2

BUILDING PROCEDURAL TEXTURES

DARWYN PEACHEY

This chapter describes how to construct procedural texture functions in a variety of ways, starting from very simple textures and eventually moving on to quite elaborate ones. The discussion is intended to give you a thorough understanding of the major building blocks of procedural textures and the ways in which they can be combined.

Geometric calculations are fundamental to procedural texturing, as they are to most of 3D computer graphics. You should be familiar with 2D and 3D points, vectors, Cartesian coordinate systems, dot products, cross products, and homogeneous transformation matrices. You should also be familiar with the RGB (red, green, blue) representation of colors and with simple diffuse and specular shading models. Consult a graphics textbook (e.g., Foley et al. 1990; Hearn and Baker 1986) if any of this sounds new or unfamiliar.

INTRODUCTION

Throughout the short history of computer graphics, researchers have sought to improve the realism of their synthetic images by finding better ways to render the appearance of surfaces. This work can be divided into *shading* and *texturing*. Shading is the process of calculating the color of a pixel or shading sample from user-specified surface properties and the shading model. Texturing is a method of varying the surface properties from point to point in order to give the appearance of surface detail that is not actually present in the geometry of the surface.

Shading models (sometimes called *illumination models, lighting models,* or *reflection models*) simulate the interaction of light with surface materials. Shading models are usually based on physics, but they always make a great number of simplifying assumptions. Fully detailed physical models would be overkill for most computer graphics purposes and would involve intractable calculations.

The simplest realistic shading model, and the one that was used first in computer graphics, is the diffuse model, sometimes called the Lambertian model. A diffuse

surface has a dull or matte appearance. The diffuse model was followed by a variety of more realistic shading models that simulate specular (mirrorlike) reflection (Phong 1975; Blinn 1977; Cook and Torrance 1981; He et al. 1991). Kajiya (1985) introduced anisotropic shading models in which specular reflection properties are different in different directions. Cabral, Max, and Springmeyer (1987), Miller (1988a), and Poulin and Fournier (1990) have done further research on anisotropic shading models.

All of the shading models described above are so-called local models, which deal only with light arriving at the surface directly from light sources. In the early 1980s, most research on shading models turned to the problem of simulating *global* illumination effects, which result from indirect lighting due to reflection, refraction, and scattering of light from other surfaces or participating media in the scene. Ray-tracing and radiosity techniques typically are used to simulate global illumination effects.

Texture

At about the same time as the early specular reflection models were being formulated, Catmull (1974) generated the first textured computer graphics images. Catmull's surfaces were represented as parametric patches. Each point on the 3D surface of a parametric patch corresponds to a particular 2D point (u, v) in parameter space. This 2D-to-3D correspondence implies that a two-dimensional texture image can easily be mapped onto the 3D surface. The (u, v) parameters of any point on the patch can be used to compute a corresponding pixel location in the texture image.

For example, let's assume that you have a JPEG image with a resolution of 1024 $\times$ 512 pixels, and a patch (u, v) space that extends from 0 to 1 in each dimension. For any (u, v) point on the patch surface, you can compute the corresponding pixel location in the JPEG image by simply multiplying u times 1024 and v times 512. You can use the color of the image pixel to determine the shading of the patch surface at (u, v). In this way, the image can be *texture mapped* onto the patch. (Unfortunately, this simple point-sampling approach to texture mapping usually will result in so-called aliasing artifacts. To avoid such problems, more elaborate sampling and filtering methods are needed to accurately compute the contribution of each texture image pixel to each pixel of the final rendered graphics image.)

These first efforts proved that texture provided an interesting and detailed surface appearance, instead of a simple and boring surface of a single uniform color. It was clear that texture gave a quantum leap in realism at a very low cost in human

effort and computer time. Variations and improvements on the notion of texture mapping quickly followed.

Blinn and Newell (1976) introduced *reflection mapping* (also called *environment mapping*) to simulate reflections from mirrorlike surfaces. Reflection mapping is a simplified form of ray tracing. The reflection mapping procedure calculates the reflection direction R of a ray from the camera or viewer to the point being shaded:

$$R = 2(N \cdot V)N - V$$

where N is the surface normal and V points toward the viewer (both must be normalized). The texture image can be accessed using the "latitude" and "longitude" angles of the normalized vector $R = (x, y, z)$:

$$\theta = \tan^{-1}(y/x)$$
$$\varphi = \sin^{-1} z$$

suitably scaled and translated to fit the range of texture image pixel coordinates. If the reflection texture is chosen carefully, the texture-mapped surface appears to be reflecting an image of its surroundings. The illusion is enhanced if both the position of the shaded point and the reflection direction are used to calculate a ray intersection with an imaginary environment sphere surrounding the scene. The latitude and longitude of the intersection can be used to access the reflection texture.

Blinn (1978) introduced *bump mapping*, which made it possible to simulate the appearance of surface bumps without actually modifying the geometry. Bump mapping uses a texture pattern to modify the direction of surface normal vectors. When the resulting normals are used in the shading calculation, the rendered surface appears to have bumps and indentations. Such bumps aren't visible at the silhouette edges of the surface, since they consist only of shading variations, not geometry.

Cook (1984) described an extension of bump mapping, called *displacement mapping*, in which textures are used actually to move the surface, not just to change the normals. Moving the surface does change the normals as well, so displacement mapping often looks very much like bump mapping except that the bumps created by displacement are even visible on the silhouettes of objects.

Reeves, Salesin, and Cook (1987) presented an algorithm for producing antialiased shadows using an image texture based on a depth image of a scene rendered from the position of the light source. A stochastic sampling technique called "percentage closer filtering" was applied to reduce the effects of aliasing in the depth image.

Peachey (1985) and Perlin (1985) described space-filling textures called *solid textures* as an alternative to the 2D texture images that had traditionally been used. Gardner (1984, 1985) used solid textures generated by sums of sinusoidal functions to add texture to models of trees, terrains, and clouds. Solid textures are evaluated based on the 3D coordinates of the point being textured, rather than the 2D surface parameters of the point. Consequently, solid textures are unaffected by distortions of the surface parameter space, such as you might see near the poles of a sphere. Continuity between the surface parameterization of adjacent patches isn't a concern either. The solid texture will remain consistent and have features of constant size regardless of distortions in the surface coordinate systems.

For example, objects machined from solid wood exhibit different grain textures depending on the orientation of the surface with respect to the longitudinal growth axis of the original tree. Ordinary 2D techniques of applying wood-grain textures typically result in a veneer or "plastic wood" effect. Although each surface of a block may resemble solid wood, the unrealistic relationship between the textures on the adjacent surfaces destroys the illusion. A solid texture that simulates wood gives consistent textures on all surfaces of an object (Figure 2.1).

Recent work has focused on the problem of generating a smooth texture coordinate mapping over a complex surface, such that the mapping introduces a minimal amount of distortion in the size and shape of 2D textures applied with it. Ideally, a mapping preserves distances, areas, and angles, so that the texture is undistorted. However, in practice none of these properties is easily preserved when the textured surface is complex in shape and topology. Piponi and Borshukov (2000) describe a method called "pelting" that generates a seamless texture mapping of an arbitrary

FIGURE 2.1 A wood-grain solid texture.

subdivision surface model. Pedersen (1995) lets the user interactively position texture patches on a surface, while helping the user minimize the distortion of the mapping. Praun, Finkelstein, and Hoppe (2000) introduce "lapped textures," in which many small swatches of texture are positioned on a surface in an overlapping fashion, according to a user-defined vector field that guides the orientation and scale of the swatches. Both Turk (2001) and Wei and Levoy (2001) generate a synthetic texture from samples and "grow" the synthetic texture right on the target surface, taking into account its shape and parameterization.

Procedural Texture

From the earliest days of texture mapping, a variety of researchers used synthetic texture models to generate texture images instead of scanning or painting them. Blinn and Newell (1976) used Fourier synthesis. Fu and Lu (1978) proposed a syntactic grammar-based texture generation technique. Schacter and Ahuja (1979) and Schacter (1980) used Fourier synthesis and stochastic models of various kinds to generate texture imagery for flight simulators. Fournier, Fussell, and Carpenter (1982) and Haruyama and Barsky (1984) proposed using stochastic subdivision ("fractal") methods to generate textures. Other researchers developed statistical texture models that analyzed the properties of natural textures and then reproduced the textures from the statistical data (Gagalowicz and Ma 1985; Garber 1981).

Cook (1984) described the "shade trees" system, which was one of the first systems in which it was convenient to generate procedural textures during rendering. Shade trees enable the use of a different shading model for each surface as well as for light sources and for attenuation through the atmosphere. Because the inputs to the shading model can be manipulated procedurally, shade trees make it possible to use texture to control any part of the shading calculation. Color and transparency textures, reflection mapping, bump mapping, displacement mapping, and solid texturing can all be implemented using shade trees.

Perlin (1985) described a complete procedural texture generation language and laid the foundation for the most popular class of procedural textures in use today, namely, those based on *noise,* a stochastic texture generation primitive.

Turk (1991) and Witkin and Kass (1991) described synthetic texture models inspired by the biochemical processes that produce (among other effects) pigmentation patterns in the skins of animals.

Sims (1991a) described a very novel texture synthesis system in which procedural textures represented as LISP expressions are automatically modified and combined by a genetic programming system. By interactively selecting among the

resulting textures, the user of the system can direct the simulated evolution of a texture in some desired direction.

All of the texture synthesis methods mentioned in this section might be called "procedural." But just what *is* a procedural texture?

Procedural versus Nonprocedural

The definition of procedural texture is surprisingly slippery. The adjective *procedural* is used in computer science to distinguish entities that are described by program code rather than by data structures. For instance, in artificial intelligence there is a distinction between procedural representations of knowledge and declarative ones (see, for example, section 7.3 in Rich 1983). But anything we do with computers has a procedural aspect at some level, and almost every procedure takes some parameters or inputs that can be viewed as the declarative part of the description. In the mapping of a texture image onto a surface, the procedural component is the renderer's texture mapping module, and the declarative component is the texture image.

It is tempting to define a procedural texture as one that is changed primarily by modifying the algorithm rather than by changing its parameters or inputs. However, a procedural texture in a black box is still a procedural texture, even though you might be prevented from changing its code. This is true of procedural textures that are provided to you in a non-source-code form as part of a proprietary commercial renderer or texture package. Some rendering systems allow the user to create new procedural textures and modify existing procedural textures, but many others do not.

One major defining characteristic of a procedural texture is that it is *synthetic*—generated from a program or model rather than just a digitized or painted image. But image textures can be included among procedural textures in a procedural texture language that incorporates image-based texture mapping as one of its primitive operations. Some very nice procedural textures can be based on the procedural combination, modification, or distortion of image textures. The question "How procedural are such textures?" is difficult to answer and hinges on the apparent amount of difference between the source images and the resulting texture.

Implicit and Explicit Procedures

We can distinguish two major types of procedural texturing or modeling methods: *explicit* and *implicit* methods. In explicit methods, the procedure directly generates the points that make up a shape. In implicit methods, the procedure answers a query

about a particular point. The most common form of implicit method is the *isocurve* (in 2D) or *isosurface* (in 3D) method. A texture pattern is defined as a function F of points P in the texture space, and the pattern consists of a level set of F, that is, the set of all points at which the function has a particular value C: $\{P \mid F(P) = C\}$. For example, a simple definition of a unit circle is the isocurve model $\{P \in R^2 | P_x^2 + P_y^2 = 1\}$. Note that F must be reasonably well behaved if the function is to form a sensible pattern: we want F to be continuous and perhaps differentiable depending on the application.

Implicit geometric models traditionally have been popular in ray tracers because the problem of intersecting a ray with the model can be expressed elegantly for implicit models: given a model $F(P) = 0$ and a ray $R(t) = O + t\,D$ with origin O and direction D, the intersection point is simply $R(t_{hit})$ where t_{hit} is the smallest positive root of $F(R(t)) = 0$. On the other hand, explicit models are convenient for depth buffer renderers (Z-buffers and A-buffers) because the explicit model can directly place points into the depth buffer in arbitrary order as the model is evaluated.

In the texturing domain, implicit procedural methods seem to be best for textures that are evaluated during rendering. In both ray tracers and depth buffer renderers, texture samples usually must be evaluated in an order that is determined by the renderer, not by the texture procedure. An implicit procedure fits perfectly in such an environment because it is designed to answer a query about any point in the texture at any time. An explicit procedure wants to generate its texture pattern in some fixed order, which probably doesn't match the needs of the rendering algorithm. In most renderers, using an explicit texture routine would require running the texture procedure as a prepass and having it generate the texture image into an image buffer, where it could be looked up as necessary for texture application during rendering. Many of the advantages of procedural textures are lost if the texture must be evaluated in a prepass.

In principle the explicit and implicit methods can be used to produce the same class of texture patterns or geometric models (virtually anything), but in practice each approach has its own class of models that are convenient or feasible. Explicit models are convenient for polygons and parametric curves and patches. Implicit models are convenient for quadrics and for patterns that result from potential or force fields. Since implicit models tend to be continuous throughout a region of the modeling space, they are appropriate for continuous density and flow phenomena such as natural stone textures, clouds, and fog.

The remainder of this chapter focuses on building procedural textures that are evaluated during rendering and, therefore, on implicit procedural textures. Some of the texture synthesis methods mentioned earlier, for example, reaction-diffusion

textures or syntactically generated textures, can be very difficult to generate implicitly during rendering. Other methods, such as Fourier spectral synthesis, fit well into an implicit procedural system.

Advantages of Procedural Texture

The advantages of a procedural texture over an image texture are as follows:

- A procedural representation is extremely compact. The size of a procedural texture is usually measured in kilobytes, while the size of a texture image is usually measured in megabytes. This is especially true for solid textures, since 3D texture "images" are extremely large. Nonetheless, some people have used tomographic X-ray scanners to obtain digitized volume images for use as solid textures.
- A procedural representation has no fixed resolution. In most cases it can provide a fully detailed texture no matter how closely you look at it (no matter how high the resolution).
- A procedural representation covers no fixed area. In other words, it is unlimited in extent and can cover an arbitrarily large area without seams and without unwanted repetition of the texture pattern.
- A procedural texture can be parameterized, so it can generate a class of related textures rather than being limited to one fixed texture image.

Many of these advantages are only *potential* advantages; procedural texturing gives you the tools to gain these advantages, but you must make an effort to use them. A badly written procedural texture could sacrifice any of these potential advantages. A procedural texture evaluated before rendering has only one of these advantages, namely, that it can be parameterized to generate a variety of related textures.

Disadvantages of Procedural Texture

The disadvantages of a procedural texture as compared to an image texure are as follows:

- A procedural texture can be difficult to build and debug. Programming is often hard, and programming an implicit pattern description is especially hard in nontrivial cases.

- A procedural texture can be a surprise. It is often easier to predict the outcome when you scan or paint a texture image. Some people choose to like this property of procedural textures and call it "serendipity." Some people hate it and say that procedural textures are hard to control.
- Evaluating a procedural texture can be slower than accessing a stored texture image. This is the classic time versus space trade-off.
- Aliasing can be a problem in procedural textures. Antialiasing can be tricky and is less likely to be taken care of automatically than it is in image-based texturing.

The RenderMan Shading Language

Some renderers provide special-purpose shading languages in which program code for shading models and procedural textures can be written. The RenderMan shading language is perhaps the best known and most widely used of such languages (Pixar 1989; Hanrahan and Lawson 1990). It is a descendant of the shade trees system described by Cook (1984). The syntax of the language is C-like, but the shading language contains built-in data types and operations that are convenient for computer graphics calculations: for example, data types for points and colors, and operators for common vector operations. The shading language lets us program any aspect of the shading calculations performed by the RenderMan renderer: surface shading, light source description, atmospheric effects, and surface displacement. Shading parameters can be made to vary across the surface to generate procedural texture effects. These could consist of variations in color, transparency, surface position, surface normal, shininess, shading model, or just about anything else you can think of.

The shading language is based on an implicit programming model, in which shading procedures called "shaders" are asked to supply the color, opacity, and other properties of specified points on a surface. As discussed in the previous section, this shading paradigm is used in most renderers, both depth buffers and ray tracers. We refer to the surface point being shaded as the "shading sample point" or simply as the "sample point." The color and opacity information that is computed by the shader for the sample point sometimes is called the "shading sample."

In the remainder of this chapter, the RenderMan shading language is used for the procedural texturing examples. You should be able to understand the examples without RenderMan experience, but if you plan to write RenderMan shaders yourself, you will benefit from reading Upstill (1990).

Figures 2.2–2.7 are examples of images produced at Pixar for television commercials and films. All of the textures and lighting effects in these images were generated in the RenderMan shading language. Scanned texture images were used occasionally for product packaging materials and other graphic designs that contain text, since the shapes of letters are rather tedious to produce in an implicit procedural texture. Most of the textures are purely procedural. These examples demonstrate that the class of textures that can be generated procedurally is very large indeed.

What If You Don't Use RenderMan?

We hope that this chapter will be useful to a wide audience of people interested in procedural textures, not only to those who write RenderMan shaders. The

FIGURE 2.2 *Knickknack*. Copyright © 1989 Pixar.

FIGURE 2.3 Lifesavers, "At the Beach." Image by Pixar. Copyright © 1991 FCB/Leber Katz Partners.

FIGURE 2.4 Carefree Gum, "Bursting." Image by Pixar. Copyright © 1993 FCB/Leber Katz Partners.

FIGURE 2.6 Listerine, "Arrows." Image by Pixar. Copyright © 1994 J. Walter Thompson.

FIGURE 2.5 Listerine, "Knight." Image by Pixar. Copyright © 1991 J. Walter Thompson.

FIGURE 2.7 Listerine, "Arrows." Image by Pixar. Copyright © 1994 J. Walter Thompson.

RenderMan shading language is a convenient way to express procedural shaders, and it allows us to easily render high-quality images of the examples. However, this chapter also describes the C language implementation of many of the procedural texture building-block functions that are built into the RenderMan shading language. This should allow you to translate the shading language examples into C code that you can use in your own rendering program. The RenderMan shading language is superficially much like the C language, so you might have to look closely to see whether a given fragment of code is in C or in the shading language.

The RenderMan shading language provides functions that access image textures. It is not practical to include an implementation of those functions in this chapter. Efficient image texture mapping with proper filtering is a complex task, and the program code consists of several thousand lines of C. Most other renderers have their own texture functions, and you should be able to translate the RenderMan texture functions into the appropriate code to access the texture functions of your renderer.

PROCEDURAL PATTERN GENERATION

Sitting down to build a procedural texture can be a daunting experience, at least the first few times. This section gives some hints, guidelines, and examples that will help you stave off the anxiety of facing an empty text editor screen with no idea of how to begin. It is usually much easier to start by copying an example and modifying it to do what you want than it is to start from scratch. Most of the time, it is better to borrow code than to invent it.

Most surface shaders can be split into two components called *pattern generation* and the *shading model.* Pattern generation defines the texture pattern and sets the values of surface properties that are used by the shading model. The shading model simulates the behavior of the surface material with respect to diffuse and specular reflection.

Shading Models

Most surface shaders use one of a small number of shading models. The most common model includes diffuse and specular reflection and is called the "plastic" shading model. It is expressed in the RenderMan shading language as follows:

```
surface
plastic(float Ka = 1, Kd = 0.5, Ks = 0.5;
      float roughness = 0.1;
      color specularcolor = color (1,1,1))
{
      point Nf = faceforward(normalize(N), I);
      point V = normalize(-I);
      Oi = Os;
      Ci = Os * (Cs * (Ka * ambient()
            + Kd * diffuse(Nf))
            + specularcolor * Ks
                  * specular(Nf, V, roughness));
}
```

The following paragraphs explain the workings of the `plastic` shader in detail, as a way of introducing the RenderMan shading language to readers who are not familiar with it.

The parameters of the `plastic` shader are the coefficients of ambient, diffuse, and specular reflectance; the roughness, which controls the size of specular highlights; and the color of the specular highlights. Colors are represented by RGB

triples, specifying the intensities of the red, green, and blue primary colors as numbers between 0 and 1. For example, in this notation, `color (1,1,1)` is white.

