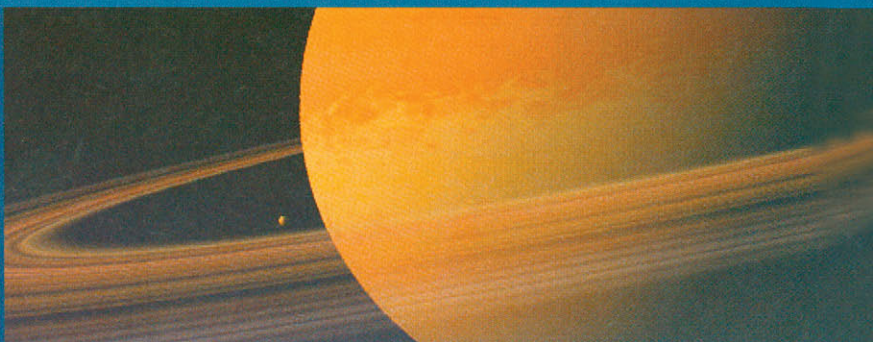


6



PRACTICAL METHODS FOR TEXTURE DESIGN

STEVEN WORLEY

INTRODUCTION

This entire book is primarily about texturing: different ways to use a mathematical formula to decide how to color the surface of an object. This chapter discusses aspects of texturing from a practical point of view, especially in dealing with the texture controls (parameters) users manipulate. This chapter shares many topics with Chapter 2, but from a different point of view.

I have written over 150 algorithmic surface textures (mostly for commercial use by other animators), and after a while you definitely get a feel for the design of the algorithms, as well as a toolbox full of utilities and ideas that make writing the textures easier. Several recurring themes and tricks occur over and over (such as mapping a computed value into a color lookup table or adding a bump-mapping effect to a color texture), and these topics form the basis of this chapter.

TOOLBOX FUNCTIONS

After building such a large library of textures, it has become clear that many new textures are just variants of each other—new ways to organize a set of stock routines together. Building blocks such as fractal noise functions, color mapping methods, and bump-mapping definitions occur in nearly every texture! This section discusses these common elements since their ubiquitous use makes them so important.

The Art of Noise

Fractal noise is, without question, the most important element currently used in procedural texturing. We won't discuss the basic implementation of the fractal noise function here, since many of the other chapters of this book discuss it in some detail (this alone is evidence of its importance in procedural texturing). Instead, we'll

discuss some enhancements and modifications to the basic noise algorithm, mostly to produce higher-quality and easier-to-use noise.

The biggest problem with the “plain” fractal noise algorithm is artifacts. The basic routine interpolates over a cubic lattice, and you’ll be able to see that lattice on your surface, especially if you are using a small number of summed scales. Purists will also note that the basic Perlin noise isn’t very isotropic, since diagonal directions have a longer distance gap between sample points than the points along the coordinate directions.

One method to hide this artifacting is to rotate each summed scale to align to a (precomputed) random orientation. Since the lattices of the different scales won’t line up, the artifacts will at least be uncorrelated and a lot less noticeable. If you want to return derivatives for each scale (for bump mapping), you’ll have to multiply the vector derivative result from each scale with the appropriate inverse rotation matrix (which is the transpose of the original rotation matrix) before you sum them.

I made these transformations by generating random rotation matrices and taking only those that point inside a single octant of a sphere. Since a 90-degree rotation (or 180 or 270) will still cause the lattices to match up, a single octant is the most rotation that is necessary.¹ Keeping the rotations confined within one octant also helps you check that there are not two similar rotations for different scales. (Remember this is a one-time precompute: you can manually sort through about 10 of these to make sure the matrices are all reasonably different.) To test what octant a rotation matrix maps to, just pump the vector (1 0 0) through the matrix and look for a vector with all positive components. You can generate the rotation matrices using the algorithm(s) in *Graphics Gems III* by Ken Shoemake and Jim Arvo. Figure 6.1 shows two examples of matrices that might be used.

This rotation trick is most useful when bump mapping, since the differentiation of the noise value clearly exposes the lattice artifacts: there are sharp discontinuities of the second derivatives along the lattice lines. When you differentiate for bump mapping, these second derivatives become discontinuous *first* derivatives and are easily visible.

Another strategy to help hide noise artifacts is to choose the lacunarity (the ratio between the sizes of successive scales) intelligently. A natural lacunarity to use is 0.5, *but don’t use this!* It’s better to use a value with a lot of digits (like 0.485743 or 0.527473), which will give you the same effective ratio. A ratio of exactly 0.5 will make the different scales “register” closely (the next smaller scale repeats exactly twice on top of the larger scale), so artifacts can appear periodically. This periodicity is broken by using a number that’s not a simple ratio.

1. Note that these are full 3D rotations, not 2D, but you know what I mean.

Rotation matrix	Inverse
$\begin{pmatrix} 0.98860 & -0.07651 & -0.12967 \\ 0.07958 & 0.99665 & 0.01871 \\ 0.12780 & -0.02881 & 0.99138 \end{pmatrix}$	$\begin{pmatrix} 0.98860 & 0.07958 & 0.12780 \\ -0.07651 & 0.99665 & -0.02881 \\ -0.12967 & 0.01871 & 0.99138 \end{pmatrix}$
$\begin{pmatrix} 0.85450 & 0.46227 & -0.23691 \\ 0.48691 & -0.55396 & 0.67530 \\ 0.18093 & -0.69240 & -0.69845 \end{pmatrix}$	$\begin{pmatrix} 0.85450 & 0.48691 & 0.18093 \\ 0.46227 & -0.55396 & -0.69240 \\ -0.23691 & 0.67530 & -0.69845 \end{pmatrix}$

FIGURE 6.1 Two random rotation matrices (and inverses).

Don't overlook enhancements or variants on the fractal noise algorithm. Extending the noise interpolation to four-dimensional space is particularly useful, as this allows you to make time-animated fractal noise. Another interesting variation I've used is to replace the random gradient and values of each noise "cell" with a lookup table of a function such as a sine wave. This lets you use your fractal noise machinery (and all the textures that have been designed to use it) with different patterns. Remember Perlin's original noise algorithm is not sacred; its usefulness should actually encourage you to experiment with its definition.

