15

FRACTAL SOLID TEXTURES: SOME EXAMPLES

F. KENTON MUSGRAVE

This chapter will describe some fractal procedural textures serving as models of natural phenomena. They are divided into the four elements of the ancients: air, fire, water, and earth. The presentation format of the code segments, for the most part, has been switched from the C++ programming language to the RenderMan (Upstill 1990) shading language.[1] The fractal functions described in the previous chapter are to be used as primitive building blocks for the textures we'll develop here, and as such, they should be implemented in the most efficient manner possible (e.g., in compiled code). If you like any of the textures we develop here enough to make them part of a standard texture library, you might want to translate them to compiled code, for efficiency. But, in general, it is quicker and easier to develop texture functions using higher-level tools such as the function graph editors in MojoWorld and Dark Tree or the RenderMan shading language. Take the texture constructions described here as starting points and modify them. Do lots of experiments and come up with your own unique textures—after all, such experimentation is how these textures came into being! Nothing here is written in stone; you can have endless fun devising variations.

The textures described in this chapter were certainly not designed as a whole, a priori, then implemented. Rather, they are generally the result of "hacking"—hours and hours of making modifications and extensions, evaluating the texture to see how it looks, making more changes, and so on. It may seem to you that this iterative loop is more artistic than scientific, and I would agree that it is, but it does share with the scientific method what Gregory Nielson calls "the basic loop of scientific discovery" (Nielson 1991): you posit a formal model (nothing, after all, is more formal than the

1. The translations from C to the RenderMan shading language are provided courtesy of Larry Gritz of ExLuna, the author of the shareware RenderMan package Blue Moon Rendering Tools (BMRT). Larry kindly updated the shaders for the second edition of this book. They have been tested and verified on both Pixar's PhotoRealistic RenderMan and BMRT.

logic of a computer program), you make observations of the behavior of the model and the system being modeled, you make modifications to improve the model, then more observations, and so on, in an iterative loop. Perhaps the main difference between science and computer graphics is, as Pat Hanrahan has pointed out, in the time required per iteration: that time is certainly much shorter when designing procedural textures than when pursuing the physical sciences. The point is, no scientific model was born perfect, and neither is any complex procedural texture. Particularly in ontogenetic modeling, again, we are not concerned with Truth but rather with visual Beauty. We are more interested in semblance than veracity. When you are more interested in the quality of the final image than in the methodology of its production, you are more an engineer than a scientist. In such an endeavor, whatever gets us the result in a reasonable amount of time, both human and computer, is a viable strategy. So we'll do whatever works.

CLOUDS

Clouds remain one of the most significant challenges in the area of modeling natural phenomena for computer graphics. While some very nice images of clouds have been rendered by Geoffrey Gardner, Richard Voss, David Ebert, Matthew Fairclough, Sang Yoon Lee, and myself, in general the modeling and rendering of clouds remains an open problem. I can't claim to have advanced the state of the art in cloud modeling much, but I have devised a few two-dimensional models[2] that are at least significant *aesthetically.* I'll describe them below, but first I'll describe the most common, quick, and easy cloud texture.

Puffy Clouds

One of the simplest and most often used fractal textures is a simple representation of thin, wispy clouds in a blue sky. All this consists of is a thresholded fBm. If our fBm function outputs values in the range $[-1, 1]$, we might specify that any value less than zero represents blue sky. This will accomplish the effect:

```
surface
puffyclouds(float Ka = 0, Kd = 0;
   float txtscale = 1;
   color skycolor = color(.15, .15, .6);
   color cloudcolor = color(1, 1, 1);
```

2. This may be a little confusing: I call these models 2D because, while they are implemented as 3D solid textures, they are designed to be evaluated on 2D surfaces. In Chapter 17, I'll show how to evaluate them as volumetric hypertextures via the QAEB algorithm.

```
    float octaves = 8, H = 0.5, lacunarity = 2;
    float threshold = 0; )
{
    float value;
    color Ct; /* Color of the surface */
    point PP; /* Surface point in shader space */

    PP = txtscale * transform("shader", P);
    /* Use fractional Brownian motion to compute a value for this point */
    value = fBm(PP, filterwidthp(PP), octaves, lacunarity, H);
    Ct = mix(skycolor, cloudcolor, smoothstep(threshold, 1, value));

    /* Shade like matte, but use color Ct */
    Ci = Ct; /* This makes the color disregard the lighting */

    /* Uncomment the next line if you want the surface to be lit */
    /* Ci = Ct * (Ka * ambient() + Kd * diffuse(faceforward(N,I))); */
}
```

This can make a good background texture for images where the view is looking up at the sky, as in Jules Bloomenthal's image of the mighty maple, shown in Figure 15.1 (Bloomenthal 1985).

FIGURE 15.1 Jules Bloomenthal's image of "the mighty maple" shows a typical use of the puffy clouds texture.

A Variety of fBm

As noted in the previous chapter, our fractals can be constructed from literally *any* basis function; the basis function we most often choose is the Perlin *noise* function. For theoretical reasons outlined in the previous chapter, with our usual lacunarity of 2.0, we use at most about 8 to 10 octaves of the basis function in our constructions. It turns out that the character of the basis function shows through clearly in such a small spectral summation. So, if we use a different basis function, we get fBm with a different character, with a different aesthetic "feel" to it. (That's why there are over 200 different basis functions available in MojoWorld, with practically infinite variations on them possible.) Take, for example, fractal mountains constructed by the popular polygon subdivision schemes. The basis function implicit in the linear interpolation between the sample points is a sawtooth wave. Thus the resulting mountains are always quite triangular and jagged. When it comes to playing with different basis functions, the possibilities are endless. Wavelets (Ruskai 1992) offer an exciting prospect for basis function manipulation; Lewis's "sparse convolution" (Lewis 1989), also called "fractal sum of pulses" in Lovejoy and Mandelbrot (1985), is a theoretically desirable approach that accommodates the use of any finite basis function. Unfortunately, it is slow in practice. Try it in MojoWorld and see for yourself.

Let me now describe one trick I sometimes play with the Perlin *noise* function, to get a variation of it with significantly altered aesthetic "feel." I call it, for lack of a better name, `DistNoise( )`, for *distorted noise.* It employs one of the oldest tricks in procedural textures: domain distortion. As it is a fundamental building block type function, I present it in C++.

```
double
DistNoise( Vector point, double distortion )
{
        Vector offset, VecNoise();
        double Noise();

        offset = point + 0.5; /* misregister domain for distortion */
        offset = VecNoise( offset ); /* get a random vector */
        offset *= distortion; /* scale the distortion */
        /* "point" is the domain; distort domain by adding "offset" */
          point += offset;

        return Noise( point );  /* distorted domain noise */
}
```

The function `VecNoise()` is a vector-valued noise function that takes a 3D vector as an argument and returns a 3D vector. This returned vector corresponds

to three separate noise values for each of its x, y, and z components. It can be constructed by simply evaluating a noise function at three different points in space: the one passed in as the argument and two displaced copies of it, as in the previous misregistration.

```
Vector
VecNoise( Vector point )
{
        Vector result;
        double Noise();

        result.x = Noise( point );
        result.y = Noise( point + 3.33 );
        result.z = Noise( point + 7.77 );

        return result;
}
```

Note that this is not the same thing as Perlin's `DNoise()`, which is specified to return the three partial derivatives of `Noise()`. The latter will be C-2 continuous, while our function is C-3 continuous.[3]

We displace the point passed to `VecNoise()` by 0.5 in each of x, y, and z, to deliberately misregister the underlying integer lattices upon which `VecNoise()` and `Noise()` are evaluated as, if you're using gradient noise, both functions have value zero at these points. Next we evaluate `VecNoise()` at the displaced point and scale the returned vector by the distortion parameter. Then we add the resultant vector to the input point; this has the net effect of distorting the input domain given to the `Noise()` function. The output is therefore also distorted.

Figure 15.2 illustrates the difference between undistorted `Noise()` and `DistNoise()`, with the distortion parameter set to 1.0. It also illustrates the difference between fBms constructed using `Noise()` and `DistNoise()`, respectively, as the basis function. That difference is subtle, but significant. To my artist's eye, the latter fBm has a sort of wispy character that looks more like certain cirrus clouds than does the "vanilla" fBm, which seems a little bland in comparison. Note, however, that `DistNoise()` is about four times as expensive to evaluate as `Noise()`, so you pay for this subtle difference. Cost aside, I have used the modified fBm to good effect in clouds, as seen in Figures 20.4 and 20.9. In the following I will refer to such fBm by the function name `DistfBm()`.

3. This business of C-2 and C-3 continuity is a mathematical point about the number of continuous derivatives a function has. If you don't know what a derivative is, just ignore this rigmarole.

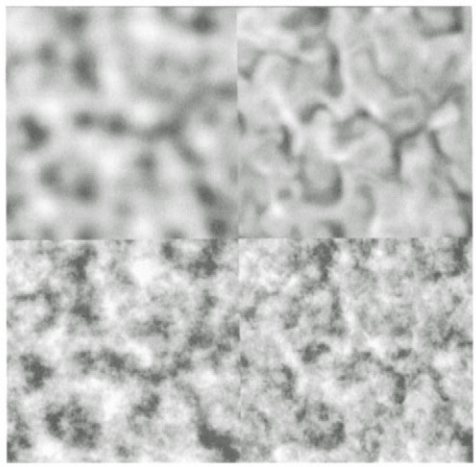

FIGURE 15.2 The upper-left square shows `Noise()`; the upper right shows `DistNoise()`. Below appears fBm constructed from each; note the subtle difference.

