16

PROCEDURAL FRACTAL TERRAINS

F. KENTON MUSGRAVE

As pointed out in Chapter 14, the same procedural constructions that we use as textures can also be used to create terrains. The only difference is that, instead of interpreting what the function returns as a color or other surface attribute, we interpret it as an altitude. This chapter will now extend the discussion of terrain models begun in Chapter 14.

Since we are designing these functions to generate terrain models external to the renderer or as QAEB primitives built into the renderer, we're switching back from the RenderMan shading language to C code for the code examples.

ADVANTAGES OF POINT EVALUATION

I first started working with procedural textures when I used them to color fractal terrains and to provide a sense of "environment," with clouds, water, and moons, as described earlier. Figure 15.8 is an example of this early work. The mountains I was making then were created with a version of polygon subdivision (hexagon subdivision) described by Mandelbrot in an appendix of *The Science of Fractal Images* (Peitgen and Saupe 1988). They have the jagged character of polygon subdivision terrains and the same-roughness-everywhere character of a homogeneous fractal dimension. Mandelbrot and I were working together at the time on including erosion features in our terrains. This led me to make some conjectures about varying the local behaviors of the terrain, which led to the two multifractal constructions I will describe next. Interestingly, I had never heard of "multifractals" when I devised these first two additive/multiplicative hybrid multifractal functions. When I showed Mandelbrot Figure 16.1 in 1991, he exclaimed in surprise, "A multifractal!" to which I astutely replied, "What's a multifractal?"[1]

What allowed me to intuitively "reinvent the (multifractal) wheel" was the flexibility implicit in our noise-based procedural fractals. Dietmar Saupe calls our Perlin

1. His cryptic retort was, "Never mind—now is not the time."

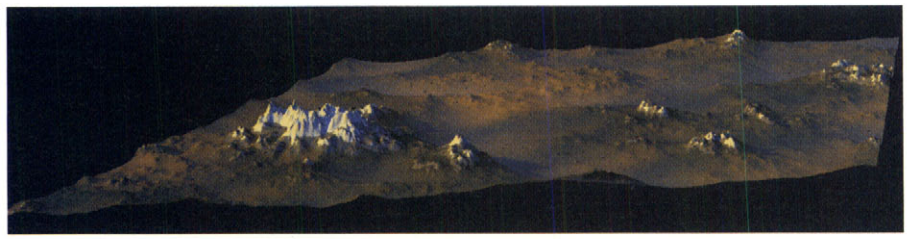


FIGURE 16.1 A multifractal terrain patch. Note the heterogeneity: plains, foothills, and mountains, all captured in a single fractal model. Copyright © 1994 F. Kenton Musgrave.

noise-based procedural fractal construction method "rescale and add" (Saupe 1989). Its distinguishing feature, he points out, is point evaluation: the fact that each sample is evaluated at a point in space, without reference to any of its neighbors. This is quite a distinction indeed, in the context of fractal terrain generation algorithms. In polygon subdivision, a given altitude is determined by a series of interpolations between neighboring points at lower frequencies (i.e., earlier steps in the iterative construction). In Fourier synthesis, the entire terrain patch must be generated all at once; no sample can be computed in isolation. In contrast, the context independence of our procedural method allows us to do whatever we please at any given point, without reference to its neighbors. There *is* interpolation involved, but it has been hidden inside the noise function, where it takes the form of Hermite spline interpolation of the gradient values at the integer lattice points (see Chapters 2, 6, and 7 for details on this). In practice, you could employ the same tricks described below to get multifractals from a polygon subdivision scheme, at least. It's not so obvious how you could accomplish similar effects with Fourier synthesis. The point is, I probably never would have thought of these multifractal constructions had I not been working in the procedural idiom.

Another distinguishing feature of terrains constructed from the noise function is that they can be rounded, like foothills or ancient mountains (see Figure 16.2). To obtain this kind of morphology from polygon subdivision, we must resort to the complexities of schemes like Lewis's "generalized stochastic subdivision" (Lewis 1987). The rounded nature of our terrains has to do with the character of the basis function; more on that later. And, as we have already shown, another distinguishing characteristic of the procedural approach is that it naturally accommodates adaptive band-limiting of spatial frequencies in the geometry of the terrain as required for rendering with adaptive level of detail, as in QAEB rendering (described in the next

FIGURE 16.2 *Carolina* illustrates a procedural model of ancient, heavily eroded mountains. Copyright © 1994 F. Kenton Musgrave.

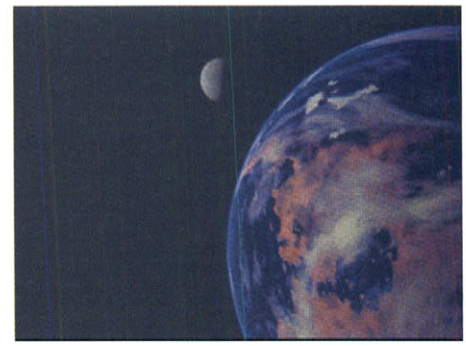
chapter). Such capability makes possible exciting applications like the planetary zoom seen in Figure 16.3.

