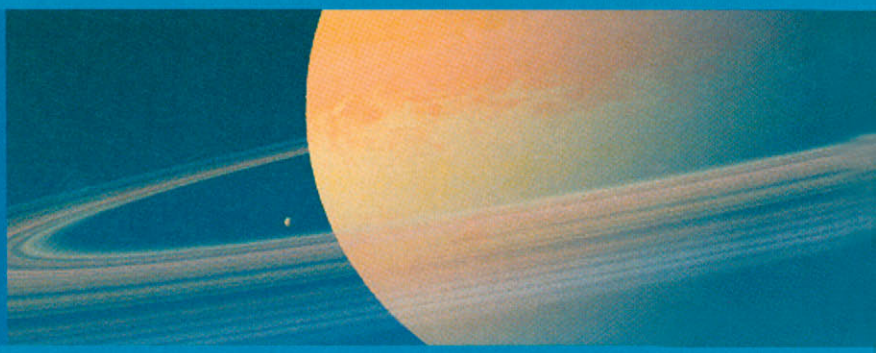
4



CELLULAR TEXTURING

STEVEN WORLEY

Procedural texturing uses fractal noise extensively. This book has multiple chapters that are virtually subtitled "More Applications of Fractal Noise." The major reason for this popularity is that noise is very *versatile*.

The noise function simply computes a single value for every location in space. We can then use that value in literally thousands of interesting ways, such as perturbing an existing pattern spatially, mapping directly to a density or color, taking its derivative for bump mapping, or adding multiple scales of noise to make a fractal version. While there are infinitely many functions that can be computed from an input location, noise's random (but controlled) behavior gives a much more interesting appearance than simple gradients or mixtures of sines.

This simple functional nature of noise makes it an adaptable tool that you might call a texture "basis" function. A basis function should be a scalar value, defined over $\Re^3$. A good basis function is useful in the same way noise is, since its value can be used in versatile, diverse methods like we do with noise. Certainly not all textures use basis functions; a brick wall pattern is not based on mapping a computed value into a color spline.

This brings us to the introduction of new basis functions based on *cellular texturing*. Their appearance and behavior are very different than noise's (which makes them a good complement to the noise appearance), but they are basis functions, like noise, and we can immediately use many of the fun ideas we've learned about for noise for the new basis.

Cellular texturing is related to randomly distributed discrete features spread through space. Noise has a "discoloration" or "mountain range" kind of feeling. Cellular textures evoke more of a "sponge," "lizard scales," "pebbles," or "flagstones" feeling. They often split space into small, randomly tiled regions called cells. Even though these regions are discrete, the cellular basis function itself is continuous and can be evaluated anywhere in space.

The behaviors of noise and cellular texturing complement each other. This chapter starts with a description of the definition of the cellular function to teach you enough to use it as a texture author. The second half of the chapter is about implementing the basis function itself. Since source code for implementation is included with this book, this section can be skipped if you wish. However, the cellular basis is a lot more "hackable" than noise's definition, so many advanced textures will rewrite parts of the algorithm; therefore, so a complete description of the classic implementation is provided.

THE NEW BASES

The cellular texturing basis functions are based on the fundamental idea of randomly scattering "feature points" throughout 3D space and building a scalar function based on the distribution of the points near the sample location. We'll define this main idea with a few simple functions.

For any location **x**, there is some feature point that lies closer to **x** than any other feature point. Define $F_1(\mathbf{x})$ as the distance from **x** to that closest feature point. Figure 4.1 shows an example of this in 2D. As **x** varies, F_1 varies continuously as the distance between the sample location and the fixed feature point varies. It's still continuous even when the calculation "switches" from one feature point to its neighbor that has now become the closest. The *derivative* of F_1 will change discontinuously at these boundaries when the two feature points are equidistant from the sample location.

These locations where the function F_1 "switches" from one feature point to the next (where its derivative is discontinuous) are along the equidistance planes that

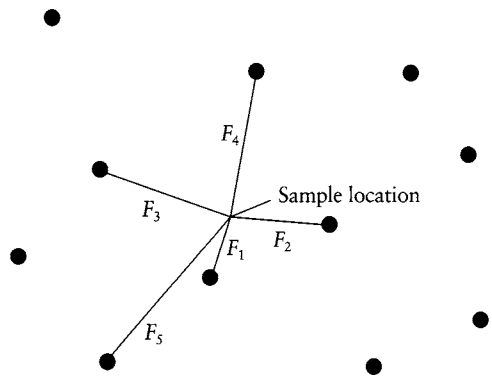


FIGURE 4.1 F_n values are the distance to the nth closest feature point.

separate two points in 3D space. These planes are exactly the planes that are computed by a Voronoi diagram, a partition of space into cellular regions.

The function $F_2(\mathbf{x})$ can be defined as the distance between the location $\mathbf{x}$ and the feature point that is the *second* closest to $\mathbf{x}$. With similar arguments as before, F_2 is continuous everywhere, but its derivative is not at those locations where the second-closest point swaps with either the first closest or third closest. Similarly, we can define $F_n(\mathbf{x})$ as the distance between $\mathbf{x}$ and the nth closest feature point.

The functions F have some interesting properties. F_n are always continuous. F_n are nondecreasing; $0 \leq F_1(\mathbf{x}) \leq F_2(\mathbf{x}) \leq F_3(\mathbf{x})$. In general, $F_n(\mathbf{x}) \leq F_{n+1}(\mathbf{x})$ by the definition of F_n. The gradient of F_n is simply the unit direction vector from the nth closest feature point to $\mathbf{x}$.

These careful definitions are very useful when we want to start making interesting textures. As with the noise function, mapping values of the function into a color and normal displacement can produce visually interesting and impressive effects. In the simplest case, $F_1(\mathbf{x})$ can be mapped into a color spline and bump. The character of F_1 is very simple, since the function increases radially around each feature point. Thus, mapping a color to small values of F_1 will cause a surface texture to place spots around each feature point—polka dots! Figure 4.2 shows this radial behavior in the upper left.