Note that we can write `DistNoise()`more tersely:

```
/*
 * Terse version of DistNoise( )
 */
double DistNoise( Vector point, double distortion )
{
        return Noise( point + distortion * VecNoise( point + 0.5 ) );
}
```

The RenderMan version looks very similar:

```
/*
 * RenderMan version of DistNoise( )
 *
 * Since the noise( ) function in RenderMan shading language has range
 * [0,1], we define a signed noise to be the noise function that Perlin
 * typically uses.
 */
#define snoise(P) (2*noise(P) - 1)

float DistNoise( point Pt, float distortion )
{
   point offset = snoise( Pt + point(0.5, 0.5, 0.5) );
   return snoise( Pt + distortion * offset );
}
```

```
/* Alternatively, it can be defined as a macro: */

#define DistNoise(Pt,distortion) \
   (snoise( Pt + distortion*snoise( Pt+point(0.5,0.5,0.5) ) )
```

Distortion for Cirrus Clouds and Global Circulation

Note that the previous construction is an example of *functional composition,* wherein functions become the arguments to other functions. Such nesting is a powerful procedural technique, as Karl Sims (1991) showed in his genetic LISP program, which creates procedural textures and which was used to create Figure 14.1; we'll investigate this paradigm in detail in Chapter 19. The idea of composition of noise functions to provide distortion has proved useful in another aspect of modeling clouds: emulating the streaming of clouds that are stretched by winds. We can get the kind of distinctive cirrus clouds seen in Figure 20.9 from the following specification:

```
/* Use signed, zero-mean Perlin noise */
#define snoise(x) ((2*noise(x))-1)

/* RenderMan vector-valued Perlin noise */
#define vsnoise(p) (2 * (vector noise(p)) - 1)

/* If we know the filter size, we can crudely antialias snoise by
 * fading to its average value at the Nyquist limit.
 */
#define filteredsnoise(p,width) (snoise(p) * (1 - smoothstep(0.2,0.6,width)))
#define filteredvsnoise(p,width) (vsnoise(p) * (1-smoothstep(0.2,0.6,width)))

surface
planetclouds(float Ka = 0.5, Kd = 0.75;
             float distortionscale = 1;
             float H = 0.7;
             float lacunarity = 2;
             float octaves = 9;
             float offset = 0;)
{
    vector Pdistortion;   /* The "distortion" vector */
    point PP;             /* Point after distortion */
    float result;         /* Fractal sum is stored here */
    float filtwidth;

    /* Transform to texture coordinates */ PP =
    transform("shader", P);
    filtwidth = filterwidthp(PP);
```

```
    /* Get "distortion" vector */
    Pdistortion = distortionscale * filteredvsnoise(PP, filtwidth);
    PP = PP + Pdistortion;
    filtwidth = filterwidthp(PP);

    /* Compute fBm */
    result = fBm(PP, filtwidth, octaves, lacunarity, H);

    /* Adjust zero crossing (where the clouds disappear) */
    result = clamp(result+offset, 0, 1);

    /* Scale density */
    result /= (1 + offset);

    /* Modulate surface opacity by the cloud value */
    Oi = result * Os;
    /* Shade like matte, but with color scaled by cloud opacity */
    Ci = Oi * (Ka * ambient() + Kd * diffuse(faceforward(normalize(N),I)));
}
```

Note that this is the same domain distortion idea that was used in `DistNoise()`, except that the distortion has greater magnitude and is at a large scale relative to the fBm.

This large-scale distortion was originally designed to provide a first approximation to the global circulation patterns in Earth's clouds as seen from space. The preceding code produced the clouds seen in Figures 20.4 and 20.9. While not a bad first approximation, it saliently lacks the eddies and swirls generated by vortices in turbulent flow.

It occurred to me that we could use an fBm-valued distortion to better emulate turbulence:

```
#define snoise(p) (2 * (float noise(p)) - 1)
#define vsnoise(p) (2 * (vector noise(p)) - 1)
#define filteredsnoise(p,width) \
   (snoise(p) * (1 - smoothstep(0.2,0.6,width)))
#define filteredvsnoise(p,width) \
   (vsnoise(p) * (1-smoothstep(0.2,0.6,width)))

/* A vector-valued antialiased fBm. */
vector
vfBm(point p;
     float filtwidth;
     uniform float maxoctaves, lacunarity, gain)
{
   uniform float i;
   uniform float amp = 1;
   varying point pp = p;
```

```
    varying vector sum = 0;
    varying float fw = filtwidth;

    for (i = 0; i < maxoctaves && fw < 1.0; i += 1) {
        sum += amp * filteredvsnoise(pp, fw);
        amp *= gain;
        pp *= lacunarity;
        fw *= lacunarity;
    }
    return sum;
}

surface
planetclouds(float Ka = 0.5, Kd = 0.75;
             float distortionscale = 1;
             float H = 0.7;
             float lacunarity = 2;
             float octaves = 9;
             float offset = 0;)
{
    vector Pdistortion; /* The "distortion" vector */
    point PP;           /* Point after distortion */
    float result;       /* Fractal sum is stored here */
    float filtwidth;

    /* Transform to texture coordinates */
    PP = transform("shader", P);
    filtwidth = filterwidthp(PP);

    /* Get "distortion" vector */
    Pdistortion = distortionscale * VfBm(PP, filtwidth, octaves, lacunarity, H);

    /* Create distorted clouds */
    PP = PP + Pdistortion;
    filtwidth = filterwidthp(PP);

    /* Compute fBm */
    result = fBm(PP, filtwidth, octaves, lacunarity, H);

    /* Adjust zero crossing (where the clouds disappear) */
    result = clamp(result+offset, 0, 1);

    /* Scale density */
    result /= (1 + offset);

    /* Modulate surface opacity by the cloud value */
    Oi = result * Os;

    /* Shade like matte, but with color scaled by cloud opacity */
    Ci = Oi * (Ka * ambient() + Kd * diffuse(faceforward(normalize(N),I)));
}
```

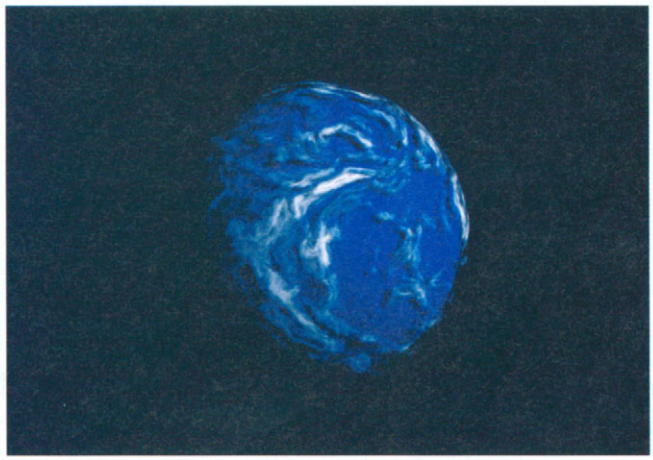

FIGURE 15.3 Clouds distorted with fBm: they look more like cotton than turbulence. Copyright © 1994 F. Kenton Musgrave.

Unfortunately, the result looks more like cotton than turbulence (see Figure 15.3), but it is an interesting experiment.

Once again, the essential element that our turbulence model lacks is vortices, or vorticity. Large-scale vortices in Earth's atmosphere occur in cyclonic and anticyclonic storm systems that are clearly visible from space. The most extreme vortices in our atmosphere occur in tornadoes and hurricanes. In Figure 15.4 we see an ontogenetic model of a hurricane, produced by the following specification:

```
surface
cyclone(float Ka = 0.5, Kd = 0.75;
        float max_radius = 1;
        float twist = 0.5;
        float scale = .7, offset = .5;
        float H = 0.675;
        float octaves = 4;)
{
    float radius, dist, angle, eye_weight, value;
    point Pt;              /* Point in texture space */
    vector PN;             /* Normalized vector in texture space */
    point PP;              /* Point after distortion */
    float filtwidth, a;

    /* Transform to texture coordinates */
    Pt = transform("shader", P);
    filtwidth = filterwidthp(Pt);
```

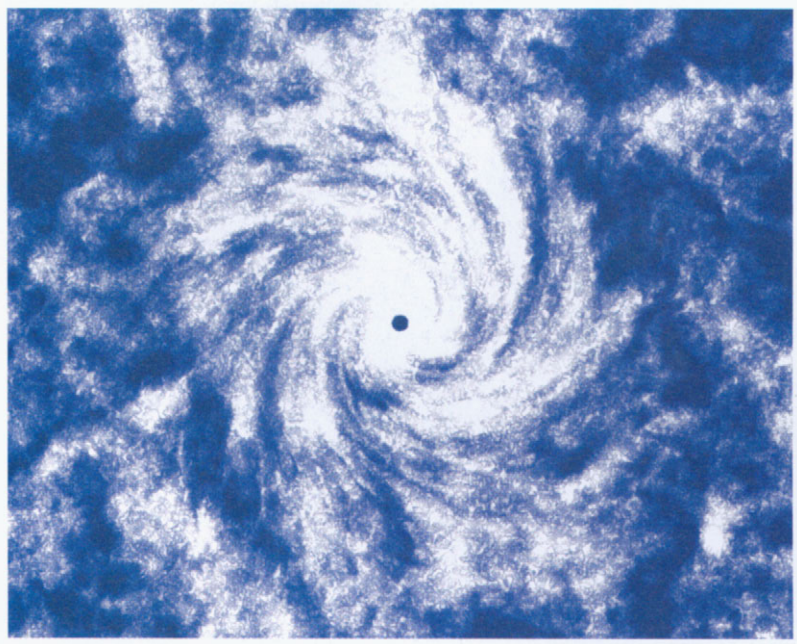

FIGURE 15.4 A first cut at including the vortices that comprise turbulence. Note that the smaller clouds are not distorted by the vortex twist, only the large-scale distribution is. Copyright © 1994 F. Kenton Musgrave.

```
/* Rotate hit point to "cyclone space" */
PN = normalize(vector Pt);
radius = sqrt(xcomp(PN)*xcomp(PN) + ycomp(PN)*ycomp(PN));

if (radius < max_radius) { /* inside of cyclone */
  /* invert distance from center */
  dist = pow(max_radius-radius, 3);
  angle = PI + twist * TWOPI * (max_radius-dist) / max_radius;
  PP = rotate(Pt, angle, point(0,0,0), point(0,0,1));
  /* Subtract out "eye" of storm */
  if (radius < 0.05*max_radius) { /* if in "eye" */
     eye_weight = ( . 1*max_radius-radius) * 10; /* normalize */
     /* invert and make nonlinear */
     eye_weight = pow(1 - eye_weight, 4);
  }
  else eye_weight = 1;
}
else { /* outside of cyclone */
  PP = Pt;
  eye_weight = 0;
}
```

```
    if (eye_weight > 0) { /* if in "storm" area */
      /* Compute clouds */
      a = DistfBm(PP, filtwidth, octaves, 2, H, 1);
      value = abs(eye_weight * (offset + scale * a));
    }
    else value = 0;

    /* Thin the density of the clouds */
    Oi = value * Os;

    /* Shade like matte, but with color scaled by cloud opacity */
    Ci = Oi * (Ka * ambient() + Kd * diffuse(faceforward(normalize(N),I)));
}
```