Any RenderMan surface shader can reference a large collection of built-in quantities such as `P`, the 3D coordinates of the point on the surface being shaded, and `N`, the surface normal at `P`. The normal vector is perpendicular to the tangent plane of the surface at `P` and points toward the outside of the surface. Because surfaces can be two-sided, it is possible to see the inside of a surface; in that case we want the normal vector to point toward the camera, not away from it. The built-in function `faceforward` simply compares the direction of the incident ray vector `I` with the direction of the normal vector `N`. `I` is the vector from the camera position to the point `P`. If the two vectors `I` and `N` point in the same direction (i.e., if their dot product is positive), `faceforward` returns `-N` instead of `N`.

The first statement in the body of the shader declares and initializes a surface normal vector `Nf`, which is normalized and faces toward the camera. The second statement declares and initializes a "viewer" vector `V` that is normalized and gives the direction to the camera. The third statement sets the output opacity `Oi` to be equal to the input surface opacity `Os`. If the surface is somewhat transparent, the opacity is less than one. Actually, `Os` is a color, an RGB triple that gives the opacity of the surface for each of the three primary colors. For an opaque surface, `Os` is `color(1,1,1)`.

The final statement in the shader does the interesting work. The output color `Ci` is set to the product of the opacity and a color.[1] The color is the sum of an ambient term and a diffuse term multiplied by the input surface color `Cs`, added to a specular term whose color is determined by the parameter `specularcolor`. The built-in functions `ambient`, `diffuse`, and `specular` gather up all of the light from multiple light sources according to a particular reflection model. For instance, `diffuse` computes the sum of the intensity of each light source multiplied by the dot product of the direction to the light source and the surface normal `Nf` (which is passed as a parameter to `diffuse`).

The plastic shading model is flexible enough to include the other two most common RenderMan shading models, the "matte" model and the "metal" model, as special cases. The matte model is a perfectly diffuse reflector, which is equivalent to plastic with a `Kd` of 1 and a `Ks` of 0. The metal model is a perfectly specular reflector,

1. The color is multiplied by the opacity because RenderMan uses an "alpha blending" technique to combine the colors of partially transparent surfaces, similar to the method described by Porter and Duff (1984).

which is equivalent to plastic with a `Kd` of 0, a `Ks` of 1, and a `specularcolor` the same as `Cs`. The `specularcolor` parameter is important because it has been observed that when illuminated by a white light source, plastics and other dielectric (insulating) materials have white highlights while metals and other conductive materials have colored highlights (Cook and Torrance 1981). For example, gold has a gold-colored highlight.

The `plastic` shader is a good starting point for many procedural texture shaders. We will simply replace the `Cs` in the last statement of the shader with a new color variable `Ct`, the texture color that is computed by the pattern generation part of the shader.

Pattern Generation

Pattern generation is usually the hard part of writing a RenderMan shader because it involves figuring out how to generate a particular texture pattern procedurally.

If the texture pattern is simply an image texture, the shader can call the built-in shading language function `texture`:

```
Ct = texture("name.tx",s,t);
```

The shading language `texture` function looks up pixel values from the specified image texture `"name.tx"` and performs filtering calculations as needed to prevent aliasing artifacts. The `texture` function has the usual 2D texture space with the texture image in the unit square. The built-in variables `s` and `t` are the standard RenderMan texture coordinates, which by default range over the unit interval [0, 1] for any type of surface. The shading language also provides an `environment` function whose 2D texture space is accessed using a 3D direction vector that is converted internally into 2D form to access a latitude-longitude or cube-face environment map (Greene 1986).

When the texture image is suitable, there is no easier or more realistic way to generate texture patterns. Unfortunately, it is difficult to get a texture image that is suitable for many texture applications. It would be nice if all desired textures could be implemented by finding a sample of the actual material, photographing it, and scanning the photograph to produce a beautiful texture image. But this approach is rarely adequate by itself. If a material is not completely flat and smooth, the photograph will capture information about the lighting direction and the light source. Each bump in the material will be shaded based on its slope, and in the worst case, the bumps will cast shadows. Even if the material is flat and smooth, the photograph often will record uneven lighting conditions, reflections of the environment

surrounding the material, highlights from the light sources, and so on. This information generates incorrect visual cues when the photograph is texture mapped onto a surface in a scene with simulated lighting and environmental characteristics that differ from those in the photograph. A beautiful photograph often looks out of place when texture mapped onto a computer graphics model.

Another problem with photographic texture images is that they aren't infinitely large. When a large area must be covered, copies of the texture are placed side by side. A seam is usually visible because the texture pixels don't match from the top to the bottom or the left to the right. Sometimes retouching can be used to make the edges match up seamlessly. Even when this has been done, a large area textured with many copies of the image can look bad because it is obvious that a small amount of texture data has been used to texture a large area. Prominent features in the texture map are replicated over and over in an obviously repetitive way. Such problems can be avoided by making sure that the texture photograph covers a large area of the texture, but this will result in visible pixel artifacts in a rendered image that magnifies a tiny part of the texture. (As an alternative approach, you might consider the remarkably successful algorithm described by Wei and Levoy (2000), which analyzes a small texture image and then synthesizes a larger texture image that resembles the smaller image, but does not duplicate it.)

The right way to use image textures is to design the shader first and then go out and get a suitable texture photograph to scan. The lighting, the size of the area photographed, and the resolution of the scanning should all be selected based on the application of the shader. In many cases the required texture images will be bump altitude maps, monochrome images of prominent textural features, and so on. The texture images may be painted by hand, generated procedurally, or produced from scanned material after substantial image processing and retouching.

Procedural pattern generators are more difficult to write. In addition to the problems of creating a small piece of program code that produces a convincing simulation of some material, the procedural pattern generator must be antialiased to prevent fine details in the pattern from aliasing when seen from far away.

Procedural pattern generators are harder to write than texture image shaders, but they have several nice properties. It is usually easy to make the texture cover an arbitrarily large area without seams or objectionable repetition. It is easy to separate small-scale shape effects such as bumps and dimples from color variations; each can be generated separately in the procedural shader.

Writing procedural pattern generators is still an art form; there is no recipe that will work every time. This is a programming task in which the problem is to generate the appearance of some real-world material. The first step is to go out and examine

the real material: its color and color variations, its reflection properties, and its surface characteristics (smooth or rough, bumpy or pitted). Photographs that you can take back to your desk are very valuable. Architectural and design magazines and books are a good source of pictures of materials.

Texture Spaces

The RenderMan shading language provides many different built-in coordinate systems (also called *spaces)*. A coordinate system is defined by the concatenated stack of transformation matrices that is in effect at a given point in the hierarchical structure of the RenderMan geometric model.

- The `current` space is the one in which shading calculations are normally done. In most renderers, `current` space will turn out to be either `camera` space or `world` space, but you shouldn't depend on this.
- The `world` space is the coordinate system in which the overall layout of your scene is defined. It is the starting point for all other spaces.
- The `object` space is the one in which the surface being shaded was defined. For instance, if the shader is shading a sphere, the `object` space of the sphere is the coordinate system that was in effect when the `RiSphere` call was made to create the sphere. Note that an object made up of several surfaces all using the same shader might have different object spaces for each of the surfaces if there are geometric transformations between the surfaces.
- The `shader` space is the coordinate system that existed when the shader was invoked (e.g., by an `RiSurface` call). This is a very useful space because it can be attached to a user-defined collection of surfaces at an appropriate point in the hierarchy of the geometric model so that all of the related surfaces share the same `shader` space.

In addition, user-defined coordinate systems can be created and given names using the `RiCoordinateSystem` call. These coordinate systems can be referenced by name in the shading language.

It is very important to choose the right texture space when defining your texture. Using the 2D surface texture coordinates (s, t) or the surface parameters (u, v) is fairly safe, but might cause problems due to nonuniformities in the scale of the parameter space (e.g., compression of the parameter space at the poles of a sphere). Solid textures avoid that problem because they are defined in terms of the 3D coordinates of the sample point. If a solid texture is based on the `camera` space coordinates

of the point, the texture on a surface will change whenever either the camera or the object is moved. If the texture is based on `world` space coordinates, it will change whenever the object is moved. In most cases, solid textures should be based on the `shader` space coordinates of the shading samples, so that the texture will move properly with the object. The `shader` space is defined when the shader is invoked, and that can be done at a suitable place in the transformation hierarchy of the model so that everything works out.

It is a simplification to say that a texture is defined in terms of a single texture space. In general a texture is a combination of a number of separate "features," each of which might be defined in terms of its own *feature space*. If the various feature spaces that are used in creating the texture are not based on one underlying texture space, great care must be exercised to be sure that texture features don't shift with respect to one another. The feature spaces should have a fixed relationship that doesn't change when the camera or the object moves.

Layering and Composition

The best approach to writing a complex texture pattern generator is to build it up from simple parts. There are a number of ways to combine simple patterns to make complex patterns.

One technique is *layering*, in which simple patterns are placed on top of one another. For example, the colors of two texture layers could be added together. Usually, it is better to have some texture function control how the layers are combined. The shading language `mix` function is a convenient way of doing this.

```
C = mix(C0, C1, f);
```

The number `f`, between 0 and 1, is used to select one of the colors `C0` and `C1`. If `f` is 0, the result of the `mix` is `C0`. If `f` is 1, the result is `C1`. If `f` is between 0 and 1, the result is a linearly interpolated mixture of `C0` and `C1`. The `mix` function is defined as

```
color
mix(color C0, color C1, float f)
{
      return (1-f)*C0 + f*C1;
}
```

`C0` and `C1` can be fixed colors, or they can be two subtextures. In either case, they are combined under the control of the number `f`, which is itself the result of some procedural texture function.

When two colors are multiplied together in the shading language, the result is a color whose RGB components are the product of the corresponding components from the input colors. That is, the red result is the product of the red components of the two inputs. Color multiplication can simulate the filtering of one color by the other. If color `C0` represents the transparency of a filter to red, green, and blue light, then `C0*C1` represents the color `C1` as viewed through the filter.

Be careful when using a four-channel image texture that was created from an RGBA image (an image with an opacity or "alpha" channel) because the colors in such an image are normally premultiplied by the value of the alpha channel. In this case, it is not correct simply to combine the RGB channels with another color under control of the alpha channel. The correct way to merge an RGBA texture over another texture color `Ct` is

```
color C;
float A;

C = color texture("mytexture",s,t);
A = texture("mytexture"[3],s,t);
result = C + (1-A) * Ct;
```

`C` is the image texture color, and `A` is the alpha channel of the image texture (channel number 3). Since `C` has already been multiplied by `A`, the expression `C + (1-A*Ct)` is the right way to *lerp*[2] between `C` and `Ct`.

Another way to combine simple functions to make complex functions is *functional composition,* using the outputs of one or more simple functions as the inputs of another function. For example, one function generates a number that varies between 0 and 1 in a particular way, and this number is used as the input to another function that generates different colors for different values of its numerical input. One function might take inputs that are points in one texture space and produce output points in another space that are the input to a second function; in this case, the first function transforms points into the feature space needed by the second function. Composition is very powerful and is so fundamental to programming that you really can't avoid using it.

The computer science literature concerning *functional programming* is a good source of techniques for combining functions (Ghezzi and Jazayeri 1982). Functional languages such as LISP (Winston and Horn 1984) rely heavily on composition and related techniques.

---

2. In computer graphics, linear interpolation is colloquially called *lerping.*

The remainder of this section presents a series of primitive functions that are used as building blocks for procedural textures. The presentation includes several examples of the use of these primitives in procedural texture shaders.

Steps, Clamps, and Conditionals

From the earlier discussion on methods of combining primitive operations to make procedural patterns, it should be clear that functions taking parameters and returning values are the most convenient kind of primitive building blocks. Steps and clamps are conditional functions that give us much the same capabilities that `if` statements give us. But steps and clamps are often more convenient, simply because they are functions.

The RenderMan shading language function `step(a,x)` returns the value 0 when `x` is less than `a` and returns 1 otherwise. The `step` function can be written in C as follows:

```
float
step(float a, float x)
{
      return (float) (x >= a);
}
```

A graph of the `step` function is shown in Figure 2.8.

The main use of the `step` function is to replace an `if` statement or to produce a sharp transition between one type of texture and another type of texture. For example, an `if` statement such as

```
if (u < 0.5)
      Ci = color (1,1,.5);
else
      Ci = color ( .5,.3,1);
```

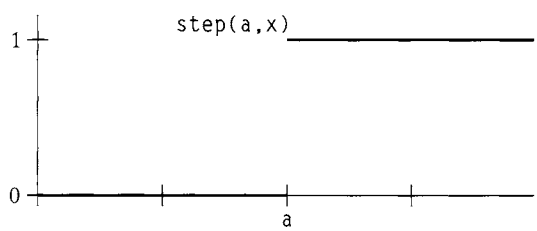

FIGURE 2.8 The `step` function.

can be rewritten to use the `step` function as follows:

```
Ci = mix(color (1,1,.5), color (.5,.3,1), step(0.5, u));
```

Later in this chapter when we examine antialiasing, you'll learn how to create an antialiased version of the `step` function. Writing a procedural texture with a lot of `if` statements instead of step functions can make antialiasing much harder.[3]

Two `step` functions can be used to make a rectangular pulse as follows:

```
#define PULSE(a,b,x) (step((a),(x)) - step((b),(x)))
```

This preprocessor macro generates a pulse that begins at `x = a` and ends at `x = b`. A graph of the pulse is shown in Figure 2.9.

The RenderMan shading language function `clamp(x,a,b)` returns the value `a` when `x` is less than `a`, the value of `x` when `x` is between `a` and `b`, and the value `b` when `x` is greater than `b`. The `clamp` function can be written in C as follows:

```
float
clamp(float x, float a, float b)
{
      return (x < a ? a: (x > b ? b : x));
}
```

A graph of the `clamp` function is shown in Figure 2.10.

The well-known `min` and `max` functions are closely related to `clamp`. In fact, `min` and `max` can be written as `clamp` calls, as follows:

```
min(x, b) ≡ clamp(x, x, b)
```

and

```
max(x, a) ≡ clamp(x, a, x)
```

Alternatively, `clamp` can be expressed in terms of `min` and `max`:

```
clamp(x, a, b) ≡ min(max(x, a), b)
```

3. Another reason for using `step` instead of `if` in RenderMan shaders is that it encourages you to compute the inputs of a conditional everywhere, not just in the fork of the conditional where they are used. This can avoid problems in applying image textures and other area operations.

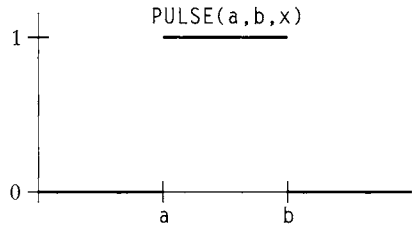

FIGURE 2.9 Two steps used to make a pulse.

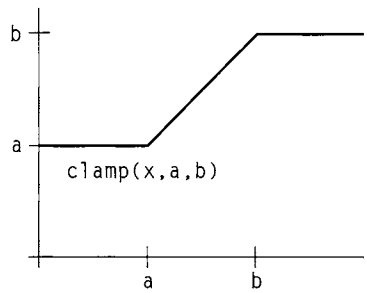

FIGURE 2.10 The `clamp` function.

For completeness, here are the C implementations of `min` and `max`:

```
float
min(float a, float b)
{
      return (a < b ? a : b);
}
float
max(float a, float b)
{
      return (a < b ? b : a);
}
```

Another special conditional function is the `abs` function, expressed in C as follows:

```
float
abs(float x)
{
      return (x < 0 ? -x : x);
}
```

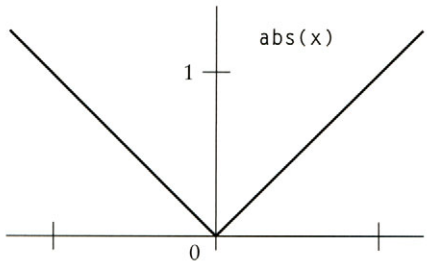

FIGURE 2.11 The abs function.

A graph of the abs function is shown in Figure 2.11. The abs function can be viewed as a rectifier; for example, it will convert a sine wave that oscillates between −1 and 1 into a sequence of positive sinusoidal pulses that range from 0 to 1.

In addition to the "pure" or "sharp" conditionals step, clamp, min, max, and abs, the RenderMan shading language provides a "smooth" conditional function called smoothstep. This function is similar to step, but instead of a sharp transition from 0 to 1 at a specified threshold, smoothstep(a,b,x) makes a gradual transition from 0 to 1 beginning at threshold a and ending at threshold b. In order to do this, smoothstep contains a cubic function whose slope is 0 at a and b and whose value is 0 at a and 1 at b. There is only one cubic function that has these properties for a = 0 and b = 1, namely, the function $3x^2 - 2x^3$.

Here is a C implementation of smoothstep, with the cubic function expressed according to Horner's rule:[4]

```
float
smoothstep(float a, float b, float x)
{
      if (x < a)
            return 0;
      if (x >= b)
            return 1;
      x = (x - a)/(b - a);
      return (x*x * (3 - 2*x));
}
```

A graph of the smoothstep function is shown in Figure 2.12.

4. Horner's rule is a method of nested multiplication for efficiently evaluating polynomials.

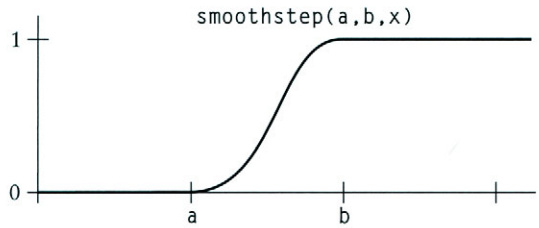

FIGURE 2.12 The smoothstep function.

The smoothstep function is used instead of step in many procedural textures because sharp transitions often result in unsightly artifacts. Many of the artifacts are due to aliasing, which is discussed at length later in this chapter. Sharp transitions can be very unpleasant in animated sequences because some features of the texture pattern appear suddenly as the camera or object moves (the features are said to "pop" on and off). Most of the motion in an animated sequence is carefully "eased" in and out to avoid sudden changes in speed or direction; the smoothstep function helps to keep the procedural textures in the scene from changing in equally unsettling ways.

Periodic Functions

The best-known periodic functions are sin and cos. They are important because of their close ties to the geometry of the circle, to angular measure, and to the representation of complex numbers. It can be shown that other functions can be built up from a sum of sinusoidal terms of different frequencies and phases (see the discussion on "Spectral Synthesis" on page 48).

sin and cos are available as built-in functions in C and in the RenderMan shading language. Some ANSI C implementations provide single-precision versions of sin and cos, called sinf and cosf, which you might prefer to use to save computation time. A graph of the sin and cos functions is shown in Figure 2.13.

Another important periodic function is the mod function. mod(a,b) gives the positive remainder obtained when dividing a by b. C users beware! Although C has a built-in integer remainder operator "%" and math library functions fmod and fmodf for double and float numbers, all of these are really remainder functions, not modulus functions, in that they will return a negative result if the first operand, a, is negative. Instead, you might use the following C implementation of mod:

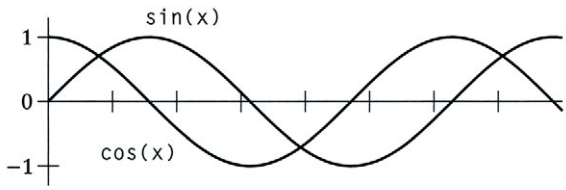

FIGURE 2.13 The `sin` and `cos` functions.

```
float
mod(float a, float b)
{
   int n = (int)(a/b);
   a -= n*b;
   if (a < 0)
     a += b;
   return a;
}
```

A graph of the periodic sawtooth function `mod(x,a)/a` is shown in Figure 2.14. This function has an amplitude of one and a period of `a`.

