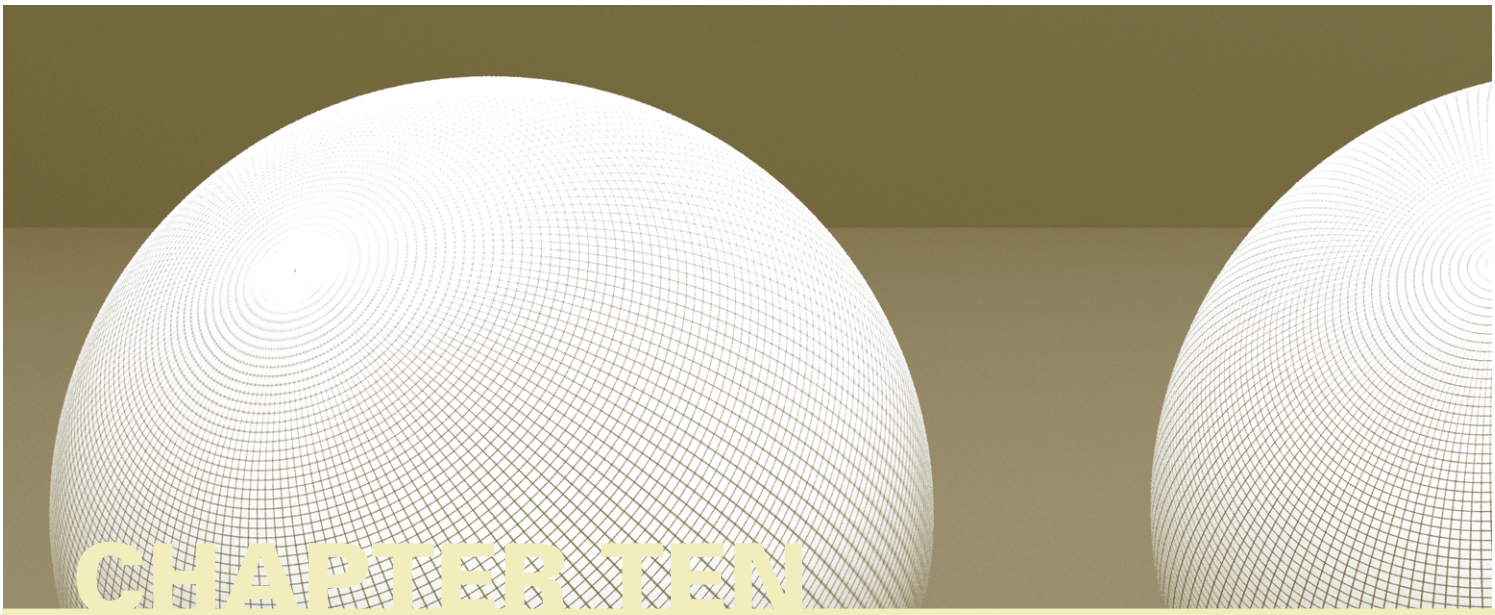
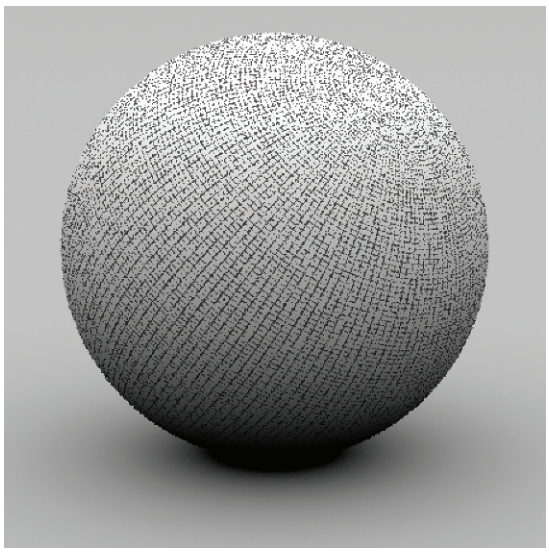