Color Mappings

Looking in our toolbox of common routines, the continual use of mappings makes this topic very important to discuss. Most textures use a paradigm that computes a value such as fractal noise and then uses this value to decide what color to apply to your object. In simpler textures this added color is always of a single shade, and the noise value is used to determine some "strength" from 0 to 1, which is used to determine how much to cross-fade the original surface color with the applied texture color.

This mapping allows the user quite a bit of control over the applied color, and its simplicity makes it both easy to implement and easy for the user to control even with raw numeric values as inputs. One mapping method I have used with great success defines four "transition" values. These transition values control the position and shape of a certain mapping function that turns the fractal noise (or other function) value into an output value from 0 to 1. Figure 6.2 shows the shape of this function. Two transitions, labeled T1 and T2, are each defined by a beginning value and ending value. By setting the different levels where these transitions occur, a large variety of mappings can be made from gradients to step functions to more complex "bandpass" shapes.

The implementation of such a mapping is trivial. The mapping is worth discussing mostly because this method is so useful in practice, since it is easy for users to

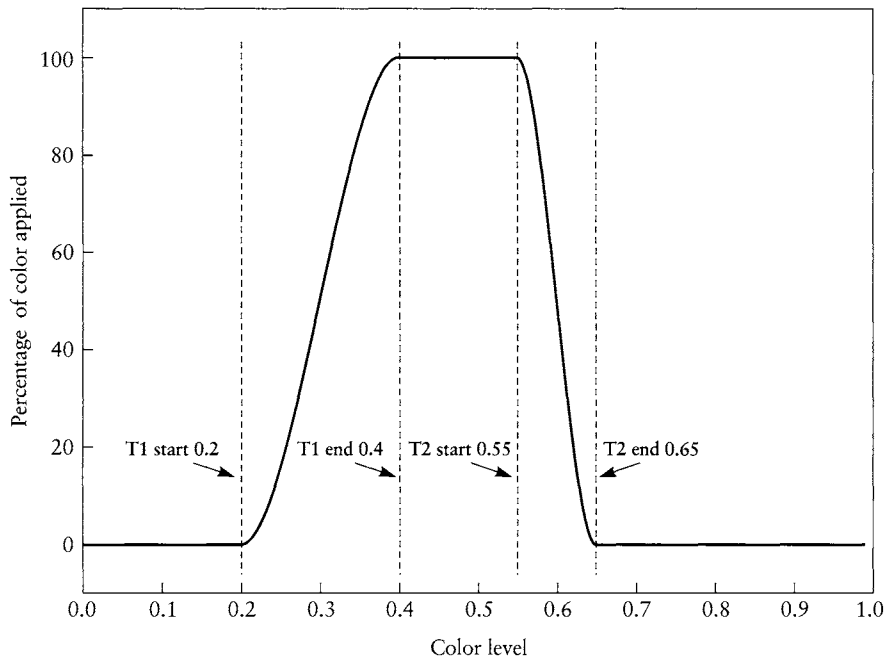


FIGURE 6.2 A mapping defined by four user values.

manipulate even numerically. A more sophisticated mapping method is much more general: color splines.

A color spline is simply an arbitrary mapping of an input value to an output color.² Often this mapping consists of a piecewise continuous interpolation between color values using a spline. Any number of spline knots (and node colors) can be defined, so the method is truly arbitrary. It does become very difficult for a user to control or manipulate an arbitrary spline mapping method without a good graphical interface; columns of color values and knot positions aren't easily visualizable.

2. You might recognize that the word “color” throughout this chapter (and book) is always used loosely. Most object surface attributes can be controlled through textures, and while the object’s diffuse reflection color is the most often modified value, any of the other surface attributes are controllable using the exact same methods. It’s just a lot easier to say “color” than “surface attributes specified by some vector that you can modify with your texture interface.”

Bump-Mapping Methods

The most important element of *impressive* textures is the use of coordinated bump mapping. I'll repeat that: impressive and useful textures use bump mapping. Gouged-out depressions, ridges, or bumps are always more realistic and interesting than a simple layer of color paint. Luckily, nearly any color texture can have bump mapping added to it. Feedback from users shows that bump-mapping ability is unquestionably the most useful “extra” feature that a texture can have.

It's usually not difficult to add bump mapping to a basic texture. If you have a function that maps a scalar field to a value (like a color-mapped fractal noise value or a perturbed sine for marble veins), you can take the derivative of the scalar function in all three directions (x , y , z), add these values to the surface normal, then renormalize it.³ You should also add a parameter for users to control the apparent height of the bump. This is just multiplied to the derivative before the addition so the user can turn off the bump mapping by using a zero value or make the bump “go the other way” by using a negative value.

In the case of geometric figures (say, a hexagon mesh, or even a plain, checkerboard), there's not really a function to take the derivative of. In this case, I like making a “ridge,” which is basically a line that follows along the exterior boundary of the figure at a fixed (user-defined) distance inside the boundary. This ridge basically makes a bevel around the outside rim of the figure, allowing the user to make the figure look raised or depressed into the surface. Figure 6.3 shows an example of how a simple bevel makes a flat pattern appear three-dimensional.

But a simple triangular bevel is awfully plain. It's good to give the user more control over the shape of this outer rim to make a variety of shapes, from a boxy groove to a circular caulk bead. In the case of a checkerboard, a user might make a concave rounded depression, which would make the squares look like they were surrounded by mortared joints.

I've tried several ideas to give the user control over the shapes of these ridges, but one has turned out to be the most useful. You don't want to have too many parameters for the user to juggle, but you still want to give them enough control to make a variety of joints. I've found that a combination of just four parameters works very well in defining a variety of bevel shapes.

The first parameter is the ridge width. What is the total width of the seam or bevel? In many (most!) cases this bevel is going to be butted up against another seam,

3. This is technically wrong! Really you need to do a little more work to get “correct” results, as discussed on page 170.