Much more interesting behavior begins when we examine F_2 and F_3 (upper right and lower left in Figure 4.2). These have more rapid changes and internal structure and are slightly more visually interesting. These too can be directly mapped into colors and bumps, but they can also produce even more interesting patterns by forming linear combinations with each other. For example, since $F_2 \geq F_1$ for all $\mathbf{x}$, the function $F_2(\mathbf{x}) - F_1(\mathbf{x})$ is well defined and very interesting, as shown in the bottom right of the figure. This combination has a value of 0 where $F_1 = F_2$, which occurs at the Voronoi boundaries. This allows an interesting way to make a latticework of connected ridges, forming a veinlike tracery.

We have interesting patterns in the basis functions F_1, F_2, F_3 and now we see that the linear combination $F_2 - F_1$ forms yet another basis. This leads us to experiment with other linear combinations, such as $2F_3 - F_2 - F_1$ or $F_1 + F_2$. In fact, most linear combinations tend to be interesting! The best way to see the different possible appearances is to generate multiple different linear combinations randomly and simply look at them, and save the ones that appeal to you. A user interface that lets you edit the linear coefficients is also fun, but a simple button to generate random coefficients is better. (It seems more useful to see brand-new patterns each time than to edit four mystery sliders that are difficult to "aim" to any goal.)

F_4 and other high n start looking similar, but the lower values of n (up to 4) are quite interesting and distinct. More importantly, linear combinations of these F_n

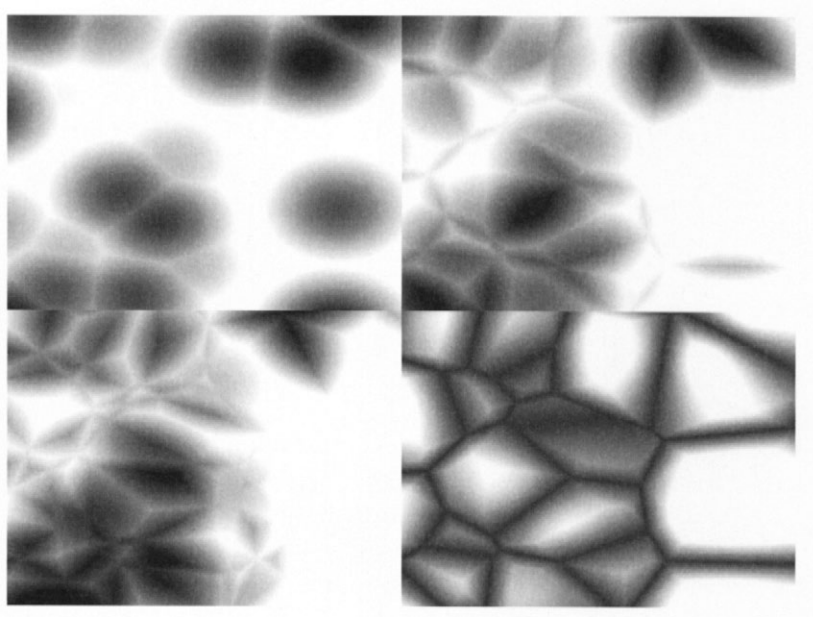


FIGURE 4.2 Gradient-mapped samples of F_1, F_2, F_3, and $F_2 - F_1$.

have more "character" than the plain F_n, particularly differences of two or more simple bases. Since the range of the function will change, it's easiest to evaluate 1000 samples or so and learn the minimum and maximum ranges to expect, and remap this value to a more stable 0–1 range. For "final" textures, this normalization only has to be precomputed once and hardwired into the source code afterwards.

Figure 4.3 shows 20 sample surfaces that are all just examples of combinations of these low-n basis functions ($C_1F_1 + C_2F_2 + C_3F_3 + C_4F_4$ for various values of C_n).

These patterns are interesting and useful, but we can also use the basis functions to make *fractal* versions, much like noise is used to produce fractal noise. By computing multiple "scales" of the function at different weights and scaling factors, a more visually complex appearance can be made. This is a simple loop, computing a function $G_n = \sum 2^{-i} F_n(2^i \mathbf{x})$ for moderate ranges of i ($i = 0$–5), and using G_n as the index for colors and bumps.

The fractal versions of any of the basic basis function combinations become extremely appealing. Figure 4.4 shows a fractal version of F_1 forming the spotted pattern and bumps on the hide of a creature. Fractal noise is used for the tongue, and a linear gradient is applied to the main body for color variation. Other fractal versions of primitives are shown in the row of cut tori in Figure 4.5.

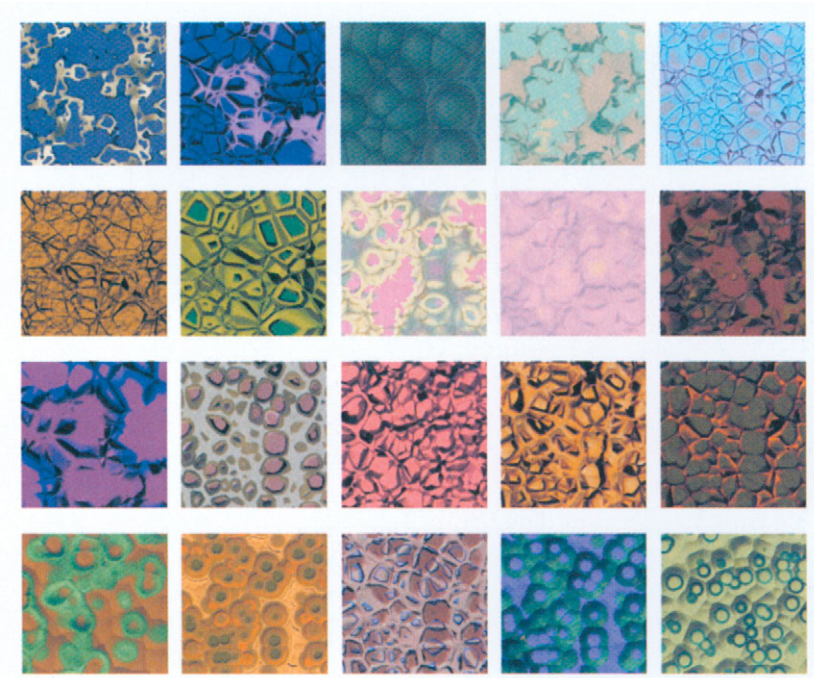


FIGURE 4.3 A variety of example appearances formed by linear combinations of the F_n functions.