CHAPTER TEN

10 TEXTURE

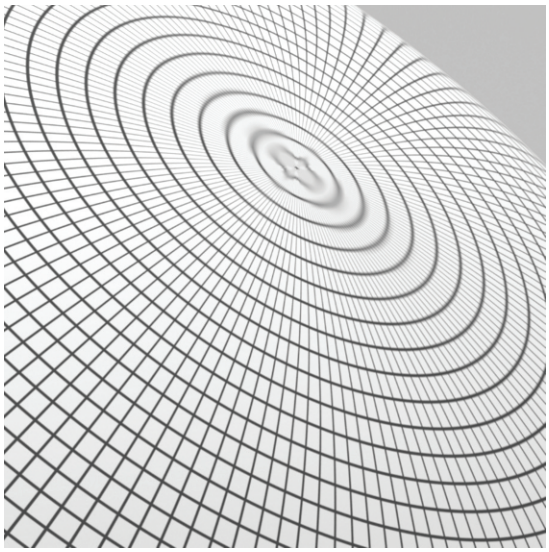
We will now describe a set of interfaces and classes that allow us to incorporate *texture* into our material models. Recall that the materials in Chapter 9 are all based on various parameters that describe their characteristics (diffuse reflectance, glossiness, etc.). Because real-world material properties typically vary over surfaces, it is necessary to be able to describe these patterns in some manner. In `pbrt`, because the texture abstractions are defined in a way that separates the pattern generation methods from the material implementations, it is easy to combine them in arbitrary ways, thereby making it easier to create a wide variety of appearances.

In `pbrt`, a texture is an extremely general concept: it is a function that maps points in some domain (e.g., a surface's (u, v) parametric space or (x, y, z) object space) to values in some other domain (e.g., spectra or the real numbers). A wide variety of implementations of texture classes are available in the system. For example, `pbrt` has textures that represent zero-dimensional functions that return a constant in order to accommodate surfaces that have the same parameter value everywhere. Image map textures are two-dimensional functions of (s, t) parameter values that use a 2D array of pixel values to compute values at a particular point (they are described in Section 10.4). There are even texture functions that compute values based on the values computed by other texture functions.

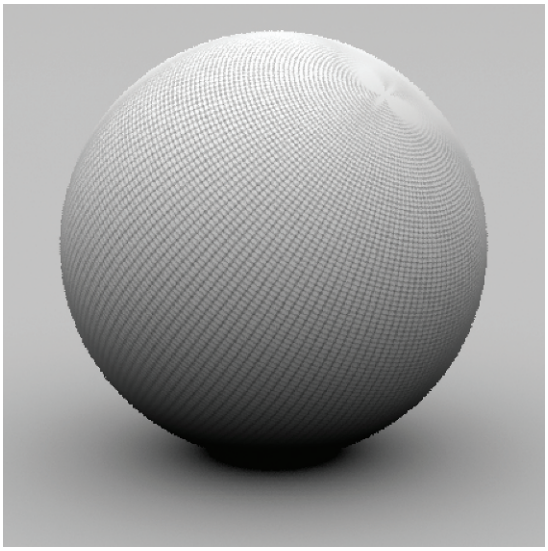
Textures may be a source of high-frequency variation in the final image. Figure 10.1 shows an image with severe aliasing due to a texture. Although the visual impact of this aliasing can be reduced with the nonuniform sampling techniques from Chapter 7, a better solution to this problem is to implement texture functions that adjust their frequency content based on the rate at which they are being sampled. For many texture functions, computing a reasonable approximation to the frequency content and antialiasing in this manner aren't too difficult and are substantially more efficient than reducing aliasing by increasing the image sampling rate.



(a)



(b)



(c)

Figure 10.1: Texture Aliasing. (a) An image of a grid texture on a sphere with one sample per pixel has severe aliasing artifacts. (b) A zoomed-in area from near the top of the sphere gives a sense of how much high-frequency detail is present between adjacent pixel sample positions. (c) The texture function has taken into account the image sampling rate to prefilter its function and remove high-frequency detail, resulting in an antialiased image, even with a single sample per pixel.

The first section of this chapter will discuss the problem of texture aliasing and general approaches to solving it. We will then describe the basic texture interface and illustrate its use with a few simple texture functions. Throughout the remainder of the chapter, we will present a variety of more complex texture implementations, demonstrating the use of a number of different texture antialiasing techniques along the way.