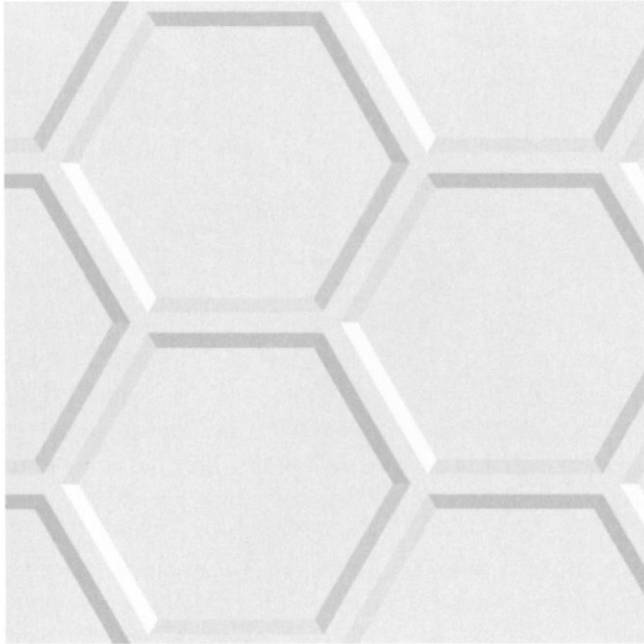


FIGURE 6.3 A ridged hexagon mesh.

so you might want to halve the number internally. (An example of this is a checkerboard: every tile is next to its neighbor. It's easier for the user to think about the total width of the joint instead of the width of the part of the joint in just one square.)

The second parameter is what I call the “plateau width.” This is a distance along the outside of the ridge that isn't affected. This allows users to make joints that have a flat part in the middle of a seam, sort of a valley between the cliff walls that are formed by the bevel. This plateau obviously must be less than the total width of the bevel.

The last two parameters control the *shape* of the bevel. If you think of the ridge as being a fancy step function (it starts low, ends high, and does something in between), you want to be able to define that transition by a straight line, a smooth S curve, or anything in between. I've found that using a smooth cubic curve over the transition allows the users to define most useful shapes by setting just two numbers, the slopes of the curve at the start and end of the transition. Slopes of 0.0 would make a smooth blending with the rest of the surface, and if both slopes were 1.0, the

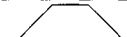


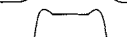

Bottom slope	Top slope	Plateau?	Ridge shape
1.0	1.0	Yes	
0.0	0.0	Yes	
0.0	1.0	Yes	
-1.0	-1.0	Yes	
0.0	0.0	No	

FIGURE 6.4 Bump-mapped ridge shapes for different slope controls.

transition would be a classic straight-line bevel. This is best shown in Figure 6.4, which also shows how the plateau is used.

The ridges shown are just examples, since there's really a continuum of curves that can be made. The slopes can even be continuously changed over time to animate the bevel morphing its shape.

These slope controls are obviously useful for making different ridge profiles, but how do you actually convert the parameters into numbers that can be used for bump mapping? The best way is to use the simple Hermite blending curves. Hermite curves are a simple type of spline, defined by a cubic polynomial over a range from 0 to 1. A cubic polynomial is very convenient since it's cheap to compute and has four degrees of freedom to control its shape. We can completely define this cubic curve by setting the starting and ending values and slopes.

This is perfect for us, since our ridge is basically a smooth curve that starts at a low height (0) and ends up high (1). We have the user specify the starting and ending slopes, defining the curve's last two degrees of freedom. The slope of this curve controls the amount of bump added to the surface normal.⁴ We'll parameterize the width of the bevel from 0 to 1 to make using the Hermite spline easy.

We have four values (start and end heights, start and end slopes) we can use to construct the cubic polynomial by adding together weighted copies of the four

4. "Adding" is a tricky word! A crude literal addition of the derivative to the normal won't give you correct results (although they'll look OK sometimes). Again, see the bump-mapping coordinate discussion on page 170 to understand how bump mapping should be done correctly.

Hermite “blending functions.” The sum, $F(t)$, is the unique cubic polynomial that satisfies the four constraints.

These blending functions are called P_1 , P_4 , R_1 , and R_4 , and represent the values of $F(0)$, $F(1)$, $F'(0)$, and $F'(1)$, respectively (see Figure 6.5).

$$P_1 = 2t^3 - 3t^2 + 1$$

$$P_4 = -2t^3 + 3t^2$$

$$R_1 = t^3 - 2t^2 + t$$

$$R_4 = t^3 - t^2$$

$$P'_1 = 6t^2 - 6t$$

$$P'_4 = -6t^2 + 6t$$

$$R'_1 = 3t^2 - 4t + 1$$

$$R'_4 = 3t^2 - 2t$$

In the case of the ridge, the start and end values will always be fixed to 0 and 1, respectively. The user supplies the two slopes (call them s_b and s_t for bottom and top). The curve that represents our ridge shape is therefore $P_1 + s_b R_1 + s_t R_4$. The derivative of the curve at the hit point is $P'_1(t) + s_b R'_1(t) + s_t R'_4(t)$. This tells us the slope of our ridge at the hit point: the weight we use when adding to the surface normal.

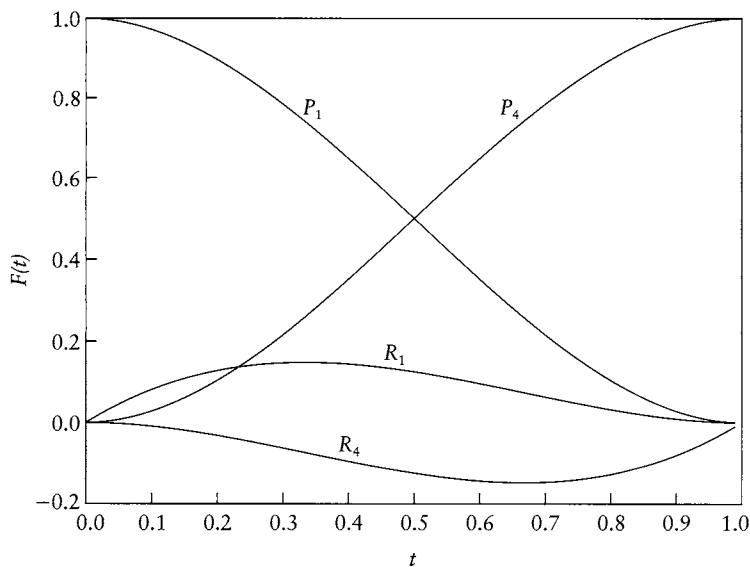


FIGURE 6.5 Hermite blending functions.