Here we have a single vortex; to model general turbulent flow we require a fractal hierarchy of vortices. Note that we've modeled the clouds on two different scales here: a distorted large-scale distribution comprising a weighting function, which is applied to smaller-scale cloud features. This construction is based on my subjective study of clouds and storms seen from space (see Kelley 1988 for many lovely examples of such images). The idea of undistorted small features corresponds to the phenomenon of viscosity, which, as our poem indicates, damps turbulence at small scales. This may also be seen as a first step in the direction of multifractal models, as our fractal behavior is different at different scales, and therefore may require more than one value or measure to characterize the fractal behavior.

The Coriolis Effect

A salient feature of atmospheric flow on Earth, and even more so on Venus, is that it is strongly affected by the *Coriolis effect*—a shearing effect caused by the conservation of angular momentum as air masses move north and south. The atmosphere is moving around the planet and, like a spinning skater who spins faster as she pulls her arms and legs in closer, the air must spin around the planet faster as it moves toward the poles and more slowly as it moves toward the equator. For a given angular momentum, angular velocity (spin rate) varies as the inverse square of radius. In this respect, the following model could be thought of as a physically accurate model, but I would still call it an ontogenetic, or at best an empirical, model.

Using our modified fBm and a Coriolis distortion, we can model Venus (as imaged at ultraviolet wavelengths) quite well (see Figure 15.5).

```
surface
venus(float Ka = 1, Kd = 1;
      float offset = 1;
```

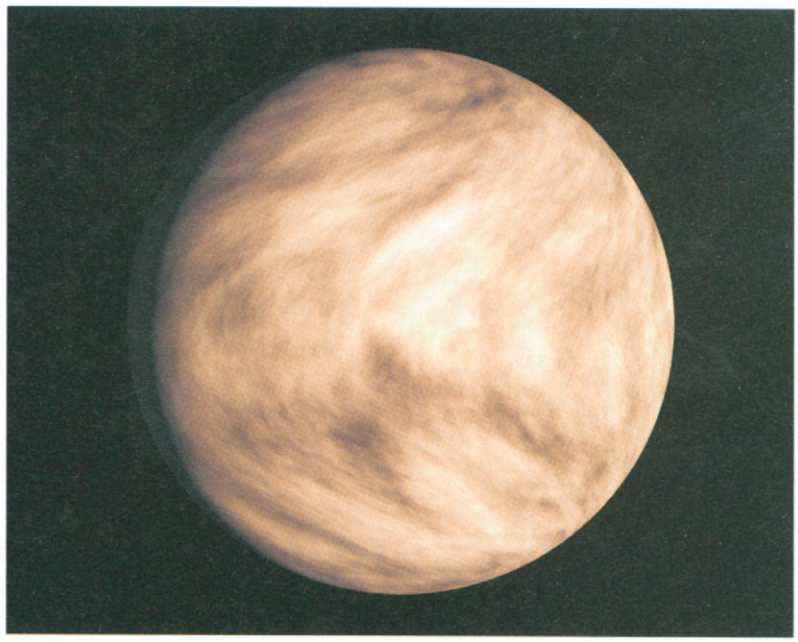

FIGURE 15.5 A strong Coriolis twist characterizes the cloud patterns on Venus. Copyright © 1994 F. Kenton Musgrave.

```
      float scale = 0.6;
      float twist = 0.22;
      float H = 0.65;
      float octaves = 8;)
{
   point Ptexture;   /* the shade point in texture space */
   vector PtN;       /* normalized version of Ptexture */
   point PP;         /* Point after rotation by Coriolis twist */
   float rsq;        /* Used in calculation of twist */
   float angle;      /* Twist angle */
   float value;      /* Fractal sum is stored here */
   float filtwidth;  /* Filter width for antialiasing */

   /* Transform to texture coordinates */
   Ptexture = transform("shader", P);
   filtwidth = filterwidthp(Ptexture);

   /* Calculate Coriolis twist distortion */
   PtN = normalize(vector Ptexture);
   rsq = xcomp(PtN)*xcomp(PtN) + ycomp(PtN)*ycomp(PtN);
```

```
    angle = twist * TWOPI * rsq;
    PP = rotate(Ptexture, angle, point(0,0,0), point(0,0,1));

    /* Compute Coriolis-distorted clouds */
    value = abs(offset + scale * fBm(PP,filtwidth,octaves,2,H));

    /* Shade like matte, but with color scaled by cloud color */
    Oi = Os;
    Ci = Oi * (value * Cs) *
         (Ka * ambient() + Kd *
         diffuse(faceforward(normalize(N),1)));
}
```

FIRE

Fire is yet another example of turbulent flow. I can't claim to have modeled fire particularly well, but I do have a fire texture that I can share with you. There are two peculiar features in this model. The first is the use of a "ridged" fBm, as discussed in the next chapter. The second is a distortion that varies exponentially with height, which is meant to model the upward acceleration of the hot gases of the flames.

```
surface
flame(float distortion = 0;
      float chaosscale = 1;
      float chaosoffset = 0;
      float octaves = 7;
      float flameheight = 1;
      float flameamplitude = .5; )
{
    point PP, PQ;
    float chaos;
    float cmap;
    float fw;

    PQ = PP = transform("shader", P);
    PQ *= point(1, 1, exp(-zcomp(PP)));
    fw = filterwidthp(PQ);

    chaos = DistfBm(PQ, fw, octaves, 2, 0.5, distortion);
    chaos = abs(chaosscale*chaos + chaosoffset);
    cmap = 0.85*chaos + 0.8 * (flameamplitude - flameheight * zcomp(PP);
    Ci = color spline(cmap,
                  color(0, 0, 0),        color(0, 0, 0),
                  color(27, 0, 0),       color(54, 0, 0),
                  color(81, 0, 0),       color(109, 0, 0),
                  color(136, 0, 0),      color(166, 5, 0),
                  color(189, 30, 0),     color(211, 60, 0),
                  color(231, 91, 0),     color(238, 128, 0),
```

```
                color(244, 162, 12),   color(248, 187, 58),
                color(251, 209, 115),  color(254, 236, 210),
                color(255, 241, 230),  color(255, 241, 230)) / 255;
}
```

While this certainly doesn't fully capture the rich structure of real flames, it's not too bad as a first approximation. In Figure 15.6 several transparent layers of this fire texture were sandwiched with some of Przemek Prusinkiewicz's early L-system tree models (Prusinkiewicz and Lindenmayer 1990). Note that the colors are critical to the realism of the results. What you want is a color map that represents a range of colors and intensities of black-body radiators, going from black to cherry red, through orange and yellow, to white.[4] Also, if you scale the flame structure down so that the high frequencies are more visible, this texture takes on a kind of astrophysical character—the ridges in the fBm form filaments and loops that, while not exactly realistic, have a distinctive character reminiscent of certain emission nebulae where stars are being born and dying. Try constructing this function without using `abs()` and observe the difference—it's actually more realistic and less surreal.

WATER

To my chagrin, people often say of my landscape images, "The water is the best part!" Well, I'm here to tell you that it's the cheapest and easiest trick I ever did with procedural textures. Let me show you how.

Noise Ripples

I originally developed this texture in a ray tracer that didn't have displacement mapping. In that context, it can be implemented as a bump map. This requires that you build a vector-valued fBm function, which is no problem as it's really just three independent fBms, one for each of *x*, *y*, and *z*, or one fBm constructed from `VecNoise()`, as we saw earlier in the definition of `VfBm()`. In MojoWorld and RenderMan, displacement mapping is available, so the implementation is simpler: it

4. Black-body radiators are a concept from physics: they are theoretically ideal (and therefore nonexistent in nature) objects that are completely without color of their own, regardless of their temperature. Thus, as you raise their temperature, they glow with a color that represents the ideal thermal, or Planck, spectrum for their temperature. Real objects heated enough to glow at visible wavelengths tend to be complicated both by their underlying (i.e., cold) nonblack color and by emission lines—certain wavelengths that have enhanced emission (or absorption) due to quantum effects. Most fires we see are made more yellow than a black-body radiator at the same temperature by sodium emission lines.

FIGURE 15.6 *Forest Fire* illustrates a fire texture. Color is critical in getting the look of fire. Note that the 1 − abs(noise) basis function gives rise to the sinews in the flames. Copyright © F. Kenton Musgrave.

requires only a scalar-valued displacement to perturb the surface, rather than a vector-valued perturbation for the surface normal.