10.1 SAMPLING AND ANTIALIASING

The sampling task from Chapter 7 is a frustrating one since the aliasing problem was known to be unsolvable from the start. The infinite frequency content of geometric edges and hard shadows *guarantees* aliasing in the final images, no matter how high the image sampling rate. (Although the visual impact of this remaining aliasing can be reduced to be unobjectionable with a sufficient number of well-placed samples.) Fortunately, for textures things are not this difficult from the start: either there is often a convenient analytic form of the texture function available, which makes it possible to remove excessively high frequencies before sampling it, or it is possible to be careful when evaluating the function so as not to introduce high frequencies in the first place. When this problem is carefully addressed in texture implementations, as is done through the rest of this chapter, there is usually no need for more than one sample per pixel in order to render an image without texture aliasing.

Two problems must be addressed in order to remove aliasing from texture functions:

1. The sampling rate in texture space must be computed. The screen space sampling rate is known from the image resolution and pixel sampling rate, but here we need to determine the resulting sampling rate on a surface in the scene in order to find the rate at which the texture function is being sampled.
2. Given the texture sampling rate, sampling theory must be applied to guide the computation of a texture value that doesn't have higher-frequency variation than can be represented by the sampling rate (e.g., by removing excess frequencies beyond the Nyquist limit from the texture function).

These two issues will be addressed in turn throughout the rest of this section.

10.1.1 FINDING THE TEXTURE SAMPLING RATE

Consider an arbitrary texture function that is a function of position, $T(\mathrm{p})$, defined on a surface in the scene. If we ignore the complications introduced by visibility issues—the possibility that another object may occlude the surface at nearby image samples, or that the surface may have a limited extent on the image plane—this texture function can also be expressed as a function over points (x, y) on the image plane, $T(f(x, y))$, where $f(x, y)$ is the function that maps image points to points on the surface. Thus, $T(f(x, y))$ gives the value of the texture function as seen at image position (x, y).

As a simple example of this idea, consider a 2D texture function $T(s, t)$ applied to a quadrilateral that is perpendicular to the z axis and has corners at the world space points (0, 0, 0), (1, 0, 0), (1, 1, 0), and (0, 1, 0). If an orthographic camera is placed looking down the z axis such that the quadrilateral precisely fills the image plane and if points p

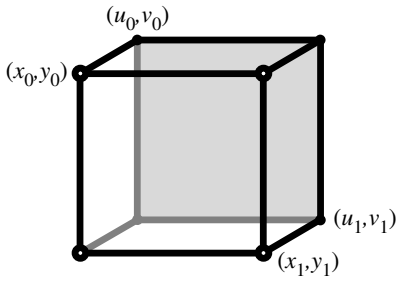


Figure 10.2: If a quadrilateral is viewed with an orthographic perspective such that the quadrilateral precisely fills the image plane, it's easy to compute the relationship between the sampling rate in (x, y) pixel coordinates and the texture sampling rate.

on the quadrilateral are mapped to 2D (s, t) texture coordinates by

$$s = \mathrm{p}_x \qquad t = \mathrm{p}_y,$$

then the relationship between (s, t) and screen (x, y) pixels is straightforward:

$$s = \frac{x}{x_r} \qquad t = \frac{y}{y_r},$$

where the overall image resolution is (x_r, y_r) (Figure 10.2). Thus, given a sample spacing of one pixel in the image plane, the sample spacing in (s, t) texture parameter space is $(1/x_r, 1/y_r)$, and the texture function must remove any detail at a higher frequency than can be represented at that sampling rate.

This relationship between pixel coordinates and texture coordinates, and thus the relationship between their sampling rates, is the key bit of information that determines the maximum frequency content allowable in the texture function. As a slightly more complex example, given a triangle with (s, t) texture coordinates at the vertices and viewed with a perspective projection, it's possible to analytically find the differences in s and t across the sample points on the image plane. This is the basis of basic texture map antialiasing in specialized graphics hardware.