By applying `mod` to the inputs of some other function, we can make the other function periodic too. Take any function, say, $f(x)$, defined on the interval from 0 to 1 (technically, on the half-open interval [0, 1]). Then f (`mod(x,a)/a`) is a periodic function. To make this work out nicely, it is best if $f(0) = f(1)$ and even better if the derivatives of f are also equal at 0 and 1. For example, the pulse function `PULSE(0.4,0.6,x)` can be combined with the `mod` function to get the periodic square wave function `PULSE(0.4,0.6,mod(x,a)/a)` with its period equal to `a` (see Figure 2.15).

It's often preferable to use another `mod`-like idiom instead of `mod` in your shaders. We can think of `xf = mod(a,b)/b` as the fractional part of the ratio `a/b`. In many cases it is useful to have the integer part of the ratio, `xi`, as well.

```
float xf, xi;
xf = a/b;
xi = floor(xf);
xf -= xi;
```

The function `floor(x)` returns the largest integer that is less than or equal to `x`. Since the `floor` function is a built-in part of both the C math library and the

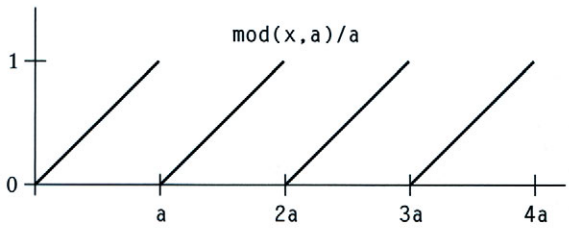

FIGURE 2.14 The periodic function `mod(x,a)/a`.

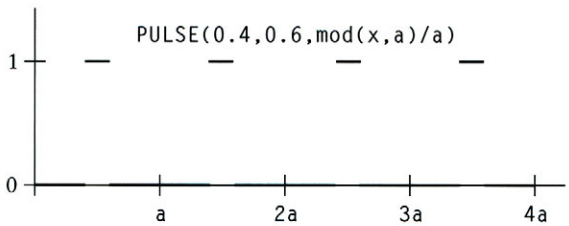

FIGURE 2.15 How to make a function periodic.

RenderMan shading language, this piece of code will work equally well in either language. Some versions of ANSI C provide a `floorf` function whose argument and result are single-precision float numbers. In C, the following macro is an alternative to the built-in `floor` function:

```
#define FLOOR(x) ((int)(x) - ((x) < 0 && (x) != (int)(x)))
```

`FLOOR` isn't precisely the same as `floor`, because `FLOOR` returns a value of `int` type rather than `double` type. Be sure that the argument passed to `FLOOR` is the name of a variable, since the macro may evaluate its argument up to four times.

A closely related function is the ceiling function `ceil(x)`, which returns the smallest integer that is greater than or equal to `x`. The function is built into the shading language and the C math library. ANSI C provides the single-precision version `ceilf`. The following macro is an alternative to the C library function:

```
#define CEIL(x) ((int)(x) + ((x) > 0 && (x) != (int)(x)))
```

Splines and Mappings

The RenderMan shading language has a built-in spline function, which is a one-dimensional Catmull-Rom interpolating spline through a set of so-called *knot* values. The parameter of the spline is a floating-point number.

```
result = spline(parameter,
  knot1, knot2, . . . , knotN-1, knotN);
```

In the shading language, the knots can be numbers, colors, or points (but all knots must be of the same type). The result has the same data type as the knots. If `parameter` is 0, the result is `knot2`. If `parameter` is 1, the result is `knotN-1`. For values of `parameter` between 0 and 1, the value of `result` interpolates smoothly between the values of the knots from `knot2` to `knotN-1`. The `knot1` and `knotN` values determine the derivatives of the spline at its end points. Because the spline is a cubic polynomial, there must be at least four knots.

Here is a C language implementation of `spline` in which the knots must be floating-point numbers:

```
/* Coefficients of basis matrix. */
#define CR00      -0.5
#define CR01      1.5
#define CR02      -1.5
#define CR03      0.5
#define CR10      1.0
#define CR11      -2.5
#define CR12      2.0
#define CR13      -0.5
#define CR20      -0.5
#define CR21      0.0
#define CR22      0.5
#define CR23      0.0
#define CR30      0.0
#define CR31      1.0
#define CR32      0.0
#define CR33      0.0

float
spline(float x, int nknots, float *knot)
{
      int span;
      int nspans = nknots - 3;
      float c0, c1, c2, c3; /* coefficients of the cubic.*/
      if (nspans < 1){/* illegal */
```

```
        fprintf(stderr, "Spline has too few knots.\n");
        return 0;
    }
    /* Find the appropriate 4-point span of the spline. */
    x = clamp(x, 0, 1) * nspans;
    span = (int) x;
    if (span >= nknots - 3)
        span = nknots - 3;
    x -= span;
    knot += span;

    /* Evaluate the span cubic at x using Horner's rule. */

    c3 = CR00*knot[0] + CR01*knot[1] + CR02*knot[2] + CR03*knot[3];
    c2 = CR10*knot[0] + CR11*knot[1] + CR12*knot[2] + CR13*knot[3];
    c1 = CR20*knot[0] + CR21*knot[1] + CR22*knot[2] + CR23*knot[3];
    c0 = CR30*knot[0] + CR31*knot[1] + CR32*knot[2] + CR33*knot[3];

    return ((c3*x + c2)*x + c1)*x + c0;

}
```

A graph of a particular example of the `spline` function is shown in Figure 2.16.

This code can easily be adapted to work with knots that are colors or points. Just do the same thing three times, once for each of the components of the knots. In other words,

```
spline(parameter, (x1,y1,z1), . . . , (xN,yN,zN))
```

is exactly equivalent to

```
(spline(parameter, x1, . . . , xN),
spline(parameter, y1, . . . , yN),
spline(parameter, z1, . . . , zN))
```

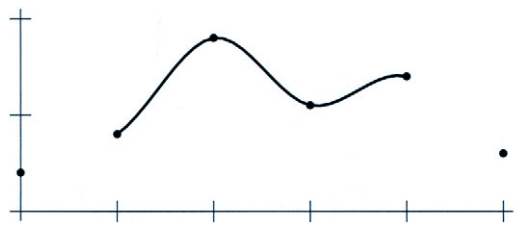

FIGURE 2.16 An example of the `spline` function.

The `spline` function is used to map a number into another number or into a color. A spline can approximate any function on the [0, 1] interval by giving values of the function at equally spaced sample points as the knots of the spline. In other words, the spline can interpolate function values from a table of known values at equally spaced values of the input parameter. A spline with colors as knots can be used as a *color map* or *color table.*

An example of this technique is a shader that simulates a shiny metallic surface by a procedural reflection map texture. The shader computes the reflection direction `R` of the viewer vector `V`. The vertical component of `R` in `world` space is used to look up a color value in a spline that goes from brown earth color below to pale bluish-white at the horizon and then to deeper shades of blue in the sky. Note that the shading language's built-in `vtransform` function properly converts a direction vector from the current rendering space to another coordinate system specified by name.

```
#define BROWN  color (0.1307,0.0609,0.0355)
#define BLUE0  color (0.4274,0.5880,0.9347)
#define BLUE1  color (0.1221,0.3794,0.9347)
#define BLUE2  color (0.1090,0.3386,0.8342)
#define BLUE3  color (0.0643,0.2571,0.6734)
#define BLUE4  color (0.0513,0.2053,0.5377)
#define BLUE5  color (0.0326,0.1591,0.4322)
#define BLACK  color (0,0,0)
surface
metallic( )
{
      point Nf = normalize(faceforward(N, I));
      point V = normalize(-I);
      point R; /* reflection direction */
      point Rworld; /* R in world space */
      color Ct;
      float altitude;

      R = 2 * Nf * (Nf . V) - V;
      Rworld = normalize(vtransform("world", R));
      altitude = 0.5 * zcomp(Rworld) + 0.5;
      Ct = spline(altitude,
            BROWN, BROWN, BROWN, BROWN, BROWN,
            BROWN, BLUE0, BLUE1, BLUE2, BLUE3,
            BLUE4, BLUE5, BLACK);
      Oi = Os;
      Ci = Os * Cs * Ct;
}
```

FIGURE 2.17 A spline-based reflection texture.

Figure 2.17 is an image shaded with the metallic reflection map shader.

Since `mix` functions and so many other selection functions are controlled by values that range over the [0, 1] interval, mappings from the unit interval to itself can be especially useful. Monotonically increasing functions on the unit interval can be used to change the distribution of values in the interval. The best-known example of such a function is the "gamma correction" function used to compensate for the nonlinearity of CRT display systems:

```
float
gammacorrect(float gamma, float x)
{
      return pow(x, 1/gamma);
}
```

Figure 2.18 shows the shape of the gamma correction function for `gamma` values of 0.4 and 2.3. If `x` varies over the [0, 1] interval, then the result is also in that interval. The zero and one end points of the interval are mapped to themselves. Other values are shifted upward toward one if `gamma` is greater than one, and shifted downward toward zero if `gamma` is between zero and one.

Perlin and Hoffert (1989) use a version of the gamma correction function that they call the `bias` function. The `bias` function replaces the `gamma` parameter with a parameter `b`, defined such that `bias(b,0.5) = b`.

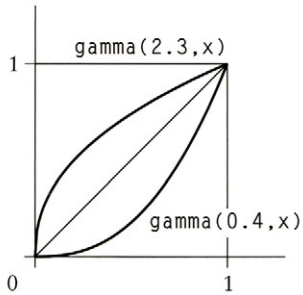

FIGURE 2.18 The gamma correction function.

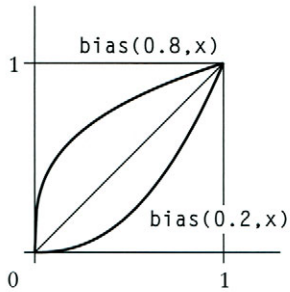

FIGURE 2.19 The `bias` function.

```
float
bias(float b, float x)
{
   return pow(x, log(b)/log(0.5));
}
```

Figure 2.19 shows the shape of the `bias` function for different choices of `b`.

Perlin and Hoffert (1989) present another function to remap the unit interval. This function is called `gain` and can be implemented as follows:

```
float
gain(float g, float x)
{
   if (x < 0.5)
     return bias(1-g, 2*x)/2;
   else
     return 1 - bias(1-g, 2 - 2*x)/2;
}
```

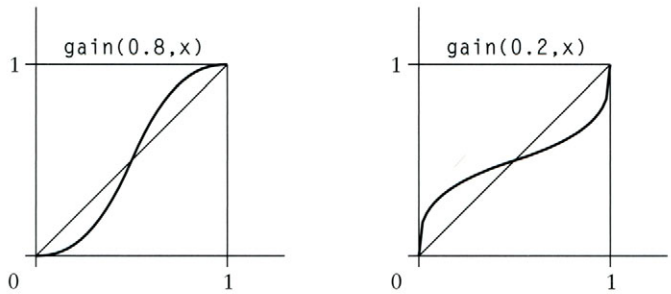

FIGURE 2.20 The `gain` function.

Regardless of the value of `g`, all `gain` functions return 0.5 when `x` is 0.5. Above and below 0.5, the `gain` function consists of two scaled-down `bias` curves forming an S-shaped curve. Figure 2.20 shows the shape of the `gain` function for different choices of `g`.

Schlick (1994) presents approximations to `bias` and `gain` that can be evaluated more quickly than the power functions given here.

Example: Brick Texture

One of the standard texture pattern clichés in computer graphics is the checkerboard pattern. This pattern was especially popular in a variety of early papers on antialiasing. Generating a checkerboard procedurally is quite easy. It is simply a matter of determining which square of the checkerboard contains the sample point and then testing the parity of the sum of the row and column to determine the color of that square.

This section presents a procedural texture generator for a simple brick pattern that is related to the checkerboard but is a bit more interesting. The pattern consists of rows of bricks in which alternate rows are offset by one-half the width of a brick. The bricks are separated by a mortar that has a different color than the bricks. Figure 2.21 is a diagram of the brick pattern.

The following is a listing of the shading language code for the brick shader, with explanatory remarks inserted here and there.

```
#define BRICKWIDTH          0.25
#define BRICKHEIGHT         0.08
#define MORTARTHICKNESS     0.01

#define BMWIDTH             (BRICKWIDTH+MORTARTHICKNESS)
#define BMHEIGHT            (BRICKHEIGHT+MORTARTHICKNESS)
```

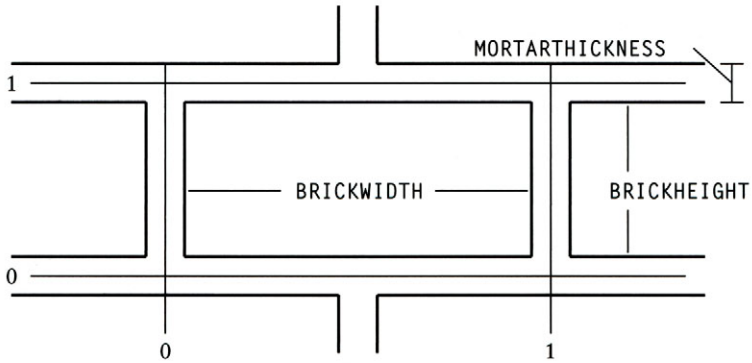

FIGURE 2.21 The geometry of a brick.

```
#define MWF                 (MORTARTHICKNESS*0.5/BMWIDTH)
#define MHF                 (MORTARTHICKNESS*0.5/BMHEIGHT)

surface brick(
    uniform float Ka = 1;
    uniform float Kd = 1;
    uniform color Cbrick = color (0.5, 0.15, 0.14);
    uniform color Cmortar = color (0.5, 0.5, 0.5);
    )
{
          color Ct;
          point Nf;
          float ss, tt, sbrick, tbrick, w, h;
          float scoord = s;
          float tcoord = t;

          Nf = normalize(faceforward(N, I));

          ss = scoord / BMWIDTH;
          tt = tcoord / BMHEIGHT;

          if (mod(tt*0.5,1) > 0.5)
          ss += 0.5; /* shift alternate rows */
```

The texture coordinates `scoord` and `tcoord` begin with the values of the standard texture coordinates `s` and `t`, and then are divided by the dimensions of a brick (including one-half of the mortar around the brick) to obtain new coordinates `ss` and `tt` that vary from 0 to 1 within a single brick. `scoord` and `tcoord` become the coordinates of the upper-left corner of the brick containing the point being shaded.

Alternate rows of bricks are offset by one-half brick width to simulate the usual way in which bricks are laid.

```
    sbrick = floor(ss); /* which brick? */
    tbrick = floor(tt); /* which brick? */
    ss -= sbrick;
    tt -= tbrick;
```

Having identified which brick contains the point being shaded, as well as the texture coordinates of the point within the brick, it remains to determine whether the point is in the brick proper or in the mortar between the bricks.

```
    w = step(MWF,ss) - step(1-MWF,ss);
    th = step(MHF,tt) - step(1-MHF,tt);

    Ct = mix(Cmortar, Cbrick, w*h);

    /* diffuse reflection model */
    Oi = Os;
    Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(Nf));
}
```

The rectangular brick shape results from two pulses (see page 28), a horizontal pulse `w` and a vertical pulse `h`. `w` is zero when the point is horizontally within the mortar region and rises to one when the point is horizontally within the brick region. `h` does the same thing vertically. When the two values are multiplied together, the result is the logical AND of `w` and `h`. That is, `w * h` is nonzero only when the point is within the brick region both horizontally and vertically. In this case, the `mix` function switches from the mortar color `Cmortar` to the brick color `Cbrick`.

The shader ends by using the texture color `Ct` in a simple diffuse shading model to shade the surface. Figure 2.22 is an image rendered with the brick texture from this example.

Bump-Mapped Brick

Now let's try our hand at some procedural bump mapping. Recall that bump mapping involves modifying the surface normal vectors to give the appearance that the surface has bumps or indentations. How is this actually done? We will examine two methods.

Blinn (1978), the paper that introduced bump mapping, describes how a bump of height $F(u, v)$ along the normal vector N can be simulated. The modified or "perturbed" normal vector is $N' = N + D$. The perturbation vector D lies in the tangent

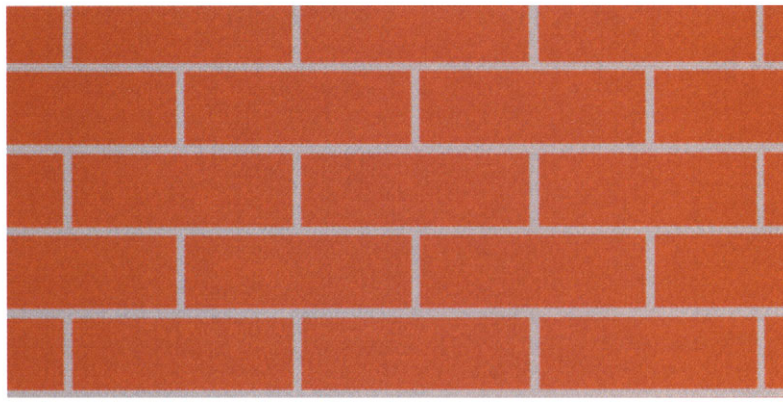

FIGURE 2.22 The brick texture.

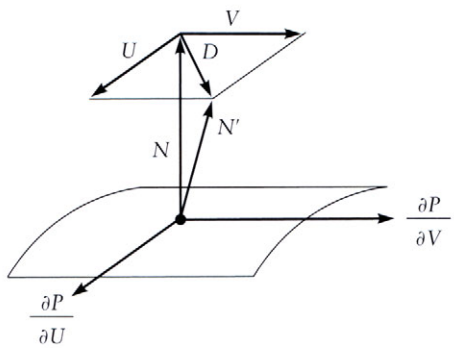

FIGURE 2.23 The geometry of bump mapping.

plane of the surface and is therefore perpendicular to N. D is based on the sum of two separate perturbation vectors U and V (Figure 2.23).

$$U = \frac{\partial F}{\partial u}\left(N \times \frac{\partial P}{\partial v}\right)$$

$$V = -\left(\frac{\partial F}{\partial v}\left(N \times \frac{\partial P}{\partial u}\right)\right)$$

$$D = \frac{1}{|N|}(U + V)$$

Let's analyze the expression for U. Note that the cross product

$$N \times \frac{\partial P}{\partial v}$$

is perpendicular to N and therefore lies in the tangent plane of the surface. It is also perpendicular to the partial derivative of P, the surface position, with respect to v. This derivative lies in the tangent plane and indicates the direction in which P changes as the surface parameter v is increased. If the parametric directions are perpendicular (usually they are only approximately perpendicular), adding a perturbation to N along the direction of this cross product would tilt N as if there were an upward slope in the surface along the u direction. The partial derivative $\partial F/\partial u$ gives the slope of the bump function in the u direction.

This technique looks somewhat frightening, but is fairly easy to implement if you already have the normal vector and the parametric derivatives of P. In the RenderMan shading language, `N`, `dPdu`, and `dPdv` contain these values. The differencing operators `Du` and `Dv` allow you to approximate the parametric derivatives of any expression. So Blinn's method of bump mapping could be implemented using the following shading language code:

```
float F; point U, V, D;
F = /* fill in some bump function here */
U = Du(F) * (N ^ dPdv);
V = -(Dv(F) * (N ^ dPdu));
D = 1/length(N) * (U + V);
Nf = N + D;
Nf = normalize(faceforward(Nf, I));
```

Then use `Nf` in the shading model just as you normally would. The resulting surface shading should give the appearance of a pattern of bumps determined by `F`.