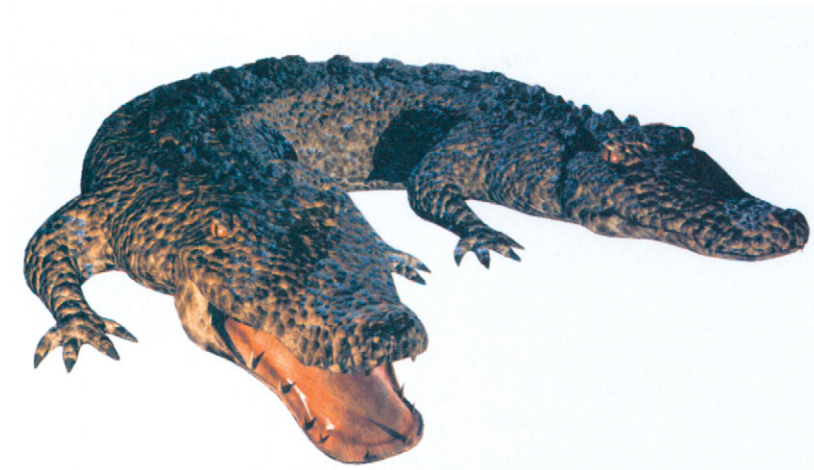


FIGURE 4.4 Natural-looking reptile hide using fractal-versions of the F_n functions.

FIGURE 4.5 More examples of fractal combinations.

The fractal version of F_1 is perhaps the most useful. Applied solely as a bump map, the surface becomes crumpled like paper or tinfoil. This surface has been extremely popular with artists as a way to break up a smooth surface, providing a subtle roughening with an appearance unlike fractal noise bumps. A surprising discovery was that a reflective, bumped-map plane with this "crumple" appearance bears an excellent resemblance to seawater, as shown in Figure 4.6. This bump-only fractal texture has become extremely popular in many renderers.

Since the cellular texture is a *family* of bases, it's fun to try more advanced incestuous combinations! Nonlinear combinations of simple polynomial products such as F_1F_2 or $F_3^2 - F_2^2$ are also interesting and useful texture bases. Again, renormalizing by empirically testing the output range makes the new basis easier to apply to color maps.

If the F_1 function returns a unique ID number to represent the closest feature point's identity, this number can be used to form features that are constant over a cell, for example, to shade the entire cell a single constant color. When combined with bumping based on $F_2 - F_1$, quite interesting flagstonelike surfaces can be easily generated. Figure 4.7 shows this technique, which also uses fractal noise discoloration in each cell. Note that unlike Miyata (1990), no precomputation is necessary and the surface can be applied on any shaped 3D object.

Bump mapping of the flagstonelike areas is particularly effective, and it is cheap to add since the gradient of F_n is just the radial unit vector pointing away from the appropriate feature point toward the sample location.

IMPLEMENTATION STRATEGY

It's not necessary to understand how to implement the cellular texture basis function in order to use it. But more than noise, the basis seems to encourage modifications and adaptations of the main algorithm to produce new effects, often with a very

FIGURE 4.6 Sea surface formed from bump-mapped fractal F_1 functions.

different behavior than the original version. The following sections describe my implementation method, hopefully to allow you to modify it to make your own alternatives. The source code is also commented, but like all software, understanding the algorithm first will make understanding the actual code much easier.[1]

1. If you do create interesting extensions or speedups, please contact me at *steve@worley.com!*

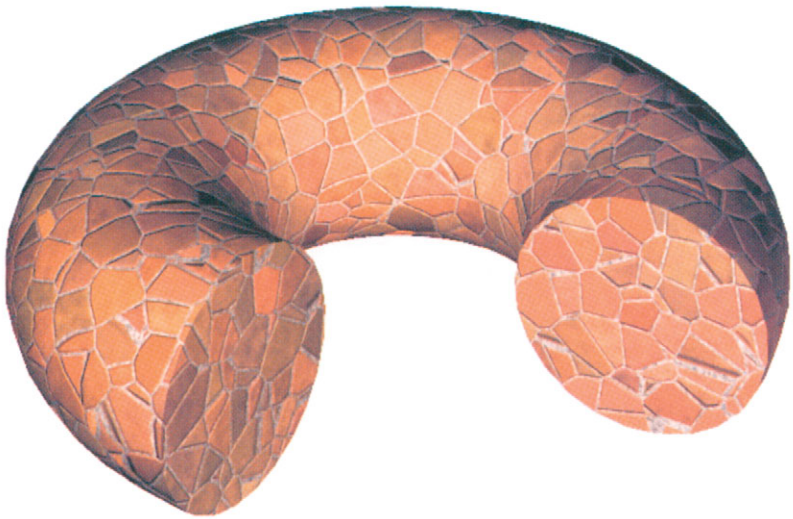


FIGURE 4.7 3D flagstone texture shows Voronoi cells.

The first step is to define how feature points are spread through space. The density and distribution of points will change the character of the basis functions. Our first assumption is that we want an *isotropic* function, to avoid any underlying pattern aligned with the world's axes. A simple idea like adding a point at every integer gridpoint and jittering their locations is easy to implement, but that underlying grid will bias the pattern, and it will be easy to see that "array" point layout.