THE HEIGHT FIELD

Terrain models in computer graphics generally take the form of *height fields*. A height field is a two-dimensional array of altitude values at regular intervals (or *post spacings*, as geographers call them). So it's like a piece of graph paper, with altitude values stored at every point where the lines cross.[2]

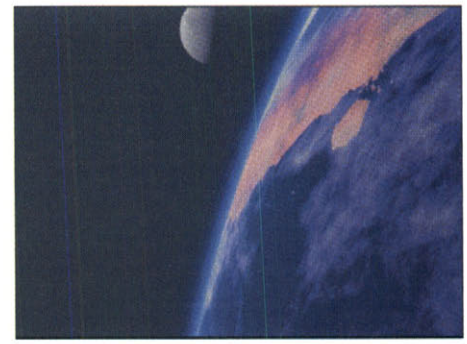
2. This is the simplest, but not the most efficient, storage scheme for terrains. Extended areas of nearly constant slope can be represented with fewer samples, for instance. Decimation algorithms (Schroeder, Zarge, and Lorensen 1992) are one way to reduce the number of samples in a height field and thus its storage space requirement. It may be desirable to resample such a model before rendering, however, to get back to the regular array of samples that facilitates fast rendering schemes such as grid tracing. For an animation of this zoom, see *www.kenmusgrave.com/animations.html*.

(a)



(b)

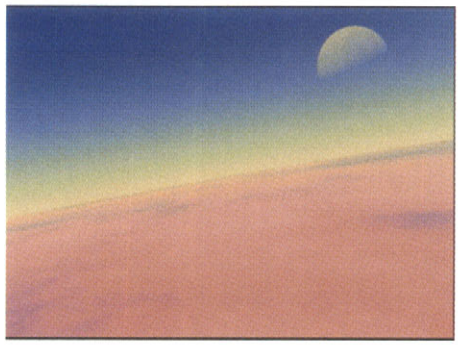
(c)



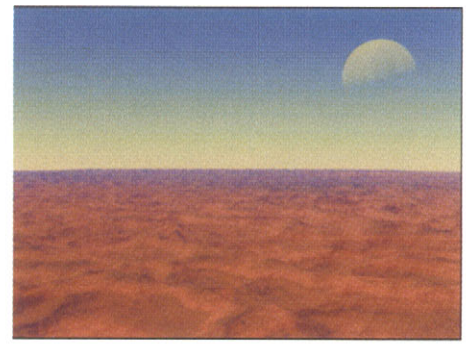
(d)

(e)

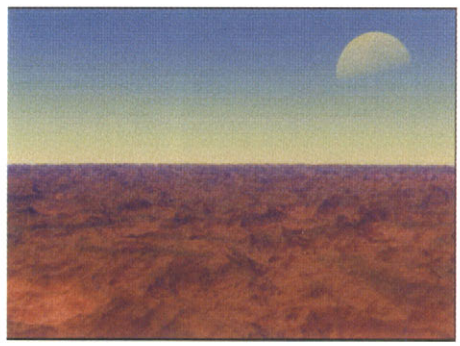
FIGURE 16.3 A series of frames from the *Gaea Zoom* animation demonstrate the kind of continuous adaptive level of detail possible with the models presented in the text. The MPEG animation is available at *www.kenmusgrave.com/animations.html*. Copyright © F. Kenton Musgrave.



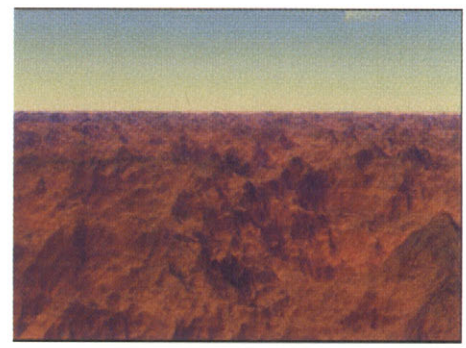
(f)



(g)



(h)



(i)

There is one, and only one, altitude value at each grid point. Thus there can be no caves or overhangs in a height field. This limitation may seem easy to overcome, but it's not. In fact, the problem is hard enough that I issued a challenge in the first edition of this book, back in 1994, offering $100 to the first person to come up with an "elegant, general solution" to it. The reward was won in early 1998 by Manuel Gamito, who came over from Portugul to work with us in the MetaCreations Skunk Works for a year. Manuel's award-winning image, an entirely procedural model rendered in a modified version of the minimal ray tracer I wrote to develop the QAEB algorithm, appears in Figure 17.2. Unfortunately, that image took about a day to render!