Fortunately, the shading language provides a more easily remembered way to implement bump mapping and even displacement mapping.

```
float F;
point PP;

F = /* fill in some bump function here */

PP = P + F * normalize(N);
Nf = calculatenormal(PP);
Nf = normalize(faceforward(Nf, I));
```

In this code fragment, a new position `PP` is computed by moving along the direction of the normal a distance determined by the bump height `F`. Then the built-in

function `calculatenormal` is used to compute the normal vector of the modified surface `PP`. `calculatenormal(PP)` does nothing more than return the cross product of the parametric derivatives of the modified surface:

```
point
calculatenormal(point PP)
{
      return Du(PP) ^ Dv(PP);
}
```

To create actual geometric bumps by displacement mapping, you use very similar shading language code:

```
float F;

F = /* fill in some bump function here */

p = p + F * normalize(N);
N = calculatenormal(P);
```

Instead of creating a new variable `PP` that represents the bumped surface, this code assigns a new value to the original surface position `P`. In the shading language this means that the positions of points on the surface are actually moved by the shader to create bumps in the geometry. Similarly, the true normal vector `N` is recomputed so that it matches the displaced surface properly. We've omitted the last line that computes `Nf` because displacement mapping should be done in a separate displacement shader, not in the surface shader.[5]

To get a better understanding of bump mapping, let's add bump-mapped mortar grooves to our brick texture. The first step is to design the shape of the groove profile, that is, the vertical cross section of the bump function. Figure 2.24 is a diagram of the profile of the bricks and mortar grooves.

In order to realistically render the mortar groove between the bricks, we want the brick shader to compute a procedural bump-mapping function that will be used to adjust the normal vector before shading. To this end, we add the following code to the brick shader, immediately before the last statement (the one that computes `Ci` from the shading model).

5. Pixar's PhotoRealistic RenderMan renderer requires you to specify a *displacement bound* that tells the renderer what the maximum value of the bump height or other displacement will be. This is fully explained in the user's manual for the renderer.

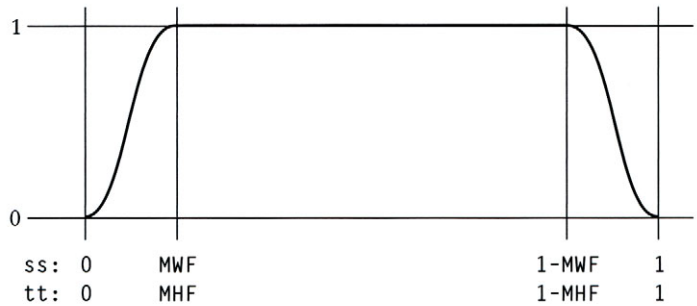

FIGURE 2.24 Brick and groove profile.

```
/* compute bump-mapping function for mortar grooves */
sbump = smoothstep(0,MWF,ss) - smoothstep(1-MWF,1,ss);
tbump = smoothstep(0,MHF,tt) - smoothstep(1-MHF,1,tt);
stbump = sbump * tbump;
```

The first two statements define the bump profile along the *s* and *t* directions independently. The first `smoothstep` call in each statement provides the positive slope of the bump function at the start of the brick, and the last `smoothstep` call in each statement provides the negative slope at the end of the brick. The last statement combines the `sbump` vertical groove and `tbump` horizontal groove to make an overall bump value `stbump`.

```
/* compute shading normal */
Nf = calculatenormal(P + normalize(N) * stbump);
Nf = normalize(faceforward(Nf, I));
Oi = Os;
Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(Nf));
```

Finally, the shading normal `Nf` is computed based on the bump height as described earlier in this section. The shader ends as before by using the texture color `Ct` and bump-mapped normal `Nf` in a diffuse shading model to shade the surface. Figure 2.25 is an image of the bump-mapped brick texture.

There is a subtle issue hidden in this example. Recall that the shader displaces the surface position by a bump height `stbump` along the normal vector. Since the built-in normal vector `N` was used without modification, the displacement is defined in the shader's `current` space, not in `shader` space. Even though the bump function itself is locked to the surface because it is defined in terms of the `s` and `t` surface texture coordinates, the height of the bumps could change if the object is scaled relative

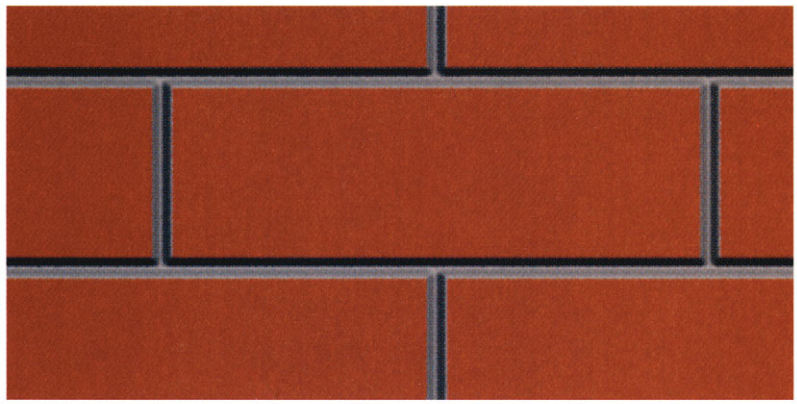

FIGURE 2.25 The bump-mapped brick texture.

to the `world` space. To avoid this problem, we could have transformed the surface point and normal vector into `shader` space, done the displacement there, and transformed the new normal back to `current` space, as follows:

```
point Nsh, Psh;
Psh = transform("shader", P);
Nsh = normalize(ntransform("shader", N));
Nsh = calculatenormal(Psh + Nsh * stbump);
Nf = ntransform("shader", "current", Nsh);
Nf = normalize(faceforward(Nf, I));
```

Note the use of `ntransform` rather than `transform` to transform normal vectors from one space to another. Normal vectors are transformed differently than points or direction vectors (see pages 216–217 of Foley et al. 1990). The second `ntransform` uses two space names to request a transformation from `shader` space to `current` space.

Example: Procedural Star Texture

Now let's try to generate a texture pattern that consists of a yellow five-pointed star on a background color `Cs`. The star pattern seems quite difficult until you think about it in polar coordinates. This is an example of how choosing the appropriate feature space makes it much easier to generate a tricky feature.

Figure 2.26 shows that each point of a five-pointed star is 72 degrees wide. Each half-point (36 degrees) is described by a single edge. The end points of the edge are a

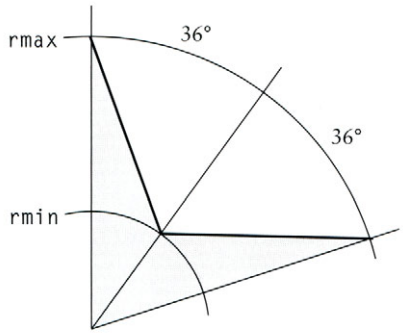

FIGURE 2.26 The geometry of a star.

point at radius `rmin` from the center of the star and another point at radius `rmax` from the center of the star.

```
surface
star(
      uniform float Ka = 1;
      uniform float Kd = 1;
      uniform color starcolor = color (1.0000,0.5161,0.0000);
      uniform float npoints = 5;
      uniform float sctr = 0.5;
      uniform float tctr = 0.5;
    )
{
      point Nf = normalize(faceforward(N, I));
      color Ct;
      float ss, tt, angle, r, a, in_out;
      uniform float rmin = 0.07, rmax = 0.2;
      uniform float starangle = 2*PI/npoints;
      uniform point p0 = rmax*(cos(0),sin(0), 0);
      uniform point pi = rmin*
          (cos(starangle/2),sin(starangle/2),0);
      uniform point d0 = pi - p0; point d1;
      ss = s - sctr; tt = t - tctr;
      angle = atan(ss, tt) + PI;
      r = sqrt(ss*ss + tt*tt);
```

At this point, the shader has computed polar coordinates relative to the center of the star. These coordinates `r` and `angle` act as the feature space for the star.

```
    a = mod(angle, starangle)/starangle;
    if (a >= 0.5)
          a = 1 - a;
```

Now the shader has computed the coordinates of the sample point `(r,a)` in a new feature space: the space of one point of the star. `a` is first set to range from 0 to 1 over each star point. To avoid checking both of the edges that define the "V" shape of the star point, sample points in the upper half of the star point are reflected through the center line of the star point. The new sample point `(r,a)` is inside the star if and only if the original sample point was inside the star, due to the symmetry of the star point around its center line.

```
    dl = r*(cos(a), sin(a),0) - p0;
    in_out = step(0, zcomp(d0^d1) );
    Ct = mix(Cs, starcolor, in_out);
    /* diffuse ("matte") shading model */
    Oi = Os;
    Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(Nf));
}
```

To test whether `(r,a)` is inside the star, the shader finds the vectors `d0` from the tip of the star point to the `rmin` vertex and `d1` from the tip of the star point to the sample point. Now we use a handy trick from vector algebra. The cross product of two vectors is perpendicular to the plane containing the vectors, but there are two directions in which it could point. If the plane of the two vectors is the (x, y) plane, the cross product will point along the positive z-axis or along the negative z-axis. The direction in which it points is determined by whether the first vector is to the left or to the right of the second vector. So we can use the direction of the cross product to decide which side of the star edge `d0` the sample point is on.

Since the vectors `d0` and `d1` have z components of zero, the cross product will have x and y components of zero. Therefore, the shader can simply test the sign of `zcomp(d0^d1)`. We use `step(0,zcomp(d0^d1))` instead of `sign(zcomp(d0^d1))` because the `sign` function returns -1, 0, or 1. We want a binary (0 or 1) answer to the query "Is the sample point inside or outside the star?" This binary answer, `in_out`, is used to select the texture color `Ct` using the `mix` function, and the texture color is used to shade the sample point according to the diffuse shading model.

Figure 2.27 is an image rendered using the `star` shader.

Spectral Synthesis

Gardner (1984, 1985) demonstrated that procedural methods could generate remarkably complex and natural-looking textures simply by using a combination of sinusoidal component functions of differing frequencies, amplitudes, and phases. The theory of Fourier analysis tells us that functions can be represented as a sum

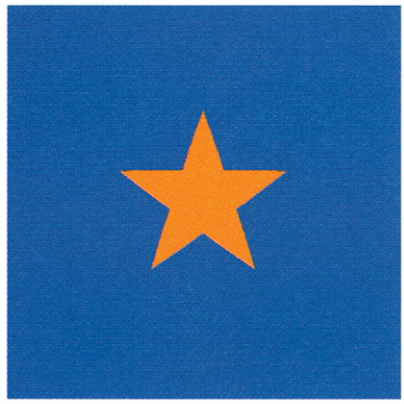

FIGURE 2.27 The star texture pattern.

of sinusoidal terms. The Fourier transform takes a function from the temporal or spatial domain, where it is usually defined, into the *frequency domain,* where it is represented by the amplitude and phase of a series of sinusoidal waves (Bracewell 1986; Brigham 1988). When the series of sinusoidal waves is summed together, it reproduces the original function; this is called the *inverse Fourier transform.*

Spectral synthesis is a rather inefficient implementation of the inverse discrete Fourier transform, which takes a function from the frequency domain back to the spatial domain. Given the amplitude and phase for each sinusoidal component, we can sum up the waves to get the desired function. The efficient way to do this is the inverse fast Fourier transform (FFT) algorithm, but that method generates the inverse Fourier transform for a large set of points all at once. In an implicit procedural texture we have to generate the inverse Fourier transform for a single sample point, and the best way to do that seems to be a direct summation of the sine wave components.

In procedural texture generation, we usually don't have all of the frequency domain information needed to reconstruct some function exactly. Instead, we want a function with some known characteristics, usually its power spectrum, and we don't care too much about the details of its behavior. It is possible to take a scanned image of a texture, compute its frequency domain representation using a fast Fourier transform, and use the results to determine coefficients for a spectral synthesis procedural texture, but in our experience that approach is rarely taken.

One of Gardner's simplest examples is a 2D texture that can be applied to a flat sky plane to simulate clouds. Here is a RenderMan shader that generates such a texture:

```
#define NTERMS 5
surface cloudplane(
    color cloudcolor = color (1,1,1);
    )
{
      color Ct;
      point Psh;
      float i, amplitude, f;
      float x, fx, xfreq, xphase;
      float y, fy, yfreq, yphase;
      uniform float offset = 0.5;
      uniform float xoffset = 13;
      uniform float yoffset = 96;

      Psh = transform("shader", P);
      x = xcomp(Psh) + xoffset;
      y = ycomp(Psh) + yoffset;

      xphase = 0.9; /* arbitrary */
      yphase = 0.7; /* arbitrary */
      xfreq = 2 * PI * 0.023;
      yfreq = 2 * PI * 0.021;
      amplitude = 0.3;
      f = 0;
      for (i = 0; i < NTERMS; i += 1) {
            fx = amplitude *
                  (offset + cos(xfreq * (x + xphase)));
            fy = amplitude *
                  (offset + cos(yfreq * (y + yphase)));
            f += fx * fy;
            xphase = PI/2 * 0.9 * cos (yfreq * y);
            yphase = PI/2 * 1.1 * cos (xfreq * x);

            xfreq *= 1.9 + i * 0.1; /* approximately 2 */
            yfreq *= 2.2 - i * 0.08; /* approximately 2 */
            amplitude *= 0.707;
      }
      f = clamp(f, 0, 1);

      Ct = mix(Cs, cloudcolor, f);
      Oi = Os;
      Ci = Os * Ct;
}
```

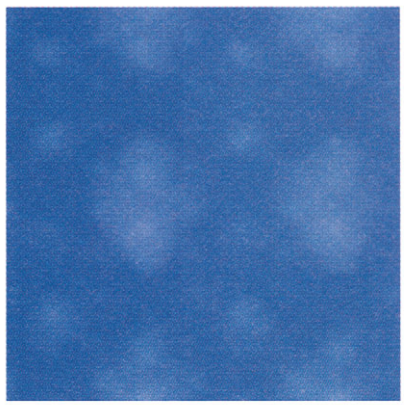

FIGURE 2.28 The cloud plane texture pattern.

This texture is a sum of five components, each of which is a cosine function with a different frequency, amplitude, and phase. The frequencies, amplitudes, and phases are chosen according to rules discovered by Gardner in his experiments. Gardner's technique is somewhat unusual for spectral synthesis in that the phase of each component is coupled to the value of the previous component in the other coordinate (for example, the x phase depends on the value of the preceding y component).

Making an acceptable cloud texture in this way is a battle to avoid regular patterns in the texture. Natural textures usually don't have periodic patterns that repeat exactly. Spectral synthesis relies on complexity to hide its underlying regularity and periodicity. There are several "magic numbers" strewn throughout this shader in an attempt to prevent regular patterns from appearing in the texture. Fourier spectral synthesis using a finite number of sine waves will always generate a periodic function, but the period can be made quite long so that the periodicity is not obvious to the observer. Figure 2.28 is an image rendered using the `cloudplane` shader with `Cs` set to a sky-blue color.

What Now?

You could go a long way using just the methods described so far. Some of these techniques can produce rich textures with a lot of varied detail, but even more variety is possible. In particular, we haven't yet discussed the noise function, the most popular of all procedural texture primitives. But first, let's digress a bit and examine one of the most important issues that affect procedural textures, namely, the difficulties of aliasing and antialiasing.

ALIASING AND HOW TO PREVENT IT

Aliasing is a term from the field of signal processing. In computer graphics, aliasing refers to a variety of image flaws and unpleasant artifacts that result from improper use of sampling. The staircaselike "jaggies" that can appear on slanted lines and edges are the most obvious examples of aliasing. The next section presents an informal discussion of basic signal processing concepts, including aliasing. For a more rigorous presentation, see Oppenheim and Schafer (1989), a standard signal processing textbook.

Signal Processing

As shown in Figure 2.29, a continuous signal can be converted into a discrete form by measuring its value at equally spaced sample points. This is called *sampling*. The original signal can be reconstructed later from the sample values by interpolation.

Sampling and reconstruction are fundamental to computer graphics.[6] Raster images are discrete digital representations of the continuous optical signals that nature delivers to our eyes and to our cameras. Synthetic images are made by sampling of geometric models that are mathematically continuous. Of course, our image signals are two-dimensional. Signal processing originally was developed to deal with the one-dimensional time-varying signals encountered in communications. The field of image processing is in essence the two-dimensional extension of signal processing techniques to deal with images.

Fortunately for computer graphics, the process of sampling and reconstruction is guaranteed to work under certain conditions, namely, when the amount of information in the original signal does not exceed the amount of information that can be captured by the samples. This is known as the *sampling theorem*. The amount of information in the original signal is called its *bandwidth*. The amount of information that can be captured by the samples is dependent upon the *sampling rate*, the number of sample points per unit distance. Unfortunately for computer graphics, the conditions for correct sampling and reconstruction are not always easy to meet, and when they are not met, aliasing occurs.

The theory of Fourier analysis tells us that a function can be represented as a sum of sinusoidal components with various frequencies, phases, and amplitudes. The Fourier transform converts the function from its original form in the "spatial

6. Mitchell and Netravali (1988) includes an excellent discussion of the many places in which sampling and reconstruction arise in computer graphics.

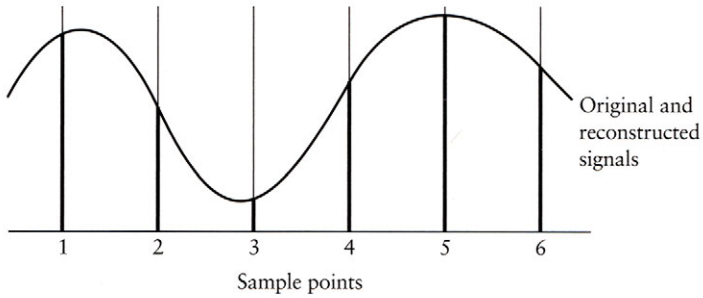

FIGURE 2.29 Sampling and reconstruction.

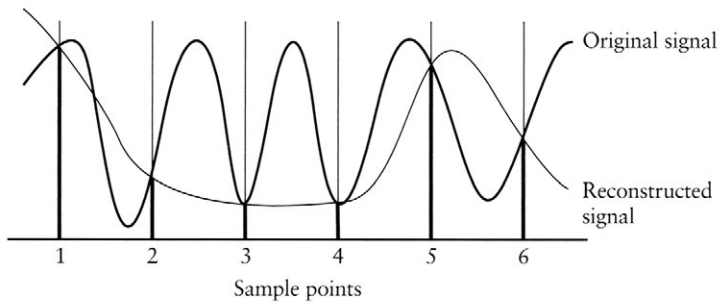

FIGURE 2.30 Aliasing.

domain" into a set of sinusoidal components in the "frequency domain." A signal with limited bandwidth will have a maximum frequency in its frequency domain representation. If that frequency is less than or equal to one-half of the sampling rate, the signal can be correctly sampled and reconstructed without aliasing. Aliasing will occur if the maximum frequency exceeds one-half of the sampling rate (this is called the *Nyquist frequency*). The maximum frequency in the reconstructed signal cannot exceed the Nyquist frequency, but the energy contributed to the original signal by the excessively high frequency components does not simply disappear. Instead, it appears in the reconstructed signal as erroneous lower-frequency energy, which is called an *alias* of the high-frequency energy in the original signal. Figure 2.30 illustrates this situation. The original signal varies too often to be adequately captured by the samples. Note that the signal reconstructed from the samples is quite different from the original signal. The problem of aliasing can be addressed by changing the sample points to be closer together, or by modifying the original signal to eliminate the high frequencies. If it is possible to increase the sampling rate, that is always beneficial. With more samples, the original signal can be reconstructed more accurately.

The Nyquist frequency threshold at which aliasing begins is increased, so the frequencies in the signal might now be below the Nyquist frequency.

Unfortunately, there is always some practical limit on the resolution of an image due to memory space or display limitations, and the sampling rate of an image is proportional to its resolution. It is impossible for an image to show details that are too small to be visible at the resolution of the image. Therefore, it is vital to take excessively high frequencies out of the original signal so that they don't show up as aliases and detract from the part of the signal that *can* be seen given the available resolution.

There is another reason why increasing the sampling rate is never a complete solution to the problem of aliasing. Some signals have unlimited bandwidth, so there is no maximum frequency. Sharp changes in the signal, for example, a `step` function, have frequency components of arbitrarily high frequency. No matter how great the image resolution, increasing the sampling rate to any finite value cannot eliminate aliasing when sampling such signals. This is why sloped lines are jaggy on even the highest resolution displays (unless they have been antialiased properly). Resolution increases alone can make the jaggies smaller, but never can eliminate them.