For more complex scene geometry, camera projections, and mappings to texture coordinates, it is much more difficult to precisely determine the relationship between image positions and texture parameter values. Fortunately, for texture antialiasing, we don't need to be able to evaluate $f(x, y)$ for arbitrary (x, y), but just need to find the relationship between changes in pixel sample position and the resulting change in texture sample position at a particular point on the image. This relationship is given by the partial derivatives of this function, $\partial f/\partial x$ and $\partial f/\partial y$. For example, these can be used to find a first-order approximation to the value of f,

$$f(x', y') \approx f(x, y) + (x' - x)\frac{\partial f}{\partial x} + (y' - y)\frac{\partial f}{\partial y}.$$

If these partial derivatives are changing slowly with respect to the distances $x' - x$ and $y' - y$, this is a reasonable approximation. More importantly, the values of these partial derivatives give an approximation to the change in texture sample position for a shift of one pixel in the x and y directions, respectively, and thus directly yield the texture sampling rate. For example, in the previous quadrilateral example, $\partial s/\partial x = 1/x_r$, $\partial s/\partial y = 0$, $\partial t/\partial x = 0$, and $\partial t/\partial y = 1/y_r$.

The key to finding the values of these partial derivatives in the general case lies in the `RayDifferential` structure, which was defined in Section 2.5.1. This structure is initialized for each camera ray by the `Camera::GenerateRayDifferential()` method; it contains not only the ray actually being traced through the scene, but also two additional rays, one offset horizontally one pixel sample from the camera ray and the other offset vertically by one pixel sample. All of the geometric ray intersection routines use only the main camera ray for their computations; the auxiliary rays are ignored (this is easy to do because `RayDifferential` is a subclass of `Ray`).

Here we will use the offset rays to estimate the partial derivatives of the mapping $\mathrm{p}(x, y)$ from image position to world space position and the partial derivatives of the mappings $u(x, y)$ and $v(x, y)$ from (x, y) to (u, v) parametric coordinates, giving the partial derivatives of world space positions $\partial \mathrm{p}/\partial x$ and $\partial \mathrm{p}/\partial y$ and the partial derivatives of (u, v) parametric coordinates $\partial u/\partial x$, $\partial v/\partial x$, $\partial u/\partial y$, and $\partial v/\partial y$. In Section 10.2, we will see how these can be used to compute the screen space derivatives of arbitrary quantities based on p or (u, v) and consequently the sampling rates of these quantities. The values of these partial derivatives at the intersection point are stored in the `DifferentialGeometry` structure. They are declared as `mutable`, since they are set in a method that takes a `const` differential geometry object.

Camera::
GenerateRayDifferential() 303
DifferentialGeometry 102
DifferentialGeometry::
ComputeDifferentials() 505
DifferentialGeometry::
dpdx 505
DifferentialGeometry::
dpdy 505
DifferentialGeometry::
dudx 505
DifferentialGeometry::
dudy 505
DifferentialGeometry::
dvdx 505
DifferentialGeometry::
dvdy 505
Intersection::GetBSDF() 484
Intersection::
GetBSSRDF() 484
Material 483
Ray 66
RayDifferential 69
RayDifferential::
hasDifferentials 69
Vector 57

⟨*DifferentialGeometry Public Data*⟩ +≡ 102

```
mutable Vector dpdx, dpdy;
mutable float dudx, dvdx, dudy, dvdy;
```

⟨*Initialize* `DifferentialGeometry` *from parameters*⟩ +≡ 102

```
dudx = dvdx = dudy = dvdy = 0;
```

The `DifferentialGeometry::ComputeDifferentials()` method computes these values. It is called by `Intersection::GetBSDF()` and `Intersection::GetBSSRDF()` before the `Material` `GetBSDF()` or `GetBSSRDF()` method is called so that these values will be available for any texture evaluation routines that are called by the material. Because ray differentials aren't available for all rays traced by the system (e.g., rays starting from light sources traced for photon mapping), the `hasDifferentials` field of the `RayDifferential` must be checked before these computations are done. If the differentials are not present, then the derivatives are all assumed to be zero (which will eventually lead to unfiltered point sampling of textures).