The water model I've always used requires only a reflective silver plane in the foreground and this simple bump or displacement map applied to the surface:

```
displacement
ripples(float Km = 1, octaves = 2;)
{
   float offset;
   point PP;

   /* Do the calculations in shader space */
   PP = transform( "shader", P );

   /* Get fractal displacement, scale by Km */
   offset = Km * fBm( PP, 3.0, 2.0, octaves );
```

```
    /* Displace the surface point and recompute the normal */
    P += offset * normalize( N );
    N = calculatenormal( P );
}
```

We have perturbed the surface normal with a degenerate—in the mathematical sense—fractal function of only two octaves. (Recall that we said in the previous chapter that a fractal ought to have at least three levels of self-similarity to be called a fractal; this one only has two. Of course, you may use more or less octaves, as your artistry dictates.) You can stretch this function by scaling it up in one of the lateral directions, to get an impression of waves. For such a simple texture, it works surprisingly well (see Figures 15.7 and 15.8).

Wind-Blown Waters

In nature, ripples on water so calm are rarely homogeneous as those given by the previous function. Usually, on the large scale, ripples are modulated by the blowing

FIGURE 15.7 *Spirit Lake* shows a noise-based water bump map. The terrain model is the first multifractal model described in Chapter 16. Copyright © F. Kenton Musgrave.

FIGURE 15.8 *Lethe* is a polygon subdivision terrain model, hence the jagged appearance. It illustrates the water, sedimentary rock strata, and moon textures described in this chapter. Copyright © F. Kenton Musgrave.

wind. Blowing wind is turbulent flow and is therefore fractal. We can apply a large-scale, as compared to the ripples, fractal weighting function to the previous ripples to generate a nice approximation of breeze-blown water:

```
displacement
windywave(float Km = 0.1;
          float txtscale = 1;
          float windfreq = 0.5;
          float windamp = 1;
          float minwind = 0.3)
{
   float offset;
   point PP;
   float wind;
   float turb;
   float filtwidth;
```

```
    PP = txtscale * windfreq * transform("shader", P);
    filtwidth = filterwidthp(PP);

    /* get fractal displacement*/
    offset = Km * fBm(PP,filtwidth,2,2,0.5);
    PP *= 8; filtwidth *= 8;

    /* calculate wind field */
    turb = turbulence(PP, filtwidth, 4, 2, 0.5);
    wind = minwind + windamp * turb;

    /* displace the surface*/
    N = calculatenormal(P+wind * offset * normalize(N));
}
```

The weighting function we use is, yet again, a variety of fBm. We use the most generic version, as there's no need for the subtleties of any of the variations we've developed. (Turbulence is actually multifractal, however, so it might be worthwhile to experiment with multifractal weighting functions.) We need only a few octaves in this fBm, since we don't really want to generate fine-scale detail with the weighting function.

The result of the `Windywave()` texture is illustrated in Figures 15.9 and 15.18.

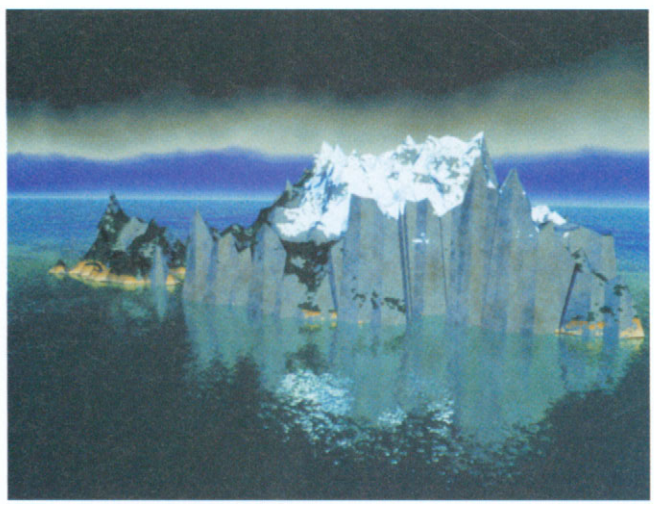

FIGURE 15.9 The effects of breezes on the water are illustrated in *Sea Rock*. Copyright © 1987 F. Kenton Musgrave.

EARTH

Now let's look at earthly textures. Well, at least two out of three will be literally earthly; the third will be a moon.

Sedimentary Rock Strata

Early in my career of rendering fractal landscapes, I didn't have the capacity to ray-trace terrains with very many triangles in them. Thus the "mountains" were quite chunky, and the triangles used to tessellate the surface were quite large and obviously flat, unlike any terrain I have ever seen. One of the early textures I devised to distract the eye's attention from this intrinsically bogus geometry was an imitation of sedimentary rock strata:

```
surface
strata(float Ka = 0.5, Kd = 1;
       float txtscale = 1;
       float zscale = 2;
       float turbscale = 0.1;
       float offset = 0;
       float octaves = 8;)
{
   color Ct;
   point PP;
   float cmap;
   float turb;
   PP = txtscale * transform("shader", P);

    /*turbation by fBm */
    turb = fBm(PP, filterwidthp(PP), octaves, 2, 0.5);

    /*use turb and z to index color map */
    cmap = zscale * zcomp(PP) + turbscale * turb - offset;
    Ct = color spline(mod(cmap, 1),
             color(166, 131, 70),   color(166, 131, 70),
             color(204, 178, 127),  color(184, 153, 97),
             color(140, 114, 51),   color(159, 123, 60),
             color(204, 178, 127),  color(230, 180, 80),
             color(192, 164, 110),  color(172, 139, 80),
             color(102, 76, 25),    color(166, 131, 70),
             color(201, 175, 124),  color(181, 150, 94),
             color(161, 125, 64),   color(177, 145, 87),
             color(170, 136, 77),   color(197, 170, 117),
             color(180, 100, 50),   color(175, 142, 84),
             color(197, 170, 117),  color(177, 145, 87),
```

```
            color(170, 136, 77),   color(186, 156, 100),
            color(166, 131, 70),   color(188, 159, 104),
            color(168, 134, 74),   color(159, 123, 60),
            color(195, 167, 114),  color(175, 142, 84),
            color(161, 125, 64),   color(197, 170, 117),
            color(177, 145, 87),   color(177, 145, 87)) / 255;

    /* Shade like matte, but with color scaled by cloud color and opacity */
    Oi = Os;
    Ci = Oi * Cs * Ct * (Ka * ambient() + Kd *
    diffuse(faceforward(normalize(N),I)));
}
```

The key idea here is to index a color lookup table by altitude[5] and to perturb that altitude index with a little fBm. The geologic analogs to this are soft-sediment deformation, in which layers of sediment are distorted before solidifying into sedimentary rock. It's closely related to Ken Perlin's famous *marble* texture.

The color lookup table is loaded with a color map that contains bands of color that you, the artist, deem appropriate for representing the different layers of rock. Both aesthetics and my observations of nature indicate that the colors of the various layers should be quite similar and subdued, with one or two layers that really stand out tossed in to provide visual interest. For an example of this, see the red and yellow bands in Figure 15.10.

Gaea: Building an Entire Planet

In fact, you can build an entire Earth-like planet with a single procedural texture. (We now call such planets MojoWorlds; this text was originally written some 10 years ago.) Not surprisingly, such an ambitious texture gets rather complex. And, of course, it is quite fractal. In fact, fractals are used in three ways in this texture: as a displacement map to provide continents, oceans, and a fractal coastline; as a perturbation to a climate-by-latitude color map (much like our earlier rock strata map) providing an interesting distribution of mountains, deserts, forests, and so on; and finally as a color perturbation, to ameliorate lack of detail in areas that would all be the same color because they share the same lookup table index.

Now let's see how such a complex procedural texture evolves, step by step. The first step in creating an earth is to create continents and oceans. This can be

5. In my original C implementation of these texture functions, the color maps are stored in 256-entry lookup tables. Larry Gritz has used RenderMan's functionality to replace those tables with splines in the code that appears in this text.

FIGURE 15.10 *Bryce* illustrates a sedimentary rock strata texture. Note the red and yellow strata that provide visual interest. The terrain model is a polygon subdivision erosion model described by Mandelbrot (Peitgen and Saupe 1988). Copyright © F. Kenton Musgrave.

accomplished by quantizing an fBm texture: A parameter threshold controls the "sea level"; any value less than threshold is considered to be below sea level and is therefore colored blue. This gives us the effect seen in Figure 15.11(a). Note that in the following code, there is a Boolean-valued parameter `multifractal`. This gives us the option of creating heterogeneous terrain and coastlines—see how the fractal dimension of the coasts varies in Figure 20.4.

Next we provide a color lookup table to simulate climatic zones by latitude; see Figure 15.11(b). Our goal is to have white polar caps and barren, gray sub-Arctic zones blending into green, temperate-zone forests, which in turn blend into buff-colored desert sands representing equatorial deserts. (Of course, you can use whatever colors you please.) The coloring is accomplished with a splined color ramp, indexed by the latitude of the ray/earth intersection point.

This rough coloring-by-latitude is then fractally perturbed, as in Figure 15.11(c). This is accomplished simply by adding fBm to the latitude value to perturb it, as with the rock strata texture. (This is just another example of the powerful tool of domain

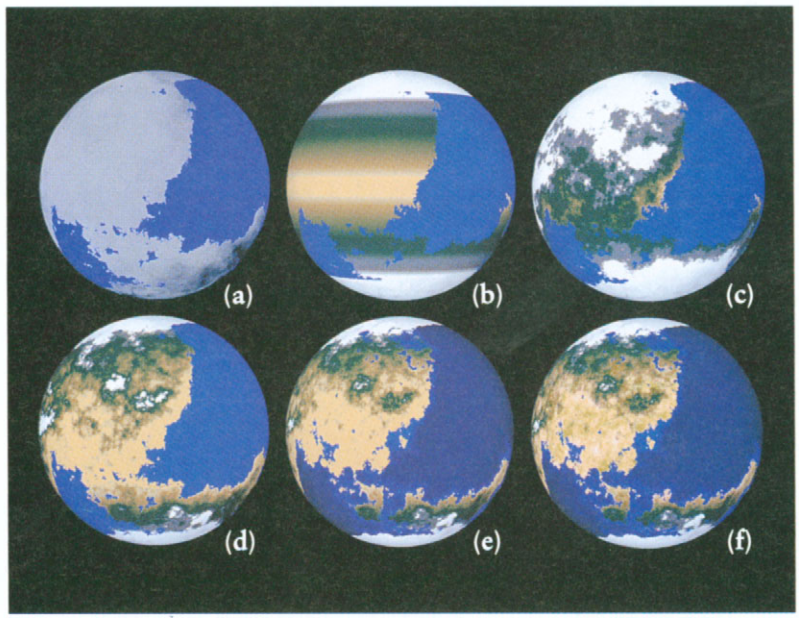

FIGURE 15.11 A sequence of stages in the development of a planetary structure. Copyright © F. Kenton Musgrave.

distortion.) We also take into account the displacement map, so that the "altitude" of the terrain may affect the climate. Note that altitude and latitude represent two independent quantities that could be used as parameters to a two-dimensional color map; to date I have used only a one-dimensional color spline for simplicity.