Since aliasing can't always be solved by increasing the sampling rate, we are forced to figure out how to remove high frequencies from the signal before sampling. The technique is called *low-pass filtering* because it passes low-frequency information while eliminating higher-frequency information.[7] The visual effect of low-pass filtering is to blur the image. The challenge is to blur the image as little as possible while adequately attenuating the unwanted high frequencies.

It is often difficult to low-pass-filter the signal before sampling. A common strategy in computer graphics is to *supersample* or *oversample* the signal, that is, to sample it at a higher rate than the desired output sampling rate. For example, we might choose to sample the signal four times for every output sample. If the signal were reconstructed from these samples, its maximum possible frequency would be four times the Nyquist frequency of the output sampling rate. A discrete low-pass filter can be applied to the "supersamples" to attenuate frequencies that exceed the Nyquist frequency of the output sampling rate. This method alleviates aliasing from frequencies between the output Nyquist frequency and the Nyquist frequency of the supersamples. Unfortunately, frequencies higher than the Nyquist frequency of the supersamples will still appear as aliases in the reconstructed signal.

7. In practice, effective antialiasing often requires a lower-frequency filtering criterion than the Nyquist frequency because the filtering is imperfect and the reconstruction of the signal from its samples is also imperfect.

An alternative approach to antialiasing is to supersample the signal at irregularly spaced points. This is called *stochastic sampling* (Cook, Porter, and Carpenter 1984; Cook 1986; Lee, Redner, and Uselton 1985; Dippé and Wold 1985). The energy from frequencies above the Nyquist frequency of the supersamples appears in the reconstructed signal as random noise rather than as a structured low-frequency alias. People are far less likely to notice this noise in the rendered image than they are to notice a low-frequency alias pattern. But it is preferable to low-pass-filter the signal before sampling because in that case no noise will be added to the reconstructed signal.

In summary, to produce an antialiased image with a specified resolution, the most effective strategy is to remove excessively high frequencies from the signal by low-pass filtering before sampling. If it isn't possible to filter the signal, the best strategy is to stochastically supersample it at as high a rate as is practical, and apply a discrete low-pass filter to the supersamples. The next section discusses ways to build low-pass filtering into procedural textures to eliminate aliasing artifacts.

You might wonder why aliasing is a problem in procedural textures. Doesn't the renderer do antialiasing? In fact, most renderers have some antialiasing scheme to prevent aliasing artifacts that result from sharp edges in the geometric model. Renderers that support image textures usually include some texture antialiasing in the texture mapping software. But these forms of antialiasing do not solve the aliasing problem for procedural textures.

The best case for "automatic" antialiasing of procedural textures is probably a stochastic ray tracer or, in fact, any renderer that stochastically supersamples the procedural texture. Rendering in this way is likely to be slow because of the many shading samples that are required by the supersampling process. And in the end, stochastic supersampling can only convert aliases into noise, not eliminate the unwanted high frequencies completely. If we can build a better form of antialiasing into the procedural texture, the result will look cleaner and the renderer can be freed of the need to compute expensive supersamples of the procedural texture.

PhotoRealistic RenderMan performs antialiasing by stochastically sampling the scene geometry and filtering the results of the sampling process (Cook, Carpenter, and Catmull 1987). The geometry is converted into a mesh of tiny quadrilaterals, and shading samples are computed at each vertex of the mesh before the stochastic sampling takes place. The vertices of the mesh are a set of samples of the location of the surface at equally spaced values of the surface parameters (u, v). Many shaders can be viewed as signal-generating functions defined on (u, v). A shader is evaluated at the mesh vertices, and the resulting colors and other properties are assigned to the mesh. This is really a sampling of the shader function at a grid of (u, v) values and its

reconstruction as a colored mesh of quadrilaterals. If the frequency content of the shader exceeds the Nyquist frequency of the mesh vertex (u, v) sampling rate, aliases will appear in the mesh colors. The reconstructed mesh color function is resampled by the stochastic sampling of the scene geometry. But once aliases have been introduced during the shader sampling process, they can never be removed by subsequent pixel sampling and filtering.

The separation of shader sampling from pixel sampling in PhotoRealistic RenderMan is advantageous because it permits a lower sampling rate for the shader samples than for the pixel samples. Shader samples are usually much more expensive to evaluate than pixel samples, so it makes sense to evaluate fewer of them. But this increases the need to perform some type of antialiasing in the shader itself; we can't rely on the stochastic supersampling of pixel samples to alleviate aliasing in procedural textures.

When image textures are used in the RenderMan shading language, the antialiasing is automatic. The texture system in the renderer filters the texture image pixels as necessary to attenuate frequencies higher than the Nyquist frequency of the (u, v) sampling rate in order to avoid aliasing.[8]

The brick texture from earlier in the chapter provides a concrete example of the aliasing problem. Figure 2.31 shows how the brick texture looks when the sampling rate is low. Notice that the width of the mortar grooves appears to vary in different parts of the image due to aliasing. This is the original version of the texture, without bump-mapped grooves. Later in the chapter we'll see how to add antialiasing techniques to the brick texture to alleviate the aliases.

Methods of Antialiasing Procedural Textures

By now you should be convinced that some form of antialiasing is necessary in procedural texture functions. In the remainder of this section we'll examine various ways to build low-pass filtering into procedural textures: clamping, analytic prefiltering, integrals, and alternative antialiasing methods. Clamping is a special-purpose filtering method that applies only to textures created by spectral synthesis. Analytic prefiltering techniques are ways to compute low-pass-filtered values for some of the primitive functions that are used to build procedural textures. One class of analytic prefiltering methods is based on the ability to compute the integral of the texture

8. See Feibush, Levoy, and Cook (1980), Williams (1983), Crow (1984), and Heckbert (1986a, 1986b) for detailed descriptions of methods for antialiasing image textures.

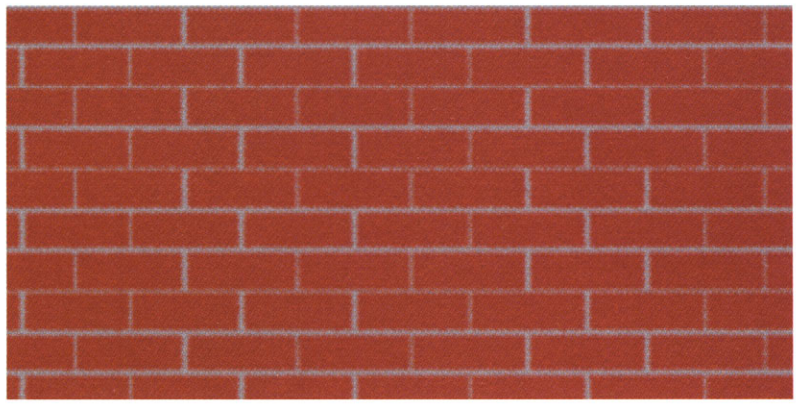

FIGURE 2.31 Aliasing in the brick texture.

function over a rectangular region. Finally, we'll consider alternatives to low-pass filtering that can be used when proper filtering is not practical.

Some procedural texture primitives are inherently band-limited; that is, they contain only a limited, bounded set of frequencies. `sin` is an obvious example of such a function. The `texture` function and its relatives have built-in filtering, so they are also band-limited. Unfortunately, some common language constructs such as `if` and `step` create sharp changes in value that generate arbitrarily high frequencies. Sharp changes in the shading function must be avoided. `smoothstep` is a smoothed replacement for `step` that can reduce the tendency to alias. Can we simply replace `step` functions with `smoothstep` functions?

The `smoothstep` function has less high-frequency energy than `step`, but using a particular `smoothstep` as a fixed replacement for `step` is not an adequate solution. If the shader is tuned for a particular view, the `smoothstep` will alias when the texture is viewed from further away because the fixed-width `smoothstep` will be too sharp. On the other hand, when the texture is viewed from close up, the `smoothstep` edge is too blurry. A properly antialiased edge should look equally sharp at all scales. To achieve this effect, the `smoothstep` width must be varied based on the sampling rate.

Determining the Sampling Rate

To do low-pass filtering properly, the procedural texture function must know the sampling rate at which the renderer is sampling the texture. The sampling rate is just the reciprocal of the spacing between adjacent samples in the relevant texture space

or feature space. This is called the *sampling interval.* For simple box filtering, the sampling interval is also the usual choice for the width of the box filter.

Obviously, the sampling interval cannot be determined from a single sample in isolation. Earlier parts of this chapter have presented a model of procedural texture in which the implicit texture function simply answers queries about the surface properties at a single sample point. The procedural texture is invoked many times by the renderer to evaluate the texture at different sample points, but each invocation is independent of all of the others.

To determine the sampling rate or sampling interval without changing this model of procedural texture, the renderer must provide some extra information to each invocation of the procedural texture. In the RenderMan shading language, this information is in the form of built-in variables called `du` and `dv` and functions called `Du` and `Dv`. The `du` and `dv` variables give the sampling intervals for the surface parameters (u, v). If the texture is written in terms of (u, v), the filter widths can be taken directly from `du` and `dv`.

In most cases, procedural textures are written in terms of the standard texture coordinates (s, t), which are scaled and translated versions of (u, v), or in terms of texture coordinates computed from the 3D coordinates of the surface point P in some space. In these cases, it is harder to determine the sampling interval, and the functions `Du` and `Dv` must be used. `Du(a)` gives an approximation to the derivative of some computed quantity `a` with respect to the surface parameter u. Similarly, `Dv(a)` gives an approximation to the derivative of some computed quantity `a` with respect to the surface parameter v. By multiplying the derivatives by the (u, v) sampling intervals, the procedural texture can estimate the sampling interval for a particular computed texture coordinate `a`. In general, it is not safe to assume that the texture coordinate changes only when u changes or only when v changes. Changes along both parametric directions have to be considered and combined to get a good estimate, `awidth`, of the sampling interval for `a`:

```
awidth = abs(Du(a)*du) + abs(Dv(a)*dv);
```

The sum of the absolute values gives an upper bound on the sampling interval; if this estimate is in error, it tends to make the filter too wide so that the result is blurred too much. This is safer than making the filter too narrow, which would allow aliasing to occur.

It is desirable for the sampling interval estimate to remain constant or change smoothly. Sudden changes in the sampling interval result in sudden changes in the texture filtering, and that can be a noticeable and annoying flaw in itself. Even if the

derivatives `Du` and `Dv` are accurate and change smoothly, there is no guarantee that the renderer's sampling intervals in (u, v) will also behave themselves. Many renderers use some form of adaptive sampling or adaptive subdivision to vary the rate of sampling depending on the apparent amount of detail in the image. In PhotoRealistic RenderMan, adaptive subdivision changes the shader sampling intervals depending on the size of the surface in the image. A surface seen in perspective could have sudden changes in sampling intervals between the nearer and more distant parts of the surface. A renderer that uses adaptive sampling based on some estimate of apparent detail might end up using the values returned by the procedural texture itself to determine the appropriate sampling rates. That would be an interesting situation indeed—one that might make proper low-pass filtering in the texture a very difficult task.

The remedy for cases in which the renderer's sampling interval is varying in an undesirable way is to use some other estimate of the sampling interval, an estimate that is both less accurate and smoother than the one described above. One such trick is to use the distance between the camera and the surface position to control the low-pass filtering:

```
awidth = length(I) * k;
```

The filter width (sampling interval estimate) is proportional to the distance from the camera (`length(I)`), but some experimentation is needed to get the right scaling factor `k`.

It is especially tricky to find the right filter width to antialias a bump height function for a bump-mapping texture. Since the bump height affects the normal vector used in shading, specular highlights can appear on the edges of bumps. Specular reflection functions have quite sharp angular falloff, and this sharpness can add additional high frequencies to the color output of the shader that are not in the bump height function. It might not be sufficient to filter the bump height function using the same low-pass filter that would be used for an ordinary texture that changes only the color or opacity. A wider filter probably is needed, but determining just how much wider it should be is a black art.

Clamping

Clamping (Norton, Rockwood, and Skolmoski 1982) is a very direct method of eliminating high frequencies from texture patterns that are generated by spectral synthesis. Since each frequency component is explicitly added to a spectral synthesis

texture, it is fairly easy to omit every component whose frequency is greater than the Nyquist frequency.

Let's begin with the following simple spectral synthesis loop, with a texture coordinate `s`:

```
value = 0;
for (f = MINFREQ; f < MAXFREQ; f *= 2)
      value += sin(2*PI*f*s)/f;
```

The loop begins at a frequency of `MINFREQ` and ends at a frequency less than `MAXFREQ`, doubling the frequency on each successive iteration of the loop. The amplitude of each sinusoidal component is the reciprocal of its frequency.

The following version is antialiased using the simplest form of clamping. The sampling interval in `s` is `swidth`.

```
value = 0;
cutoff = clamp(0.5/swidth, 0, MAXFREQ);
for (f = MINFREQ; f < cutoff; f *= 2)
      value += sin(2*PI*f*s)/f;
```

In this version the loop stops at a frequency less than `cutoff`, which is the Nyquist frequency for the sampling rate `1/swidth`. In order to avoid "pops," sudden changes in the texture as the sampling rate changes (e.g., as we zoom in toward the textured surface), it is important to fade out each component gradually as the Nyquist frequency approaches the component frequency. The following texture function incorporates this gradual fade-out strategy:

```
value = 0;
cutoff = clamp(0.5/swidth, 0, MAXFREQ);
for (f = MINFREQ; f < 0.5*cutoff; f *= 2)
      value += sin(2*PI*f*s)/f;
fade = clamp(2*(cutoff-f)/cutoff, 0, 1);
value += fade * sin(2*PI*f*s)/f;
```

The loop ends one component earlier than before, and that last component (whose frequency is between `0.5*cutoff` and `cutoff`) is added in after the loop and is scaled by `fade`. The `fade` value gradually drops from 1 to 0 as the frequency of the component increases from `0.5*cutoff` toward `cutoff`. This is really a result of changes in `swidth` and therefore in `cutoff`, rather than changes in the set of frequency components in the texture pattern.

Note that the time to generate the spectral synthesis texture pattern will increase as the sampling rate increases, that is, as we look more closely at the texture pattern. More and more iterations of the synthesis loop will be executed as the camera

approaches the textured surface. The example code incorporates MAXFREQ as a safety measure, but if MAXFREQ is reached, the texture will begin to look ragged when viewed even more closely.

Clamping works very well for spectral synthesis textures created with sine waves. It is hard to imagine a clearer and more effective implementation of low-pass filtering! But when the spectral synthesis uses some primitive that has a richer frequency spectrum of its own, clamping doesn't work as well.

If the primitive contains frequencies higher than its nominal frequency, the low-pass filtering will be imperfect and some high-frequency energy will leak into the texture. This can cause aliasing.

Even if the primitive is perfectly band-limited to frequencies lower than its nominal frequency, clamping is imperfect as a means of antialiasing. In this case, clamping will eliminate aliasing, but the character of the texture may change as high frequencies are removed because each component contains low frequencies that are removed along with the high frequencies.

Analytic Prefiltering

A procedural texture can be filtered explicitly by computing the convolution of the texture function with a filter function. This is difficult in general, but if we choose a simple filter, the technique can be implemented successfully. The simplest filter of all is the box filter; the value of a box filter is simply the average of the input function value over the area of the box filter.

To compute the convolution of a function with a box filter the function must be integrated over the area under the filter. This sounds tough, but it's easy if the function is simple enough. Consider the step function shown in Figure 2.8. The step function is rather ill-behaved because it is discontinuous at its threshold value. Let's apply a box filter extending from x to x + w to the function step(b,x). The result is the box-filtered step function, boxstep(a,b,x), where a = b - w (Figure 2.32). The value of boxstep is the area under the step function within the filter box. When the entire filter is left of b (that is, x ≥ b), the value is 0. When the entire filter is right of b (that is, x > b), the value is 1. But boxstep is "smoother" than the step function; instead of being a sharp, discontinuous transition from 0 to 1 at b, boxstep is a linear ramp from 0 to 1 starting at a and ending at b. The slope of the ramp is 1/w.

The boxstep function can be written as a preprocessor macro in C or the shading language as follows:

```
#define boxstep(a,b,x) clamp(((x)-(a))/((b)-(a)),0,1)
```

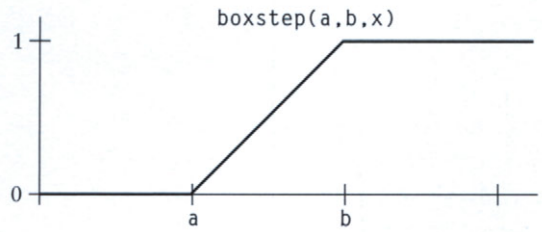

FIGURE 2.32 Box-filtering the `step` function.

Now the `step(b,x)` can be replaced with `boxstep(b-w,b,x)`. If the filter width `w` is chosen correctly, the `boxstep` function should reduce aliasing compared to the `step` function.

Better Filters

Now we know how to generate a box-filtered version of the `step` function, but the box filter is far from ideal for antialiasing. A better filter usually results in fewer artifacts or less unnecessary blurring. A better alternative to `boxstep` is the `smoothstep` function that was discussed earlier in this chapter. Filtering of the step with a first-order filter (box) gives a second-order function, namely, the linear ramp. Filtering of the step with a third-order filter (quadratic) gives a fourth-order function, namely, the cubic `smoothstep`. Using `smoothstep` to replace `step` is like filtering with a quadratic filter, which is a better approximation to the ideal sinc filter than the box filter is.

The `boxstep` macro is designed to be plug-compatible with `smoothstep`. The call `boxstep(WHERE-swidth, WHERE, s)` can be replaced with the call `smoothstep(WHERE-swidth, WHERE, s)`. This is the filtered version of `step(WHERE, s)`, given a filter extending from `s` to `s+swidth`.

Using the `smoothstep` cubic function as a filtered step is convenient and efficient because it is a standard part of the shading language. However, there are other filters and other filtered steps that are preferable in many applications. In particular, some filters such as the sinc and Catmull-Rom filters have *negative lobes*—the filter values dip below zero at some points. Such filters generally produce sharper texture patterns, although ringing artifacts are sometimes visible. A Catmull-Rom filter can be convolved with a step function (which is equivalent to integrating the filter function) to produce a `catstep` filtered step function that has been used with good results (Sayre 1992).

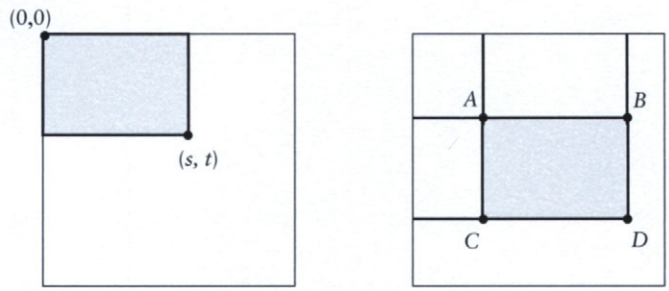

FIGURE 2.33 The summed-area table: (a) table entry (s, t) stores area of shaded region; (b) four entries A, B, C, D are used to compute shaded area.

Integrals and Summed-Area Tables

Crow (1984) introduced the *summed-area table* method of antialiasing image textures. A summed-area table is an image made from the texture image. As illustrated in Figure 2.33(a), the pixel value at coordinates (s, t) in the summed-area table is the sum of all of the pixels in the rectangular area $(0{:}s, 0{:}t)$ in the texture image. Of course, the summed-area table might need higher-precision pixel values than those of the original texture to store the sums accurately.

The summed-area table makes it easy to compute the sum of all of the texture image pixels in any axis-aligned rectangular region. Figure 2.33(b) shows how this is done. The pixel values at the corners of the region A, B, C, D are obtained from the summed-area table (four pixel accesses). The sum over the desired region is then simply $D + A - B - C$. This sum divided by the area of the region is the average value of the texture image over the region. If the region corresponds to the size and position of a box filter in the (s, t) space, the average value from the summed-area table calculation can be used as an antialiased texture value. The cost of the antialiasing is constant regardless of the size of the region covered by the filter, which is very desirable.