So, explicitly, the “ridge” algorithm is as follows:

1. Decide if the hit point lies within the bevel width. You’ll need to come up with a measure of how close a hit point is to the “edge” of the figure.
2. If the hit is outside the bevel width or inside the plateau width, exit. The surface is flat at this location.
3. Set N_{add} equal to the normal vector perpendicular to the bevel. (This is the vector that points from the hit point toward the closest edge.) For example, on a checkerboard, if the hit point is on a bevel on the right (+X) side of a square, the normal would be (1, 0, 0), and the bevel on the bottom (−Y) of the square would have a normal of (0, −1, 0).
4. Set t to the normalized (dimensionless) distance along the slope. (If the hit point was d units from the edge, $t = 1.0 - (d - \text{plateau width}) / (\text{ridge width} - \text{plateau width})$.)
5. Compute s , the slope of the cubic curve at the hit point. This is $P'_1(t) + s_b R'_1(t) + s_t R'_t(t)$, where s_b and s_t are user parameters.
6. Weight N_{add} by sA and add it to the hit point’s surface normal. A is yet another user parameter, controlling the amplitude of the bump effect.
7. Renormalize the surface normal.

This is actually pretty easy to implement. The hardest part is probably determining the distance to the edge of a figure like a hexagon.

Note that this kind of bump mapping doesn’t have to be restricted to geometric figures! It’s easy to add to 3D scalar field functions, too. This lets you use something like fractal noise to make weird squiggly canyons or raise marble veins out of the surface of a stone. In this case the t variable is usually set by a high and low value defined by the user. The derivative of the field provides the N_{add} vector.

THE USER INTERFACE

Writing an algorithm that makes an interesting surface is only half the battle! One of the overlooked aspects of texture design is defining the user-accessible parameters. Even when an environment has a fancy GUI for manipulating the texture, the user is usually faced with setting a whole bunch of parameters with vague labels. The usability of your textures can be dramatically increased by using some thought and planning during your implementation. This is especially true when designing the

interface that you or users will use to control the texture. This interface often is not a fancy previewer, but simply a set of parameter values or even a crude text file. (In 1992, text file parameter definition was common. It's scary, but in 2002, it's still common!) Even with such basic interfaces, careful parameter design can make the texture significantly easier to use.

Parameter Ranges

It might seem obvious, but the user should be provided with suggestions for acceptable ranges for each parameter. Some ranges are easy enough to define, like colors or a percentage level. But, since all interesting textures seem to use ad hoc code that even the programmer doesn't quite understand, there are always going to be mystery constants that the user has to play with to get different effects. It's awfully inelegant to have a parameter named "squigginess" that only makes useful patterns when it is between 55 and 107.⁵

One way to help the user control vague parameters like this is to *remap* a parameter's range. If "squigginess" is just a mystery constant (which happens to be interesting over that 55–107 range), it's silly to expect users to enter these weird values. If you find that a parameter like this has one particular range of usability, you can use a linear transformation to remap the range: the user would enter a number from 0 to 1 for "squigginess," not a value in the nonintuitive internal range. The new range of values provide the exact same amount of control as before, but at least the user knows that values like 0.3 or 0.9 might be interesting to try. (They can also experiment and try – 0.2 or 1.1, of course, but they'll understand if the result isn't useful.) The remapped range then serves as an indirect guide to selecting useful parameter settings.

Remapping linear ranges like this is easy, especially when you're remapping from 0 to 1. Since the transformation is linear, it's just a multiply and an addition. If x is between 0 and 1, you can map to the values from L and H by the simple formula $L + x(H - L)$.

This does *not* mean that you should make all parameters map to a 0–1 range! RGB colors are often easiest to enter as three 0–255 values. The width of a check in a checkerboard texture should be measured in the coordinate system's units. Common sense should be your guide.

5. This may seem like a contrived example, but veteran texture authors know it's not. Mystery constants seem to be an integral, recurring aspect of texture design.

Color Table Equalization

Color splines were discussed on page 182. Fractal noise textures in particular work well with this mapping method. However, while it is very easy to say, “Just feed the fractal noise value into a lookup table,” how is this table designed? Obviously, it depends on whether you’re making a texture that will be manipulated with a fancy GUI or one that is a subroutine the user has specified with raw parameter values. In practice, it usually boils down to the latter, since the GUI is just a front end for setting numeric parameters.

The biggest problem users find with setting color splines or levels is the fact that it is hard to know how much effect their color choices will have. This is critically true for renderers that can’t quickly preview textures interactively. The biggest obstacle to users in setting color levels is that they do not know how much impact their choices will have on the rendered surface. Say that the users can define a color scheme where fractal noise below the value of -0.5 is black, a value above 0.5 is white, and in between values are a smooth gray transition. The problem is that the users have no idea how much of their object is going to be colored! Is the surface going to be dominated by huge patches of black and white, with narrow gray transitions in between? Or will it be gray values everywhere that never quite get to solid black or white?

Of course, the user can experiment and try to find the right levels to start and end color applications, but it is frustrating to have such an arbitrary, nonlinear scale! It is even worse, since if you change the number of noise octaves added onto the fractal sum, the scale changes again! If you think about it, it would be much easier if the users could deal with constant *percentage* levels and not weird, unknown ranges. Thus, a user could set the “lower” 20% of the noise to map to black, the “upper” 20% to white, and the remaining 60% to a gray gradient. This way the user has some idea how large each color range will be on the surface.

Thus there are two steps to making the noise levels “behave.” First, they need to be normalized so that the amplitude ratio between scales and the total number of scales don’t make the noise ranges change too much. The second part consists of making a table to map these values to percentages. The second step is really just a histogram equalization or a cumulative probability function.

