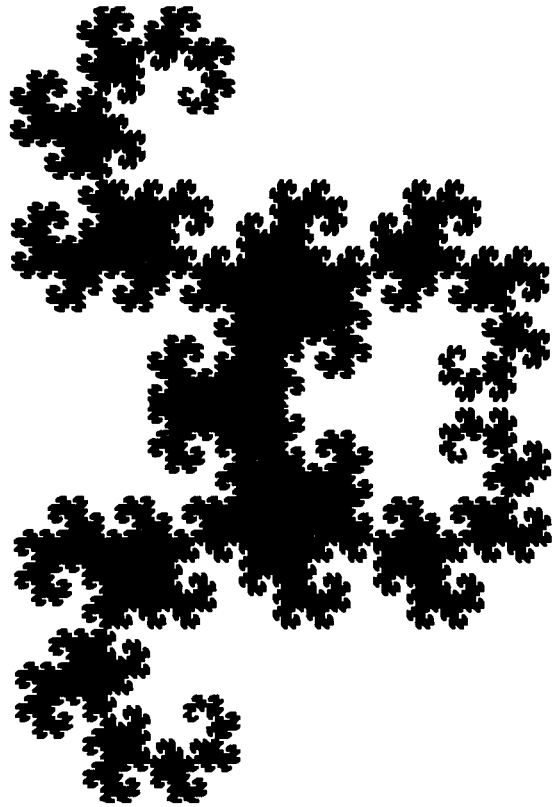


**Figure 8.2-2** Two-dimensional cloud formation.

between the two bracketed bounds (compare Exercise 8.2-2).

Of course the different pictures in Figure 8.2-3 are not easily reproducible. We wrote a program to draw an endless sequence of such grass-structures, and selected interesting pictures from the results. You can collect your best grass- and cloud-structures on an external storage medium (optical, hard, or floppy disk).

For experiments with grass-structures, use the above description (see Exercise 8.2-2) as the basis for your own discoveries. The pictures were generated with the fixed value `depth` = 7, `angle` = 20 or 10, and random side lengths.

Modern computers sometimes have the facility to merge or combine parts of pictures with others. It is also possible, using drawing programs such as MacPaint (Macintosh) or special programs on MS-DOS or Unix machines, to mix parts of pictures and work on them further.

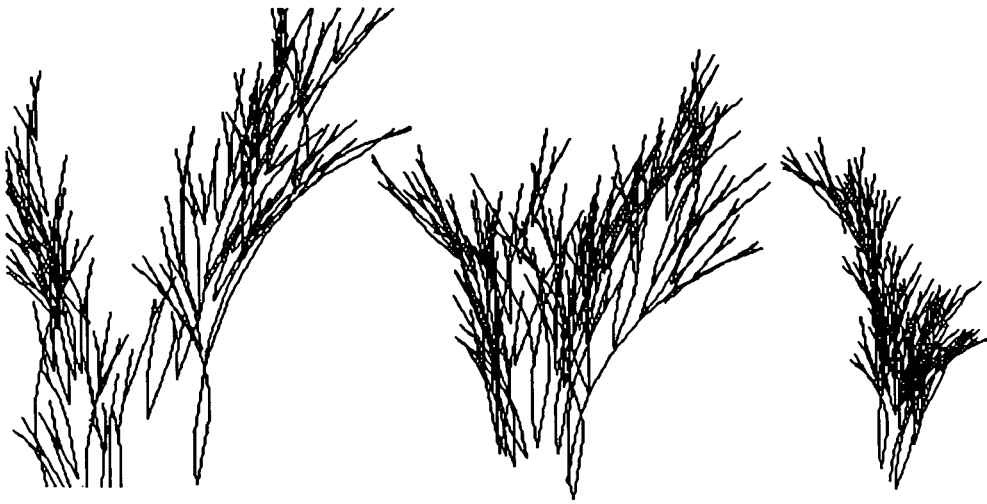


**Figure 8.2-3** Different grasses.

Having obtained the many small units that arise in nature, we wish to put them together into a complete landscape. The principle for generating landscapes was explained in the Computer Recreations column of *Scientific American*, see Dewdney (1986a).

Start with a triangle in space. The midpoints of the three sides are displaced upwards or downwards according to a random method. If these three new points are joined, we have in general four different triangles, with their own particular sides and corners. The method is carried out in turn for the resulting small triangles. To obtain a realistic effect, the displacment of the midpoints should not be as great as the first time. From 1 triangle with 3 vertices, we thus obtain 4 triangles with 6 vertices, 16 triangles with 15 vertices, 64 triangles with 45 vertices. After 6 iterations we have 4096 tiny triangles contained by 2145 vertices. Now we draw these. As you see in Figure 8.2-4, the result is in fact reminiscent of mountains. For reasons of visibility we have drawn only two of the three sides of each triangle. Also, we have supplemented the picture to form a rectangle.