Next we add an exponentiation parameter to the color spline index computed earlier, to allow us to "drive back the glaciers" and expand the deserts to a favorable balance, as in Figure 15.11(d).

We now modify the oceans, adjusting the sea level for a pleasing coastline and making the color a function of "depth" to highlight shallow waters along the coastlines, as seen in Figure 15.11(e). Depth of the water is calculated in exactly the same way as the "altitude" of the mountains—as the magnitude of the bump vector in the direction of the surface normal. This depth value is used to darken the blue of the water almost to black in the deepest areas. It might also be desirable to modify the surface properties of the texture in the ocean areas, specifically the specular highlight, as this significantly affects the appearance of Earth from space (again, see Kelley 1988), although I haven't yet tried it.

Finally, we note that the "desert" areas about the equator in Figure 15.11(e) are quite flat and unrealistic in appearance. Earth, by contrast, features all manners of random mottling of color. By interpreting an fBm function as a color perturbation, we can add significantly to the realism of our model: compare Figure 15.11(e) and 15.11(f). This added realism is of an artistic nature—the color mottling of Earth does not, in general, resemble fBm—but it is nevertheless aesthetically effective.

This code accomplishes all of the above:

```
#define N_OFFSET 0.7
#define VERY_SMALL 0.0001
surface
terran(float Ka = .5, Kd = .7;
       float spectral_exp = 0.5;
       float lacunarity = 2, octaves = 7;
       float bump_scale = 0.07;
       float multifractal = 0;
       float dist_scale = .2;
       float offset = 0;
       float sea_level = 0;
       float mtn_scale = 1;
       float lat_scale = 0.95;
       float nonlinear = 0;
       float purt_scale = .9;
       float map_exp = 0;
       float ice_caps = 0.9;
       float depth_scale = 1;
       float depth_max = .5;
       float mottle_limit = 0.75;
       float mottle_scale = 20;
       float mottle_dim = .25;
       float mottle_mag = .02;)
{
   point PP, P2;
   vector PtN;
   float chaos, latitude, purt;
   color Ct;
   point Ptexture, tp;
   uniform float i;
   float o, weight;   /* Loop variables for fBm calculation */
   float bumpy;
   float filtwidth, fw;

   /* Do all shading in shader space */
   Ptexture = transform("shader", P);
   filtwidth = filterwidthp(Ptexture);
   PtN = normalize(vector Ptexture); /* Normalize Ptexture to radius 1.0 */
```

```
/*********************************************************************
* First, figure out where we are in relation to the oceans/mountains.
* Note: this section of code must be identical to "terranbump" if you
* expect these two shaders to work well together.
*********************************************************************/

if (multifractal == 0) {   /* use a "standard" fBm bump function */
   bumpy = fBm(Ptexture, filtwidth, octaves, lacunarity, spectral_exp);
} else {                   /* use a "multifractal" fBm bump function */
   /* get "distortion" vector, as used with clouds */
   Ptexture += dist_scale * filteredvsnoise(Ptexture, filtwidth);
   /* compute bump vector using MfBm with displaced point */
   o = spectral_exp; tp = Ptexture;
   fw = filtwidth;
   weight = abs(filteredDistNoise(tp, fw, 1.5));
   bumpy = weight * filteredsnoise(tp, fw);

   /* Construct a multifractal */
   for (i = 1; i < octaves && weight >= VERY_SMALL && fw < 1; i += 1) {
      tp *= lacunarity;
      fw *= filtwidth;
      /* get subsequent values, weighted by previous value */
      weight *= o * (N_OFFSET + snoise(tp));
      weight = clamp(abs(weight), 0, 1);
      bumpy += snoise(tp) * min(weight, spectral_exp);
      o *= spectral_exp;
   }
}

/* get the "height" of the bump, displacing by offset */
chaos = bumpy + offset;
/* set bump for land masses (i.e., areas above "sea level") */
if (chaos > sea_level) {
   chaos *= mtn_scale;
   P2 = P + (bump_scale * bumpy) * normalize(N);
} else P2 = P;
N = calculatenormal(P2);

/*********************************************************************
* Step 2: Assign a climate type, roughly by latitude.
*********************************************************************/

/* make climate symmetric about equator */
latitude = abs(zcomp(PtN));

/* Fractally purturb color map offset using "chaos"
 * "nonlinear" scales purturbation-by-z
 * "purt_scale" scales overall purturbation
 */
```

```
    latitude += chaos*(nonlinear*(1-latitude) + purt_scale);
    if (map_exp > 0 ) /* Perform nonlinear "driving the glaciers back" */
      latitude = lat_scale * pow(latitude,map_exp);
  else latitude *= lat_scale;

    if (chaos > sea_level) {
      /* Choose color of land based on the following spline.
       * Ken originally had a huge table. I was too lazy to type it in,
       * so I used a scanned photo of the real Earth to select some
       * suitable colors.—Larry Gritz
       */
      Ct = spline(latitude,
            color(.5, .39, .2),
            color(.5, .39, .2),
            color(.5, .39, .2),
            color(.2, .3, 0 ),
            color( .085, .2, .04),
            color(.065, .22, .04),
            color(.5, .42, .28),
            color(.6, .5, .23),
            color(1,1,1),
            color(1,1,1));

      /* Mottle the color to provide visual interest */
      if (latitude < mottle_limit) {
         PP = mottle_scale * Ptexture;
         purt = fBm(PP, mottle_scale*filtwidth, 6, 2, mottle_dim);
         Ct += (mottle_mag * purt) * (color(0.5, 0.175, 0.5));
      }
    }
    else {
      /* Oceans */
      Ct = color(.12, .3, .6);
      if (ice_caps > 0 && latitude > ice_caps)
         Ct = color(1,1,1); /* Ice color */
      else {
         /* Adjust color of water to darken deeper seas */
         chaos -= sea_level;
         chaos *= depth_scale;
         chaos = max(chaos, -depth_max);
         Ct *= (1+chaos);
      }
    }

    /* Shade using matte model */
    Ci = Ct * (Ka * ambient() + Kd * diffuse(faceforward(normalize(N),I)));
    Oi = Os; Ci *= Oi;
}
```

Selene

Now I'll show you an example of extreme hackery in pursuit of a specific visual end: my texture that can turn a featureless gray sphere into an imitation of the Moon, replete with lunar highlands, maria, and a single rayed crater.[6] I reasoned that one crater, if spectacular enough, would suffice to give an overall impression of moonliness. In retrospect, I guess it was, because this image caused Digital Domain to contact Larry and me for help in modeling the Moon for the movie *Apollo 13*. This in turn led to my being employed there, where I got to experience firsthand the madness and excitement of the Hollywood digital effects business, working on movies like *Titanic, Dante's Peak,* and *Air Force One*. Heady stuff—everyone should try it for a year or two!

Back to the moon texture: The highlands/maria part of the texture is just a simpler (in that it is not multifractal) variation on the continent/ocean part of the above texture. The rayed crater is the interesting part. It is a classic example of ontogenetic modeling taken to an extreme: the dynamics of such an impact and its resulting visual consequences would be very difficult to model physically, yet the phenomenon is essential to the appearance of the Moon. So I set out to construct a reasonable visual semblance through whatever chicanery I could devise. The resulting crater has two parts, which I originally implemented as two separate C functions: the bump-mapped crater rim and the rays, which are simply a surface color texture.

The crater bump map consists of a central peak, a common substrate-rebound feature seen in large impact craters; an unperturbed crater floor, which is resurfaced with smooth lava in the real craters; and a ring that delineates the edge of the crater. The ring is designed to have a steep slope on the inside and a more gradual one on the outside, again in emulation of the structure of real impact craters. Furthermore, the outside ring has a texture of random ridges, concentric with the crater. This is in emulation of compression features in the crust around the edge of the crater, formed when the shock of impact "bulldozed" the crater out from the center. We obtain this texture with radially compressed fBm. The composite result of these features is a detailed, fairly realistic model of a crater. Since it is applied as a bump map, as with ray tracers, you must be careful not to try to view the crater at a very low, glancing

6. If you've never seen a rayed crater before, just grab a pair of binoculars next time there's a full moon, and have a look. They're all over the place, they're not particularly subtle, and some have rays that reach more than halfway around the globe of the Moon. The rays are the result of splattering of ejecta from the impact that made the crater.

angle, for then the lack of geometric detail becomes evident. A problem with bump maps is that since they do not affect geometry, bump-mapped features cannot obscure one another, as the raised near rim of a real crater would obscure your view of the far rim at low angles of view. If your renderer supports displacement maps or QAEB primitives, you can model the geometry correctly.