⟨*DifferentialGeometry Method Definitions*⟩ +≡

```
void DifferentialGeometry::ComputeDifferentials(
        const RayDifferential &ray) const {
    if (ray.hasDifferentials) {
        ⟨Estimate screen space change in p and (u, v) 507⟩
    }
```

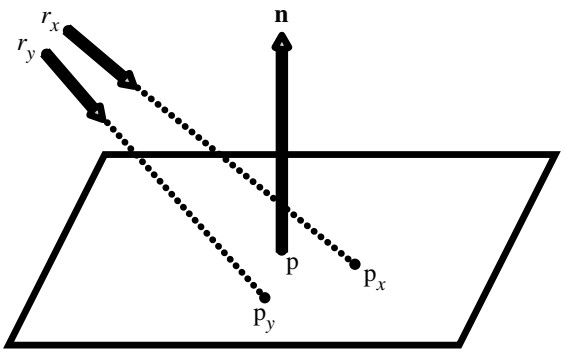


Figure 10.3: By approximating the local surface geometry at the intersection point with the tangent plane through p, approximations to the points at which the auxiliary rays r_x and r_y would intersect the surface can be found by finding their intersection points with the tangent plane p_x and p_y.

```
    else {
        dudx = dvdx = 0.;
        dudy = dvdy = 0.;
        dpdx = dpdy = Vector(0,0,0);
    }
}
```

The key to computing these estimates is the assumption that the surface is locally flat with respect to the sampling rate at the point being shaded. This is a reasonable approximation in practice, and it is hard to do much better. Because ray tracing is a point-sampling technique, we have no additional information about the scene in between the rays we traced. For highly curved surfaces or at silhouette edges, this approximation can break down, though it is rarely a source of noticeable error in practice. For this approximation, we need the plane through the point intersected by the main ray that is tangent to the surface. This plane is given by the implicit equation

$$ax + by + cz + d = 0,$$

where $a = \mathbf{n}_x$, $b = \mathbf{n}_y$, $c = \mathbf{n}_z$, and $d = -(\mathbf{n} \cdot \mathrm{p})$. We can then compute the intersection points p_x and p_y between the auxiliary rays r_x and r_y with this plane (Figure 10.3). These new points give an approximation to the partial derivatives of position on the surface $\partial \mathrm{p}/\partial x$ and $\partial \mathrm{p}/\partial y$, based on forward differences:

$$\frac{\partial \mathrm{p}}{\partial x} \approx \mathrm{p}_x - \mathrm{p}, \quad \frac{\partial \mathrm{p}}{\partial y} \approx \mathrm{p}_y - \mathrm{p}.$$

Because the differential rays are offset one pixel sample in each direction, there's no need to divide these differences by a Δ value, since $\Delta = 1$.

⟨*Estimate screen space change in* p *and* (u, v)⟩ ≡ 505
⟨*Compute auxiliary intersection points with plane* 507⟩
```
dpdx = px - p;
dpdy = py - p;
```
⟨*Compute* (u, v) *offsets at auxiliary points* 508⟩

The ray–plane intersection algorithm gives the t value where a ray described by origin o and direction **d** intersects a plane described by $ax + by + cz + d = 0$:

$$t = \frac{-((a, b, c) \cdot \mathrm{o}) + d}{(a, b, c) \cdot \mathbf{d}}.$$

To compute this value for the two auxiliary rays, the plane's d coefficient is computed first. It isn't necessary to compute the a, b, and c coefficients, since they're available in `dg.nn`. We can then apply the formula directly.

⟨*Compute auxiliary intersection points with plane*⟩ ≡ 507
```
float d = -Dot(nn, Vector(p.x, p.y, p.z));
Vector rxv(ray.rxOrigin.x, ray.rxOrigin.y, ray.rxOrigin.z);
float tx = -(Dot(nn, rxv) + d) / Dot(nn, ray.rxDirection);
Point px = ray.rxOrigin + tx * ray.rxDirection;
Vector ryv(ray.ryOrigin.x, ray.ryOrigin.y, ray.ryOrigin.z);
float ty = -(Dot(nn, ryv) + d) / Dot(nn, ray.ryDirection);
Point py = ray.ryOrigin + ty * ray.ryDirection;
```

Using the positions p_x and p_y, an approximation to their respective (u, v) coordinates can be found by taking advantage of the fact that the surface's partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ form a (not necessarily orthogonal) coordinate system on the plane and that the coordinates of the auxiliary intersection points in terms of this coordinate system are their coordinates with respect to the (u, v) parameterization (Figure 10.4). Given