The summed-area table is really a table of the integral of the original texture image over various regions of the (s, t) space. An analogous antialiasing method for procedural textures is to compute the definite integral of the procedural texture over some range of texture coordinate values, rather than computing the texture value itself. For example, a procedural texture function $f(x)$ on some texture coordinate x might have a known indefinite integral function $F(x)$. If the desired box filter width is

w_x, the expression $(F(x) - F(x - w_x))/w_x$ might be used as a filtered alternative to the texture value $f(x)$. Integrals for many of the basic building blocks of procedural textures are easy to compute, but a few are tricky.[9]

Example: Antialiased Brick Texture

As an example of the application of these techniques, let's build antialiasing into the brick texture described earlier in this chapter.

The first step is to add the code needed to determine the filter width. The width variables must be added to the list of local variable declarations:

```
float swidth, twidth;
```

To compute the filter widths, we can add two lines of code just before the two lines that compute the brick numbers `sbrick` and `tbrick`:

```
swidth = abs(Du(ss)*du) + abs(Dv(ss)*dv);
twidth = abs(Du(tt)*du) + abs(Dv(tt)*dv);
sbrick = floor(ss); /* which brick? */
tbrick = floor(tt); /* which brick? */
```

The actual antialiasing is done by replacing the following two lines of the original shader that determine where to change from mortar color to brick color:

```
w = step(MWF,ss) - step(1-MWF,ss);
h = step(MHF,tt) - step(1-MHF,tt);
```

with an antialiased version of the code:

```
w = boxstep(MWF-swidth,MWF,ss)
  - boxstep(1-MWF-swidth,1-MWF,ss);
h = boxstep(MHF-twidth,MHF,tt)
  - boxstep(1-MHF-twidth,1-MHF,tt);
```

This is just the same code using `boxstep` instead of `step`. If the texture pattern consisted of a single brick in an infinite field of mortar, this would be sufficient. Unfortunately, more is required in order to handle a periodic pattern like the brick texture. The brick texture depends on a `mod`-like folding of the texture coordinates to convert a single pulse into a periodic sequence of pulses. But a wide filter positioned

9. The noise functions described in the later section "Making Noises" are among the tricky ones to integrate.

inside one brick can overlap another brick, a situation that is not properly accounted for in this periodic pulse scheme.

To solve the aliasing problem in a more general way, we can apply the integration technique described in the previous section. The integral of a sequence of square wave pulses is a function that consists of upward-sloping ramps and plateaus. The ramps correspond to the intervals where the pulses have a value of 1, and the plateaus correspond to the intervals where the pulses have a value of 0. In other words the slope of the integral is either 0 or 1, depending on the pulse value. The slope is the derivative of the integral, which is obviously the same as the original function.

The integrals of the periodic pulse functions in the `ss` and `tt` directions are given by the following preprocessor macros:

```
#define frac(x)         mod((x),1)
#define sintegral(ss)   (floor(ss)*(1-2*MWF) + \
                        max(0,frac(ss)-MWF))
#define tintegral(tt)   (floor(tt)*(1-2*MHF) + \
                        max(0,frac(tt)-MHF))
```

These are definite integrals from 0 to `ss` and 0 to `tt`. The `ss` integral consists of the integral of all of the preceding complete pulses (the term involving the floor function) plus the contribution of the current partial pulse (the term involving the fractional part of the coordinate).

To compute the antialiased value of the periodic pulse function, the shader must determine the value of the definite integral over the area of the filter. The value of the integral is divided by the area of the filter to get the average value of the periodic pulse function in the filtered region.

```
w = (sintegral(ss+swidth) - sintegral(ss))/swidth;
h = (tintegral(tt+twidth) - tintegral(tt))/twidth;
```

When using this method of antialiasing, you should remove the following lines of code from the shader:

```
ss -= sbrick;
tt -= tbrick;
```

because the `floor` and `mod` operations in the integrals provide the necessary periodicity for the pulse sequence. Forcing `ss` and `tt` to lie in the unit interval interferes with the calculation of the correct integral values.

Figure 2.34 shows the antialiased version of the brick texture, which should be compared with the original version shown in Figure 2.31. The widths of the mortar grooves are more consistent in the antialiased version of the texture.

Alternative Antialiasing Methods

Building low-pass filtering into a complicated procedural texture function can be far from easy. In some cases you might be forced to abandon the worthy goal of "proper" filtering and fall back on some alternative strategy that is more practical to implement.

One simple alternative to low-pass filtering is simply to blend between two or more versions of the texture based on some criterion related to sampling rate. For example, as the sampling rate indicates that the samples are getting close to the rate at which the texture begins to alias, you can fade your texture toward a color that is the average color of the texture. This is clearly a hack; the transition between the detailed texture and the average color might be quite obvious, although this is probably better than just letting the texture alias. The transition can be smoothed out by using more than two representations of the texture and blending between adjacent pairs of textures at the appropriate sampling rates.

A more sophisticated antialiasing method is to supersample the texture pattern in the procedural texture itself. When the shader is asked to supply the color of a sample point, it will generate several more closely spaced texture samples and combine them in some weighted sum that implements a low-pass filter. As mentioned earlier, supersampling will at least decrease the sampling rate at which aliasing

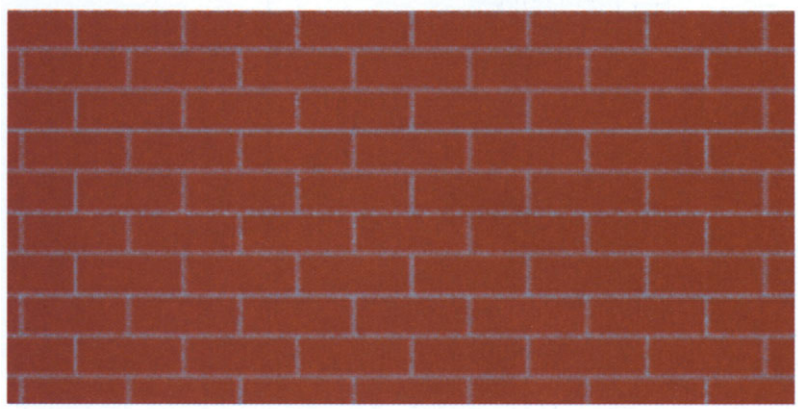

FIGURE 2.34 Box-filtered version of the brick texture.

begins. If the positions of the supersamples are jittered stochastically, the aliases will tend to be broken up into noise that might not be objectionable.

Supersampling in the procedural texture can be complicated and expensive, but it might be more efficient than supersampling implemented by the renderer. The procedural texture can limit the amount of code that is executed for each texture sample and therefore do a more lightweight version of supersampling.

MAKING NOISES

To generate irregular procedural textures, we need an irregular primitive function, usually called `noise`. This is a function that is apparently stochastic and will break up the monotony of patterns that would otherwise be too regular. When we use terms like "random" and "stochastic" in this discussion, we almost always mean to say "apparently random" or "pseudorandom." True randomness is unusual in computer science, and as you will see, it is actually undesirable in procedural textures.

We discussed the importance of aliasing and antialiasing before covering irregular patterns because issues related to antialiasing are of key importance in the design of stochastic texture primitives.

The obvious stochastic texture primitive is *white noise*, a source of random numbers, uniformly distributed with no correlation whatsoever between successive numbers. White noise can be generated by a random physical process, such as the thermal noise that occurs within many analog electronic systems. Try tuning a television to a channel on which no station is currently broadcasting if you want to see a good approximation to white noise.

A pseudorandom number generator produces a fair approximation to white noise. But is white noise really what we need? Alas, a bit of thought reveals that it is not. White noise is never the same twice. If we generate a texture pattern on the surface of some object, let's say a marble pattern, we certainly will want the pattern to stay the same frame after frame in an animation or when we look at the object from a variety of camera positions. In fact, we need a function that is apparently random but is a repeatable function of some inputs. Truly random functions don't have inputs. The desired stochastic texture primitive will take texture coordinates as its inputs and will always return the same value given the same texture coordinates.

Luckily, it isn't hard to design such a function. Looking into the literature of hashing and pseudorandom number generation, we can find several ways to convert a set of coordinate numbers into some hashed value that can be treated as a pseudorandom number (PRN). Alternatively, the hashed value can be used as an index into a table of previously generated PRNs.

Even this repeatable sort of white noise isn't quite what is needed in a stochastic texture primitive. The repeatable pseudorandom function has an unlimited amount of detail, which is another way of saying that its values at adjacent points are completely independent of one another (uncorrelated). This sounds like what we want, but it proves to be troublesome because of the prevalence of point sampling in computer graphics. If we view an object from a new camera angle, the positions of the sample points at which the texture function is evaluated will change. A good PRN function will change its value markedly if the inputs change even slightly. Consequently, the texture will change when the camera is moved, and we don't want that to happen.

Another way to look at this problem is in terms of aliasing. White noise has its energy spread equally over all frequencies, including frequencies much higher than the Nyquist frequency of the shading samples. The sampling rate can never be high enough to capture the details of the white noise.

To keep our procedural textures stable and to keep them from aliasing, we need a stochastic function that is smoother than white noise. The solution is to use a low-pass-filtered version of white noise.[10] In the remainder of this chapter, we refer to these filtered noise functions simply as `noise` functions.

The properties of an ideal `noise` function are as follows:

- `noise` is a repeatable pseudorandom function of its inputs.
- `noise` has a known range, namely, from -1 to 1.
- `noise` is band-limited, with a maximum frequency of about 1.
- `noise` doesn't exhibit obvious periodicities or regular patterns. Such pseudorandom functions are always periodic, but the period can be made very long and therefore the periodicity is not conspicuous.
- `noise` is *stationary*—that is, its statistical character should be translationally invariant.
- `noise` is *isotropic*—that is, its statistical character should be rotationally invariant.

The remainder of this section presents a number of implementations of noise that meet these criteria with varying degrees of success.

---

10. Low-pass-filtered noise is sometimes called *pink noise,* but that term is more properly applied to a stochastic function with a $1/f$ power spectrum.

Lattice Noises

Lattice noises are the most popular implementations of noise for procedural texture applications. They are simple and efficient and have been used with excellent results. Ken Perlin's *noise* function (Perlin 1985), "the function that launched a thousand textures," is a lattice noise of the gradient variety; an implementation equivalent to Perlin's is described on page 75.[11]

The generation of a lattice noise begins with one or more uniformly distributed PRNs at every point in the texture space whose coordinates are integers. These points form the *integer lattice*. The necessary low-pass filtering of the noise is accomplished by a smooth interpolation between the PRNs. To see why this works, recall that the correct reconstruction of a signal from a set of samples can never contain frequencies higher than the Nyquist frequency of the sample rate. Since the PRNs at the integer lattice points are equally spaced samples of white noise and since reconstruction from samples is a form of interpolation, it is reasonable to expect that the interpolated function will be approximately band-limited below the Nyquist frequency of the lattice interval. The quality of the resulting `noise` function depends on the nature of the interpolation scheme.

All lattice noises need some way to generate one or more pseudorandom numbers at every lattice point. The `noise` functions in this chapter use a table of PRNs that is generated the first time `noise` is called. To find the PRNs in the table that are to be used for a particular integer lattice point `(ix,iy,iz)`, we'll use the following code:

```
#define TABSIZE          256
#define TABMASK          (TABSIZE-1)
#define PERM(x)          perm[(x)STABMASK]
#define INDEX(ix,iy,iz) PERM((ix)+PERM((iy)+PERM(iz)))
```

The macro `INDEX` returns an index into an array with `TABSIZE` entries. The selected entry provides the PRNs needed for the lattice point. Note that `TABSIZE` must be a power of two so that performing `i&TABMASK` is equivalent to `i%TABSIZE`. As noted on page 31, using `i%TABSIZE` isn't safe, because it will yield a negative result if `i` is negative. Using the bitwise AND operation "&" avoids this problem.

The array `perm` contains a previously generated random permutation of the integers from zero to `TABMASK` onto themselves. Feeding sequential integers through the

11. In case you wish to compare the implementations, note that Ken describes his *noise* function in detail in Chapter 12.

permutation gives back a pseudorandom sequence. This hashing mechanism is used to break up the regular patterns that would result if `ix`, `iy`, and `iz` were simply added together to form an index into the `noiseTab` table. Here is a suitable `perm` array:

```
static unsigned char perm[TABSIZE] = {
225, 155, 210, 108, 175, 199, 221, 144, 203, 116,  70, 213,  69, 158,  33, 252,
  5,  82, 173, 133, 222, 139, 174,  27,   9,  71,  90, 246,  75, 130,  91, 191,
169, 138,   2, 151, 194, 235,  81,   7,  25, 113, 228, 159, 205, 253, 134, 142,
248,  65, 224, 217,  22, 121, 229,  63,  89, 103,  96, 104, 156,  17, 201, 129,
 36,   8, 165, 110, 237, 117, 231,  56, 132, 211, 152,  20, 181, 111, 239, 218,
170, 163,  51, 172, 157,  47,  80, 212, 176, 250,  87,  49,  99, 242, 136, 189,
162, 115,  44,  43, 124,  94, 150,  16, 141, 247,  32,  10, 198, 223, 255,  72,
 53, 131,  84,  57, 220, 197,  58,  50, 208,  11, 241,  28,   3, 192,  62, 202,
 18, 215, 153,  24,  76,  41,  15, 179,  39,  46,  55,   6, 128, 167,  23, 188,
106,  34, 187, 140, 164,  73, 112, 182, 244, 195, 227,  13,  35,  77, 196, 185,
 26, 200, 226, 119,  31, 123, 168, 125, 249,  68, 183, 230, 177, 135, 160, 180,
 12,   1, 243, 148, 102, 166,  38, 238, 251,  37, 240, 126,  64,  74, 161,  40,
184, 149, 171, 178, 101,  66,  29,  59, 146,  61, 254, 107,  42,  86, 154,   4,
236, 232, 120,  21, 233, 209,  45,  98, 193, 114,  78,  19, 206,  14, 118, 127,
 48,  79, 147,  85,  30, 207, 219,  54,  88, 234, 190, 122,  95,  67, 143, 109,
137, 214, 145,  93,  92, 100, 245,   0, 216, 186,  60,  83, 105,  97, 204,  52

};
```

This hashing technique is similar to the permutation used by Ken Perlin in his *noise* function.

Ward (1991) gives an implementation of a lattice noise in which the lattice PRNs are generated directly by a hashing function rather than by looking in a table of random values.

Value Noise

Given a PRN between −1 and 1 at each lattice point, a noise function can be computed by interpolating among these random values. This is called *value noise*. The following routine will initialize a table of PRNs for value noise:

```
#define RANDMASK 0xVfffffff
#define RANDNBR ((random() & RANDMASK)/(double) RANDMASK)
```

```
float valueTab[TABSIZE];

void
valueTabInit(int seed)
{
      float *table = valueTab;
      int i;
      srandom(seed);
      for(i = 0; i < TABSIZE; i++)
      *table++ = 1. - 2.*RANDNBR;
}
```

Given this table, it is straightforward to generate the PRN for an integer lattice point with coordinates `ix`, `iy`, and `iz`:

```
float
vlattice(int ix, int iy, int iz)
{
      return valueTab[INDEX(ix,iy,iz)];
}
```

The key decision to be made in implementing value noise is how to interpolate among the lattice PRNs. Many different methods have been used, ranging from linear interpolation to a variety of cubic interpolation techniques. Linear interpolation is insufficient for a smooth-looking noise; value noise based on linear interpolation looks "boxy," with obvious lattice cell artifacts. The derivative of a linearly interpolated value is not continuous, and the sharp changes are obvious to the eye. It is better to use a cubic interpolation method so both the derivative and the second derivative are continuous. Here is a simple implementation using the cubic Catmull-Rom spline interpolation function shown on page 34:

```
float
vnoise(float x, float y, float z)
{
      int ix, iy, iz;
      int i, j, k;
      float fx, fy, fz;
      float xknots[4], yknots[4], zknots[4];
      static int initialized = 0;

      if (!initialized) {
         valueTabInit(665);
         initialized = 1;
         }
      ix = FLOOR(x);
      fx = x - ix;
```

```
    iy = FLOOR(y);
    fy = y - iy;

    iz = FLOOR(z);
    fz = z - iz;

    for (k = -1; k <= 2; k++) {
      for (j = -1; j <= 2; j++) {
        for (i = -1; i <= 2; i++)
           xknots[i+1] = vlattice(ix+i,iy+j,iz+k);
        yknots[j+1] = spline(fx, 4, xknots);
      }
      zknots[k+1] = spline(fy, 4, yknots);
    }
    return spline(fz, 4, zknots);
}
```

Since this is a cubic Catmull-Rom `spline` function in all three dimensions, the spline has 64 control points, which are the vertices of the 27 lattice cells surrounding the point in question. Obviously, this interpolation can be quite expensive. It might make sense to use a modified version of the `spline` function that is optimized for the special case of four knots and a parameter value that is known to be between 0 and 1.

A graph of a 1D sample of `vnoise` is shown in Figure 2.35(a), and an image of a 2D slice of the function is shown in Figure 2.36(a). Figure 2.37(a) shows its power spectrum. The noise obviously meets the criterion of being band-limited; it has no significant energy at frequencies above 1.

Many other interpolation schemes are possible for value noise. Quadratic and cubic B-splines are among the most popular. These splines don't actually interpolate the lattice PRN values; instead they approximate the values, which may lead to a narrower oscillation range (lower amplitude) for the B-spline noise. The lattice convolution noise discussed on page 78 can be considered a value noise in which the interpolation is done by convolving a filter kernel with the lattice PRN values.

Lewis (1989) describes the use of *Wiener interpolation* to interpolate lattice PRNs. Lewis claims that Wiener interpolation is efficient and provides a limited amount of control of the `noise` power spectrum.

Gradient Noise

Value noise is the simplest way to generate a low-pass-filtered stochastic function. A less obvious method is to generate a pseudorandom gradient vector at each lattice point and then use the gradients to generate the stochastic function. This is called

gradient noise. The *noise* function described by Perlin (1985) and Perlin and Hoffert (1989) was the first implementation of gradient noise. The RenderMan shading language `noise` function used in the irregular texture examples later in this chapter is a similar implementation of gradient noise.

The value of a gradient noise is 0 at all of the integer lattice points. The pseudorandom gradients determine its behavior between lattice points. The gradient method uses an interpolation based on the gradients at the eight corners of a single lattice cell, rather than the 64-vertex neighborhood used in the cubic interpolation method described in the previous section.