The correct way to eliminate this bias is to keep the idea of splitting space into cubes, but choosing the number of points *inside* each cube in a way that will completely obscure the underlying cube pattern. We'll analyze this separately later.

Dicing Space

Since space will be filled with an infinite number of feature points, we need to be able to generate and test just a limited region of space at a time. The easiest way to do this is to dice space into cubes and deal with feature points inside each cube. This allows us to look at the points near our sample by examining the cube that the sample location is in plus the immediate neighbor cubes. An example is shown in Figure 4.8, where the "X" marks our sample location and dots show feature points in each cube. We can ignore the more distant cubes as long as we're assured that the feature points will lie within the 3×3 grid of local cubes.

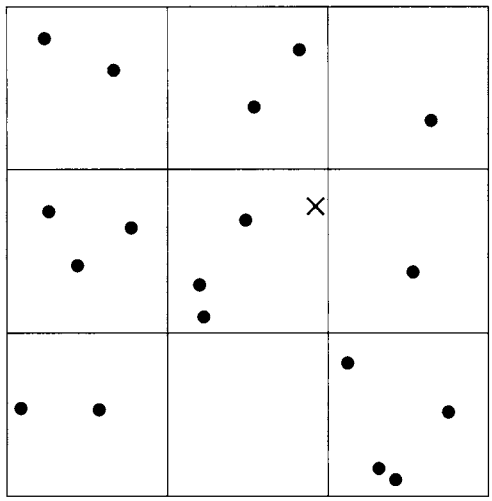


FIGURE 4.8 Searching for local feature points in neighboring cubes of space.

Each "cube" in space can be uniquely represented by its integer coordinates, and by simple floor operations we can determine, for example, that a point like (1.2, 3.33, 2.3) lies within the cube indexed by (1, 3, 2).

Now we determine how many and where feature points lie inside this cube. The "random" number we use to determine the number of points in a cube obviously must be unique to that cube and reproducible at need. There is a similar requirement in the noise function, which also uses a cubic lattice with fixed values associated with each gridpoint. We also need this seed to generate the location of the feature points themselves.

The solution to this problem is to hash the three integer coordinates of a cube into a single 32-bit integer that is used as the seed for a fast random number generator. This is easy to compute as something like $702395077x + 915488749y + 2120969693z \bmod 2^{32}$. The constants are random but chosen to be odd, and not simple multiples of each other mod 2^{32}. Like linear congruential random number generators, the *low-order* bits of this seed value are not very random.

We compute the number of points in the cube using this seed to pick a value from a short lookup table of 256 possibilities. This hardwired array of possible point populations is carefully precomputed (as described on page 145) to give us the "keep the points isotropic" magic property we desire. We use the *high-order* bits from our seed to index into the table to decide how many feature points to add into the cube. Since our table is a nice length of 256, we can just use the eight high-order

bits of the seed value to get the index. (The original 1996 paper used a different method for picking the point count, but the lookup table is both easier and more versatile.)

Neighbor Testing

Next, we compute the locations of the m feature points inside the sample cube. Again, these are values that are random, but fixed for each cube. We use the already initialized random number generator to compute the XYZ locations of each of the feature points. These coordinates are relative to the base of the cube and always range from 0 to 1 for each coordinate.

As we generate each point, we compute its distance to the original function evaluation location and keep a sorted list of the n smallest distances we've seen so far. As we test each point in turn, we effectively do an insertion sort to include the new point in the current list. This sounds expensive, but for typical values of n of 2 or 3, it only takes one or two comparisons and is not a major contributor to the algorithm's evaluation time.

This procedure finds the closest feature points and the values of F_1 to F_n for the points within the current cube of space (the one that contains the hit point). However, the feature points within a *neighboring* cube could quite possibly contain a feature point even closer than the ones we have found already, so we must iterate among the boundary cubes too. We could just test all 26 immediate neighboring cubes, but by checking the closest distance we've computed so far (our tentative nth closest feature distance) we can throw out whole rows of cubes at once by deciding when no point in the neighbor cube could possibly contribute to our list.

Figure 4.9 shows this elimination in a 2D example. The central cube has three feature points. We compute F_1 based on the feature point closest to the sample location (marked by "X"). We don't know yet what points are in the adjoining cubes marked by "?," but if we examine a circle of radius F_1, we can see that it's possible that the cubes labeled A, B, and C *might* contribute a closer feature point, so we have to check them. If we want to make sure our computation of F_2 is accurate, we have to look at a larger circle and additionally test cubes D and E. In practice, we can make these decisions quickly since it's easy to compare the current F distance to the distance of the neighbor cubes (especially since we can just compare the squared distances, which are faster to compute).

This kind of analysis also shows us that we need sufficient feature point density to make sure that one layer of neighbor cubes is enough to guarantee that we've found the closest point. We *could* start checking cubes that are even more distant

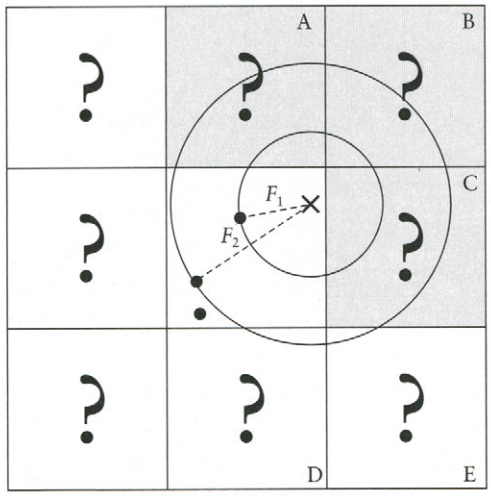


FIGURE 4.9 We don't need to test neighbor cubes that are too distant.

neighbors, but that becomes more and more expensive. Checking all neighbors requires at most $3^3=27$ cubes, but including more distant neighbors 2 cubes away would need $5^3=125$ cubes.