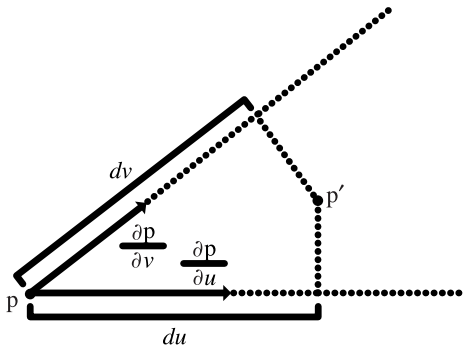


Figure 10.4: An estimate of the difference in (u, v) parametric coordinates from p to p′ can be found by finding the coordinates of p′ with respect to the coordinate system defined by p, $\partial \mathrm{p}/\partial u$, and $\partial \mathrm{p}/\partial v$.

DifferentialGeometry::dpdx 505
DifferentialGeometry::dpdy 505
DifferentialGeometry::nn 102
DifferentialGeometry::p 102
Dot() 60
Point 63
RayDifferential::rxOrigin 69
Vector 57

a position p' on the plane, we can compute its position with respect to the coordinate system by

$$\mathrm{p}' = \mathrm{p} + \Delta_u \frac{\partial \mathrm{p}}{\partial u} + \Delta_v \frac{\partial \mathrm{p}}{\partial v},$$

or, equivalently,

$$\begin{pmatrix} \mathrm{p}'_x - \mathrm{p}_x \\ \mathrm{p}'_y - \mathrm{p}_y \\ \mathrm{p}'_z - \mathrm{p}_z \end{pmatrix} = \begin{pmatrix} \partial \mathrm{p}_x/\partial u & \partial \mathrm{p}_x/\partial v \\ \partial \mathrm{p}_y/\partial u & \partial \mathrm{p}_y/\partial v \\ \partial \mathrm{p}_z/\partial u & \partial \mathrm{p}_z/\partial v \end{pmatrix} \begin{pmatrix} \Delta_u \\ \Delta_v \end{pmatrix}.$$

A solution to this linear system of equations for one of the auxiliary points $\mathrm{p}' = \mathrm{p}_x$ or $\mathrm{p}' = \mathrm{p}_y$ gives the corresponding screen space partial derivatives $(\partial u/\partial x, \partial v/\partial x)$ or $(\partial u/\partial y, \partial v/\partial y)$, respectively.

This linear system has three equations with two unknowns—that is, it's overconstrained. We need to be careful since one of the equations may be degenerate—for example, if $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ are in the xy plane such that their z components are both zero, then the third equation will be degenerate. To deal with this case, because we only need two equations to solve the system, we'd like to choose the two that won't lead to a degenerate system. An easy way to do this is to take the cross product of $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$, see which coordinate of the result has the largest magnitude, and use the other two. Their cross product is already available in `nn`, so using this approach is straightforward. Even after all this, it may happen that the linear system has no solution (usually due to the partial derivatives not forming a coordinate system on the plane). In that case, all that can be done is to return arbitrary values.

⟨*Compute* (u, v) *offsets at auxiliary points*⟩ ≡ 507

```
⟨Initialize A, Bx, and By matrices for offset computation 508⟩
if (!SolveLinearSystem2x2(A, Bx, &dudx, &dvdx)) {
    dudx = 0.; dvdx = 0.;
}
if (!SolveLinearSystem2x2(A, By, &dudy, &dvdy)) {
    dudy = 0.; dvdy = 0.;
}
```

⟨*Initialize* `A`, `Bx`, *and* `By` *matrices for offset computation*⟩ ≡ 508

```
float A[2][2], Bx[2], By[2];
int axes[2];
if (fabsf(nn.x) > fabsf(nn.y) && fabsf(nn.x) > fabsf(nn.z)) {
    axes[0] = 1; axes[1] = 2;
}
else if (fabsf(nn.y) > fabsf(nn.z)) {
    axes[0] = 0; axes[1] = 2;
}
else {
    axes[0] = 0; axes[1] = 1;
}
⟨Initialize matrices for chosen projection plane 509⟩
```

DifferentialGeometry::dudx 505
DifferentialGeometry::dudy 505
DifferentialGeometry::dvdx 505
DifferentialGeometry::dvdy 505
DifferentialGeometry::nn 102
SolveLinearSystem2x2() 1020