We avoid describing a program here: instead we have given an example of a complete program in §11.3. Because the computation of the four points takes quite a long time, we have only specified the displacements. So, for example, with the same data-set you can draw a sunken or a raised landscape. Or you can ignore everything below some particular value of the displacment and draw just single dots. In this way an impression of lakes and seas is created, as in Figure 8.2-4. In fact these pictures are already quite remarkable. Of course there are many other gadgets in the LucasFilm

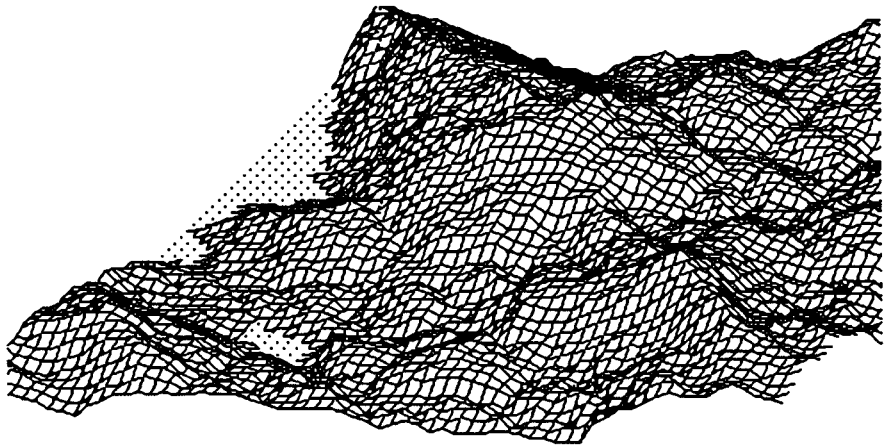


**Figure 8.2-4** Fractal landscape with lake and mountains.

studio's box of tricks. Even the computer-graphical methods are improved from day to day, so that fiction and reality merge smoothly into each other. But that is another story.

## Computer Graphics Experiments and Exercises for §8.2

### Exercise 8.2-1

With the aid of the above description, develop a program to represent twigs:

```
PROCEDURE twig (depth, side, angle : integer);
BEGIN
   IF depth > 0 THEN
   BEGIN
      turn (-angle);
      forward (2*side);
      twig (depth-1, side, angle);
      backward (2*side); turn (2*angle);
      forward (side);
      twig (depth-1, side, angle);
      backward (side); turn (-angle);
   END;
END;
```

### Exercise 8.2.-2

Experiment with grass structures too:

```
PROCEDURE randomTwig (depth, side, angle: integer);
   CONST
      delta = 5;
BEGIN
   IF depth > 0 THEN
   BEGIN
      turn (-angle);
      forward (2 * random (side, side+delta));
      randomTwig (depth-1, side, angle);
      forward(-2 * random (side, side+delta));
      turn (2*angle);
      forward (random (side, side+delta));
      randomTwig (depth-1, side, angle);
      forward (-random (side, side+delta));
      turn (-angle);
   END;
END;
```

As data for the two exercises, use $7 \leq$ `depth` $\leq 12$, $10 \leq$ `side` $\leq 20$, $10 \leq$ `angle` $\leq 20$.

Try to implement a suitable procedure `random`.

If you modify the program descriptions in other ways, you can generate still more pictures.

## 8.3 Graftals

Besides fractal structures, nowadays people in the LucasFilm studios or university computer graphics laboratories also do experiments with *graftals*. These are mathematical structures which can model much more professionally the things we drew in the previous section: plants and trees. Like fractals, graftals are characterised by self-similarity and great richness of form under small changes of parameters. For graftals, however, there is no mathematical formula such as those we have found for the simple fractal structures we have previously investigated. The prescription for generating graftals is given by so-called *production rules*. This concept comes from information theory. The grammatical structure of progamming languages is defined by production rules. With production rules it is possible to express how a language is built up. We use something similar to model natural structures, when we wish to lay down formal rules for the way they are constructed.

The words of our language for generating graftals are strings formed from the symbols '0', '1', and square brackets '[', ']'. For instance, the string 01[11[01]] represents a graftal.

A production rule (substitution rule) might resemble the following:

$0 \rightarrow 1[0]1[0]0$

$1 \rightarrow 11$

[ → [
] → ]

The rule given here is only an example. The rule expresses the fact that the string to the left of the arrow can be replaced by the string to its right. If we apply this rule to the string 1[0]1[0]0, we get

11[1[0]1[0]0]11[1[0]1[0]0]1[0]1[0]0.

