9.8 RANDOM FRACTALS

Chaos and chance are words to describe phenomena of which we are ignorant. Sven G. Carlson

The fractal shapes described so far are completely deterministic: No random elements are used in generating them, and their shapes are completely predictable (although very complicated). In graphics, the term *fractal* has become widely associated with randomly generated curves and surfaces that exhibit a degree of self-similarity. These curves are used to provide "naturalistic" shapes for representing objects such as coastlines, rugged mountains, grass, and fire.

9.8.1 Fractalizing a Segment

Perhaps the simplest random fractal is formed by recursively roughening or "fractalizing" a line segment. At each step, each line segment is replaced with a "random elbow." Figure 9.56 shows this process applied to the line segment S having endpoints A and B. S is replaced by the two segments from A to C and from C to B. For a fractal curve, point C is randomly chosen along the perpendicular bisector L of S. The elbow lies randomly on one or the other side of the "parent" segment AB.

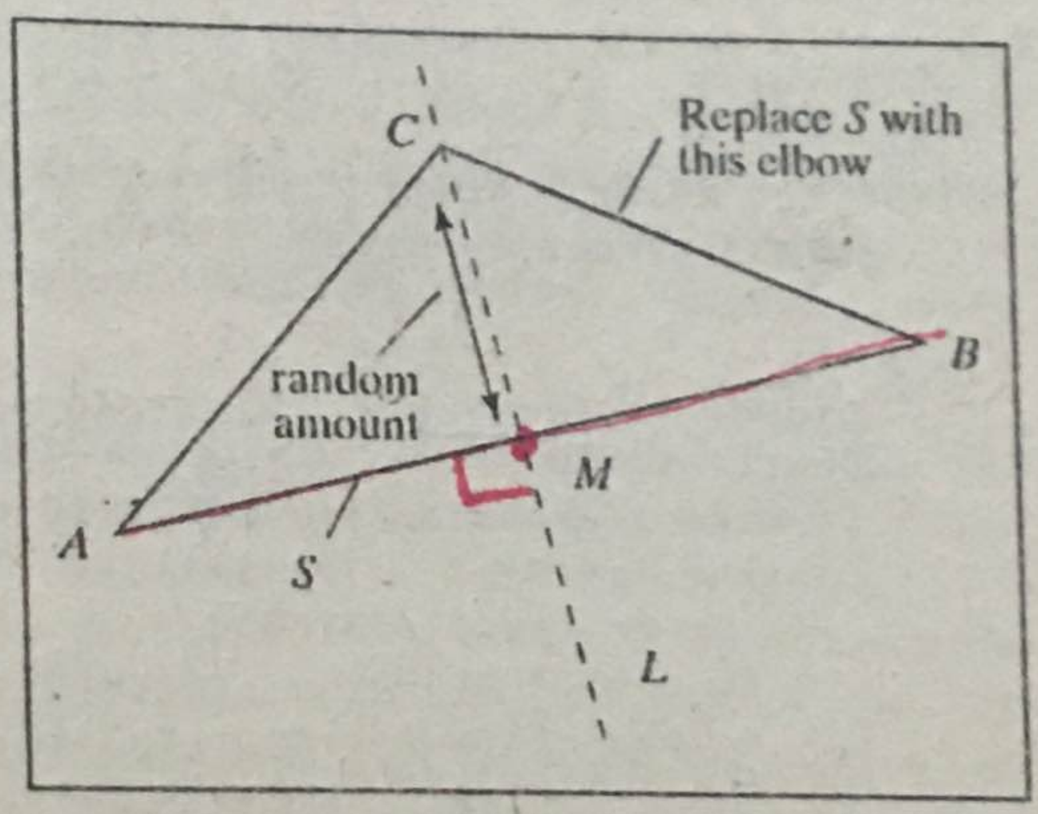


FIGURE 9.56 Fractalizing with a random elbow.

Three stages in the fractalization of a segment are shown in Figure 9.57. In the first stage, the midpoint of AB is perturbed to form point C. In the next stage, each of the two segments has its midpoint perturbed to form points D and E. In the final stage, the new points $F \ldots I$ are added. (How many points are there in total after k stages?)