The regular (i.e., evenly spaced) samples of the height field accommodate efficient ray-tracing schemes such as *grid tracing* (Musgrave 1988) and quad tree (Kajiya 1983a) spatial subdivision. A detailed discussion of such a rendering scheme is beyond the scope of this book; if you're interested in that, see Musgrave (1993). I've always preferred to ray-trace my landscapes, but if you lack the computational horsepower for that, there are some very nice non-ray-tracing terrain renderers, such as Vistapro, available for home computers. If you'd like to try your hand at ray-tracing height fields, you can buy MetaCreations' Bryce, Animatek World Builder, or World Tool Set. Or you can pick up Craig Kolb's public domain Rayshade ray tracer, which features a very nice implementation of hierarchical grid tracing for height fields. The hierarchical approach captures the best aspects of grid tracing (i.e., low memory overhead) and of quadtree methods (i.e., speed). For multiple renderings of a static height field—as in fly-by animations—the PPT algorithm is the fastest rendering method (Paglieroni 1994).

There are several common file formats for height field data. There is the DEM (digital elevation map) format of the U.S. Geological Survey (USGS) height fields, which contain measured elevation data corresponding to the "quad" topographic maps available from the USGS, which cover the entire United States. The U.S. military establishment has their DTED (digital terrain elevation data) files, which are similar, but are likely to include terrains outside of the United States and its territories. While you may render such data as readily as synthetic fractal terrains, as a synthesist (if you will), I consider the use of "real" data to be cheating! My goal is to synthesize a detailed and familiar-looking reality, entirely from scratch. Therefore, I have rarely concerned myself with measured data sets; I have mostly worked with terrains that I have synthesized myself.

As I have usually worked alone, with no programming assistance, I generally prefer to implement things in the quickest, simplest manner I can readily devise so that I can get on to making pictures. Thus my home-grown height field file format (which is also used by Rayshade) is very simple: it is a binary file containing first a single integer (4 bytes), which specifies the size of the (square) height field, followed by n^2 floats (4 bytes each), where n is the value of the leading integer. I append any additional data I wish to store, such as the minimum and maximum values of the height field, and perhaps the random number generator seed used to generate it, after the elevation data. While far more compact than an ASCII format for the same data, this is still not a particularly efficient storage scheme. Matt Pharr, of ExLuna, has implemented an improved file format, along with conversion routines from my old format to his newer one. In the new scheme, there is a 600-byte header block for comments and documentation of the height field. The elevation data is stored as shorts (2 bytes), with the values normalized and quantized into integers in the range

$[0, 2^{16} - 1]$. The minimum and maximum altitudes over the height field are also stored, so that the altitude values may be restored to their floating-point values at rendering time by the transformation

$$z = \frac{a(z_{max} - z_{min})}{2^{16} - 1} + z_{min}$$

where a is the quantized and scaled altitude value, z is the decoded floating-point value, and z_{min} and z_{max} are the min/max values of the height field. Pharr's code also obviates the big-endian/little-endian byte-order problem that can crop up when transferring binary files between different computers, as well as automatically taking care of transfers between 32-bit and 64-bit architectures. Pharr's code is available on the Internet via anonymous ftp at *cs.princeton.edu*. If you intend to render many height fields, it is worth picking up, as it saves about half of the space required to store a given height field.

HOMOGENEOUS fBm TERRAIN MODELS

The origin of fractal mountains in computer graphics is this: Mandelbrot was working with fBm in one dimension (or one-point-something dimensions, if you must), like the plot we saw in Figure 14.2. He noted that, at a fractal dimension of about 1.2 (the second trace from the top in Figure 14.2), the trace of this function resembled the skyline of a jagged mountain range. In the true spirit of ontogenetic modeling, he reasoned that, if this function were extended to two dimensions, the resulting surface should resemble mountains. Indeed it did, and thus were born fractal mountains for computer graphics. Figure 16.4 is a crude example of such a simple fractal terrain model.

Again, there is no known causal relationship between the shape of real mountains and the shape of this fractal function; the function simply resembles mountains, and does so rather closely. Of course there are many features in real mountains, such as drainage networks and other erosion features, that are not present in this first model. Much of my own research has been toward including such features in synthetic terrain models, largely through procedural methods (Musgrave 1993).