Another rule (1111[11]11[111]1) represents, for instance, a tree or part of a tree with a straight segment 7 units long. Each 1 counts 1 unit, and in particular the numbers in brackets each represent a twig of the stated length. Thus the first open bracket represents a twig of length 2. It begins after the first 4 units of the main stem. The 4 units of the main stem end at the first open bracket. The main stem grows a further 2 units. Then there is a twig of length 3, issuing from the 6th unit of the main stem. The main stem is then finally extended by 1 unit (Figure 8.3-1).

This example shows how by using and interpreting a particular notation, namely 1111[11]11[111]1, a simple tree structure can be generated. Our grammar here has the alphabet {1,[,]}. The simplest graftal has the alphabet {0,1,[,]}. The '1' and '0' characterise segments of the structure. The brackets represent twigs.

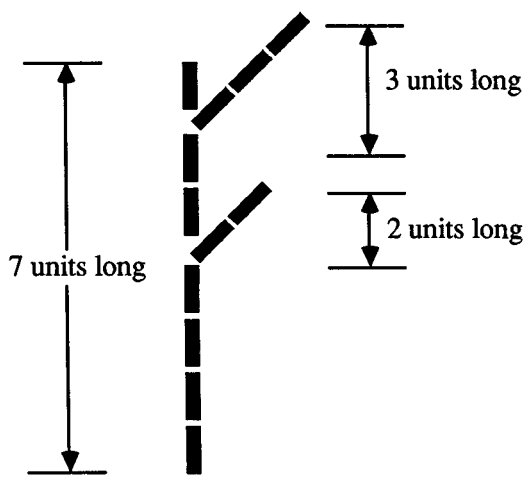



**Figure 8.3-1** The structure 1111[11]11[111]1.

Once we know how to interpret these bracketed structures, all we require is a method for producing structures with many twigs by systematically applying production rules. This variation on the idea just explained can be obtained as follows:

We agree that each part of the structure is generated from a triple of zeros and ones, such as 101 or 111. Each triple stands for a binary number, from which, with the aid of

the appropriate production rule, the graftal is constructed.

For example, '101' expresses in binary the number 5, and '111' the number 7.

We provide a table of binary numbers to help you remember:

| Decimal | Binary |
|---|---|
| 0 | 000 |
| 1 | 001 |
| 2 | 010 |
| 3 | 011 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |

**Table 8.3-1** Decimal and binary numbers (0 - 7).

We express the production rule for building the graftal in terms of such binary numbers. Because there are $2^3$ possible combinations of digits {0,1}, we must specify 8 production rules.

An example of such an 8-fold production rule is:

**0.1.0.1.0.00[01]0.0**

or, otherwise written,

| position: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| rule: | **0.** | **1.** | **0.** | **1.** | **0.** | **00[01].** | **0.** | **0** |
| binary number: | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |

An example will explain how the construction is carried out.

We take each individual digit of the rule and generate the binary triple. Compare the triple with the production rule. Replace the digit with the corresponding sequence from the rule. To do this, the formation of the triples must also be governed by a rule. This takes (e.g.) the following form:

For a single zero or one at the beginning, a 1 is added to left and right. For a pair of numbers, first a 1 is added to the left and then the pair is repeated with a 1 added to the right.

Start with a 1. We add 1 to left and right: this gives 111. If we had started with 0 we would have got 101.

Our production rule consists mainly of the strings '0' or '00[01]'. A '1' generates '0' by applying the rule, a '0' generates '00[01]'.

Applying the rule to these strings then leads to complicated forms.

We follow the development through several generations, beginning with '1':

***Generation 0***

1 → transformation → 111

111 → rule → **0**

(An isolated '1' at the start is extended to left and right by '1'.)

***Generation 1***

**0** → transformation → 101

101 → rule → **00[01]**

(An isolated '0' at the start is extended to left and right by '1'.)

***Generation 2***

**00**[01] → transformation→ 100 001 [001 011]

100 001 [001 011] → rule → **01[11]**

(For pairs of digits 1 is added to the left and to the right.)

***Generation 3***

**01**[11] → transformation → 101 011[111 111]

101 011[111 111] → rule → **00[01] 1 [0 0]**

(If the main track is broken by a branch, the last element of the main branch is used for pair-formation.)

***Generation 4***

**00**[01] **1** [0 0] → transformation → 100 001 [001 011] 011 [100 001]

100 001 [001 011] 011 [100 001] → rule → **01[11]1[01]**

(if the main track is broken by a branch, the last element of the main branch is used for pair-formation.)

***Generation 5***

**01**[11]1[01] → transformation → 101 011 [111 111] 111 [101 011]

101 011 [111 111] 111 [101 011] → rule → **00[01] 1 [0 0] 0 [00[01] 1]**

***Generation 6***

would then be 01[11]0[01]00[01][01[11]1].