After we've checked all of the necessary neighbors, we've finished computing our values of F. If we computed F_n, we were effectively finding values for all F_1, F_2, . . . , F_n simultaneously, which is very convenient. The code included with this book returns all these values, plus the delta vectors corresponding to each feature point, plus a unique ID integer for each point (equal to the hashed cube ID plus the index of the feature point as it was computed). These all tend to be useful when using the function to form textures. If you don't need the extra information, you can modify the source code to return only the F_n values to gain a small speedup.

The Subtle Population Table

The desire for an isotropic distribution of points in space requires careful design. It can be done by choosing the number of points in each cube randomly but using a *Poisson distribution*. This distribution gives the probability of finding a specific number of points in a region when given a mean density. There may be more or less than this expected number of points in the region; the exact probabilities of any number of points in a region can be computed by using the discrete Poisson distribution function. If we generate the points inside of each cube randomly, with the

population based on the Poisson probabilities, the feature points will be truly isotropic and the texture function will have no grid bias at all.

Each cube in space may contain zero, one, or more feature points. We determine this on the fly quite simply by noting that the Poisson random distribution function describes the exact probabilities of each of the possible number of feature points occurring in the cube. For a mean density of λ feature points per unit volume, the probability of m points occurring in a unit cube is $P(\lambda,m) = \{(\lambda^{-m}\, e^{\lambda}\, m!)\}^{-1}$. Thus we can tabulate the probabilities for $m = 0, 1, 2, 3, \ldots$ easily. Computation is aided by the convenient recurrence relations $P(\lambda, 0) = e^{-\lambda}$ and $P(\lambda, m) = \frac{\lambda}{m} P(\lambda, m-1)$.

The value of λ to use is an interesting design decision. A high λ will tend to have a lot of feature points in each cube, and it will take extra time to generate and test the points. But a low λ will tend to have to evaluate more of the neighbor cubes. What's the best balance for speed? We're aided by the fact that we can change the feature scale of our function by simply scaling the input location by a constant. So no matter what λ we choose, we can renormalize the function to hide the specific λ. For the convenience of texture authors, I like to normalize the function such that the mean value of F_1 is equal to 1.0. This is similar to Perlin's decision to make the noise function vary over a characteristic distance of 1.0.

The obvious way to choose λ is to simply try different values and find which one gives the fastest average evaluation speed! But we do have an extra limitation. If λ is too low (less than 2.0 or so), then it's possible that the feature points are so sparse that the central 27 cubes are not enough to guarantee that we've found the closest point. We could start testing more distant cubes, but it's much more convenient to just use a high enough density to insure that the central 27 are sufficient. Also, notice that if we require the accurate computation of higher-order n values of F_n, we'll need a higher density of points to keep those more distant points within our "one cube" distance limit. Experiments (and some quick analysis) show that computation of an accurate F_n requires a density of $\lambda \approx n$. If the density is too low, our function fails and starts to sometimes return values for F_n that are discontinuous across the cube boundaries. This is not good.

Unfortunately, this lower limit for λ affects our efficiency, since by Murphy's law, fastest evaluation usually happens at lower λ values. So in practice, we can cheat by taking our carefully computed Poisson distribution and corrupting it to try to reduce the cases that cause problems. These are the cases when the population m is a low value of 0 or 1 (it's the low densities that don't give us enough candidates, and those empty cubes are particularly unhelpful). We could just *clip* the lowest allowable density to be at least 1. This helps evaluation efficiency enormously, since it allows us to use lower values of λ without causing the artifacts of discontinuous boundaries.

However, this clipping *violates our careful isotropic function design.* But if we choose to do this, we can at least do it knowledgeably and know that we can restore isotropy if we want to spend a little more CPU time to do it.

For actual implementation, we generate a set of integers that follow this Poisson density. By randomly selecting from this precomputed array, we can quickly compute cube populations that follow our chosen distribution. Since this table is precomputed, we can tweak and tune it as much as we like.

In practice, I've compromised speed versus isotropy by first generating a distribution following an ideal Poisson distribution. I then *bias* the distribution against those evil low populations by randomly increasing a fraction of the 0 and 1 populations by one. I can run lots of empirical tests to look for discontinuities indicating the density isn't sufficient and also at what average speed the function runs. For accurate computation of F_2, using a density of $\lambda = 1.60$ and incrementing 75% of the 0 and 1 populations is enough to prevent visible artifacts and give a good evaluation speed. Since most texturing usually only uses F_1 to F_4, we can tune for F_4 and find that $\lambda = 2.50$ and again a 75% random increment works. This is the precomputed table that's used in the example code for this book. There's also commented code for generating these lookup tables.

This population distribution seems overanalyzed, but the end results are worth it. While it's possible to simply use a constant population of, say, $m = 3$ and skip the table completely, the subtle axis bias can show up as noticeable artifacts that are difficult to analyze and understand. The Poisson table takes negligible extra evaluation overhead but gives you a noticeably higher-quality basis.

Extensions and Alternatives

The cellular noise function itself is extremely extendable and customizable. This is in contrast to Perlin noise, which tends to be viewed as a black box; you rarely have to open the hood to tinker with its engine. Even multifractals, introduced by Ken Musgrave, still use noise's basic function in its basic form.

A variety of small speedups can be made to the basic cellular basis algorithm and implementation, including making a version hardwired to return only specific F_n values, using fixed-point computation, using parallel vector CPU instructions such as MMX and SSE, changing the boundary cube segmentation to use hexagonal packing, and more.