The crater ray construction is inspired by the famous Doc Edgerton photo of the splashing milk drop. (You know, the one where it looks like a crown, with all the little droplets evenly spaced around the rim of the splash.) This high degree of regularity inspired me to build my texture in a similar way: with a number of rays, evenly spaced, with only a small random displacement in their spacing. The rays take the form of weighting functions, the width of which asymptotically approaches zero as we move out along the length of the ray. There is a discontinuity in the weighting function between rays, but this has never been visible in practice. The weighting is applied to an fBm splatter texture, which is stretched out *away* from the crater, exactly opposite of the compression texture outside the rim of the crater.

This crater ray texture looked much too regular to be realistic. So next I tried a fractal (fBm) splatter for the rays; the result was much too irregular. It seems that the behavior of nature lies somewhere between the two. I eventually settled on a combination of both: I use fractal splatter for the short-range ejecta and the ray scheme for the long-range ejecta. The combination looks reasonably good, although I think it would benefit from further, artful randomization of the ray structure. At the time I developed this model, I didn't need to inspect the resulting moon up close—it was designed to serve as a backdrop for earthly scenes—so I called it "good enough" and moved on, only slightly embarrassed to have put so much time and effort into such an entirely ad hoc model.

The code for the lunar texture follows. Figures 15.12 and 20.4 illustrate what it can look like.

```
surface
luna (float Ka = .5, Kd = 1;
      float lacunarity = 2, octaves = 8, H = 0.7;
      color highland_color = .7;
      float maria_basecolor = .7, maria_color = .1;
      float highland_threshold = -0.2;
      float highland_altitude = 0.01, maria_altitude = 0.004;
      float peak_rad = .0075, inner_rad = .01, rim_rad = .02, outer_rad = .05;
      float peak_ht = 0.005, rim_ht = 0.003;)
{
   float radial_dist;
   point PP;
   float chaos;
   color Ct;
```

FIGURE 15.12 The rayed crater is prominent on this procedurally textured moon. Copyright © 1994 F. Kenton Musgrave.

```
float temp;
point vv;
float uu, ht;
float lighten;
point NN;
float pd; /* pole distance */
float raydist;
float filtwidth;
float omega;
PQ = P;

PP = transform("shader", P);
filtwidth = filterwidthp(PP);
NN = normalize(N);

radial_dist = sqrt(xcomp(PP)*xcomp(PP) + ycomp(PP)*ycomp(PP));
omega = pow(lacunarity, (-.5)-H);
chaos = fBm(PP, filtwidth, octaves, lacunarity, omega);

/*** Get overall maria/highlands texture ************************/

chaos = fBm (PP, H, lacunarity, octaves);
```

```
/* Start out with the surface color, then modify as needed */
Ct = Cs;

/* Ensure that the crater is in one of the maria */
temp = radial_dist;
if (temp < 1)
   chaos -= .3 * (1 - smoothstep(0, 1, temp));

/* Determine highlands and maria */
if (chaos > highland_threshold) {
   PQ += chaos * highland_altitude * NN;
   Ct += highland_color * chaos;
} else {
   PQ += chaos * maria_altitude * NN;
   Ct *= maria_basecolor + maria_color * chaos;
}

/*** Add crater ************************************************/

pd = 1-v;
vv = vector(xcomp(PP)/radial_dist, 0, zcomp(PP)/radial_dist);
lighten = 0;
if (pd < peak_rad) {                    /* central peak */
   uu = 1 - pd/peak_rad;
   ht = peak_ht * smoothstep(0, 1, uu);
} else if (pd < inner_rad) {            /* crater floor */
   ht = 0;
} else if (pd < rim_rad) {              /* inner rim */
   uu = (pd-inner_rad) / (rim_rad - inner_rad);
   lighten = .75*uu;
   ht = rim_ht * smoothstep(0, 1, uu);
} else if (pd < outer_rad) {            /* outer rim */
   uu = 1 - (pd-rim_rad) / (outer_rad-rim_rad);
   lighten = .75*uu*uu;
   ht = rim_ht * smoothstep(0, 1, uu*uu);
} else ht = 0;

/* Lighten the higher areas */
PQ += ht * NN;
lighten *= 0.2;
Ct += color(lighten,lighten,lighten);

/* Add some noise to simulate rough features */
if (uu > 0) {
   if (pd < peak_rad) {                 /* if on central peak */
   vv = 5*PP + 3 * vv;
   ht = fBm(vv, filterwidthp(vv), 4, 2, 0.833);
   PQ += 0.0025 * uu*ht * NN;
 } else {
   vv = 6*PP + 3 * vv;
```

```
        ht = fBm(vv, filterwidthp(vv), 4, 2, 0.833);
        if (radial_dist > rim_rad) uu *= uu;
        PQ += 0.0025 * (0.5*uu + 0.5*ht) * NN;
      }
    }

    /*** Generate crater rays ************************************/

    lighten = 0;
    if (pd >= rim_rad && pd < 0.4) {
        float fw = filterwidth(u);
        lighten = smoothstep(.15, .5, filteredsnoise(62*u, 62*fw) );
        raydist = 0.2 + 0.2 * filteredsnoise(20 * mod(u+0.022,1), 20*u);
        lighten *= (1 - smoothstep(raydist-.2, raydist, pd));
    }
    lighten =0.2 * clamp(lighten, 0, 1);
    Ct += color(lighten, lighten, lighten);

    /* Shade like matte */
    Ci = Ct * (Ka * ambient() + Kd * diffuse(faceforward(normalize(N),I)));
    Oi = Os; Ci *= Oi;
}
```

RANDOM COLORING METHODS

Good painters evoke worlds of color in a painting, and even in a single brush stroke. Van Gogh, who painted with a palette knife, not a brush, executed several paintings in the morning, took a long lunch, then did several more in the afternoon. Painting in a hurry, he didn't mix his paints thoroughly before applying a thick blob to the canvas. Thus each stroke has a universe of swirling color within it. Seurat's pointillism is another form of what painters call *juxtaposition*, or the use of a lot of different colors to average to some other color. This is part of the visual complexity that, to me, distinguishes good paintings from most computer graphics.

Generating visual complexity is pretty much the name of my game, so I've devised some methods for procedurally generating complexity in color. They utilize the same kinds of fractal functions that we use in modeling natural phenomena.

Random fBm Coloring

My first attempt at random coloring was simply fBm interpreted as color. For this, you start with a vector-valued fBm that returns three values and interpret that 3-vector as an RGB color. This can work pretty well, as seen in Figure 20.9. Unfortunately, since fBm has a Gaussian distribution with an expected value of zero, when one of the values (say, red) is fairly far from zero, the other two are likely to be close

to zero. This yields a preponderance of red, green, and blue blotches. We want more control and color variation than that.

The GIT Texturing System

I wanted to obtain a rich, fractal variation in color detail, similar to the juxtaposition in a van Gogh stroke or a local area in a Seurat. This juxtaposition should average to a user-specified color, even though that color may not be present anywhere in the resulting palette. We also want easy user control of the color variation in the juxtaposition palette. Control of this palette is accomplished by manipulating the values of a 4×4 transformation matrix, but this is too mathematically abstract to constitute an effective user interface (UI), to say the least. We desire a simple and intuitive UI that an artist can use effectively, without knowledge of the underlying math.

Long ago I gave this idea the wonderfully unpretentious—not!—moniker "generalized Impressionistic texture," or GIT for short. (We need more TLAs—three-letter acronyms.) The GIT matrix generator system takes the form of a time-varying swarm of color samples in a color space, usually the RGB color cube. The center of the swarm is translated (moved) to the position in the color space of the desired average color of the resulting palette. A swarm of color sample chips is then manipulated to obtain the desired variation in color. This is accomplished by rotating and scaling the principal axes of color variation within the color space.[7] If you think of the scattering of color samples as lying within an ellipsoid, or an elongated M&M, the principal axes correspond to the length, width, and thickness of the M&M. After some manipulation, you might have a major axis of variation along, for instance, blue-to-yellow, with less significant variations along two other perpendicular axes in the chosen color space. The idea is that you can have a lot of different colors present, with their average clearly specified and smooth interpolation between them well defined.[8] The major variations can be along any straight line in color space, not just red, green, or blue; this is the flexibility of the system.

The simplest underlying mathematical model requires that the three axes of color variation be mutually perpendicular. In this model we build a standard 4×4

7. Which color space you use is important, as it affects the character of available color variations. To date, we've only implemented the RGB color cube. It is a little more challenging to display other color spaces as polyhedra, due to their nonrectilinear shapes. It shouldn't be too hard to do, though.

8. The average isn't quite as clear as it may seem, as the ellipsoid can violate the bounds of the color space. You can handle this by either clipping to the boundaries or reflecting them back into the color space (the solution we've used). Either solution will skew the average color away from that at the center of the ellipsoid, which marks the presumed average. There's no easy fix for this.

transformation matrix encoding rotation, translation, and scaling. The transformation matrix is built using interactive controls that let the user manipulate the scatter plot of color samples in the three-dimensional color space. In our implementation, the scatter plot is constantly changing: it is a circular queue of vector values representing offsets from the center of the ellipsoid, which represents the average color. When a sample reaches the end of the queue, it is replaced with another random sample. The random samples are gotten by evaluating vector-valued fBm at random points in its three-dimensional domain. We use about 100 samples at a time, the actual number being controlled by a slider. More samples span the range of colors more accurately, but the ones in front tend to obscure those behind, so you can't see the entire range of colors being spanned. The idea behind the circular queue is that by having the random samples constantly changing with time, you can get a pretty good idea of the range of colors spanned just by watching for a while.

When the desired distribution of color variations has been determined interactively, the resulting transformation matrix is used by a texture routine in a renderer. The result is a procedural solid texture with wonderfully rich variations in color. The colors of the mountains in Figures 15.13, 15.14, 16.6, 20.18, and 20.20 come from such GIT textures. I find them very pleasing because they finally start to capture in synthetic imagery some of the color complexity and subtlety we see in paintings.