In order to make the above clearer, let us consider generations 3 and 4 (see Figure 8.3-2).

Generation 3 comprises 01[11].

Take the pair 01 and extend left by 1 to get 101.

Take the pair 01 and extend right by 1 to get 011.

The brackets follow.

Take the pair 11 and extend left by 1 to get 111.

Take the pair 11 and extend right by 1 to get 111.

Generation 4 comprises 00[01]1[01]. At generation 4 there is a difficulty. The main branch (**001**) is broken by a side-branch (brackets): **00**[01]1[01].

Take the pair 00 and extend left by 1 to get 100.

Take the pair 00 and extend right by 1 to get 001.

Take the pair 01 and extend left by 1 to get 100.

Take the pair 01 and extend right by 1 to get 011.

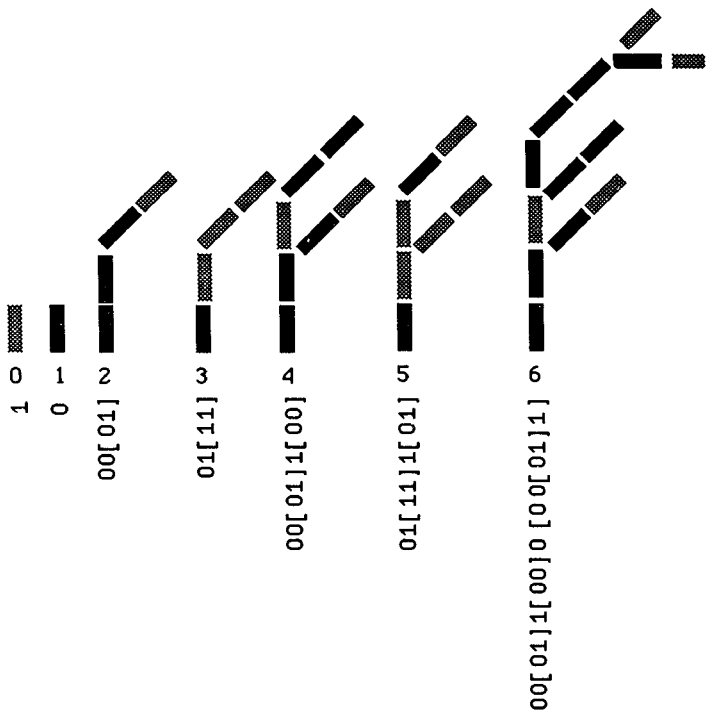



**Figure 8.3-2** Construction of a graftal.

At the next digit 1 we must attach the previous digit 0 from the main branch. If we do this we get the pair 01. Now we have:

Take the 01 and extend right by 1 to get 011.

The two remaining digits [01] then give 101 and 011.

Figure 8.3-2 shows how this graftal builds up over 6 generations.

Now that this example has clarifed the principles for constructing graftals, we will exhibit some pictures of such structures.

In Figure 8.3-4 we show the development of a graftal from the 4th to the 12th generation. A sequence up to the 13th generation is shown in Figure 8.3-5.

We should point out that the development of graftal structures takes a lot of computation - hence time. In fact, the later generations can tie up your computer for an entire night. Your computer may also exceed the available RAM storage capacity (about 1 MB), so that generations this high cannot be computed (see the description in §11.3).