Fractal Dimension

As pointed out in Chapter 14 and as illustrated by Figure 14.2, fractal dimension can be thought of as a measure of the *roughness* of a surface. The higher the fractal dimension, the rougher the surface. Figure 16.5 illustrates how the roughness of an

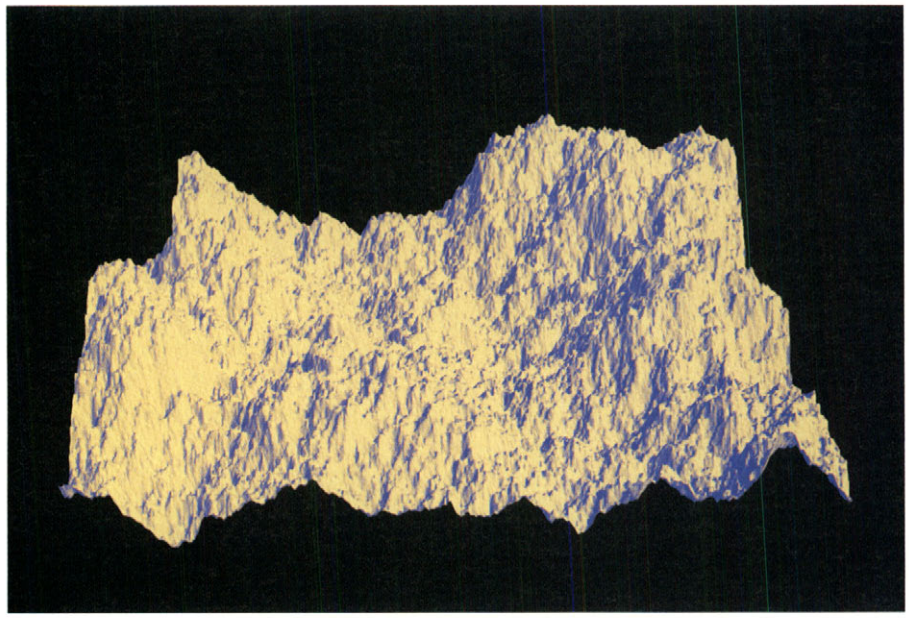


FIGURE 16.4 A terrain patch with homogeneous fractal dimension (of ~2.2).

fBm surface varies with fractal dimension: at the left edge of the patch, the fractal dimension is 2.0; on the right it is 3.0. The most interesting thing about this patch is that it is not planar (i.e., flat) on the left, nor does it fill all of the space on the right. So we see that the formal definition of fractal dimension for fBm does not capture all of the useful fractal behavior available from the construction: the kind of rolling foothills that would occur off the left end of this patch are indeed self-similar and thus fit our heuristic definition of "fractal." Yet they do not fit the formal mathematical definition of fractal dimension (at least not the one for fBm).[3] This is a good example of how fractals defy precise definition and sometimes require that we "paint with a broad brush" so that we don't unnecessarily exclude relevant phenomena. Many researchers in computer graphics and other fields have substituted terms such as "stochastic" and "self-similar" for "fractal" because of this poor fit with formal definitions, but this is probably not appropriate: there are few useful stochastic

3. It's worth noting that different methods for measuring fractal dimension may give slightly different results when applied to the same fractal. So even formal methods may not agree about the limits of fractal behavior and the exact values of quantitative measurements of fractal behavior.

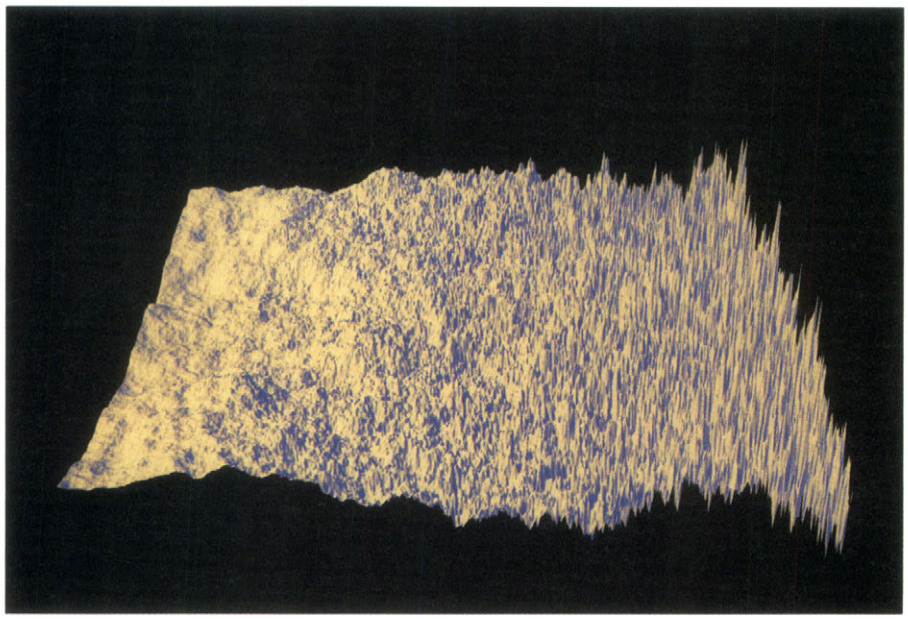


FIGURE 16.5 In this patch, the fractal dimension varies from 2.0 on the left to 3.0 on the right.

models of visual natural phenomena that do not feature self-similarity, and self-similar models are best characterized as fractal, formal technicalities notwithstanding.