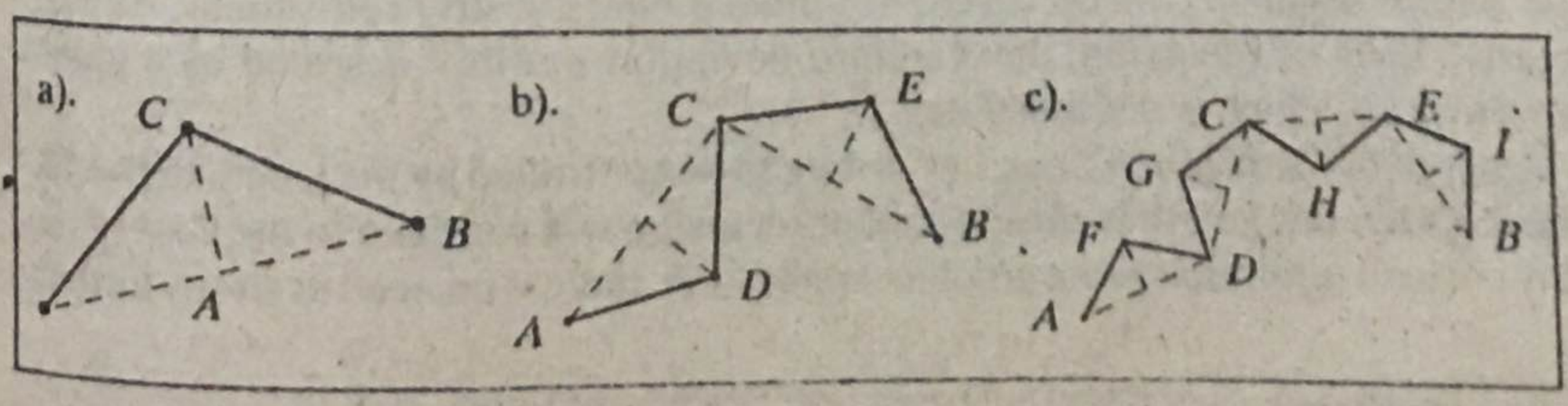


FIGURE 9.57 Steps in the fractalization process.

How do we actually perform fractalization in a program? In Figure 9.56, Line L passes through the midpoint M of segment S and is perpendicular to it. We saw in Chapter 4 that any point C along L has the parametric form

$$C(t) = M + (B - A)^{\perp} t, \tag{9.21}$$

for some value of t, where the midpoint $M = (A + B)/2$. The distance of C from M is $|B - A||t|$, which is proportional to both t and the length of S. So, to produce a point C on the random elbow, we let t be computed randomly. If t is positive, the elbow lies to one side of AB; if t is negative, it lies to the other side.

For most fractal curves, t is modeled as a Gaussian random variable with a zero mean and some standard deviation. Using a mean of zero causes, with equal probability, the elbow to lie above or below the parent segment.

The routine `fract()` shown in Figure 9.58 generates curves that approximate actual fractals. The routine recursively replaces each segment in a random elbow with a smaller random elbow. A simple stopping criterion is used: When the length of the segment is small enough (specifically, when the square of its length is smaller than some global value *minLenSq* set by the user), the segment is drawn using `cvs.lineTo()`, where `cvs` is a Canvas object (see Chapter 3). The variable t is made to be approximately Gaussian in its distribution by summing together 12 uniformly distributed random values lying between 0 and 1 [Knuth98] The result has a mean value of 6 and a variance of 1. The mean value is then shifted to 0, and the variance is scaled as necessary.

FIGURE 9.58 Fractalizing a line segment.

```
d fract(Point2 A, Point 2 B, double stdDev)
/ generate a fractal curve from A to B.
   double xDiff = A.x - B.x, yDiff = A.y - B.y;
   Point2 C;
   if(xDiff * XDiff + YDiff * yDiff < minLenSq)
         cvs.lintTo(B.x, B.y);
   else
   {
         stdDev *= factor;      // scale stdDev by factor
         double t = 0;
         // make a gaussian variate t lying between 0 and 12.0
         for(int i = 0; i , 12; i++)
               t += rand()/32768.0;
         t = (t - 6) * stdDev;   // shift the mean to 0 and sc
         C.x = 0.5 * (A.x + B.x) - t * (B.y - A.y);
         C.y = 0.5 * (A.y + B.y) + t * (B.x - A.x);
         fract(A, C, stdDev);
         fract(C, B, stdDev);
   }
```

Note that because the offset expressed in Equation (9.21) is proportional to the length of the parent segment, fractal curves are indeed (statistically) self-similar. At each successive level of recursion, the standard deviation `stdDev` is scaled by a global factor `factor`, which is discussed next.

The depth of recursion in `fract()` is seen to be controlled by the length of the line segment. Often this length is chosen so that recursion will continue to the limit of the display device's resolution. For a graphics application, there is no need to go any further.