Our implementation of gradient noise begins by using the following routine to initialize the table of pseudorandom gradient vectors:

```
#include <math.h>

float gradientTab[TABSIZE*3];
```

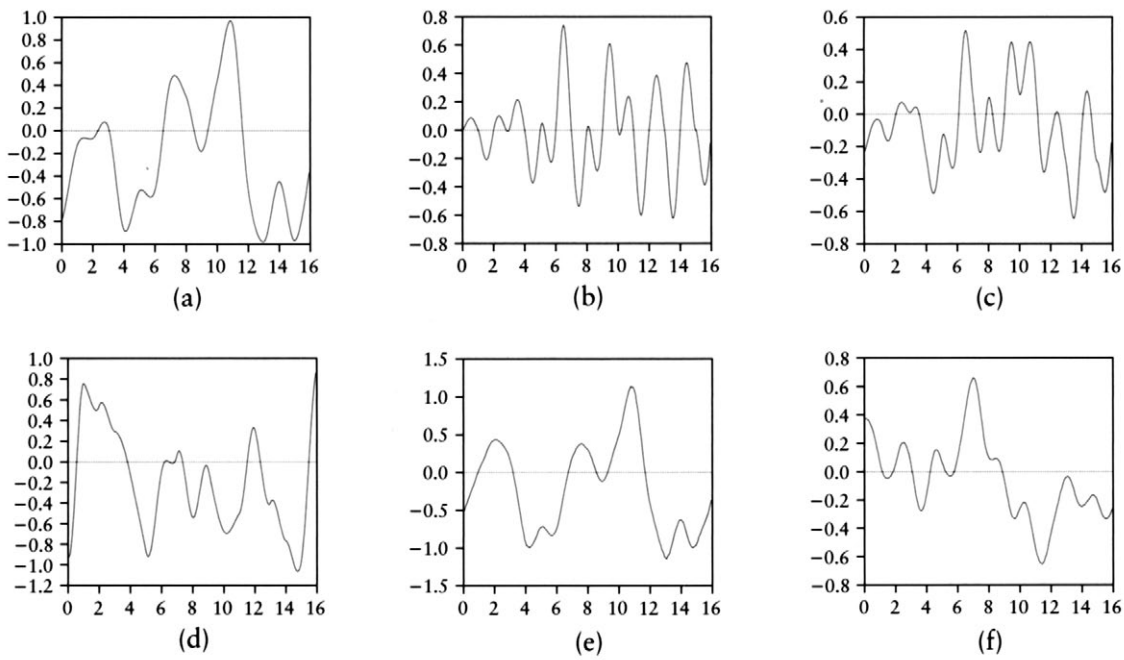

FIGURE 2.35 Graphs of various noises: (a) `vnoise`; (b) `gnoise` (Perlin's noise); (c) `vnoise + gnoise`; (d) Ward's Hermite noise; (e) `vcnoise`; (f) `scnoise`.

```
void
gradientTabInit(int seed)
{
      float *table = gradientTab;
      float z, r, theta;
      int i;

      srandom(seed);
      for(i = 0; i < TABSIZE; i++) {
            z = 1. - 2.*RANDNBR;
            /* r is radius of x,y circle */
            r = sqrtf(1 - z*z);
            /* theta is angle in (x,y) */
            theta = 2 * M_PI * RANDNBR;
```

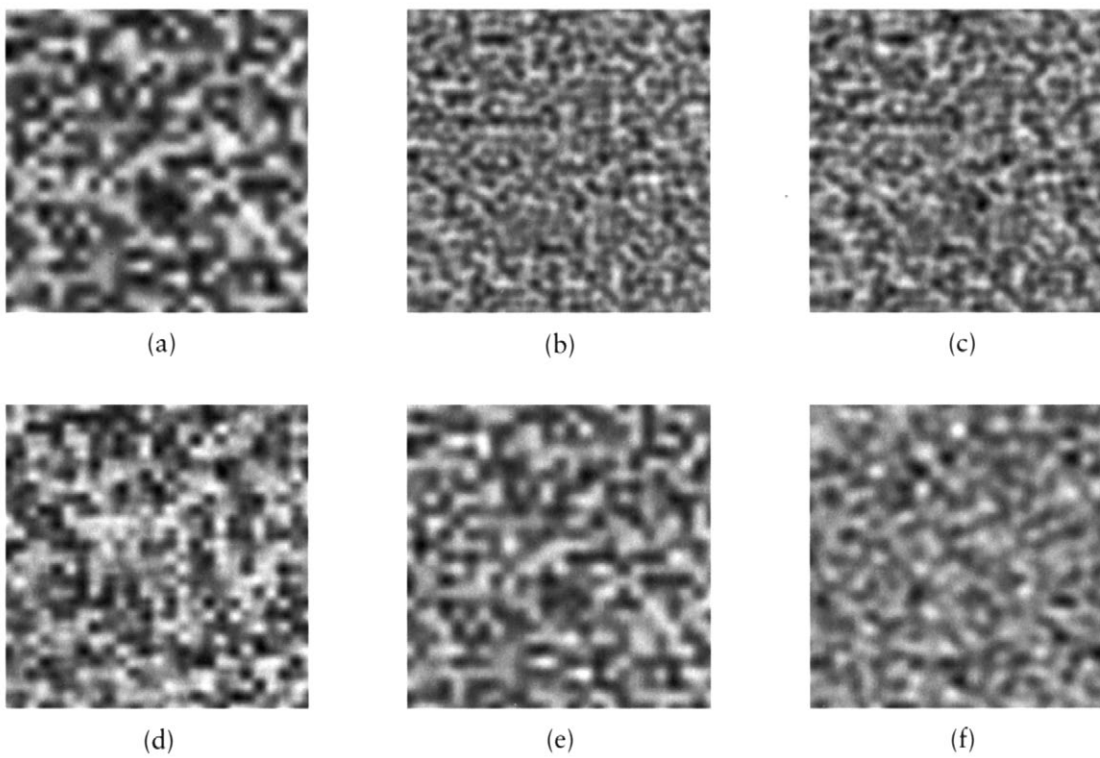

FIGURE 2.36 2D slices of various noises: (a) `vnoise`; (b) `gnoise` (Perlin's noise); (c) `vnoise + gnoise`; (d) Ward's Hermite noise; (e) `vcnoise`; (f) `scnoise`.

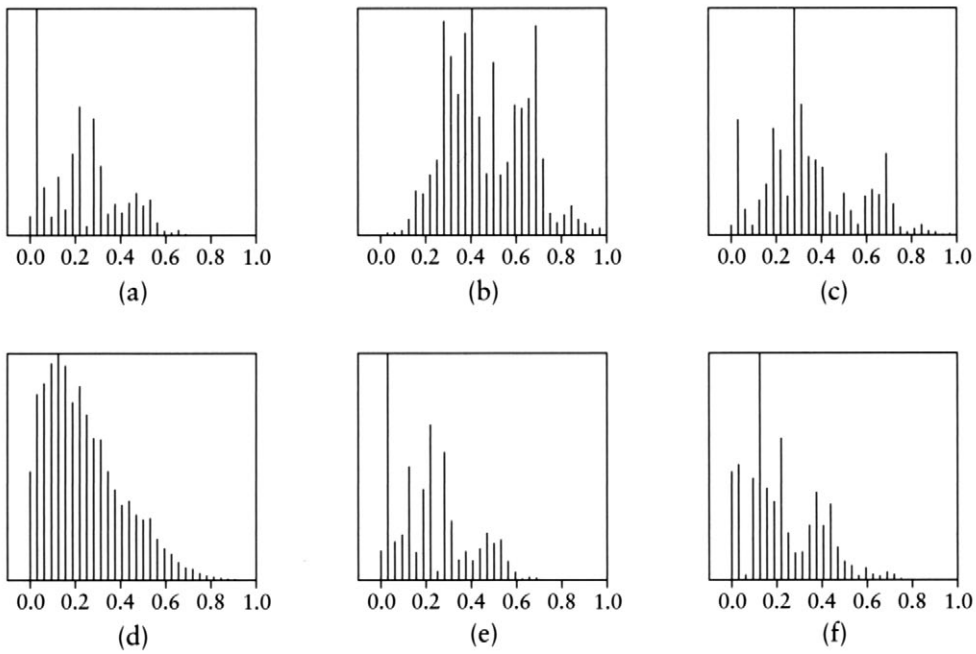

FIGURE 2.37 The power spectra of various noises: (a) `vnoise`; (b) `gnoise` (Perlin's noise); (c) `vnoise + gnoise`; (d) Ward's Hermite noise; (e) `vcnoise`; (f) `scnoise`.

```
            *table++ = r * cosf(theta);
            *table++ = r * sinf(theta);
            *table++ = z;
        }
}
```

This method of generating the gradient vectors attempts to produce unit vectors uniformly distributed over the unit sphere. It begins by generating a uniformly distributed `z` coordinate that is the sine of the latitude angle. The cosine of the same latitude angle is the radius `r` of the circle of constant latitude on the sphere. A second PRN is generated to give the longitude angle `theta` that determines the `x` and `y` components of the gradient.

Perlin's *noise* implementation uses a different scheme of generating uniformly distributed unit gradients. His method is to generate vectors with components between -1 and 1. Such vectors lie within the cube that bounds the unit sphere. Any vector whose length is greater than 1 lies outside the unit sphere and is discarded.

Keeping such vectors would bias the distribution in favor of the directions toward the corners of the cube. These directions have the greatest volume within the cube per solid angle. The remaining vectors are normalized to unit length.

The following routine generates the value of gradient noise for a single integer lattice point with coordinates `ix`, `iy`, and `iz`. The value `glattice` of the gradient noise for an individual lattice point is the dot product of the lattice gradient and the fractional part of the input point relative to the lattice point, given by `fx`, `fy`, and `fz`.

```
float
glattice(int ix, int iy, int iz,
float fx, float fy, float fz)
{
      float *g = &gradientTab[INDEX(ix,iy,iz)*3];
      return g[0]*fx + g[1]*fy + g[2]*fz;
}
```

Eight `glattice` values are combined using smoothed trilinear interpolation to get the gradient noise value. The linear interpolations are controlled by a `smoothstep`-like function of the fractional parts of the input coordinates.

```
#define LERP(t,x0,x1) ((x0) + (t)*((x1)-(x0)))
#define SMOOTHSTEP(x) ((x)*(x)*(3 - 2*(x)))

float
gnoise(float x, float y, float z)
{
      int ix, iy, iz;
      float fx0, fx1, fy0, fy1, fz0, fz1;
      float wx, wy, wz;
      float vx0, vx1, vy0, vy1, vz0, vz1;

      static int initialized = 0;

      if (!initialized) {
            gradientTabInit(665);
            initialized = 1;
      }
      ix = FLOOR(x);
      fx0 = x - ix;
      fx1 = fx0 - 1;
      wx = SMOOTHSTEP(fx0);
      iy = FLOOR(y);
      fy0 = y - iy;
      fy1 = fy0 - 1;
      wy = SMOOTHSTEP(fy0);
```

```
    iz = FLOOR(z);
    fz0 = z - iz;
    fz1 = fz0 - 1;
    wz = SMOOTHSTEP(fz0) ;

    vx0 = glattice(ix,iy,iz,fx0,fy0,fz0);
    vx1 = glattice(ix+1,iy,iz,fx1,fy0,fz0);
    vy0 = LERP(wx, vx0, vx1);
    vx0 = glattice(ix,iy+1,iz,fx0,fy1,fz0);
    vx1 = glattice(ix+1,iy+1,iz,fx1,fy1,fz0);
    vy1 = LERP(wx, vx0, vx1);
    vz0 = LERP(wy, vy0, vy1);
    vx0 = glattice(ix,iy,iz+1,fx0,fy0,fz1);
    vx1 = glattice(ix+1,iy,iz+1,fx1,fy0,fz1);
    vy0 = LERP(wx, vx0, vx1);
    vx0 = glattice(ix,iy+1,iz+1,fx0,fy1,fz1);
    vx1 = glattice(ix+1,iy+1,iz+1,fx1,fy1,fz1);
    vy1 = LERP(wx, vx0, vx1);
    vz1 = LERP(wy, vy0, vy1);

    return LERP(wz, vz0, vz1);
}
```

Figure 2.35(b) is a graph of a 1D sample of a gradient noise, and Figure 2.36(b) shows a 2D slice of the noise. Figure 2.37(b) shows its power spectrum. Most of the energy of gradient noise comes from frequencies between 0.3 and 0.7. There is more high-frequency energy in gradient noise than in value noise and less low-frequency energy. These are consequences of the fact that gradient noise has zeros at each lattice point and therefore is forced to change direction at least once per lattice step.

Value-Gradient Noise

A gradient noise is zero at all of the integer lattice points. This regular pattern of zeros sometimes results in a noticeable grid pattern in the gradient noise. To avoid this problem without losing the spectral advantages of gradient noise, we might try to combine the value and gradient methods to produce a value-gradient noise function.

One implementation of value-gradient noise is simple: it is just a weighted sum of a value noise and a gradient noise. Some computation can be saved by combining the two functions and sharing common code such as the `INDEX` calculation from the integer coordinates and the calculation of `ix`, `fx0`, and so on. Figure 2.35(c) shows a graph of a weighted sum of our Catmull-Rom `vnoise` and the gradient `gnoise`, and Figure 2.36(c) shows a 2D slice of it. Figure 2.37(c) shows the power spectrum

of this function. The slice image looks a little less regular than the gradient noise slice, presumably because the regular pattern of zero crossings has been eliminated.

A more sophisticated form of value-gradient noise is based on cubic Hermite interpolation. The Hermite spline is specified by its value and tangent at each of its end points. For a value-gradient noise, the tangents of the spline can be taken from the gradients. Ward (1991) gives the source code for just such a value-gradient noise. (Ward states that this is Perlin's *noise* function, but don't be fooled—it is quite different.) Figures 2.35(d) and 2.36(d) show Ward's Hermite noise function, and Figure 2.37(d) shows its power spectrum.[12] The power spectrum is remarkably regular, rising quite smoothly from DC to a frequency of about 0.2 and then falling smoothly down to 0 at a frequency of 1. Since the power is spread quite widely over the spectrum and the dominant frequencies are quite low, this noise function could be difficult to use in spectral synthesis.

Lattice Convolution Noise

One objection to the lattice noises is that they often exhibit axis-aligned artifacts. Many of the artifacts can be traced to the anisotropic nature of the interpolation schemes used to blend the lattice PRN values. What we'll call *lattice convolution noise* is an attempt to avoid anisotropy by using a discrete convolution technique to do the interpolation. The PRNs at the lattice points are treated as the values of random impulses and are convolved with a radially symmetrical filter. In the implementation that follows, we'll use a Catmull-Rom filter with negative lobes and a radius of 2. This means that any lattice point within a distance of two units from the input point must be considered in doing the convolution. The convolution is simply the sum of the product of each lattice point PRN value times the value of the filter function based on the distance of the input point from the lattice point.

Here is an implementation of lattice convolution noise called `vcnoise`. It begins with the filter function `catrom2`, which takes a squared distance as input to avoid the need to compute square roots. The first time `catrom2` is called, it computes a table of Catmull-Rom filter values as a function of squared distances. Subsequent calls simply look up values from this table.

12. This analysis of Ward's noise function is actually based on the source code that is provided on the diskette that accompanies *Graphics Gems IV*. The code on the diskette seems to be a newer version of Ward's routine with an improved interpolation method.

```
static float catrom2(float d)
{
#define SAMPRATE 100 /* table entries per unit distance */
#define NENTRIES (4*SAMPRATE+1)
      float x;
      int i;
      static float table[NENTRIES];
      static int initialized = 0;
      if (d >= 4)
            return 0;
      if (!initialized) {
            for (i = 0; i < NENTRIES; i++){
                  x = i/(float) SAMPRATE;
                  x = sqrtf(x);
                  if (x < 1)
                        table[i] = 0.5 * (2+x*x*(-5+x*3));
                  else
                        table[i] = 0.5 * (4+x*(-8+x*(5-x)));
            }
            initialized = 1;
      }
      d = d*SAMPRATE + 0.5;
      i = FLOOR(d);
      if (i >= NENTRIES)
            return 0;
      return table[i];
}

float
vcnoise(float x, float y, float z)
{
      int ix, iy, iz;
      int i, j, k;
      float fx, fy, fz;
      float dx, dy, dz;
      float sum = 0;
      static int initialized = 0;
      if (!initialized) {
            valueTabInit(665);
            initialized = 1;
      }
      ix = FLOOR(x);
      fx = x - ix;
      iy = FLOOR(y);
      fy = y - iy;
      iz = FLOOR(z);
      fz = z - iz;
```

```
        for (k = -1; k <= 2; k++) {
              dz = k - fz;
              dz = dz*dz;
              for (j = -1; j <= 2; j++) {
                    dy = j - fy;
                    dy = dy*dy;
                    for (i = -1; i <= 2; i++){
                          dx = i - fx;
                          dx = dx*dx;
                          sum += vlattice(ix+i,iy+j,iz+k)
                                * catrom2(dx + dy + dz);
                    }
              }
        }
        return sum;
}
```

Figure 2.35(e) shows a graph of `vcnoise`, and Figure 2.36(e) shows a 2D slice of it. Figure 2.37(e) shows its power spectrum. The spectrum is not unlike that of the other value noises. Perhaps this is not surprising since it is essentially a value noise with a different interpolation scheme. Some degree of spectral control should be possible by modifying the filter shape.

Sparse Convolution Noise

There are several ways to generate noise functions that aren't based on a regular lattice of PRNs. One such method is called *sparse convolution* (Lewis 1984, 1989). A similar technique called *spot noise* is described by van Wijk (1991).

Sparse convolution involves building up a noise function by convolving a filter function with a collection of randomly located random impulses (a Poisson process). The scattered random impulses are considered to be a "sparse" form of white noise, hence the term "sparse convolution." The low-pass filtering of the white noise is accomplished by the filter function. The power spectrum of the sparse convolution noise is derived from the power spectrum of the filter kernel, so some control of the noise spectrum is possible by modifying the filter.

Sparse convolution is essentially the same as the lattice convolution noise algorithm described in the previous section, except that the PRN impulse values are located at pseudorandom points in each lattice cell. Here is an implementation of `scnoise`, a sparse convolution noise based on Lewis's description. The filter used is the Catmull-Rom filter described in the previous section. Three randomly placed impulses are generated in each lattice cell. A neighborhood of 125 lattice cells must

be considered for each call to the noise function because a randomly placed impulse two cells away could have a nonzero filter value within the current cell. As a result this noise function is computationally expensive.

```
static float impulseTab[TABSIZE*4];
static void
impulseTabInit(int seed)
{
      int i;
      float *f = impulseTab;
      srandom(seed); /* Set random number generator seed. */
      for (i = 0; i < TABSIZE; i++) {
            *f++ = RANDNBR;
            *f++ = RANDNBR;
            *f++ = RANDNBR;
            *f++ = 1. - 2.*RANDNBR;
      }
}

#define NEXT(h) (((h)+1) & TABMASK)
#define NIMPULSES 3

float
scnoise(float x, float y, float z)
{
      static int initialized;
      float *fp;
      int i, j, k, h, n;
      int ix, iy, iz;
      float sum = 0;
      float fx, fy, fz, dx, dy, dz, distsq;

      /* Initialize the random impulse table if necessary. */
      if (!initialized) {
            impulseTabInit(665);
            initialized = 1;
      }
      ix = FLOOR (x); fx = x - ix;
      iy = FLOOR(y);  fy = y - iy;
      iz = FLOOR (z); fz = z - iz;

      /* Perform the sparse convolution. */
      for (i = -2; i <= 2; i++) {
            for (j = -2; j <= 2; j++) {
                  for (k = -2; k <= 2; k++) {
                        /* Compute voxel hash code. */
                        h = INDEX(ix+i,iy+j,iz+k);
```

```
                        for (n = NIMPULSES; n > 0; n--, h = NEXT(h)) {
                              /* Convolve filter and impulse. */
                              fp = &impulseTab[h*4];
                              dx = fx - (i + *fp++);
                              dy = fy - (j + *fp++);
                              dz = fz - (k + *fp++);
                              distsq = dx*dx + dy*dy + dz*dz;
                              sum += catrom2(distsq) * *fp;
                        }
                  }
            }
      }

      return sum / NIMPULSES;
}
```

Figures 2.35(f) and 2.36(f) show 1D and 2D sections of `scnoise`. Figure 2.37(f) shows the power spectrum. The spectrum is similar to that of the other value noises, but the slice image appears to exhibit fewer gridlike patterns than the other noises.

Explicit Noise Algorithms

Some interesting methods of generating noises and random fractals aren't convenient for implicit procedural texture synthesis. These methods generate a large batch of noise values all at once in an explicit fashion. To use them in an implicit procedural texture during rendering, the noise values would have to be generated before rendering and stored in a table or texture image. A good example of such a technique is the midpoint displacement method (Fournier, Fussell, and Carpenter 1982). A related method of random successive additions is described by Saupe (1992). Lewis (1986, 1987) describes a generalization of such methods to give greater spectral control. Saupe (1989) shows that such methods can be more than an order of magnitude less expensive than implicit evaluation methods.

Fourier Spectral Synthesis

Another explicit method of noise generation is to generate a pseudorandom discrete frequency spectrum in which the power at a given frequency has a probability distribution that is correct for the desired noise. Then a discrete inverse Fourier transform (usually an inverse FFT) is performed on the frequency domain representation to get a spatial domain representation of the noise. Saupe (1988) and Voss (1988) describe this technique.