**Figure 8.3-3** Graftal-plant.

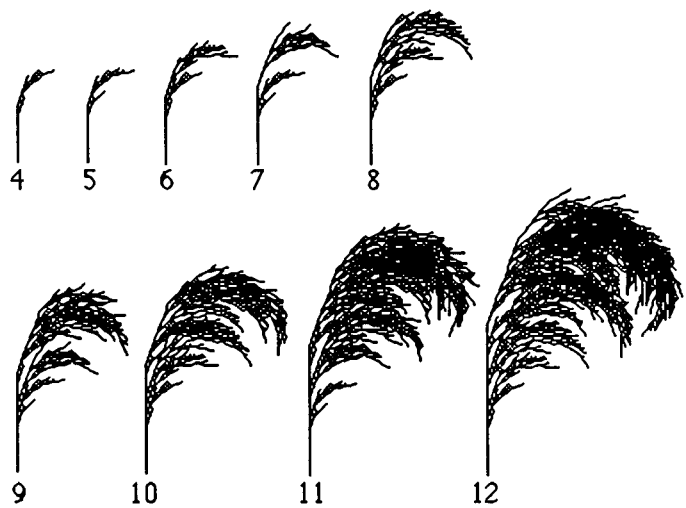



**Figure 8.3-4** Development of a graftal from the 4th to the 12th generation.

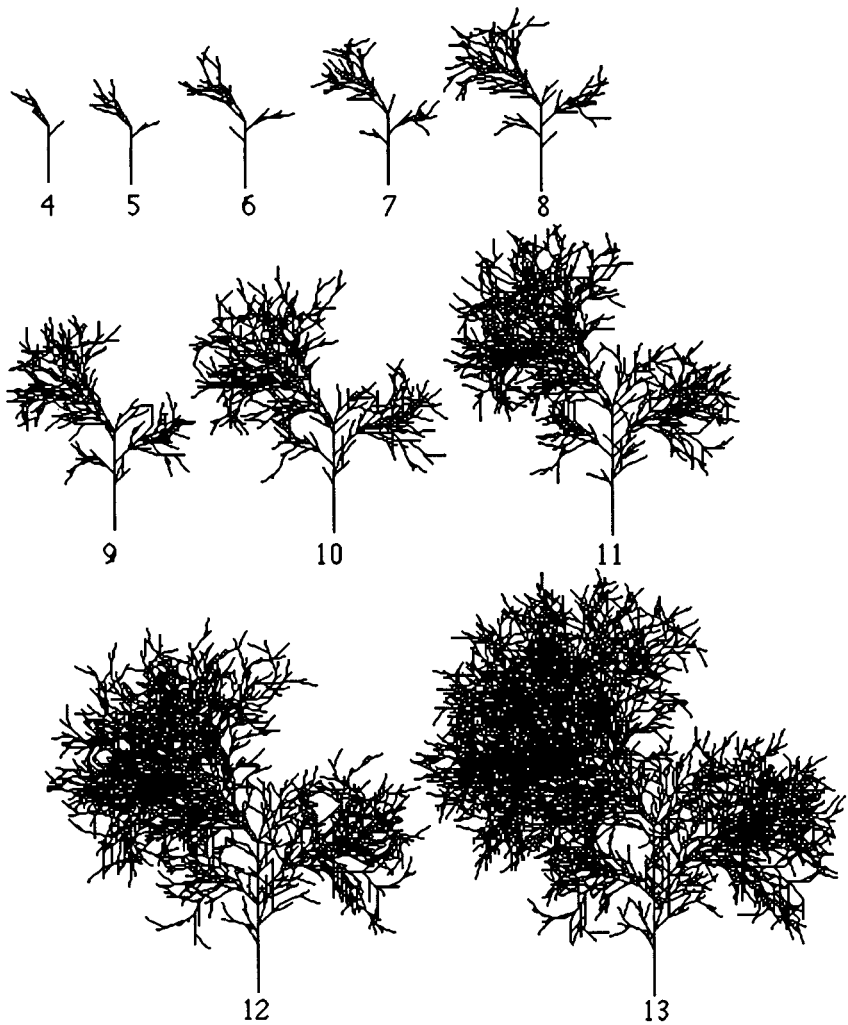



**Figure 8.3-5** A graftal from the 4th to the 13th generation.

## Computer Graphics Experiments and Exercises for §8.3

The experimental field of graftals is still not widely known. The following examples will get you started. A program for graphical representation is given in §11.3.[1]

**Exercise 8.3-1**

Experiment with graftals of the following structure:
Rule:0.1.0.11.[01].0.00[01].0.0
Angle: -40,40,-30,30
Number of generations: 10.

**Exercise 8.3-2**

Experiment with graftals of the following structure:
Rule: 0.1.0.1.0.10 [11].0.0
Angle: -30,20,-20,10
Number of generations: 10.

**Exercise 8.3-3**

Experiment with graftals of the following structure:
Rule:0.1[1].1.1.0.11.1.0
Angle: -30,30,-15,15,-5,5
Number of generations: 10.

**Exercise 8.3-4**

Experiment with graftals of the following structure:
Rule:0.1[1].1.1.0.11.1.0
Angle: -30,30,-20,20
Number of generations: 10.

**Exercise 8.3-5**

Experiment with graftals of the following structure:
Rule:0.1[01].1.1.0.00[01].1.0
Angle: -45,45,-30,20
Number of generations: 10.