Visual Effects of the Basis Function

As illustrated in the previous chapter, the small spectral sums used to create random fractals for computer graphics allow the character of the basis function to show through clearly in the result. Usually, the choice of basis function is implicit in the algorithm: it is a sine wave for Fourier synthesis, a sawtooth wave in polygon subdivision, and a piecewise-cubic Hermite spline in noise-based procedural fBm. You could use a Walsh transform and get square waves as your basis. Wavelets (Ruskai 1992) promise to provide a powerful new set of finite basis functions. And again, sparse convolution (Lewis 1989) or fractal sum of pulses (Lovejoy and Mandelbrot 1985) offer perhaps the greatest flexibility in choice of basis functions. With those methods, you could even use the profile of the kitchen sink as a basis function, leading naturally to sinkholes in the terrain.

Gavin Miller (1986) showed that the creases in terrain constructed with the most common form of polygon subdivision (i.e., subdivision of an equilateral triangle) are really an artifact of the interpolation scheme implicit in the subdivision algorithm. But I think that for the most part it has simply been overlooked that there is a basis function implicit in *any* fBm construction and that the character of that basis function shows through in the result. As shown in the previous chapter, we can use this awareness to obtain certain aesthetic effects when designing both textures and terrains.

The smoothness of the Hermite spline interpolant in the noise function allows us to generate terrains that are more rounded than those commonly seen in computer graphics previously. Figures 15.7, 16.2, and 20.9 illustrate this well. Other examples of basis function effects are seen in Figures 15.15, 16.6, 18.3, 20.18, and 20.20 and in Figures 17.4–17.6, where the ridged basis function was used to get a terrain with razorback ridges at all scales. Note that this terrain model can only be effectively rendered with adaptive level of detail, as with QAEB and other schemes (Bouville 1985; Kajiya 1983b). Without this, in a polygonal model rendered with perspective projection, nearby ridges would take on a saw-toothed appearance, as undersampled elevation values would generally lie on alternating sides of the ridgeline, and distant areas would alias on the screen due to undersampling of the complex terrain model. A nonpolygonal approach to rendering with adaptive level of detail, QAEB tracing, is described in the next chapter. It is ideal for rendering such sharp-edged terrain models.

HETEROGENEOUS TERRAIN MODELS

It would seem that before our 1989 SIGGRAPH paper (Musgrave, Kolb, and Mace 1989) it hadn't yet occurred to anyone to generate heterogeneous terrain models. Earlier published models had been monofractal, that is, composed of some form of fBm with a uniform fractal dimension. Even Voss's heterogeneous terrains (Voss 1988) represent simple exponential vertical scalings of a surface of uniform fractal dimension. As pointed out in Chapter 14, nature is decidedly not so simple and well behaved. Real landscapes are quite heterogeneous, particularly over large scales (e.g., kilometers). Except perhaps on islands, mountains rise out of smoother terrains—witness the dramatic rise of the Rocky Mountains from the relatively flat plains just to their east. Tall ranges like the Rockies, Sierras, and Alps typically have rolling foothills formed largely by the massive earthmovers known as glaciers. All natural terrains, except perhaps recent volcanic ones, bear the scars of erosion. In fact, erosion and tectonics are responsible for nearly all geomorphological features