In the description of spectral synthesis textures on page 51, the example showed that many hand-picked "random" coefficients were used to generate the cloud texture. We could think of this as the generation of a random frequency domain representation and the evaluation of the corresponding spatial function using a spectral sum to implement the discrete inverse Fourier transform. This is far less efficient than an FFT algorithm, but has the advantage that it can be evaluated a point at a time for use in an implicit procedural texture. The complexity and apparent irregularity of Gardner's textures is less surprising when they are seen to be noiselike stochastic functions in disguise!

Direct Fourier synthesis of noise is much slower than the lattice noises described earlier and is probably not practical for procedural texture synthesis. Lattice convolution and sparse convolution are other methods that offer the promise of detailed spectral control of the noise. There is a trade-off between trying to generate all desired spectral characteristics in a single call to `noise` by using an expensive method such as Fourier synthesis or sparse convolution versus the strategy of building up spectral characteristics using a weighted sum of several cheaper gradient noise components.

Gradient noise seems to be a good primitive function to use for building up spectral sums of noise components, as demonstrated in the next section. When combining multiple `noise` calls to build up a more complex stochastic function, the gradient noise gives better control of the spectrum of the complex function because gradient noise has little low-frequency energy compared to the other noise functions; its dominant frequencies are near one-half.

GENERATING IRREGULAR PATTERNS

Armed with the stochastic primitive functions from the preceding section, we can now begin to generate irregular texture patterns. Since most natural materials are somewhat irregular and nonuniform, irregular texture patterns are valuable in simulating these materials. Even manufactured materials are usually irregular as a result of shipping damage, weathering, manufacturing errors, and so on. This section describes several ways to generate irregular patterns and gives examples in the form of RenderMan shaders.

The `noise` function in the RenderMan shading language is an implementation of the lattice gradient noise described in the preceding section. The RenderMan function is unusual in that it has been scaled and offset to range from 0 to 1, instead of the more usual range of -1 to 1. This means that the RenderMan `noise` function has a value of 0.5 at the integer lattice points. The -1 to 1 range is sometimes more

convenient, and we can use the following signed noise macro in RenderMan shaders to get a noise in this range:

```
#define snoise(x) (2 * noise(x) - 1)
```

The RenderMan `noise` function can be called with a 1D, 2D, or 3D input point, and will return a 1D or 3D result (a number, a point, or a color).

Most textures need several calls to `noise` to independently determine a variety of stochastic properties of the material. Remember that repeated calls to `noise` with the same inputs will give the same results. Different results can be obtained by shifting to another position in the noise space. It is common to call

```
noise(Q * frequency + offset)
```

where `offset` is of the same type as the coordinate `Q` and has the effect of establishing a new noise space with a different origin point.

There are two approaches to generating time-dependent textures. Textures that move or flow can be produced by moving through the 3D noise space over time:

```
f = noise(P - time*D);
```

can be used to make the texture appear to move in the direction and rate given by the vector `D` as time advances. Textures that simply evolve without a clear flow direction are harder to create. A 4D `noise` function is the best approach. RenderMan's `noise` function is limited to three dimensions.[13] In some cases one or two dimensions of the `noise` result can be used to represent spatial position, so that the remaining dimension can be used to represent time.

The shading language function `pnoise` is a relative of `noise`. The noise space can be wrapped back on itself to achieve periodic effects. For example, if a period of 30 is specified, `pnoise` will give the same value for input $x + 30$ as for input x. The oscillation frequency of the noise value is unaffected. Here are some typical `pnoise` calls:

```
pnoise(f, 30 );
pnoise(s, t, 30, 30)
pnoise(P, point (10, 15, 30))
```

13. We use a 4D quadratic B-spline value noise in our in-house animation system with good results. It pays to keep the order of the interpolation fairly low for 4D noise to avoid having too many lattice point terms in the interpolation.

It is easy to implement `pnoise` by making the choice of lattice PRNs periodic with the desired period. This technique is limited to integer periods.

When generating procedural textures using any of the `noise` functions described in this chapter, it is important to develop an understanding of the range and distribution of the noise values. It can be difficult to normalize a stochastic function so that the range is exactly −1 to 1. Furthermore, most of the noise values tend to lie close to 0 with only occasional excursions to the limits of the range. A histogram of the distribution of the noise values is a useful tool in designing functions based on `noise`.

Spectral Synthesis

The discussion on page 48 showed how complex regular functions with arbitrary spectral content can be built up from sine waves. As mentioned earlier, the many pseudorandom coefficients in Gardner's spectral synthesis textures might be viewed as a way of generating a noise function by the inverse Fourier transform method. Spectral synthesis using a noise function as the primitive gives an even richer stochastic content to the texture and reduces the need to use random coefficients for each component. When a stochastic function with a particular power spectrum is needed, spectral synthesis based on `noise` can be used to generate it.

Several calls to `noise` can be combined to build up a stochastic spectral function with a particular frequency/power spectrum. A noise loop of the form

```
value = 0;
for (f = MINFREQ; f < MAXFREQ; f *= 2)
      value += amplitude * snoise(Q * f);
```

with amplitude varying as a function of frequency `f` will build up a value with a desired spectrum. `Q` is the sample point in some texture space.

Perlin's well-known `turbulence` function is essentially a stochastic function of this type with a "fractal" power spectrum, that is, a power spectrum in which amplitude is proportional to `1/f`.

```
float
fractalsum(point Q)
{
      float value = 0;
      for (f = MINFREQ; f < MAXFREQ; f *= 2)
        value += snoise(Q * f)/f;
      return value;
}
```

This isn't quite the same as `turbulence`, however. Derivative discontinuities are added to the `turbulence` function by using the absolute value of the `snoise` function. Taking the absolute value folds the function at each zero crossing, making the function undifferentiable at these points. The number of peaks in the function is doubled, since the troughs become peaks.

```
float
turbulence(point Q)
{
      float value = 0;
      for (f = MINFREQ; f < MAXFREQ; f *= 2)
        value += abs(snoise(Q * f))/f;
      return value;
}
```

Figure 2.38 shows a slice of the `fractalsum` function on the left and a slice of the `turbulence` function on the right. The `fractalsum` is very cloudlike in appearance, while `turbulence` is apparently lumpier, with sharper changes in value. Figure 2.39(a) shows the power spectrum of the `fractalsum` function, and Figure 2.39(b) shows the power spectrum of the `turbulence` function. As you might expect, the power spectra show a rapid decline in energy as the frequency increases; this is a direct result of the $1/f$ amplitude scaling in the spectral synthesis loops.

Spectral synthesis loops should use clamping to prevent aliasing. Here is a version of `turbulence` with clamping included:

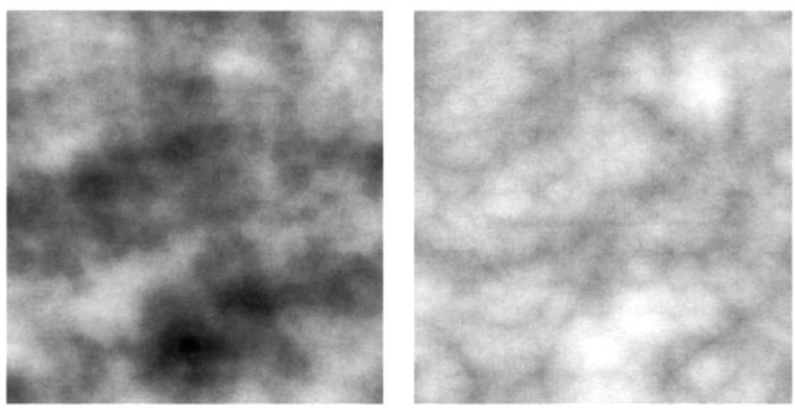

FIGURE 2.38 Slices of `fractalsum` and `turbulence` functions.

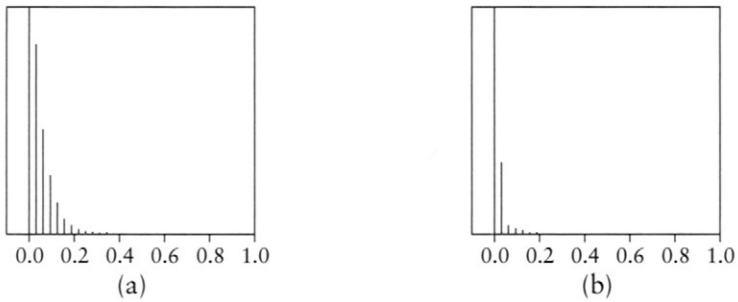

FIGURE 2.39 Power spectra of (a) `fractalsum` and (b) `turbulence` functions.

```
float
turbulence(point Q)
{
      float value = 0;
      float cutoff = clamp(0.5/Qwidth, 0, MAXFREQ);
      float fade;

      for (f = MINFREQ; f < 0.5*cutoff; f *= 2)
        value += abs(snoise(Q * f))/f;
      fade = clamp(2*(cutoff-f)/cutoff, 0, 1);
      value += fade * abs(snoise(Q * f))/f;
      return value;
}
```

Marble is a material that is typically simulated using an irregular texture based on spectral synthesis. The following marble shader uses a four-octave spectral synthesis based on `noise` to build up a stochastic value called `marble` that is similar to `fractalsum`. It is best to use a solid texture space for a marble texture, so that the texture can be used to shade curved surfaces as if they were carved out of a solid block of marble. This is accomplished by using the 3D surface point as the argument to the `noise` calls.

```
#define PALE_BLUE    color (0.25, 0.25, 0.35)
#define MEDIUM_BLUE  color (0.10, 0.10, 0.30)
#define DARK_BLUE    color (0.05, 0.05, 0.26)
#define DARKER_BLUE  color (0.03, 0.03, 0.20)
#define NNOISE       4
```

```
color
marble_color(float m)
{
      return color spline(
            clamp(2 * m + .75, 0, 1),
            PALE_BLUE, PALE_BLUE,
            MEDIUM_BLUE, MEDIUM_BLUE, MEDIUM_BLUE,
            PALE_BLUE, PALE_BLUE,
            DARK_BLUE, DARK_BLUE,
            DARKER_BLUE, DARKER_BLUE,
            PALE_BLUE, DARKER_BLUE);
}


surface blue_marble(
      uniform float Ka = 1;
      uniform float Kd = 0.8;
      uniform float Ks = 0.2;
      uniform float texturescale = 2.5;
      uniform float roughness = 0.1;
    )
{
      color Ct;
      point NN;
      point PP;
      float i, f, marble;


      NN = normalize(faceforward(N, I));
      PP = transform("shader", P) * texturescale;
      marble = 0; f = 1;
      for (i = 0; i < NNOISE; i += 1) {
            marble += snoise(PP * f)/f;
            f *= 2.17;
      }
      Ct = marble_color(marble);
      Ci = Os * (Ct * (Ka * ambient() + Kd * diffuse(NN))
            + Ks * specular(NN, normalize(-I), roughness));
}
```

The function `marble_color` maps the floating-point number `marble` into a color using a color spline. Figure 2.40 shows an example of the marble texture.

The spectral synthesis loop in this shader has no clamping control to avoid aliasing. If the texture is viewed from far away, aliasing artifacts will appear.

Note that the frequency multiplier in the spectral synthesis loop is 2.17 instead of exactly 2. This is usually better because it prevents the alignment of the lattice points of successive `snoise` components, and that tends to reduce lattice artifacts. In

FIGURE 2.40 Blue marble texture.

Chapter 6, Steve Worley describes another way to reduce lattice artifacts: randomly rotating the texture spaces of the different components.

Changing the frequency multiplier from 2 to some other value (like 2.17) affects the average size of gaps or *lacunae* in the texture pattern. In the fractal literature, this property of the texture is called *lacunarity* (Mandelbrot 1982). The scaling of the amplitude of successive components affects the *fractal dimension* of the texture. The $1/f$ amplitude scaling used here gives a $1/f^2$ scaling of the power spectrum, since power spectral density is proportional to amplitude squared. This results in an approximation to fractional Brownian motion (fBm) (Saupe 1992).

Perturbed Regular Patterns

Purely stochastic patterns tend to have an amorphous character. Often the goal is a more structured pattern with some appearance of regularity. The usual approach is to start with a regular pattern and add noise to it in order to make it look more interesting and more natural.

For example, the brick texture described on page 39 is unrealistically regular, as if the bricklayer were inhumanly precise. To add some variety to the texture, `noise` can be used to modify the relative positions of the bricks in the different rows. The following is the code from the original shader that calculates which brick contains the sample point.

```
sbrick = floor(ss); /* which brick? */
tbrick = floor(tt); /* which brick? */
ss -= sbrick;
tt -= tbrick;
```

To perturb the ss location of the bricks, we can rewrite this code as follows:

```
tbrick = floor(tt); /* which brick? */
ss += 0.1 * snoise(tbrick+0.5);
sbrick = floor(ss); /* which brick? */
ss -= sbrick;
tt -= tbrick;
```

There are a few subtle points to note here. The call to `snoise` uses `tbrick` rather than `tt` so that the noise value is constant over the entire brick. Otherwise, the stochastic offset would vary over the height of the brick and make the vertical edges of the brick wavy. Of course, we had to reorder the calculation of `sbrick` and `tbrick` so that `sbrick` can depend on `tbrick`. The value of the perturbation will change *suddenly* as `tbrick` jumps from one integer to the next. That's okay in this case, because the perturbation affects only the horizontal position of a row of bricks, and changes only in the middle of the mortar groove between the rows.

Since `snoise` is a gradient noise that is zero at all integer points, the perturbation always would be zero if the shader used `snoise(tbrick)`. Instead, it uses `snoise(tbrick + 0.5)` to sample the value of the noise halfway between the integer points, where it should have an interesting value.

The 0.1 multiplier on the `snoise` simply controls the size of the irregularity added to the texture. It can be adjusted as desired.

A more realistic version of this brick texture would incorporate some noise-based variations in the color of the bricks and mortar as well as a small stochastic bump mapping of the normal to simulate roughness of the bricks. It is easy to keep layering more stochastic effects onto a texture pattern to increase its realism or visual appeal. But it is important to begin with a simple version of the basic pattern and get that texture working reliably before attempting to add details and irregularity.

Perturbed Image Textures

Another valuable trick is to use a stochastic function to modify the texture coordinates used to access an image texture. This is very easy to do. For example, the simple texture access

```
Ct = texture("example.tx", s, t);
```

using the built-in texture coordinates `s` and `t` can be replaced with the following:

```
point Psh;
float ss, tt;
Psh = transform("shader", P);
ss = s + 0.2 * snoise(Psh);
tt = t + 0.2 * snoise(Psh+(1.5,6.7,3.4));
Ct = texture("example.tx", ss, tt);
```

In this example, `snoise` based on the 3D surface position in "shader" space is used to modify the texture coordinates slightly. Figure 2.41(a) shows the original image texture, and Figure 2.41(b) is a texture produced by perturbing the image texture with the code above.

Random Placement Patterns

A random placement pattern is a texture pattern that consists of a number of regular or irregular subpatterns or "bombs" that are dropped in random positions and orientations to form the texture. This bombing technique (Schacter and Ahuja 1979) was originally used in an explicit form to generate image textures before rendering. With some difficulty, it can be used in an implicit texture function during rendering.

The most obvious implementation is to store the positions of bombs in a table and to search the table for each sample point. This is rather inefficient and is especially hard to implement in the RenderMan shading language since the language has no tables or arrays. With a little ingenuity, we can devise a method of bombing that uses only `noise` to determine the bomb positions relevant to a sample point. The

(a) (b)

FIGURE 2.41 Perturbing an image texture: (a) original image; (b) perturbed image

texture space is divided into a grid of square cells, with a bomb located at a random position within each cell.

In the following example shader, the bomb is the star pattern created by the procedural texture on page 46.

```
#define NCELLS 10
#define CELLSIZE (1/NCELLS)
#define snoise(s,t) (2*noise((s),(t))-1)

surface wallpaper(
      uniform float Ka = 1;
      uniform float Kd = 1;
      uniform color starcolor = color (1.0000,0.5161,0.0000);
      uniform float npoints = 5;
      )
{
      color Ct;
      point Nf;
      float ss, tt, angle, r, a, in_out;
      float sctr, tctr, scell, tcell;
      uniform float rmin = 0.01, rmax = 0.03;
      uniform float starangle = 2*PI/npoints;
      uniform point p0 = rmax*(cos(0),sin(0), 0);
      uniform point pi = rmin*
            (cos(starangle/2),sin(starangle/2), 0);
      uniform point d0 = pi - p0;
      point d1;

      scell = floor(s*NCELLS);
      tcell = floor(t*NCELLS);
      sctr = CELLSIZE * (scell + 0.5
         + 0.6 * snoise(scell+0.5, tcell+0.5));
      tctr = CELLSIZE * (tcell + 0.5
         + 0.6 * snoise(scell+3.5, tcell+8.5));
      ss = s - sctr;
      tt = t - tctr;
      angle = atan(ss, tt) + PI;
      r = sqrt(ss*ss + tt*tt);
      a = mod(angle, starangle)/starangle;

      if (a >= 0.5)
            a = 1 - a;
      d1 = r*(cos(a), sin(a),0) - p0;
      in_out = step(0, zcomp(d0^d1) );
      Ct = mix(Cs, starcolor, in_out);

      /* "matte" reflection model */
      Nf = normalize(faceforward(N, I));
```

```
        Oi = Os;
        Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(Nf));
}
```

A couple of improvements can be made to this procedural texture. The requirement to have exactly one star in each cell makes the pattern quite regular. A separate noise value for each cell could be tested to see whether the cell should contain a star.

If a star is so far from the center of a cell that it protrudes outside the cell, this shader will clip off the part of the star that is outside the cell. This is a consequence of the fact that a given star is "noticed" only by sample points that lie inside the star's cell. The shader can be modified to perform the same shading tests for the eight cells that surround the cell containing the sample point. Stars that crossed over the cell edge will then be rendered correctly. The tests can be done by a pair of nested for loops that iterate over −1, 0, and 1. The nested loops generate nine different values for the cell coordinate pair `(scell, tcell)`. The star in each cell is tested against the sample point.

```
scellctr = floor(s*NCELLS);
tcellctr = floor(t*NCELLS);
in_out = 0;
for (i = -1; i <= 1; i += 1) {
        for (j = -1; j <= 1; j += 1) {
        scell = scellctr + i;
        tcell = tcellctr + j;
        if (noise(3*scell-9.5,7*tcell+7.5) < 0.55) {
                sctr = CELLSIZE * (scell + 0.5
                        + 0.6 * snoise(scell+0.5, tcell+0.5));
                tctr = CELLSIZE * (tcell + 0.5
                        + 0.6 * snoise(scell+3.5, tcell+8.5));
                ss = s - sctr;
                tt = t - tctr;

                angle = atan(ss, tt) + PI;
                r = sqrt(ss*ss + tt*tt);
                a = mod(angle, starangle)/starangle;

                if (a >= 0.5)
                        a = 1 - a;
                d1 = r*(cos(a), sin(a),0) - p0;
                in_out += step(0, zcomp(d0^d1));
        }
    }
}
Ct = mix(Cs, starcolor, step(0.5, in_out));
```

FIGURE 2.42 Random placement wallpaper texture.

The first `noise` call is used to decide whether or not to put a star in the cell. Note that the value of `in_out` can now be as high as 9. The additional `step` call in the last line converts it back to a 0 to 1 range so that the `mix` will work properly. Of course, the `step` functions in the wallpaper shader should be `smoothstep` calls with appropriate filter widths to prevent jaggy edges on the stars. Figure 2.42 shows the star wallpaper pattern generated by the shader.

The wallpaper texture in Pixar's film *Knickknack* (Figure 2.2) is a more elaborate example of a procedural texture based on bombing.

Note that the bomb subpattern used for a random placement texture doesn't have to be a procedural texture like the star. An image texture can be used instead, so that the bombing procedure can lay down copies of any picture you like, instead of the stars.

CONCLUSION

This chapter describes an approach to constructing procedural textures and a set of building blocks that you can use to build them. The range of textures that you can build is limited only by your imagination and your programming expertise. We hope that the examples in this chapter and in the rest of the book will inspire you and enable you to create wonderful imagery of your own, using procedural textures and models.