PRACTICE EXERCISES

9.8.1 Controlling the recursion depth

An alternative method for stopping the recursive calls in `fract()` is to pass it a parameter `depth`, giving the maximum depth of recursion. Show how to alter `fract()` in order to stop the recursion when a certain depth is reached.

9.8.2 Fractalize to the resolution of the device

Lines are to be fractalized and displayed on a 512-by-512-pixel display. If you want to fractalize lines so that any horizontal or vertical lines will be fractalized to the resolution of the display, what is the maximum depth of recursion that needs to be used? ■

9.8.2 Controlling the Spectral Density of the Fractal Curve

Peitgen [Peitgen88] shows that the fractal curves generated by using the algorithm of Figure 9.58 have a "power spectrai density" given by

$$S(f) = 1/f^{\beta} \tag{9.22}$$

where β, the power (exponent) of the "noise process," is a parameter the user can set to control the "jaggedness" of the fractal noise. When β is 2, the process is known as **Brownian motion**, and when β is 1, the process is called "$1/f$ noise." $1/f$ noise is *self-similar* in a statistical sense, and has been shown to be a good model for many physical processes, such as clouds, the sequence of tones in music from many cultures, and certain crystal growth. Peitgen et al. also show that the fractal dimension of such processes is:

$$D = \frac{5 - \beta}{2} \tag{9.23}$$

In the routine `fract()` of Figure 9.58, the scaling factor `factor` by which the standard deviation is scaled at each level is based on the exponent β of the fractal curve, which determines how jagged the curve is. β varies between 1 and 3: Values larger than 2 lead to smoother ("persistent") curves, and values smaller than 2 lead to more jagged, "antipersistent" curves. The value of *factor* is given by [Peitgen88]:

$$\text{factor} = 2^{(1-\beta/2)} \tag{9.24}$$

Thus, factor decreases as β increases. Some sample values are shown in Figure 9.59.

Note that `factor` = 1 when $\beta = 2$, in which case the relative standard deviation does not change from level to level. This provides a model of **Brownian motion**. For persistent curves in which $\beta > 2$, we see that `factor` < 1, so the standard deviation diminishes from level to level: The offsets of the elbows become less and less pronounced statistically. On the other hand, when $\beta < 2$, antipersistent curves are formed: `factor` is greater than 1, so the standard deviation increases from level to level, and the elbows tend to become more and more pronounced.

factor	β	D	
1.4	1	2	1/f noise
1.1	1.6	1.7	
1.0	2	1.5	Brownian motion
0.9	2.4	1.3	
0.7	3	1	

FIGURE 9.59 How factor and D depend on β.

A fractal can be drawn as shown in Figure 9.60. In this routine, `factor` is computed using the C++ library function `pow(...)`.

---

[14] In some papers, the family of "noises" having spectral density $1/f^{\beta}$ for any β between .5 and 1.5 are *all* referred to as $1/f$ noise [Peitgen88].