The normalization step is pretty easy. Ideally, you want the fractal noise, no matter how many scales or what scale size or ratio you use, to return a value over the same range. You can do this adequately well by dividing the final sum by the square root of the sum of the squares of all the amplitudes of the scales. For those who don’t want to parse that last sentence,

$$F(x) = \frac{\sum_{i=0}^{n-1} a^i N(\frac{x}{Ls^i})}{\sqrt{\sum_{i=0}^{n-1} (a^i)^2}}$$

where

F = normalized fractal noise value

L = largest scale size

s = size ratio between successive scales (usually around 0.5)

a = amplitude ratio between successive scales (usually around 0.5)

n = number of scales added (usually between 3 and 10)

x = hit location (a vector)

This normalization makes it a lot nicer when you are tweaking textures: the relative amounts of colors or size of the bumps won't be affected when you change the scale or amplitude ratio or number of summed scales.

The equalization step is a little more involved. The values of fractal noise will fall into a range, probably within -2 to 2 , with most values around 0 . It looks a lot like a normal distribution except the tails die out quickly. The histogram shown in Figure 6.6 shows the distribution of 10,000 sample values of fractal noise. In practice, more samples should be used, but with this number the slight random errors are visible to remind you this is an empirical measurement, not an analytic one.⁶

The histogram is very interesting, but we're really looking for a way to map the user's percentage level to a noise value. If, for example, 10% of all noise values fell below -0.9 , we'd know to map 10% to that -0.9 value. Zero percent would map to the very lowest noise value, and 100% would map to the very highest. This is a *cumulative* function and is used in probability and statistics quite often. Luckily, the way to explicitly measure this relation empirically is straightforward.

First, store many random values of fractal noise in an array. Computing 250,000 samples is probably sufficient, but more is even better. (This is a one-time computation, so speed doesn't matter.) Sort the values into an ascending array. If you plotted the sorted array, you'd get a curve similar to the normalization plot shown in Figure 6.7. This is now a lookup table! For example, if you used 100,000 samples, the 50,000th number in the sorted array would correspond to the 50% level: half of the

6. You may recognize that in many cases the distribution of fractal functions tends to form a normal distribution because of the statistical Law of Large Numbers. So for some functions, you can just use a Gaussian as a distribution model and use the `Erf()` function to compute the normalization values.

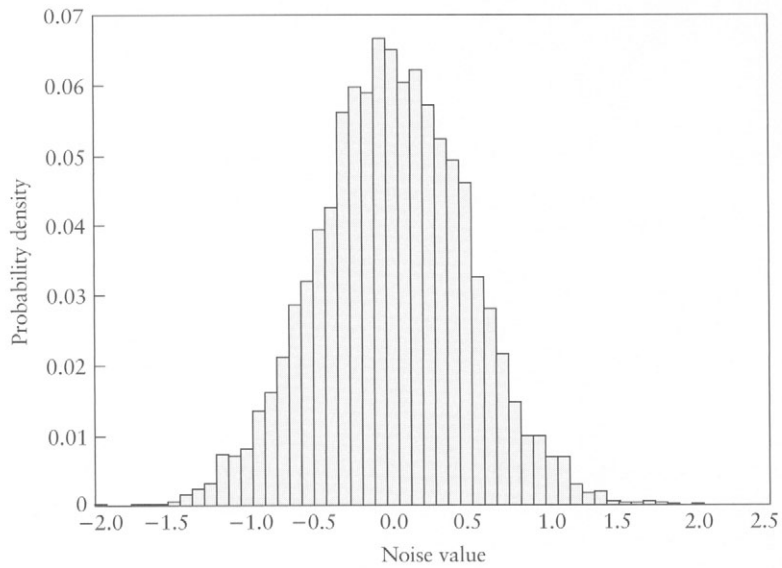


FIGURE 6.6 Histogram of 10,000 noise values.

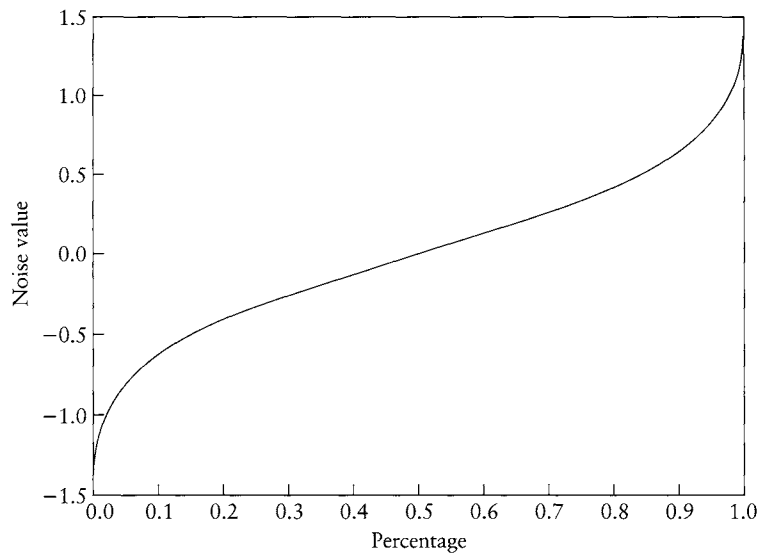


FIGURE 6.7 Percentage to noise value normalization curve.

noise values would be less than the value stored in this location in the sorted array. (It's the median!) The 10,000th number corresponds to the 10% level, since 10% of the 100,000 values are less than this value.

You obviously don't need to store 100,000 lookup values in your texture code! You can store perhaps just 11 equally spaced samples and linearly interpolate between them: a 15% level would be computed by averaging the 10% and 20% levels. Probably 16,000 values is a fine enough "grain" that you don't even need to bother interpolating.

Adding the translation from user parameter values (probably expressed in a 0 to 1 fractional value) to a noise value is easy enough, probably done as a first step and stored in a static variable. If you have a fancy GUI it can even be done as a pre-process as soon as the user has set the levels.

It is impossible to overstress the usefulness of this equalization to users. They can actually plan out the coverage of different colors fairly accurately the first time. If they want a sky 20% covered with clouds, they immediately know the settings to use. This convenience is no real computational load, but for the user it makes using the texture a *lot* more intuitive.