An Impressionistic Image Processing Filter

The GIT scheme generates solid textures that are usually applied to the surfaces of objects in a scene. In painting, juxtaposition is in image space—on the canvas—not in world space or object space, as with solid textures. Hence we, Myeong Lim and myself, sought to apply GIT texturing in image space to digital images. In this case, the matrix is determined by performing *principal components analysis* (Gonzalez and Woods 1992) to local areas in the image.[9] In principal components analysis, the Hotelling transform is applied to a scattering of data, yielding the *autocorrelation matrix* for the distribution. This matrix encodes some magic values known mathematically as the *eigenvectors,* which correspond to the principal axes described earlier, and the *eigenvalues,* which correspond to the length of those axes. More mathematical magic! The data points we provide to the Hotelling transform are the RGB values of pixels, ranging from a fairly small neighborhood around a given pixel to the entire image.

9. Explaining this is well beyond the scope of this book; see Gonzalez and Woods (1992) for details if you're interested. But be advised, it involves some pretty heavy linear algebra and statistics.

FIGURE 15.13 *Slickrock I* features a procedural terrain with adaptive level of detail and a terrain model constructed from a ridged basis function—hence the ridges everywhere and at all scales. The same subtle color perturbation has been applied to the surface as in Figure 20.9, but it has been "squashed down" vertically to make it resemble sedimentary rock strata. Copyright © F. Kenton Musgrave.

The autocorrelation matrix encodes exactly the same information as the interactively derived matrix in the GIT scheme. We use it in a similar, but slightly different, way: the juxtaposition is now expressed in a synthetic brush stroke. This is applied to the image as a blurring, yet detail-adding filter. The added detail is in the smeared colors in the synthetic brush strokes. To accommodate this added detail, we generally expand the image by a factor of four to eight in both dimensions. Standard image processing routines are used to determine lightness gradients in the input image, and the brush strokes are applied cross-gradient (Salisbury et al. 1994) to resemble an artist's strokes. Thus, if the image faded from dark at the bottom to light at the top, the strokes would be horizontal. While blurring the image underneath along the direction of the stroke (Cabral and Leedom 1993), random detail is simultaneously added in the form of fractal color juxtaposition. Figure 15.15 shows this in practice. This application of the GIT idea is probably less successful than commercially available paint programs such as Painter, but it was an interesting experiment. We never spent much time on the model for the brush strokes; this scheme could provide an interesting filter if the details were worked out.

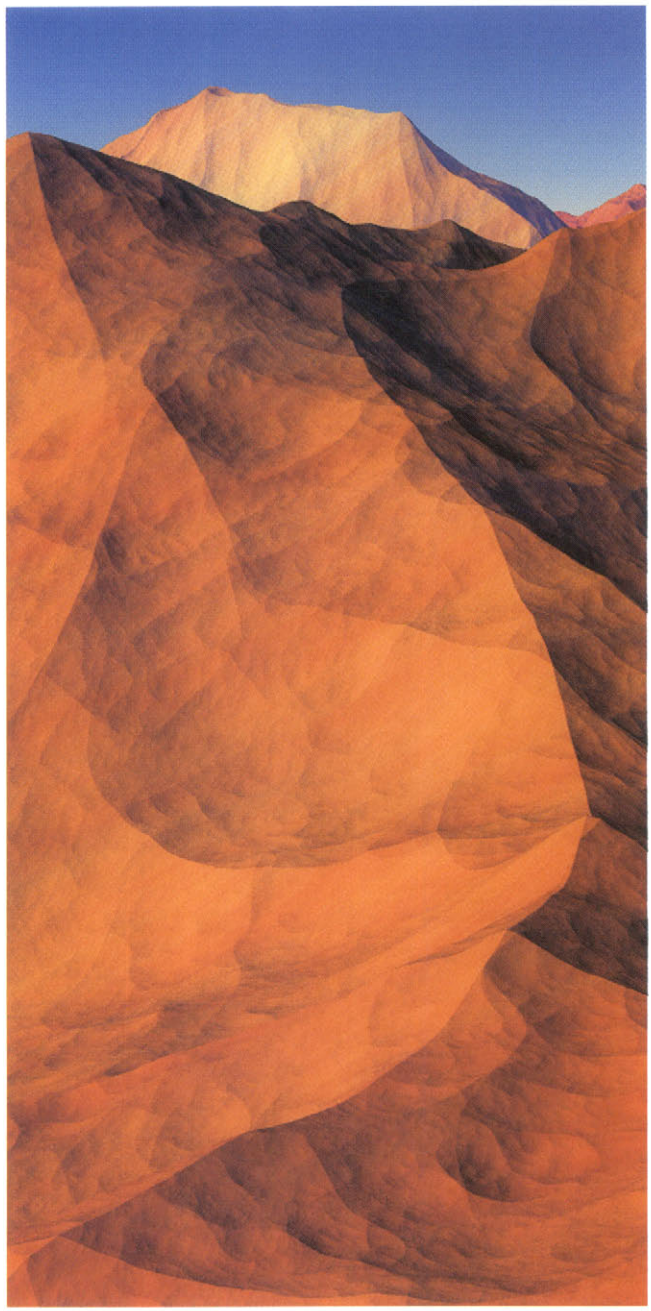

FIGURE 15.14 *Slickrock III* illustrates a GIT texture applied to a terrain. Both the terrain and the texture feature adaptive level of detail, as described in Chapter 17. Copyright © F. Kenton Musgrave.

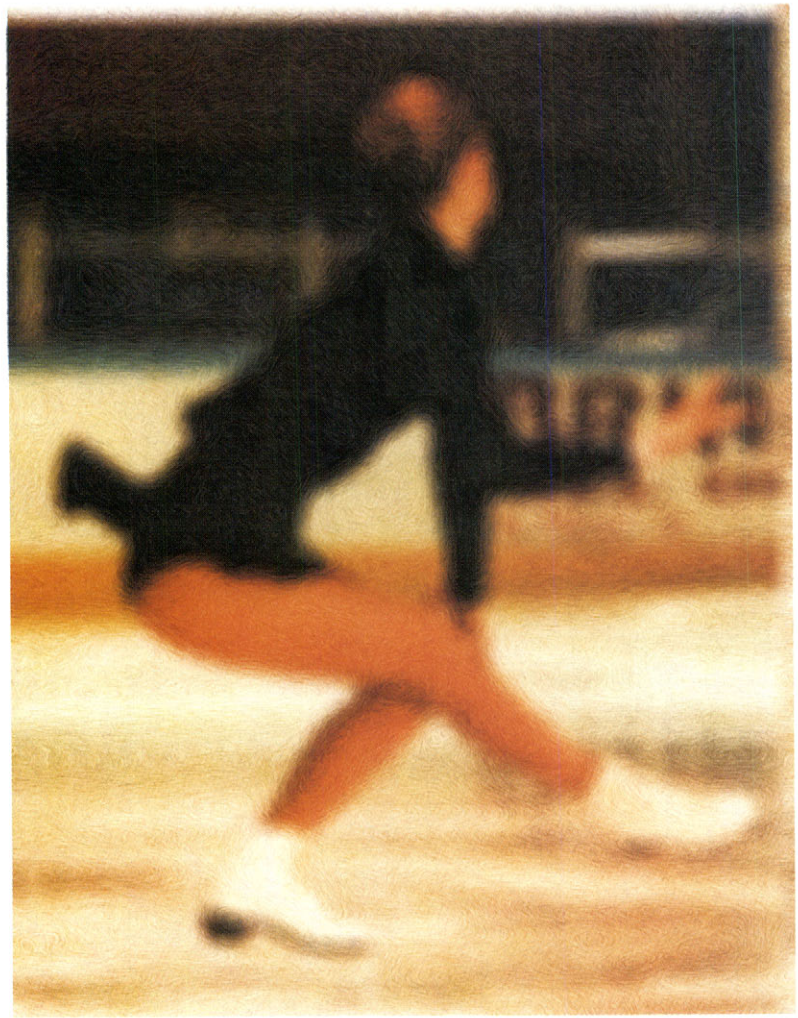

FIGURE 15.15 A GIT-processed image of Beth Musgrave skating, by Myeong Lim. The GIT processing adds random colors and the painterly quality. Copyright © F. Kenton Musgrave.

The "Multicolor" Texture

I always like to automate things as much as possible to see what the computer can be made to do on its own.[10] Being interested in the kind of painterly textures that the GIT experiments were designed to create, I set out to design a "painterly" procedural texture that is entirely random (see Figure 15.16). The result is, I think, rather

10. My considered view of the computer's role in generating art is that it is like an idiot savant assistant: it is extremely simple-minded, incapable of doing anything without the most exhaustively precise directions, but fantastically quick in what it does well—calculations, if you will. Its power in this can lead to fabulous serendipity, as we will see in Chapter 19, but the computer is never truly creative. All creativity resides in the human operator. While the computer can sometimes *appear* to be fabulously creative, it is an illusion, at least in these schemes.