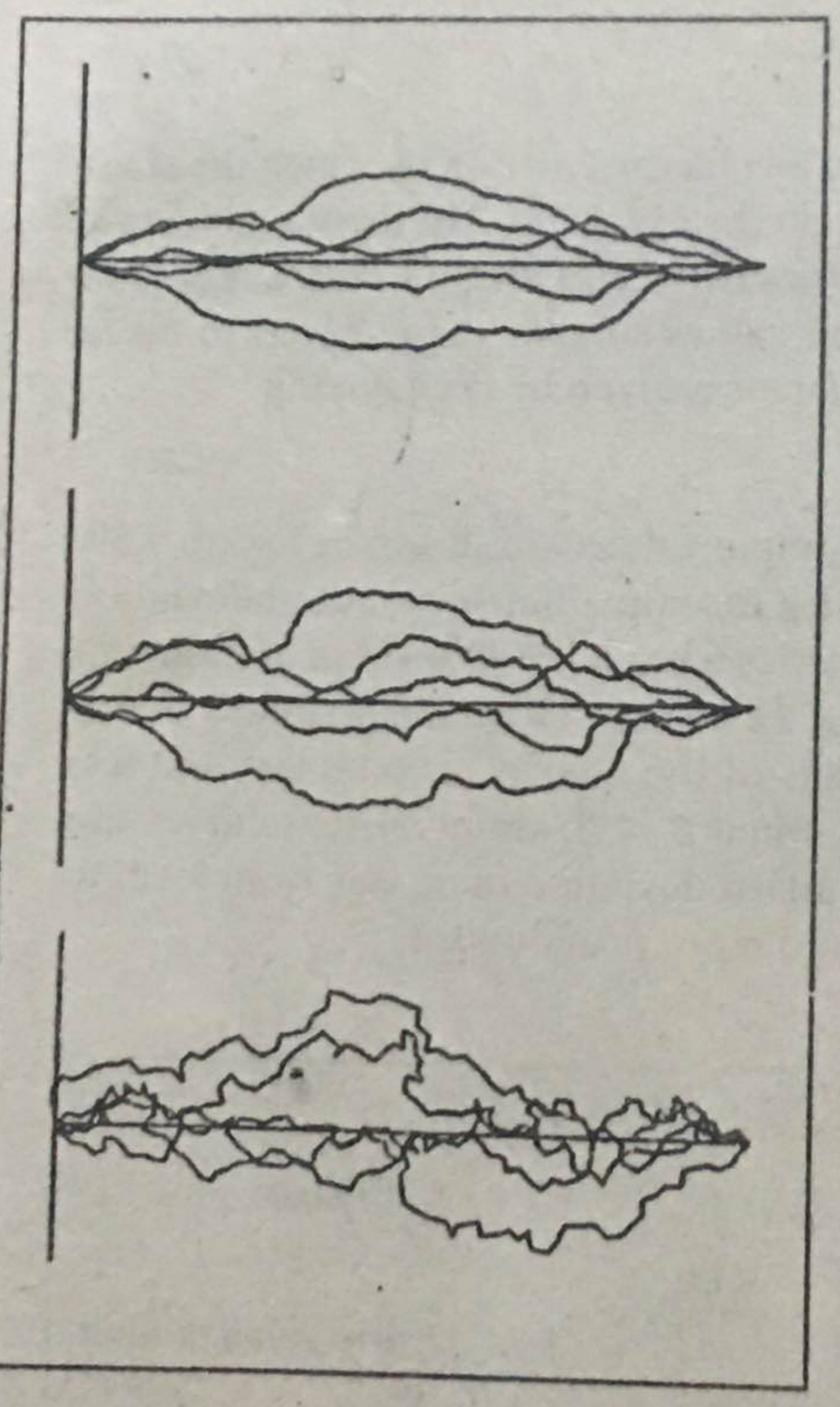
FIGURE 9.60 Drawing a fractal curve (pseudocode).

```
double MinLenSq, factor; // global variables

void drawFractal(Point2 A, Point2 B)
{
      double beta, StdDev;
      user inputs beta, MinLenSq, and the initial StdDev
      factor = pow(2.0,(1.0 - beta)/2.0);
      cvs.moveTo(A);
      fract(A, B, StdDev);
}
```

Some examples fractal curves are shown in Figure 9.61 for various values of β based on a line segment of unit length. (`minLenSq` was set to .05 and `stdDev` to 1 for these figures.) In each case, five fractal curves were generated between the same endpoints to show the possible variations. Note the pronounced effect of changing the exponent β of the fractals.

FIGURE 9.61 Examples of fractal curves. (a) $\beta = 2.4$; (b) $\beta = 2$; (c) $\beta = 1.6$.



Arbitrary polylines can be fractalized, as shown in Figure 9.62, providing a simple means of roughening a shape to make it look naturally rugged. In the figure, the coastline of Florida was crudely digitized, so that the state is approximated by only 20 points [Figure 9.62(a)]. Then each of the segments of the coastline of Florida was fractalized, whereas the inland state borders that should be straight were left unfractalized. It is fascinating to fractalize polygons that represent known geographic entities such as islands, countries, or continents, to see how natural their various borders can be made to look. One can also fractalize other shapes, such as animals and the top of a person's head, to give them a natural appearance.