Most fundamentally, the most interesting modification is to change the *definition of distance.* There are many different distance metrics that can be used in mathematics, and the Euclidean distance is just the simplest. The basis can instead be

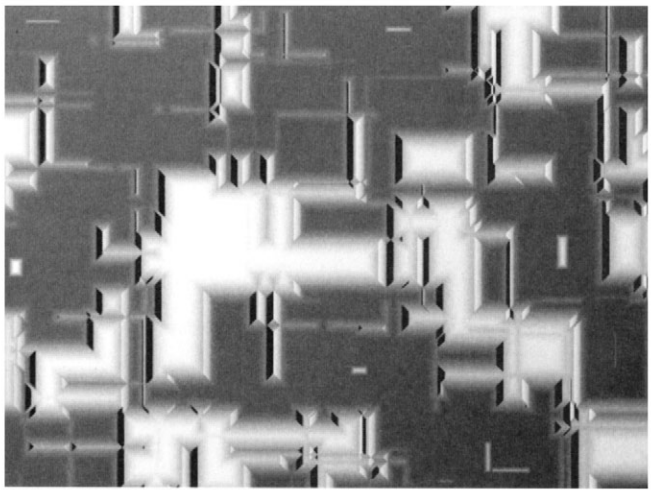


FIGURE 4.10 The F_1 "Manhattan" distance metric shows random rectangular shapes.

computed using the "Manhattan distance" between two points ($|dx| + |dy| + |dz|$), which forms regions that are rigidly rectangular but still random. These make surfaces like random right-angle channels—useful for spaceship hulls or circuit board traces. Figure 4.10 shows a nonfractal version of F_1 that uses this Manhattan distance metric. A radial coordinate version can cover spheres, creating a surface similar to the "Death Star." A whole range of cell shapes can be computed with a distance formula of the form $C_x dx^{n_x} + C_y dy^{n_y} + C_z dz^{n_z}$. Even more metrics are possible, including strange ones such as adding a small fractal noise variation to the Euclidean distance to make an oddly undulating craterscape. With these alternative metrics, the basis values are still monotonic ($F_{n+1} \geq F_n$), but the derivatives can change based on the metric you choose, which may affect your bump-mapping method.[2]

The main design constraint is that the distance metric should tend to increase as the Euclidean distance increases. This limitation is just to allow our simple "search all 27 neighboring cubes" algorithm to find the feature points within its search region. Since different metrics have different "reaches" that are not always easy to analyze, we usually use a simpler version of the basic algorithm that uses a larger

2. In practice, you can use a finite-difference bump-mapping method to allow arbitrary bumping without having to worry much about whether the derivative is easy to compute or not. See the discussion on page 170.

average density λ and checks all 27 neighbor cubes exhaustively. This means computation is slower than the optimized Euclidean distance version. After you've designed a new basis, you may be able to decrease λ and/or implement a fast rejection technique tailored to your new pattern. It's useful to make a test harness to evaluate your function millions of times randomly to measure speed as well as look for discontinuities in high F_n, which indicate you'll need a higher density for the limited 27-cube search.

Changes to other parts of the basic algorithm can produce other kinds of effects. The density of the feature points can be made to vary spatially, allowing for small, dense features in one area and larger features in another. Object geometry might be used to disperse precomputed feature spots (similar to Turk 1991) at the expense of requiring precomputation and table lookup, but gaining object surface dependence similar to the advantages Turk found. The algorithm is normally computed in 3D, but 2D variants are even faster; 4D variants can be used for animated fields, although these become very slow because of the necessity to compute up to $3^4=81$ cubes instead of $3^3=27$.