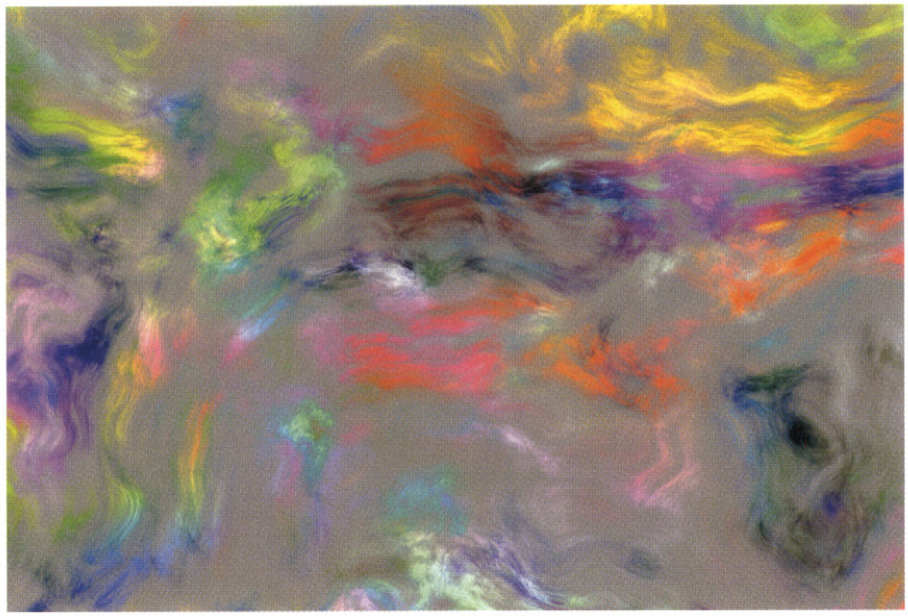

FIGURE 15.16 The "multicolor" texture attempts to capture some of the richness of color juxtaposition seen in paintings. Copyright © F. Kenton Musgrave.

striking, so I'll describe it here. It's a solution looking for a problem—I haven't found a way to include it in any images—but perhaps someday someone will find a use for it. It's also a good example of how a hacked-together texture can become brittle: small changes in the input parameters can "break" the result (meaning, your beautiful texture gets ugly).

The idea here is to generate a rich, painterly combination of random colors in a texture that looks something like flowing paint. Furthermore, we want the color to average to neutral gray. (It can actually average to any color, but I like the neutrality of 50% gray.) There are four basic elements in this texture: first, a vector-valued fBm that provides a random axis about which to rotate in color space; second, another vector-valued fBm provides a random color sample; third, a domain-distorted multifractal function modulates saturation in the random colors; and fourth, a rotation by a random amount around the random axis decorrelates the color from the RGB axes.

Let's go over this step by step. We start with a neutral gray plane. Then we get a random rotation axis from a vector-valued fBm function and store it for later use. We modulate its saturation with a multifractal function similar to the one rendered as a height field in Figure 14.3. Where that function is zero, the gray is unchanged. Where it's positive or negative, the gray becomes colored. Again, the color comes from a vector-valued fBm. As noted earlier, interpreting such an fBm vector as an RGB color doesn't give us the truly random variety of colors we want. We can get that using a random variation of the GIT scheme: simply build a random rotation matrix, corresponding to a GIT matrix, but without any translation or scaling. (The translation is inherent in the length of the fBm vector; the scaling is done by the

multifractal.) This may seem elaborate, but it has been my experience that you have to mess around with random colors a lot to get them to be truly random and to vary in a way that is pleasing to the eye.

Here's the shader code for "multicolor." As it (and some of the other textures in this chapter) is brittle, you might also want to look at the code for the original C version that appears on this book's Web site (*www.mkp.com/tm3*).

```
vector vMultifractalFunc(point p; float H, lacunarity, octaves, zero_off-set)
{
   point pos = p;
   float f = 1, i;

   vector y = 1;
   for (i = 0; i < octaves; i += 1) {
      y *= zero_offset + f * (2*vector noise(pos) - 1);
      f *= H;
      pos *= lacunarity;
   }
   return y;
}

/*
 * A multifractal multicolor texture experiment
 */
surface multicolor()
{
   vector axis, cvec, angle;
   point tx = transform("shader", P);
   float i;

   axis = 4.0 * vfBm(tx, filterwidthp(tx), 8, 2.0, 0.5);
   cvec = .5 * 5.0 * vfBm(tx*0.3, filterwidthp(tx), 7, 2.0, 0.5);
   tx += cvec;

   cvec += 4.0e5 * vMultifractalFunc(tx, 0.7, 2.0, 8, 0.2);
   angle = fBm(tx, filterwidthp(tx), 6, 2.0, 0.5);
   cvec = rotate(cvec, angle, point(0,0,0), axis);

   Ci = 0.5 + color(xcomp(cvec), ycomp(cvec), zcomp(cvec));

   /* Clamp color values to range [0..1] */
   for (i = 0; i<3; i += 1) {
      float c = abs(comp(Ci,i));
      if (c > 1)
      setcomp(Ci, i, 1-c);
   }
   Oi = Os;
   Ci *= Oi;
}
```

Ultimately, I think of this texture and the genetic textures I present in Chapter 19 as applications of the "naturalness" of the fractal functions developed earlier in this text, in pursuit of the look and feel of paintings, which I think of as being very "natural." That is, they may be man-made, but they appear very natural compared to the hard-edged artificiality of most synthetic images. The flow of the paint and the hand of the painter are very natural, and paintings are executed in a physical medium, rather than shuffling bits around for later output on some arbitrary device. This physicality is completely natural, compared to the abstractions of image synthesis. So I, personally, think of paintings as being a part of nature, as compared to what we're doing here. At any rate, I think of painters as the ancient masters of rendering, so I'm trying to learn from them by imitation. I find it fun and, when I get a good result, satisfying.

The "multicolor" texture provided the starting point for my genetic texture program described in Chapter 19. Development of this genetic program was driven partly by my desire to automate the generation of textures like "multicolor"—brittle textures like this are a real pain to design and refine—and partly by the knowledge that someday we're going to want to populate an entire virtual universe with procedurally generated planets, which, as we saw in this chapter, is also too much work to do on a planet-by-planet basis. You'll also see then that some of the fundamental functionality in my genetic program derives from the ideas I've described here about random coloration. At any rate, I think that the constructions in this chapter are mostly rather unique and nonobvious, since they involve a cross-fertilization of ideas from aesthetics and mathematics. I think that's cool. And best of all, they can make nice pictures.

PLANETARY RINGS

Rings like Saturn's are particularly easy to model (see Figures 15.17 and 15.18). You can simply employ a fractal function, evaluated as a function of radius, to control the transparency—and color, if you like—of a disk passing through the equator of the planet. The following C code is what was used to create Figure 15.18.

```
void Rings ( Vector intersect,
             double *refract,
             Color *color,
             double inner_rad, outer_rad,
             double f_dim, octaves, density_scale, offset,
             double inner_rolloff, outer_rolloff, /* in fraction of ring width */
             double text_scale, tx_offset, ty_offset, tz_offset )
{
```

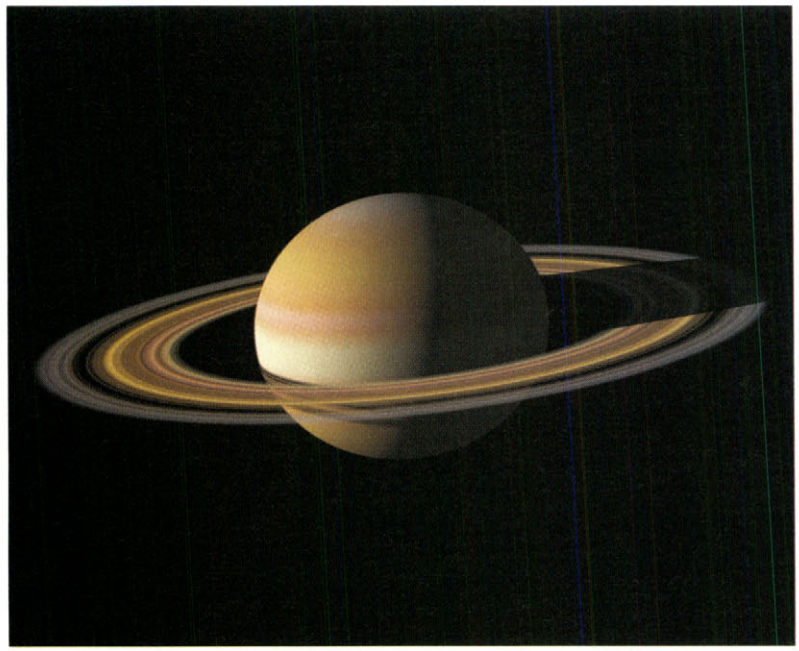

FIGURE 15.17 Planetary rings like Saturn's are easy to model by modulating density with a fractal function of radius.

FIGURE 15.18 Other style features a subtle use of the windywave shader and the model of Saturn seen in Figure 15.17.

```
    Vector  point;
    double  x_pos, y_pos, z_pos, radius, density, fBm();
    double  band_width, inner_rolloff_end, outer_rolloff_start, rolloff;

    x_pos = intersect.x;
    y_pos = intersect.y;
    z_pos = intersect.z;
    radius = sqrt (x_pos*x_pos + y_pos*y_pos + z_pos*z_pos);

    /* most of the work is in computing the inner and outer rolloff zones */
    if ( radius<inner_rad || radius>outer_rad ) {
          density = 0.0;
    } else {
          /* compute fBm texture */
          point = Vector (text_scale*radius + tx_offset, ty_offset, tz_offset);
          density = fBm( point, f_dim, 2.0, octaves );
          density = density_scale*density + offset;

          /* check for inner & outer rolloff zones */
          band_width = outer_rad - inner_rad;
          inner_rolloff_end =   inner_rad + band_width*inner_rolloff;
          outer_rolloff_start = outer_rad - band_width*outer_rolloff;
          if ( radius < inner_rolloff_end ) {
                /* get rolloff parameter in range [0..1] */
                rolloff = (radius-inner_rad)/(band_width*inner_rolloff);
                /* give rolloff desired curve */
                rolloff = 1.0 - rolloff;
                rolloff = 1.0 - rolloff*rolloff;
                /* do the rolling-off */
                density *= rolloff;
          } else if ( radius > outer_rolloff_start ) {
                /* get rolloff parameter in range [0..1] */
                rolloff = (outer_rad-radius)/(band_width*outer_rolloff);
                /* give rolloff desired curve */
                rolloff = 1.0 - rolloff;
                rolloff = 1.0 - rolloff*rolloff;
                /* do the rolling-off */
                density *= rolloff;
          }
    }
    /* clamp max & min values */
    if ( density < 0.0) density = 0.0;
    if ( density > 1.0) density = 1.0;

    transparency = 1.0 - density;

} /* Rings() */
```