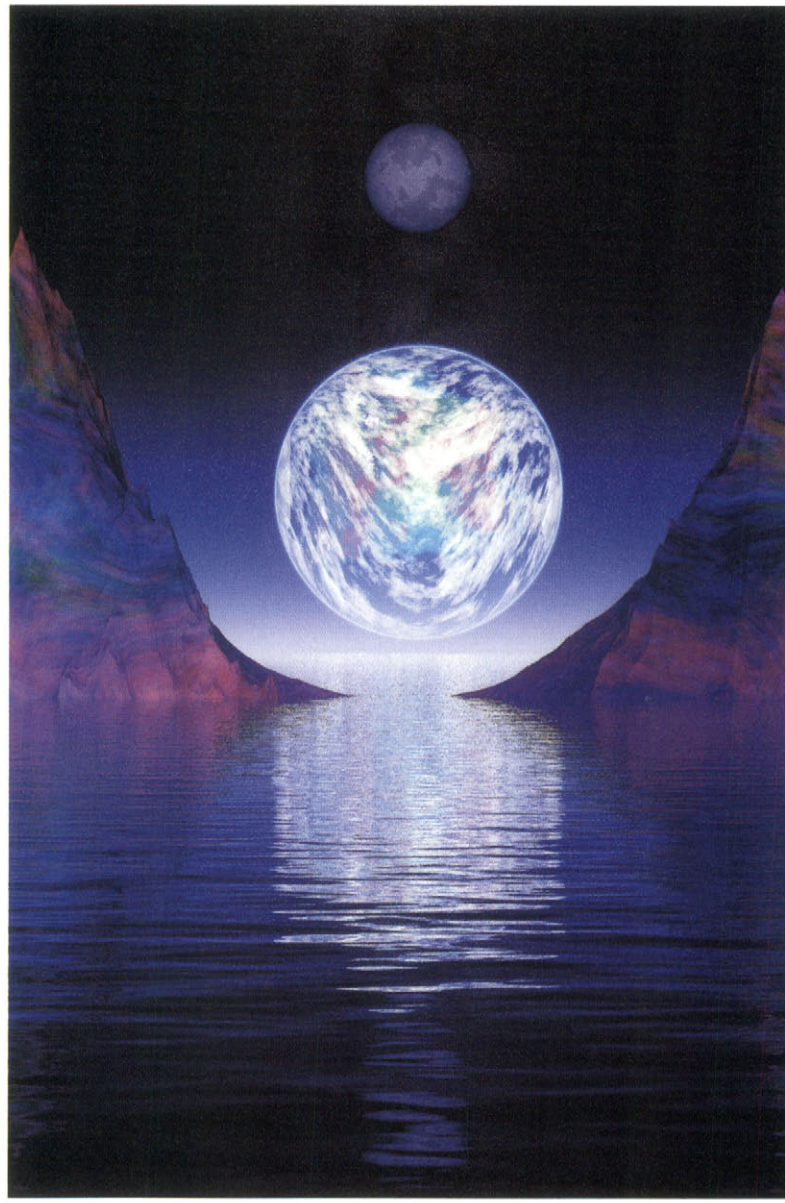


FIGURE 16.6 *Parabolic Curves in the Plane of the Ecliptic* employs most of the tricks described in Chapters 14–18, from multifractals to GIT textures to QAEB tracing. There are parabolas in the terrain, in the central valley, in the clouds, and in depth. Copyright © F. Kenton Musgrave.

on our planet, other than volcanic features, impact craters, and various features due to bioturbation (humanity's included). Some erosion features are relatively easy to model: talus slopes, for example. Others, such as drainage networks, are not so easy (Musgrave, Kolb, and Mace 1989). The rest of this chapter will describe certain ontogenetic models designed to yield a first approximation of certain erosion features, without overly compromising the elegance and computational efficiency of the original fBm model. These models are, at least loosely speaking, varieties of multifractals.

Statistics by Altitude

The observation that motivated my first multifractal model is that, in real terrains, low-lying areas sometimes tend to fill up with silt and become topographically smoother, while erosive processes may tend to keep higher areas more jagged. This can be accomplished with the following variation on fBm:

```
/*
 * Heterogeneous procedural terrain function: stats by altitude method.
 * Evaluated at "point"; returns value stored in "value".
 *
 * Parameters:
 *     "H" determines the fractal dimension of the roughest areas
 *     "lacunarity" is the gap between successive frequencies
 *     "octaves" is the number of frequencies in the fBm
 *     "offset" raises the terrain from "sea level"
 */
double
Hetero_Terrain( Vector point,
         double H, double lacunarity, double octaves, double offset )
{
    double      value, increment, frequency, remainder, Noise3();
    int         i;
    static int first = TRUE;
    static double *exponent_array;

    /* precompute and store spectral weights, for efficiency */
    if ( first ) {
        /* seize required memory for exponent_array */
        exponent_array =
        (double *)malloc( (octaves+1) * sizeof(double) );
        frequency = 1.0;
        for (i=0; i<=octaves; i++) {
             /* compute weight for each frequency */
             exponent_array[i] = pow( frequency, -H );
             frequency *= lacunarity;
         }
         first = FALSE;
```

```
    }

    /* first unscaled octave of function; later octaves are scaled */
    value = offset + Noise3( point ); point.x *= lacunarity;
    point.y *= lacunarity; point.z *= lacunarity;

    /* spectral construction inner loop, where the fractal is built */
    for (1=1; i<octaves; 1++) {
         /* obtain displaced noise value */
         increment = Noise3( point ) + offset;

         /* scale amplitude appropriately for this frequency */
         increment *= exponent_array[i];

         /* scale increment by current "altitude" of function */
         increment *= value;

         /* add increment to "value" */
         value += increment;

         /* raise spatial frequency */
         point.x *= lacunarity;
         point.y *= lacunarity;
         point.z *= lacunarity;
    } /* for */

    /* take care of remainder in "octaves" */
    remainder = octaves - (int)octaves;
    if ( remainder ) {
       /* "i" and spatial freq. are preset in loop above */
       /* note that the main loop code is made shorter here */
       /* you may want to make that loop more like this */
       increment = (Noise3( point ) + offset) * exponent_array[i];
       value += remainder * increment * value;
    }

    return( value );

} /* Hetero_Terrain() */
```

We accomplish our end by multiplying each successive octave by the current value of the function. Thus in areas near zero elevation, or "sea level," higher frequencies will be heavily damped, and the terrain will remain smooth. Higher elevations will not be so damped and will grow jagged as the iteration progresses. Note that we may need to clamp the highest value of the weighting variable to 1.0, to prevent the sum from diverging as we add in more values.