Exploring the Parameter Domain

Even as the author of a texture algorithm, determining appropriate parameter values to use can be very difficult. A mere texture *user* should have some strategies for manipulating a set of parameters.

The easiest method of setting a texture's parameters is to simply use previously interesting values! It may seem silly, but if you don't keep track of settings you have already produced, you'll always be starting from scratch whenever you wish to produce a new surface appearance. Keeping track of previous successful settings will give you a starting point for surfaces with the absolute minimum amount of future effort.

The library of settings is obviously not practical until you *build* that library first, and even then the library won't always have an example of the surface you desire. Other strategies for modifying the parameters give you significantly more control.

The most obvious but most demanding method is simply to understand the texture well and use your intelligence to choose the proper settings. When you are familiar with what each parameter does, you can predict what the modified surface will look like and just set the values appropriately. A trivial example is when you wish to change the colors applied by the texture, so you simply edit the color

parameters. However, knowledge of the texture's controls still isn't enough; if you want a surface that has a more "blobby" behavior, but there's no convenient parameter named "blobbiness," you'll be forced to determine what parameters (or set of parameters!) will produce the effect you want.

While in theory you can simply use your experience and judgment to set the parameters, in practice this can be difficult. Oddly, often the most effective editing method is just to vary different parameters wildly! Blindly adjust them! Don't be conservative! When the dust settles, look at the surface produced; it may be something you like and can use. If it's not, it only takes a few seconds to try again.

This random method is arguably just as powerful as the "forethought" editing method. Even when I feel I understand the texture well (because I wrote it!) random settings will sometimes make interesting appearances I didn't expect and couldn't even plan for. It's a great starting point, since from here you can use the "intellectual," planned editing steps. The random method will give you a wide variety of surfaces that, even if they are not immediately useful, you can still save in an attribute library; it's likely you might be able to use them sometime in the future.

The final editing strategy is what I call the "polish" step. When you have a setting you are happy with, you can use a methodical last pass to optimize your settings. The idea is that if the surface looks as good to you as possible, then changing any of the parameters must make a less perfect surface. (If it didn't, you'd have an even better surface than you started with, right?) With this in mind, you can go through each parameter sequentially. Perturb the value by a small amount, but enough to make a visible change. If you like the new pattern better, then congratulations, you've just refined your surface and improved it! If not, try adjusting the parameter a small amount the *other* way. If neither adjustment improves your surface, reset the value to what it was initially, then proceed to the next parameter. After you've gone through all the parameters, your surface is optimized; any parameter changes will make it less attractive.

While there are many strategies for editing parameters, the best strategy of all has got to be the long-term accumulation of a library of example parameter settings for each texture. You'll find new surfaces all the time when exploring (especially using the random flailing method), and if your attribute library is large, the next time you need a specific surface, you'll be that much more likely to have a good starting surface; otherwise you'd be forced to start from scratch.⁷

7. If you're designing a texture *system*, make sure to allow some sort of a user preset functionality!

Previews

Especially when you are initially developing textures, it is painful to run test after test to see how the texture is behaving. Obviously, the faster your development platform is, the better, but even then the design cycle time can be painfully slow.

I heartily agree with Ken Perlin's advice on this problem: generate *low-resolution* tests! Most problems are glaring errors, and even 200×200 pixel test images are often enough detail to judge the gross behavior of a texture very quickly. If you're in a compile-render-recode-compile loop, you might be surprised how much time is wasted just waiting for the next render, even with modern fast CPUs.

If you can't speed up your computer, the best way to help cut the design cycle time is to have the renderer display the image as it is rendering (if it can!). This way you can abort the render as soon as you know there is a problem, as opposed to waiting for the whole thing to finish before discovering there's a problem.

Even better is to design a previewer specifically for editing texture parameters. Place each parameter on a slider, and implement a progressive refinement display. This is a rendered image (perhaps just a simple plane or image of a sphere) that is rendered at a very coarse resolution. After the image has been computed, the image should be recomputed at a resolution that is twice as fine, updating the display appropriately. (Note that one-fourth of the pixels have already been computed!) This refinement continues until you've computed the texture down to the individual pixel level.

If this preview can be interrupted and restarted by changing a parameter (ideally by just grabbing and changing a slider), the design cycle for setting parameters becomes enormously faster. Even with a very slow computer, the update is interactive (due to the progressive display scheme). An editor like this increased the utility of my entire set of textures by a (conservatively estimated!) factor of five. *The utility of a texture to a user (especially a nontechnical one) is inversely proportional to the time for a complete design cycle.*

EFFICIENCY

As important and useful as it is, procedural texturing has one flaw that often limits its potential. That flaw is simply efficiency. A complex surface tends to require

complex computations to define it.⁸ In particular, the most used building block of texturing (Perlin’s fractal noise) is, unfortunately, slow.

The first and second editions of this book contained several pages of “speedup tricks” here, dealing with clever C macros to save a few microseconds. This was important in the “good old days” of 25 MHz processors. In today’s modern era, we still love speed, but it’s actually counterproductive to spend enormous effort optimizing C code when CPUs can now render a full screen of fractal noise in real time. This speed allows us to concentrate more on texture design and not programming details.

But efficiency is still important! It just means that we can focus more on the algorithms, especially caching and antialiasing. Because procedural textures are easier and faster to compute, we’re just using them more. In 1989, Perlin’s hypertextures were hyperslow. But today, most professional renderers have volumetric texture and shading effects, which are an order of magnitude more compute intensive.

TRICKS, PERVERSIONS, AND OTHER FUN TEXTURE ABUSES

Basic textures are often implemented in the same rough method. A function is passed an XYZ location in space, the current surface attributes, and perhaps some user parameters, and the texture modifies the attributes based on some internal function. This generic design is sufficient for more general surface texturing, but if you are creative you can actually produce some truly unique and interesting effects even with the limited information most textures usually have.