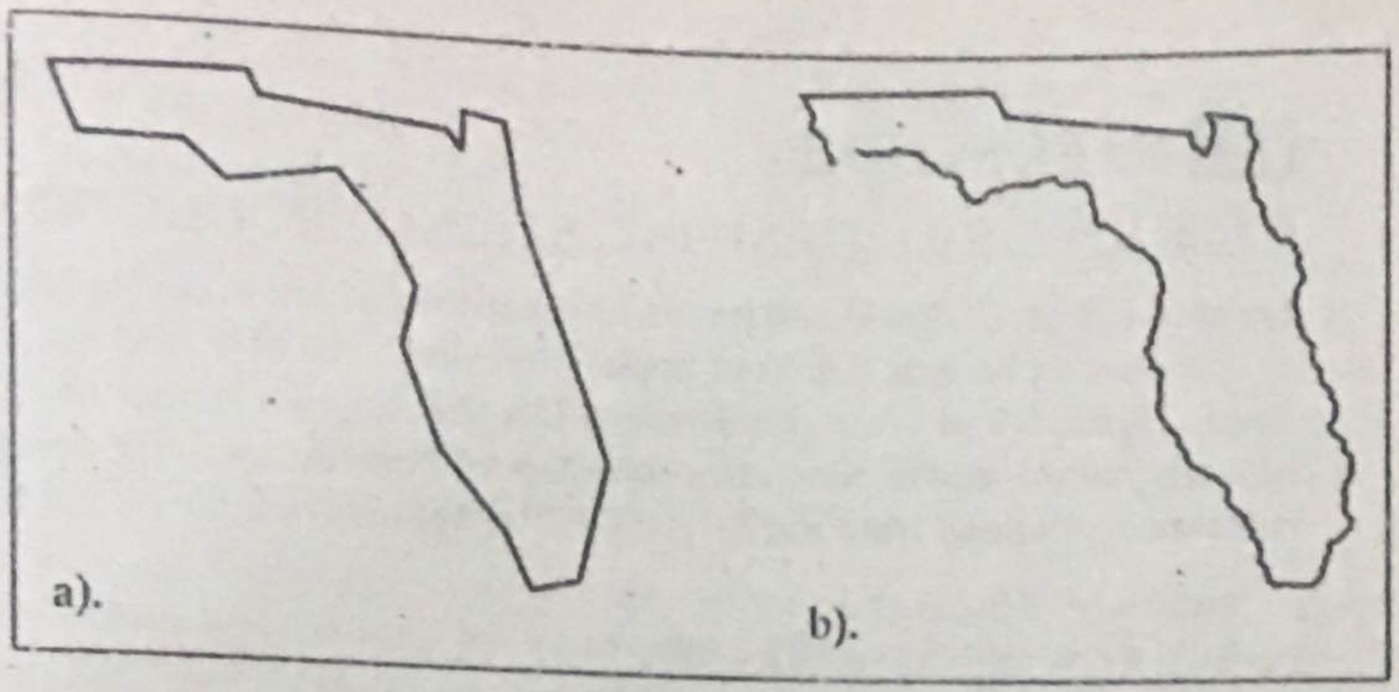


FIGURE 9.62 Generating realistic coastlines.

One of the features of fractal curves generated by pseudorandom-number generators is that they are completely repeatable. All that is required is to use the same seed each time the curve is fractalized. Each of the three sets of fractals in Figure 9.61 used the same starting seed, so they are based on exactly the same sequence of random values (except for a scaling factor based on their different variances). In this manner, a complicated shape such as the fractalized coastline can be completely described in a database by storing only

- the polypoint that describes the original line segments,
- the values of `minLenSq` and `stdDev`, and
- the seed.

From this modest amount of data, an extract replica of the fractalized curve can be regenerated at any time.

Fractal Surfaces

Surfaces, as well as lines, can be fractalized in order to generate realistic-looking mountainous terrain. Plate 31 shows some spectacular examples. Methods for generating such scenes are discussed in Case Study 9.9.

9.9 SUMMARY

This chapter examined some approaches to the infinite, using repetition and recursion. With certain figures, we can move "outward" toward infinity by placing the figures side by side forever, thus tiling the entire infinite plane. It is also possible to go "inward" toward the infinitely small, at least in principle. Recursion provides a simple mechanism for drawing ever smaller "child" versions of a figure inside the "parent" figure, down to the resolution of the display. If the proper class of figures is chosen, the patterns will become self-similar fractals: No matter how closely they are scrutinized, they will exhibit the same level of detail. Randomness can be used to provide an automatic roughness for drawing coastlines, trees, and other natural features.

An iterated function system (IFS) is a set of affine transformations that can be used to generate complex images. Playing the Chaos Game, you can generate the "strange attractor" of an IFS through a simple process of repetitively choosing one of the affine maps at random and mapping a single point. The process can be turned around, and a given image can be analyzed to produce an IFS having the image as its attractor. Since it takes so many fewer bytes to represent an IFS than to store an image, the approach has generated a great deal of interest in the field of image compression.

We explored the beguiling Mandelbrot set, whose boundary is itself a fractal curve. As one zooms in on a region near the boundary, new details emerge, and the wart pattern seems to reproduce itself over and over again. Because this process can continue forever, the Mandelbrot set is "infinitely complicated." Computer graphics provides a simple and powerful tool for exploring phenomena such as these. The pictures are generated simply, yet they can reveal enormous complexity. We also looked at how the Julia sets are formed and can be drawn algorithmically and how the Mandelbrot set is a "map" into the entire family of Julia sets.