The behavior of this function is illustrated in the terrains seen in Figures 16.2, 16.7, and 18.2.

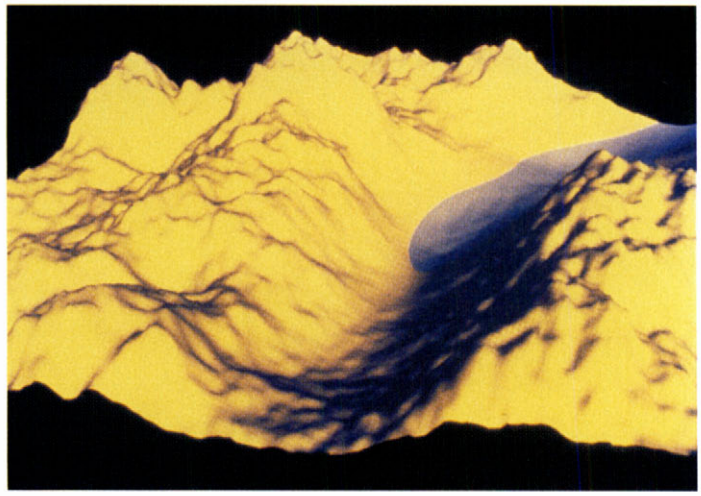


FIGURE 16.7 This multifractal terrain patch is quite smooth at "sea level" and gets rougher as altitude increases. Copyright © 1994 F. Kenton Musgrave.

A Hybrid Multifractal

My next observation was that valleys should have smooth bottoms at all altitudes, not just at sea level. It occurred to me that this could be accomplished by scaling higher frequencies in the summation by the local value of the previous frequency:

```
/* Hybrid additive/multiplicative multifractal terrain model. *
 * Some good parameter values to start with:
 *
 * H:          0.25
 * offset: 0.7
 */
double
HybridMultifractal( Vector point, double H, double lacunarity,
                    double octaves, double offset )
{
    double     frequency, result, signal, weight, remainder;
    double     Noise3();
    int  i;
    static     int first = TRUE;
    static     double *exponent_array;

    /* precompute and store spectral weights */
    if ( first ) {
        /* seize required memory for exponent_array */
        exponent_array =
          (double *)malloc( (octaves+1) * sizeof(double) );
```

```
        frequency = 1.0;
        for (i=0; i<=octaves; i++) {
        /* compute weight for each frequency */
        exponent_array[i] = pow( frequency, -H);
        frequency *= lacunarity;
    }
    first = FALSE;
}

/* get first octave of function */
result = ( Noise3( point ) + offset ) * exponent_array[0];
weight = result;

/* increase frequency */
point.x *= lacunarity;
point.y *= lacunarity;
point.z *= lacunarity;

/* spectral construction inner loop, where the fractal is built */
for (1=1; i<octaves; 1++) {
     /* prevent divergence */
     if ( weight > 1.0 ) weight = 1.0;

     /* get next higher frequency */
     signal = ( Noise3( point ) + offset ) * exponent_array[i];

     /* add it in, weighted by previous freq's local value */
     result += weight * signal;

     /* update the (monotonically decreasing) weighting value */
     /* (this is why H must specify a high fractal dimension) */
     weight *= signal;

     /* increase frequency */
     point.x *= lacunarity;
     point.y *= lacunarity;
     point.z *= lacunarity;
} /* for */

     /* take care of remainder in "octaves" */
     remainder = octaves - (int)octaves;
     if ( remainder )
         /* "i" and spatial freq. are preset in loop above */
         result += remainder * Noise3( point ) * exponent_array[i];

    return( result );

} /* HybridMultifractal() */
```

Note the offset applied to the noise function to move its range from $[-1, 1]$ to something closer to $[0, 2]$. (If your noise function has a different range, you'll need to

adjust this.) You should experiment with the values of these parameters and observe their effects.