Volume Rendering with Surface Textures

One of the sneakiest tricks is to change a surface texture into a true volume texture. This has been done since the beginning of algorithmic texturing; Geoffrey Gardner, as early as 1985, was fabulously effective at turning simple ellipsoid surfaces into apparent cloud volumes (Gardner 1985). This wasn’t a true volume integration, but it looked great!

You can actually do a true volume rendering with simple surface textures as long as you limit your surfaces to a somewhat restricted geometry. The easiest method of doing this relies on the texture knowing which direction the incoming ray is arriving from and assuming the user is applying the texture only to one type of surface like a sphere. The trick is simple: since you know the hit point of the ray, the direction of

8. The notable exceptions to this rule are patterns such as the fractal Mandelbrot set, but these have an uncontrollable complexity; you can’t easily use the Mandelbrot set to make an image of granite paving stones.

the ray, and you're assuming you're hitting a sphere (so you know the geometry), you know exactly what path the ray would take if it passed through the sphere.

Now, if you perform a volume rendering with this information (which might be an analytic integration of a fog function, a numeric integration of a 3D volume density, or some other volume-rendering technique⁹), you can just use the final output color as a *surface* color. In particular, this is the surface *emittance*; often renderers will let you define this emittance (which is sometimes called *luminosity*) or, just as useful in this case, allow you to specify the true RGB surface color the surface will be treated as (and no shading will be done). By setting this surface color to the volume density computed intensity, the illusion of a gas can be complete! The gas must be contained by the sphere that defines it, but it is a true 3D effect and as such can be viewed from any angle. The biggest caveat is that the camera cannot fly through the volume density.

Odd Texture Ideas

If you think about the abilities of textures, you might realize that textures might be useful for more than just applying a color onto the surface of an object. In particular, if your host renderer passes the current surface color to the texture, you can design routines that can manipulate that color. You might map a bitmap onto a surface, then use a “gamma correction” texture to brighten the image. This texture would just be called after the brushmap is applied and would correct the image “on demand.”

In fact, I've found quite a few “utility” textures like this that don't really apply a pattern at all. One is a solid color texture that has just two arguments, an RGB surface color and a “fade” value. By applying a solid color to an object, you can *cover up* previous textures. The fade value allows you to make the surface any opacity you want. If you linearly change this opacity over time, you can “fade in” an image map or previous texture. Or you could use the texture at a small (10%) opacity to just tint a surface's color.

Another useful texture transforms the RGB surface color to HSV, then lets the user “rotate” the hue around the color wheel. This is a cheap form of color cycling and is especially useful for animations.

You also shouldn't get stuck thinking that textures have to be fractal noise patterns or endlessly tiled figures. I've found that textures can be useful in adding fairly specific, structured features that you might think would be better implemented with an image map. One example is an LED display like a watch or calculator. The

9. Chapter 8 by David Ebert discusses this topic extensively.

texture takes a decimal number as an argument and displays it on the simulated seven-segment digits! This might seem weird, but you can now make something like an animated countdown without having to make hundreds of image maps manually.¹⁰

In a similar vein, a “radar” texture can make a sweeping line rotate around a circle tirelessly, with radar “blips” that brighten and fade realistically. All sorts of items that need to be continually updated (clocks, blinking control panel lights, simulated computer displays) can often be implemented as a semispecific surface texture.

A fun texture is the Mandelbrot set. Just map the hit point’s XY position to the complex plane. As the camera approaches the surface of the object, the texture automatically zooms the detail of the set! This is an awfully fun way to waste valuable CPU time.

Don’t get stuck in always layering an image onto the surface, either. One of my most often used textures uses fractal noise to *perturb* the brightness of the surface. A user might use a brushmap or another texture to provide surface detail, then use this “weathering” texture to add random variations to the surface. An example might be a texture that takes a fractal noise value F and scales the object surface color by a factor of $1 + \alpha F$. Even for subtle values of α the perfect hue of the surface is given some variation. This is dramatically impressive when a regular grid of squares suddenly becomes the weathered hull of an ocean liner.

2D Mapping Methods

For obvious reasons, 3D procedural textures are more versatile than 2D image maps. But modern renderers have powerful image mapping controls, which can often be great to use even with procedurals. The simple idea is to convert the 3D procedural texture into a 2D image map and allow the renderer to deal with it from there. This can speed up rendering and allow use of a procedural in 3D hardware (for OpenGL display), for video games that use real-time graphics, for solving antialiasing problems (since the renderer’s image map antialiasing will be used), and of course for renderers that simply lack the proper texturing architecture.

Converting a 3D procedural to a 2D image requires some kind of harness to generate the maps. Tools for this are becoming more common as UV mapped modeling has become the norm. This means that the model itself has encoded UV values over each patch or polygon, correlating each surface point with a 2D image location. The image generation harness can iterate over the entire object, evaluating the

10. There is a sample of my LED texture on this book’s Web site. It’s admittedly a decade-old artifact of amateur coding, but may be interesting for reference.

procedural over the surface and “writing” the color values into the 2D image map. This test harness can be crude or sophisticated, depending on how it deals with sampling the surface evenly and how it filters the result to form the 2D texture sample. But once working, this process is invaluable for many artists’ goals. The generic term for this technique is *texture baking*.

The 3D nature of the procedural texture allows many 2D effects that couldn’t be made by hand-painting because of the varying distortion that a human artist can’t compensate for. This is even true for common spherical projection maps that are just a special case of *UV* mapping (which use a formula instead of an arbitrary table to associate *UV* values with the surface.) For example, an unwrapped image of the Earth (showing continents, oceans, etc.) applied to a sphere produces a final spherical planet with the proper appearance. This polar coordinate mapping method allows the renderer to use the surface’s 3D *XYZ* point to find the 2D *UV* coordinates of the corresponding position on the image map. This mapping is an “equal-angular” mapping, not a Mercator mapping, so the transformation is particularly simple:

$$U = \frac{\tan^{-1}(X, Y)}{2\pi}$$

$$V = \frac{\sin^{-1}(Z/\sqrt{X^2 + Y^2 + Z^2})}{2\pi} + 0.5$$

For *UV* maps defined by this spherical transform, we can invert the equations and use it to convert a *UV* map into an *XYZ* position for our procedural. Assuming we are evaluating the texture on the surface of a sphere of unit radius, this reverse transformation is not difficult. Simply,