**Exercise 8.3-6**

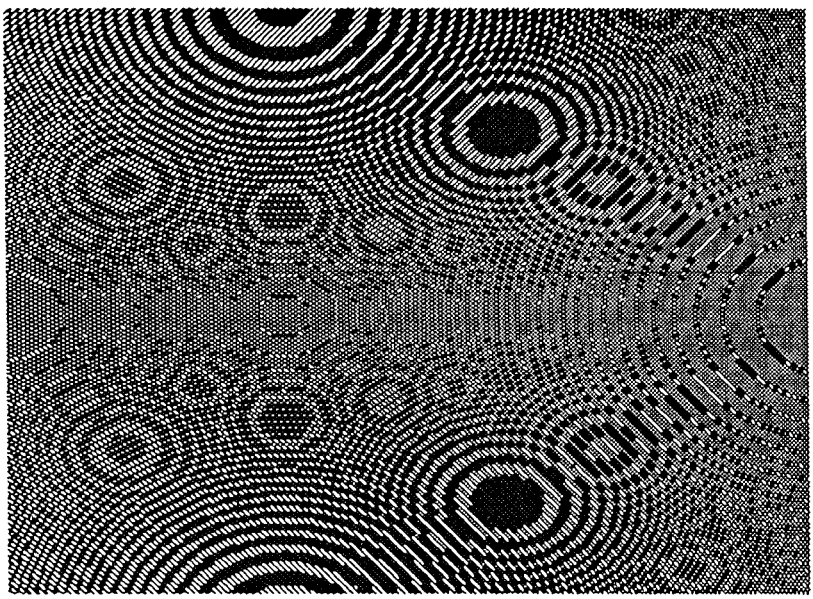
Vary the above examples in any way you choose, changing the production rule, the angle, or the number of generations.

[1]The idea of graftals has been known for some time in the technical literature; see Smith (1984), SIGGRAPH (1985). We first thought about carrying out this type of experiment on a PC after reading a beautiful essay about them. In this section we have oriented ourselves following the examples in that article: Estvanik (1986), p. 46.

## 8.4 Repetitive Designs

Now things get repetitive. What in the case of graftals required the endless application of production rules, resulting in ever finer structure, can also be generated by other - simpler - rules. The topic in this section is computer-graphical structures which can be continued indefinitely as 'repetitive designs' - rather like carpets. The generating rules are not production rules, but algorithms constructed in the simplest manner. The structures that are generated are neither fractals nor graftals, but 'repetitals', if you wish.

We found the simple algorithms in the Computer Recreations column of *Scientific American*; see Dewdney (1986b). Naturally we immediately began to experiment. The pictures we produced will not be concealed any longer:



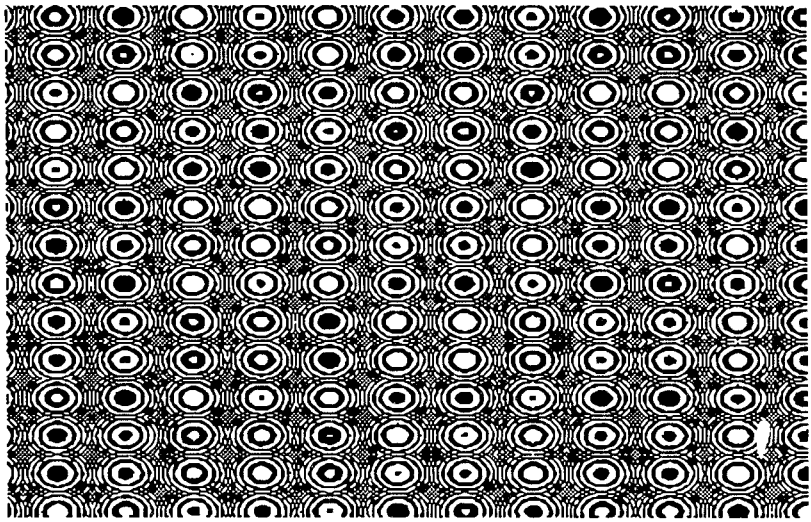
Data: 0, 10, 0, 20

**Figure 8.4-1** Interference pattern 1.

The program listing for Figures 8.4-1 to 8.4-3 is very simple:

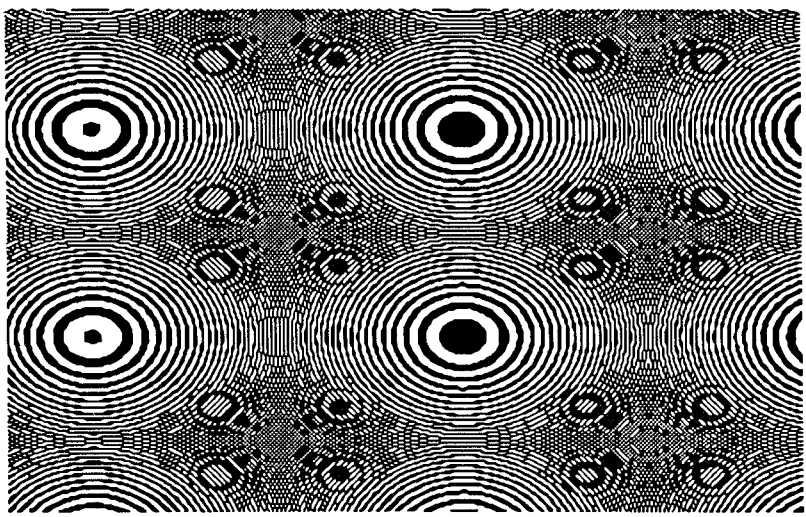
**Program Fragment 8.4-1**

```
PROCEDURE Conett;
   VAR
      i, j, c: integer;
      x, y, z: real;
BEGIN
```