An amusing facet of this function is that it *doesn't* do what I designed it to do: a valley above sea level in this function is defined not by the local value of the last frequency in the sum, as I have assumed, but by the local gradient of the function (i.e., the local tangent, the partial derivatives in x and y—however you care to view it). Put another way, in the above construction, we ignore the bias introduced by lower frequencies—we may be adding a "valley" onto the side of an already steep slope, and thus we may not get a valley at all, only a depression on the side of the slope. Nevertheless, this construction has provided some very nice, heterogeneous terrain models. Figure 16.1 illustrates a terrain model produced from the above function. Note that it begins to capture some of the large-scale heterogeneity of real terrains: we have plains, foothills, and alpine mountains, all in one patch. Figure 20.18 shows a similar construction, this time using the same ridged basis function seen in Figure 20.20: it's like Perlin's original "turbulence" function, which used the absolute value of the noise function, only it's turned upside-down, as 1 − abs(noise) so that the resulting creases stick up as ridges. The resulting multifractal terrain model is illustrated in Figures 17.4–17.6. It is generated by the following code:

```
/* Ridged multifractal terrain model.
 *
 * Some good parameter values to start with:
 *
 * H:          1.0
 * offset: 1.0
 * gain: 2.0
 */
double RidgedMultifractal( Vector point, double H, double lacunarity,
                           double octaves, double offset, double gain )
{
    double     result, frequency, signal, weight, Noise3();
    int        i;
    static int first = TRUE;
    static double *exponent_array;

    /* precompute and store spectral weights */
    if ( first ) {
         /* seize required memory for exponent_array */
         exponent_array =
                 (double *)malloc( (octaves+1) * sizeof(double) );
                 frequency = 1.0;
         for (i=0; i<=octaves; i++) {
         /* compute weight for each frequency */
         exponent_array[i] = pow( frequency, -H );
         frequency *= lacunarity;
```

```
        }
        first = FALSE;
    }

    /* get first octave */
    signal = Noise3( point );

    /* get absolute value of signal (this creates the ridges) */
    if ( signal < 0.0 ) signal = -signal;
    /* invert and translate (note that "offset" should be ~ = 1.0) */
    signal = offset - signal;
    /* square the signal, to increase "sharpness" of ridges */
    signal *= signal;
    /* assign initial values */
    result = signal;
    weight = 1.0;

    for( 1=1; i<octaves; i++ ) {
        /* increase the frequency */
        point.x *= lacunarity;
        point.y *= lacunarity;
        point.z *= lacunarity;

        /* weight successive contributions by previous signal */
        weight = signal * gain;
        if ( weight > 1.0 ) weight = 1.0;
        if ( weight < 0.0 ) weight = 0.0;
        signal = Noise3( point );
        if ( signal < 0.0 ) signal = -signal;
        signal = offset - signal;
        signal *= signal;

        /* weight the contribution */
        signal *= weight;
        result += signal *exponent_array[i];

}

    return( result );

} /* RidgedMultifractal() */
```

Multiplicative Multifractal Terrains

In Chapter 14, Figure 14.3 illustrates, as a terrain patch, the multiplicative multifractal function presented in that chapter. Qualitatively, that terrain patch appears quite similar to the statistics-by-altitude patch seen in Figure 16.1. At the time of this writing, our formal—that is, mathematical, rather than artistic—research into the mathematics of such multifractal terrain models is quite preliminary, so I have little of use to report. The multifractal construction of Chapter 14 does appear to have

some curious properties: as the value of `scale` goes from zero to infinity, the function goes from highly heterogeneous (at zero) to flat (diverging to infinity). We have not yet completed our quantitative study of the behavior, so I cannot elucidate further at this time.

For the time being, however, for the purposes of terrain synthesis it seems best to stick with the two hybrid additive multiplicative multifractal constructions presented in this chapter, rather than attempting to use the pure multifractal function presented in Chapter 14. These hybrid models may be no better understood mathematically, but they *are* better behaved as functions; that is, they don't usually need to be rescaled and are less prone to divergence.

CONCLUSION

I hope that in the last three chapters I have been able to illustrate the power of fractal geometry as a visual language of nature. We have seen that fractals can readily provide nice visual models of fire, earth, air, and water. I hope that I have also helped clarify the bounds of usefulness of fractal models for computer graphics: while fractals are not the final word in describing the world we live in, they do provide an elegant source of visual complexity for synthetic imagery. The accuracy of fractal models of natural phenomena is of an ontogenetic, rather than physical, character: they reflect morphology fairly well, but this semblance does not issue from first principles of physical law, so far as we know. The world we inhabit is more visually complex than we can hope to reproduce in synthetic imagery in the near future, but the simple, inherently procedural complexity of fractals marks a first significant step toward accomplishing such reproduction. I hope that some of the constructions presented here will be useful to you, whether in your own attempts to create synthetic worlds or in more abstract artistic endeavors.

There is plenty of work left to be done in developing fractal models of natural phenomena. Turbulence has yet to be efficiently procedurally modeled to everyone's satisfaction, and multifractals need to be understood and applied in computer graphics. Trees are distinctly fractal, yet to a large extent they still defy our ability to capture their full complexity in a simple, *efficient* model; the same goes for river systems and dielectric breakdown (e.g., lightning). Reproducing other, nonfractal manifestations of heterogeneous complexity will, no doubt, keep image synthesists busy for a long time to come. I like to think that our best synthetic images reflect directly something of our depth—and lack—of understanding of the world we live in. As beautiful and convincing as some of the images may be, they are only a first approximation of the true complexity of nature.