$$X = \cos((V - 0.5) \cdot 2\pi) \sin(U \cdot 2\pi)$$

$$Y = \sin((V - 0.5) \cdot 2\pi) \sin(U \cdot 2\pi)$$

$$Z = \sin((V - 0.5) \cdot 2\pi)$$

Figures 6.8 and 6.9 show an example of how baking can work in practice. A basketball was defined algorithmically by using a brute-force procedure to evaluate a function over the surface of a sphere. Simple rules were used to determine whether a point was in a stripe or not (this is a quick geometric decision). A large lookup table of points distributed over the surface of the sphere was examined to see if the sample point was within a “pip” on the surface.¹¹ The inverse spherical mapping transformation was used to loop over the pixels of a texture map, and depending on whether the corresponding 3D surface point is located within a line or a pip, the

11. No subtlety here, every element in the lookup table of a couple thousand points was examined. This example was not a procedural texture designed for rendering (it was to produce a one-shot image map), so efficiency was not a big concern.

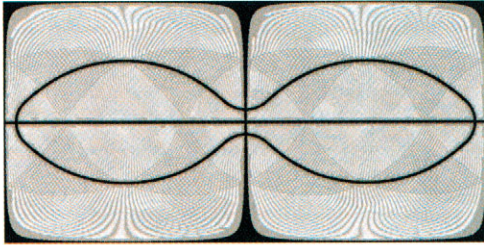


FIGURE 6.8 A 2D basketball image map.



FIGURE 6.9 The seamless wrapped ball.

image map was set to a gray value. This texture map was then used directly in a renderer.

In practice this trick works extremely well, since the predistortion of the image map exactly compensates for the renderer's subsequent distortion when wrapping. No seams are visible, and the image is not stretched; both occur when naive maps (such as digitized imagery) are used. This book's Web site contains the (admittedly old and inelegant) procedural to generate the basketball's appearance.

Note that this exact same method can be used to evaluate a texture over a cylinder or other shapes, as long as the transformation the renderer uses to map a 3D XYZ location to a UV image sample is known and an inverse transformation can be determined.

WHERE WE'RE GOING

Procedural texturing (and modeling) is certainly still in the frontier of computer graphics. It's an open-ended niche that is two decades old, but still growing in importance. The always-increasing power of computers is displacing rendering speed as the major bottleneck; the production of complex material appearances and

geometry is now possible and practical. Imagine the complexity present in a full model of a city; a single animator could not design each building and surface manually. The answer is of course procedural definition, which is why it is steadily becoming more and more important over time.¹²

But what topics will be at that frontier? There are obviously many, but I see several “holes” in classical texturing that will likely be important soon.¹³ One of the largest gaps in even simple texturing is a strategy for more intelligent textures—ones that examine their environment (and particularly the object geometry) to change their appearance. The strategy used by Greg Turk (1991) for reaction-diffusion textures is one of the early forays into this field, but the potential is enormous. Imagine designing a dragon, but having the dragon scales behave differently over the belly of the dragon than the wings. Individual scales might change size based on the curvature of the skin (where there is high curvature, the plates need to be smaller to keep the area flexible). What is a good method for doing even this size scaling? What other effects could be modulated by object geometry? What about the geometry should the textures examine? The applications are obvious, but there have been no great developments in this area. Yet.

I see another topic becoming important soon. With the complexity of textures continually increasing (both in application methods and in internal design complexity), a “black-box” texture starts to become unwieldy for users to control. The black box is a texture personified by many control parameters, almost always numbers. When textures grow, the number of controls grows. It is not unusual to have up to 30 or 40 of these parameters! Even when carefully chosen and labeled, this many levers and knobs become difficult for the user to handle! This can be minimized through a good user interface (see page 194), but even then it is easy for a user (especially a nontechnical artist) to become swamped.

The black-box design is likely to be around for quite a while; it is a convenient way to design textures for the programmer. These parameters must therefore somehow be abstracted or hidden from users to avoid overwhelming them. This probably implies a user interface similar to the ones used by both Sims (1991a) and Todd and Latham (1993). Essentially, the user is presented not with many parameters to adjust

12. When this chapter was first written in 1994, I envisioned the day that we would have procedurally built cities, and even planets that could be viewed at any level of detail from orbit to 1 meter. That day has already come. At SIGGRAPH 2001, a procedural city technique was presented, as well as MojoWorld, a commercial “build a synthetic planet at any scale” renderer. What will happen in another 10 years?

13. I wrote this same sentence in 1994, but the topics are still just as appropriate in 2002, if not more so.

but with a collection of images. Each image is computed using different parameters (which are hidden from the user), and the user empirically ranks or sorts the images to identify the ones that are most interesting. From an artist's point of view, this is obviously appealing; it is very easy to use. The abstraction can also hide "parameters" of arbitrary complexity; Sims's textures are actually giant LISP expressions, not a mere vector of numeric parameters.

The big questions that remain are what sort of method is best for deciding how to choose the parameters to make the image to present to the user, and how to use the user's preferences to guide the production of new trial settings. Todd and Latham (1993) present a rudimentary answer to this question, mainly involving parameter "momentum," but especially for complex models this can become inadequate. I feel that development of a robust method for "texture evolution" based on numeric parameter vectors will be one of the most important tools for making procedural textures more useful from a user's point of view. In particular, two topics need to be studied: accounting for past explorations of the texture's parameter space and compensating for parameter correlations (where connections between parameters will complicate a "gradient" optimum search method). The growing complexity of textures really demands a new style of interface design, and this looks like the most promising.

So, while texturing is now two decades old, it is more important than ever. In computer graphics during the 1970s, the surface visibility problem was the main frontier. In the 1980s, lighting and surface properties (including the development of radiosity and BRDF surface models) were probably the most important developments. The 1990s produced efficient global lighting and image-based rendering. It is too early to tell what this current decade will be dominated by, but my prediction, especially for the next 10 years, will be procedural object and surface definition. We finally know how to render any model; now let's make the computer help us build the models themselves.