Data: 0, 30, 0, 100

**Figure 8.4-2** Interference-pattern 2



Data: 0, 50, 0, 80

**Figure 8.4-3** Interference-pattern 3.

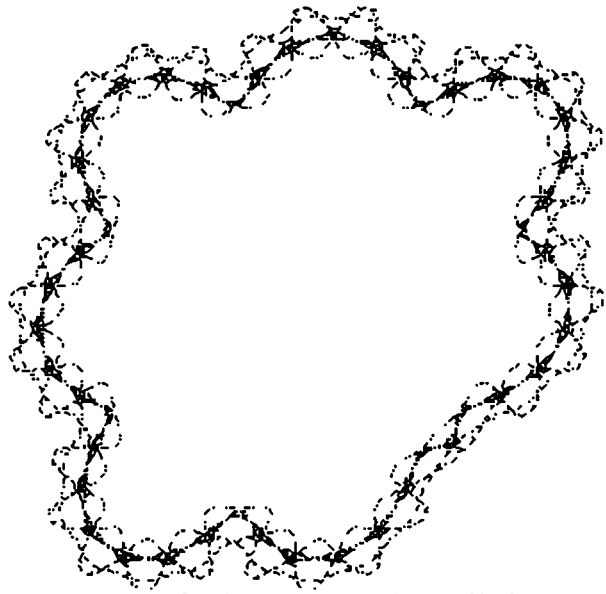



**Figure 8.4-4** Garland.

```
FOR i := 1 TO Xscreen DO
   FOR j := 1 TO Yscreen DO
   BEGIN
      x := Left + (Right-Left)*i/Xscreen;
      y := Bottom + (Top - Bottom) * j / Yscreen;
      z := sqr(x) + sqr(y);
      IF trunc (z) < maxInt THEN
      BEGIN
         c := trunc (z);
         IF NOT odd (c) THEN SetPoint (i,j);
      END;
   END;
END;
```

Input data for `Left`, `Right`, `Bottom`, `Top` are given in the figures. Again the richness of form obtained by varying parameters is astonishing. The idea for this simple algorithm is due to John E. Conett of the University of Minnesota; see Dewdney (1986b).

A quite different form of design can be obtained with Barry Martin's algorithm (Figures 8.4-4ff.). Barry Martin, of Aston University in Birmingham, devised a method

which is just as simple as the above method of John Conett. It depends on two simple formulas, which combine together the sign, absolute value, and root functions. The sign function has the value +1 or -1, depending on whether the argument $x$ is positive or negative. If $x = 0$ then the sign function equals 0.

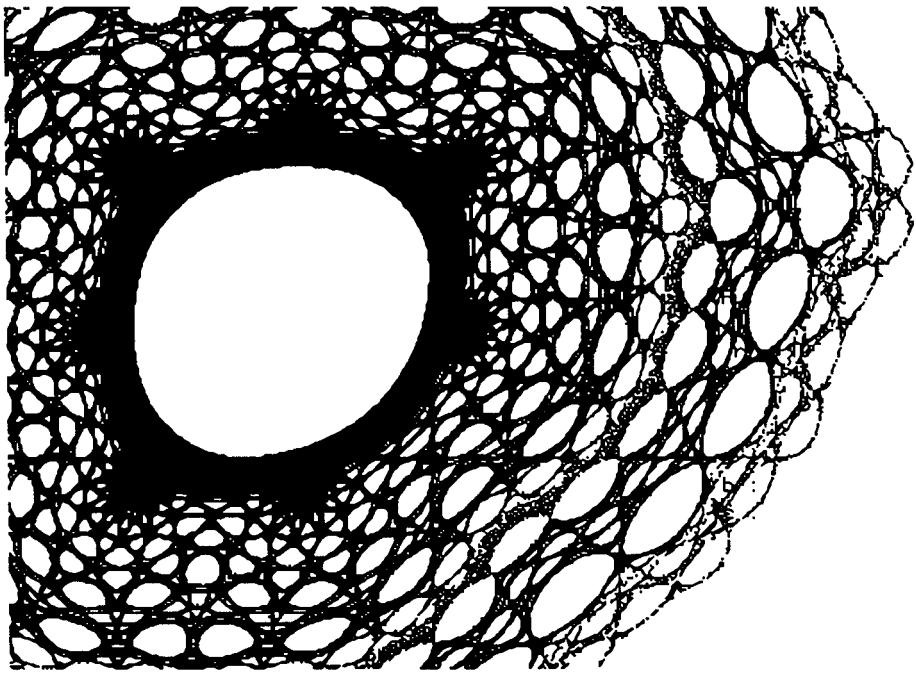


**Figure 8.4-5** Spiderweb with $a = -137$, $b = 17$, $c = -4$, $n = 1\ 898\ 687$.

The program listing for Figs 8.4-4ff. is as follows:

**Program Fragment 8.4-2**

```
FUNCTION sign (x: real) : integer;
BEGIN
   sign := 0;
   IF x <> 0 THEN
      IF x < 0 THEN
         sign := -1
      ELSE IF x > 0 THEN
         sign := 1;
END;
```

```
PROCEDURE Martin1;
   VAR
      i,j : integer;
      xOld, yOld, xNew, yNew : real;
BEGIN
   xOld := 0;
   yOld := 0;
   REPEAT
      SetUniversalPoint (xOld, yOld);
      xNew := yOld - sign (xOld)*sqrt (abs (b*xOld-c));
      yNew := a - xOld;
      xOld := xNew;
      yOld := yNew;
   UNTILButton;
END;
```

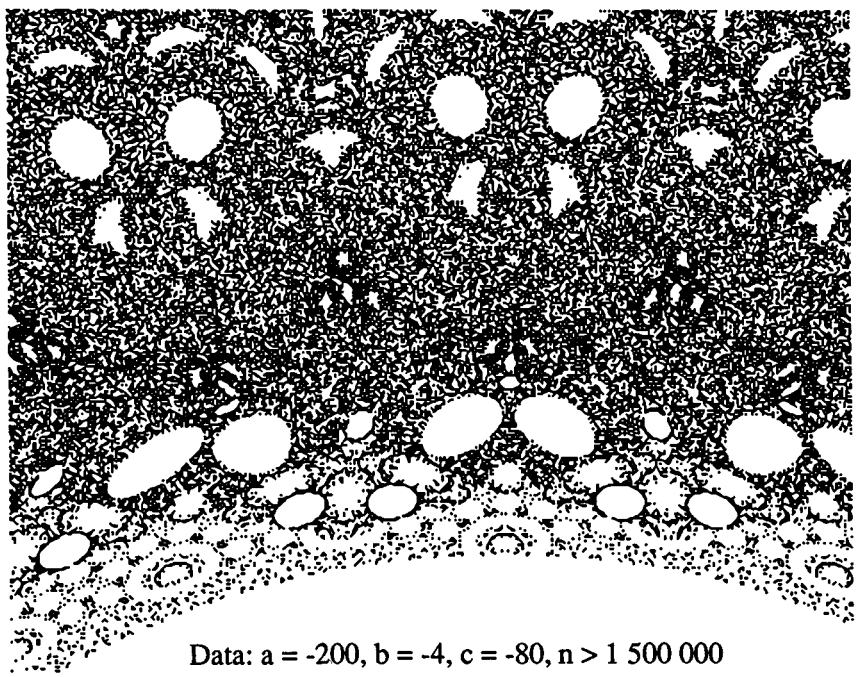



**Figure 8.4-6** Cell culture.

With these completely different pictures we bring to and end our computer-graphical experiments, and once more recommend that you experiment for yourself.

In previous chapters you have been confronted with many new concepts and meanings. It all began with experimental mathematics and measles. We reached a provisional end with fractal computer graphics and now with carpet designs. Other aspects will not be discussed further.

Until now we have made no attempt to structure our discoveries in this new science, on the border between experimental mathematics and computer graphics. We attempt this in the next chapter under the title 'step by step into chaos'. After a glance back to the 'land of infinite structures', our investigations of the relation between order and chaos will then come to an end.

In the chapters after these (Chapter 11 onwards), we turn to the solutions to the exercises, as well as giving tricks and tips which are useful for specific practical implementations on various computer systems.

## Computer Graphics Experiments and Exercises for §8.4

### Exercise 8.4-1

Implement program listing 8.4-1 and experiment with different data in the range [0,100] for the input size.

Try to find how the picture varies with parameters.

Which parameters produce which effects?

### Exercise 8.4-2

Implement program listing 8.4-2 and experiment with different data for the input variables $a$, $b$, $c$.

Try to find how the picture varies with parameters.

Which parameters produce which effects?

### Exercise 8.4-3

In program listing 8.4-2 replace the statement

```
xNew := yOld - sign (xOld)*sqrt (abs (b*xOld-c));
```

by

```
xNew := yOld - sin (x);
```

and experiment (as in Exercise 8.4-2).