SAMPLE CODE

The Web site for this book (*www.mkp.com/tm3*) contains my implementation of the cellular texturing function, as well as a utility for generating the Poisson lookup tables. It is written in vanilla C for maximum portability. It is designed to return accurate values of F_n up to and including $n = 4$. It can return higher-order values as well, but in that case you may want to replace the Poisson table with a higher density to keep it artifact free. The function returns the values of F_n, the separation vector between the sample location and the nth feature point, and a unique ID number for that feature point. You are welcomed and encouraged to use this code in your own projects, private and commercial.

```
/* Copyright 1994, 2002 by Steven Worley
   This software may be modified and redistributed without restriction
   provided this comment header remains intact in the source code.
   This code is provided with no warrantee, express or implied, for
   any purpose.

   A detailed description and application examples can be found in the
   1996 SIGGRAPH paper "A Cellular Texture Basis Function" and
   especially in the 2003 book Texturing & Modeling, A Procedural
   Approach, 3rd edition. There is also extra information on the Web
   site http://www.worley.com/cellular.html.
```

```
   If you do find interesting uses for this tool, and especially if
   you enhance it, please drop me an email at steve@worley.com.

   An implementation of the key cellular texturing basis
   function. This function is hardwired to return an average F_1 value
   of 1.0. It returns the <n> closest feature point distances
   F_1, F_2, .. F_n the vector delta to those points, and a 32-bit
   seed for each of the feature points. This function is not
   difficult to extend to compute alternative information such as
   higher-order F values, to use the Manhattan distance metric, or
   other fun perversions.

   <at>    The input sample location.
   <max_order> Smaller values compute faster. < 5, read the book to extend it.
   <F>     The output values of F_1, F_2, .. F[n] in F[0], F[1], F[n-1]
   <delta> The output vector difference between the sample point and the n-th
           closest feature point. Thus, the feature point's location is the
           hit point minus this value. The DERIVATIVE of F is the unit
           normalized version of this vector.
   <ID>    The output 32-bit ID number that labels the feature point. This
           is useful for domain partitions, especially for coloring flagstone
           patterns.

   This implementation is tuned for speed in a way that any order > 5
   will likely have discontinuous artifacts in its computation of F5+.
   This can be fixed by increasing the internal points-per-cube
   density in the source code, at the expense of slower
   computation. The book lists the details of this tuning. */

#include <math.h>
#include <stdio.h>
#include <assert.h>
#include "cell.h" /* Function prototype */

/* A hardwired lookup table to quickly determine how many feature
   points should be in each spatial cube. We use a table so we don't
   need to make multiple slower tests. A random number indexed into
   this array will give an approximate Poisson distribution of mean
   density 2.5. Read the book for the long-winded explanation. */

static int Poisson_count[256]=

{4,3,1,1,1,2,4,2,2,2,5,1,0,2,1,2,2,0,4,3,2,1,2,1,3,2,2,4,2,2,5,1,2,3,
 2,2,2,2,2,3,2,4,2,5,3,2,2,2,5,3,3,5,2,1,3,3,4,4,2,3,0,4,2,2,2,1,3,2,
 2,2,3,3,3,1,2,0,2,1,1,2,2,2,2,5,3,2,3,2,3,2,2,1,0,2,1,1,2,1,2,2,1,3,
 4,2,2,2,5,4,2,4,2,2,5,4,3,2,2,5,4,3,3,3,5,2,2,2,2,2,3,1,1,4,2,1,3,3,
 4,3,2,4,3,3,3,4,5,1,4,2,4,3,1,2,3,5,3,2,1,3,1,3,3,3,2,3,1,5,5,4,2,2,
 4,1,3,4,1,5,3,3,5,3,4,3,2,2,1,1,1,1,1,2,4,5,4,5,4,2,1,5,1,1,2,3,3,3,
 2,5,2,3,3,2,0,2,1,1,4,2,1,3,2,1,2,2,3,2,5,5,3,4,5,5,2,4,4,5,3,2,2,2,
 1,4,2,3,3,4,2,5,4,2,4,2,2,2,4,5,3,2};
```

```
/* This constant is manipulated to make sure that the mean value of F[0]
   is 1.0. This makes an easy natural "scale" size of the cellular features. */
#define DENSITY_ADJUSTMENT 0.398150

/* the function to merge-sort a "cube" of samples into the current best-found
   list of values. */
static void AddSamples(long xi, long yi, long zi, long max_order,
                       double at[3], double *F,
                       double (*delta)[3], unsigned long *ID);

/* The main function! */
void Worley(double at[3], long max_order,
            double *F, double (*delta)[3], unsigned long *ID)
{
  double x2,y2,z2, mx2, my2, mz2;
  double new_at[3];
  long int_at[3], i;

  /* Initialize the F values to "huge" so they will be replaced by the
  first real sample tests. Note we'll be storing and comparing the
  SQUARED distance from the feature points to avoid lots of slow
  sqrt() calls. We'll use sqrt() only on the final answer. */
  for (i=0; i<max_order; i++) F[i]=999999.9;

  /* Make our own local copy, multiplying to make mean(F[0])==1.0 */
  new_at[0]=DENSITY_ADJUSTMENT*at[0];
  new_at[1]=DENSITY_ADJUSTMENT*at[1];
  new_at[2]=DENSITY_ADJUSTMENT*at[2];

  /* Find the integer cube holding the hit point */
  int_at[0]=(long)floor(new_at[0]); /* A macro could make this slightly faster */
  int_at[1]=(long)floor(new_at[1]);
  int_at[2]=(long)floor(new_at[2]);

  /* A simple way to compute the closest neighbors would be to test all
     boundary cubes exhaustively. This is simple with code like:
     {
       long ii, jj, kk;
       for (ii=-1; ii<=1; ii++) for (jj=-1; jj<=1; jj++) for (kk=-1; kk<=1; kk++)
       AddSamples(int_at[0]+ii,int_at[1]+jj,int_at[2]+kk,
       max_order, new_at, F, delta, ID);
     }
     But this wastes a lot of time working on cubes that are known to be
     too far away to matter! So we can use a more complex testing method
     that avoids this needless testing of distant cubes. This doubles the
     speed of the algorithm. */

  /* Test the central cube for closest point(s). */
  AddSamples(int_at[0], int_at[1], int_at[2], max_order, new_at, F, delta, ID);
```

```
/* We test if neighbor cubes are even POSSIBLE contributors by examining the
   combinations of the sum of the squared distances from the cube's lower
   or upper corners.*/
x2=new_at[0]-int_at[0];
y2=new_at[1]-int_at[1];
z2=new_at[2]-int_at[2];
mx2=(1.0-x2)*(1.0-x2);
my2=(1.0-y2)*(1.0-y2);
mz2=(1.0-z2)*(1.0-z2);
x2*=x2;
y2*=y2;
z2*=z2;

/* Test 6 facing neighbors of center cube. These are closest and most
   likely to have a close feature point. */
if (x2<F[max_order-1])  AddSamples(int_at[0]-1, int_at[1] , int_at[2] ,
                                   max_order, new_at, F, delta, ID);
if (y2<F[max_order-1])  AddSamples(int_at[0] , int_at[1]-1, int_at[2] ,
                                   max_order, new_at, F, delta, ID);
if (z2<F[max_order-1])  AddSamples(int_at[0] , int_at[1] , int_at[2]-1,
                                   max_order, new_at, F, delta, ID);

if (mx2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1] , int_at[2] ,
                                   max_order, new_at, F, delta, ID);
if (my2<F[max_order-1]) AddSamples(int_at[0] , int_at[1]+1, int_at[2] ,
                                   max_order, new_at, F, delta, ID);
if (mz2<F[max_order-1]) AddSamples(int_at[0] , int_at[1] , int_at[2]+1,
                                   max_order, new_at, F, delta, ID);

/* Test 12 "edge cube" neighbors if necessary. They're next closest. */
if ( x2+ y2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1]-1, int_at[2] ,
                                       max_order, new_at, F, delta, ID);
if ( x2+ z2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1] , int_at[2]-1,
                                       max_order, new_at, F, delta, ID);
if ( y2+ z2<F[max_order-1]) AddSamples(int_at[0] , int_at[1]-1, int_at[2]-1,
                                       max_order, new_at, F, delta, ID);
if (mx2+my2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1]+1, int_at[2] ,
                                       max_order, new_at, F, delta, ID);
if (mx2+mz2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1] , int_at[2]+1,
                                       max_order, new_at, F, delta, ID);
if (my2+mz2<F[max_order-1]) AddSamples(int_at[0] , int_at[1]+1, int_at[2]+1,
                                       max_order, new_at, F, delta, ID);
if ( x2+my2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1]+1, int_at[2] ,
                                       max_order, new_at, F, delta, ID);
if ( x2+mz2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1] , int_at[2]+1,
                                       max_order, new_at, F, delta, ID);
if ( y2+mz2<F[max_order-1]) AddSamples(int_at[0] , int_at[1]-1, int_at[2]+1,
                                       max_order, new_at, F, delta, ID);
if (mx2+ y2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1]-1, int_at[2] ,
                                       max_order, new_at, F, delta, ID);
```

```
  if (mx2+ z2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1] , int_at[2]-1,
                                         max_order, new_at, F, delta, ID);
  if (my2+ z2<F[max_order-1]) AddSamples(int_at[0] , int_at[1]+1, int_at[2]-1,
                                         max_order, new_at, F, delta, ID);

  /* Final 8 "corner" cubes */
  if ( x2+ y2+ z2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1]-1, int_at[2]-1,
                                         max_order, new_at, F, delta, ID);
  if ( x2+ y2+mz2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1]-1, int_at[2]+1,
                                         max_order, new_at, F, delta, ID);
  if ( x2+my2+ z2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1]+1, int_at[2]-1,
                                         max_order, new_at, F, delta, ID);
  if ( x2+my2+mz2<F[max_order-1]) AddSamples(int_at[0]-1, int_at[1]+1, int_at[2]+1,
                                         max_order, new_at, F, delta, ID);
  if (mx2+ y2+ z2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1]-1, int_at[2]-1,
                                         max_order, new_at, F, delta, ID);
  if (mx2+ y2+mz2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1]-1, int_at[2]+1,
                                         max_order, new_at, F, delta, ID);
  if (mx2+my2+ z2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1]+1, int_at[2]-1,
                                         max_order, new_at, F, delta, ID);
  if (mx2+my2+mz2<F[max_order-1]) AddSamples(int_at[0]+1, int_at[1]+1, int_at[2]+1,
                                         max_order, new_at, F, delta, ID);

  /* We're done! Convert everything to right size scale */
  for (i=0; i<max_order; i++)
    {
      F[i]=sqrt(F[i])*(1.0/DENSITY_ADJUSTMENT);
      delta[i][0]*=(1.0/DENSITY_ADJUSTMENT);
      delta[i][1]*=(1.0/DENSITY_ADJUSTMENT);
      delta[i][2]*=(1.0/DENSITY_ADJUSTMENT);
    }

  return;
}

  static void AddSamples(long xi, long yi, long zi, long max_order,
                         double at[3], double *F,
                         double (*delta)[3], unsigned long *ID)
  {
  double dx, dy, dz, fx, fy, fz, d2;
  long count, i, j, index;
  unsigned long seed, this_id;

  /* Each cube has a random number seed based on the cube's ID number.
     The seed might be better if it were a nonlinear hash like Perlin uses
     for noise, but we do very well with this faster simple one.
     Our LCG uses Knuth-approved constants for maximal periods. */
  seed=702395077*xi + 915488749*yi + 2120969693*zi;

  /* How many feature points are in this cube? */
```

```
count=Poisson_count[seed>>24]; /* 256 element lookup table. Use MSB */

seed=1402024253*seed+586950981; /* churn the seed with good Knuth LCG */

for (j=0; j<count; j++) /* test and insert each point into our solution */
  {
   this_id=seed;
   seed=1402024253*seed+586950981; /* churn */

    /* compute the 0 .. 1 feature point location's XYZ */
    fx=(seed+0.5)*(1.0/4294967296.0);
    seed=1402024253*seed+586950981; /* churn */
    fy=(seed+0.5)*(1.0/4294967296.0);
    seed=1402024253*seed+586950981; /* churn */
    fz=(seed+0.5)*(1.0/4294967296.0);
    seed=1402024253*seed+586950981; /* churn */

    /* delta from feature point to sample location */
    dx=xi+fx-at[0];
    dy=yi+fy-at[1];
    dz=zi+fz-at[2];

    /* Distance computation! Lots of interesting variations are
       possible here!
       Biased "stretched" A*dx*dx+B*dy*dy+C*dz*dz
       Manhattan distance fabs(dx)+fabs(dy)+fabs(dz)
       Radial Manhattan:  A*fabs(dR)+B*fabs(dTheta)+C*dz
       Superquadratic:    pow(fabs(dx), A) + pow(fabs(dy), B) + pow(fabs(dz),C)

       Go ahead and make your own! Remember that you must insure that a
       new distance function causes large deltas in 3D space to map into
       large deltas in your distance function, so our 3D search can find
       them! [Alternatively, change the search algorithm for your special
       cases.]
    */

    d2=dx*dx+dy*dy+dz*dz; /* Euclidean distance, squared */

    if (d2<F[max_order-1]) /* Is this point close enough to rememember? */
      {
    /* Insert the information into the output arrays if it's close enough.
       We use an insertion sort. No need for a binary search to find
       the appropriate index .. usually we're dealing with order 2,3,4 so
       we can just go through the list. If you were computing order 50
       (wow!!), you could get a speedup with a binary search in the sorted
       F[] list. */

    index=max_order;
    while (index>0 && d2<F[index-1]) index--;
```

```
      /* We insert this new point into slot # <index> */

      /* Bump down more distant information to make room for this new point. */
      for (i=max_order-1; i-->index;)
         {
           F[i+1]=F[i];
           ID[i+1]=ID[i];
           delta[i+1][0]=delta[i][0];
           delta[i+1][1]=delta[i][1];
           delta[i+1][2]=delta[i][2];
         }
      /* Insert the new point's information into the list. */
      F[index]=d2;
      ID[index]=this_id;
      delta[index][0]=dx;
      delta[index][1]=dy;
      delta[index][2]=dz;
    }
  }

 return;
}
```