⟨*Initialize matrices for chosen projection plane*⟩ ≡ 508

```
A[0][0] = dpdu[axes[0]];
A[0][1] = dpdv[axes[0]];
A[1][0] = dpdu[axes[1]];
A[1][1] = dpdv[axes[1]];
Bx[0] = px[axes[0]] - p[axes[0]];
Bx[1] = px[axes[1]] - p[axes[1]];
By[0] = py[axes[0]] - p[axes[0]];
By[1] = py[axes[1]] - p[axes[1]];
```

10.1.2 FILTERING TEXTURE FUNCTIONS

It is necessary to remove frequencies in texture functions that are past the Nyquist limit for the texture sampling rate. The goal is to compute, with as few approximations as possible, the result of the *ideal texture resampling* process, which says that in order to evaluate $T(f(x, y))$ without aliasing, we must first band-limit it, removing frequencies beyond the Nyquist limit by convolving it with the sinc filter:

$$T'_b(x, y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \operatorname{sinc}(x')\operatorname{sinc}(y')T'\left(f(x + x', y + y')\right) \mathrm{d}x'\mathrm{d}y'.$$

The band-limited function in turn should then be convolved with the pixel filter $g(x, y)$ centered at the (x, y) point on the screen at which we want to evaluate the texture function:

$$T'_f(x, y) = \int_{-\mathrm{xWidth}/2}^{\mathrm{xWidth}/2}\int_{-\mathrm{yWidth}/2}^{\mathrm{yWidth}/2} g(x', y')T'_b(x + x', y + y')\mathrm{d}x'\mathrm{d}y'.$$

This gives the theoretically perfect value for the texture as projected onto the screen.[1]

In practice, there are many simplifications that can be made to this process, with little reduction in visual quality. For example, a box filter may be used for the band-limiting step, and the second step is usually ignored completely, effectively acting as if the pixel filter were a box filter, which makes it possible to do the antialiasing work completely in texture space and simplifies the implementation significantly. The EWA filtering algorithm in Section 10.4.4 is a notable exception that doesn't make all of these simplifications.

Even the box filter, with all of its shortcomings, gives acceptable results for texture filtering in many cases. The box filter can be particularly easy to use, since it can be applied analytically by computing the average of the texture function over the appropriate region. Intuitively, this is a reasonable approach to the texture filtering problem, and it can be computed directly for many texture functions. Indeed, through the rest of this chapter, we will often use a box filter to average texture function values between samples and

DifferentialGeometry::
dpdu 102

DifferentialGeometry::
dpdv 102

DifferentialGeometry::p 102

MatteMaterial 484

1 One simplification that is present in this ideal filtering process is the implicit assumption that the texture function makes a linear contribution to frequency content in the image, so that filtering out its high frequencies removes high frequencies from the image. This is true for many uses of textures—for example, if an image map is used to modulate the diffuse term of a `MatteMaterial`. However, if a texture is used to determine the roughness of a glossy specular object, for example, this linearity assumption is incorrect, since the roughness value is both inverted and used as an exponent in a microfacet BRDF. We will ignore this issue here, since it isn't easily solved in general and usually doesn't cause substantial errors. The "Further Reading" section has more discussion of this issue.

informally use the term "filter region" to describe the area being averaged over. This is the most common approach when filtering texture functions.

An alternative to using the box filter to filter texture functions is to use the observation that the effect of the ideal sinc filter is to let frequency components below the Nyquist limit pass through unchanged but to remove frequencies past it. Therefore, if we know the frequency content of the texture function (e.g., if it is a sum of terms, each one with known frequency content), then if we replace the high-frequency terms with their average values, we are effectively doing the work of the sinc prefilter. This approach is known as *clamping* and is the basis for antialiasing in the textures based on the noise function in Section 10.6.

Finally, for texture functions where none of these techniques is easily applied, the approach of last resort is *supersampling*—the function is evaluated and filtered at multiple locations near the main evaluation point, thus increasing the sampling rate in texture space. If a box filter is used to filter these sample values, this is equivalent to averaging the value of the function. This approach can be expensive if the texture function is complex to evaluate, and as with image sampling a very large number of samples may be needed to remove aliasing. Although this is a brute-force solution, it is still more efficient than increasing the image sampling rate, since it doesn't incur the cost of tracing more rays through the scene.

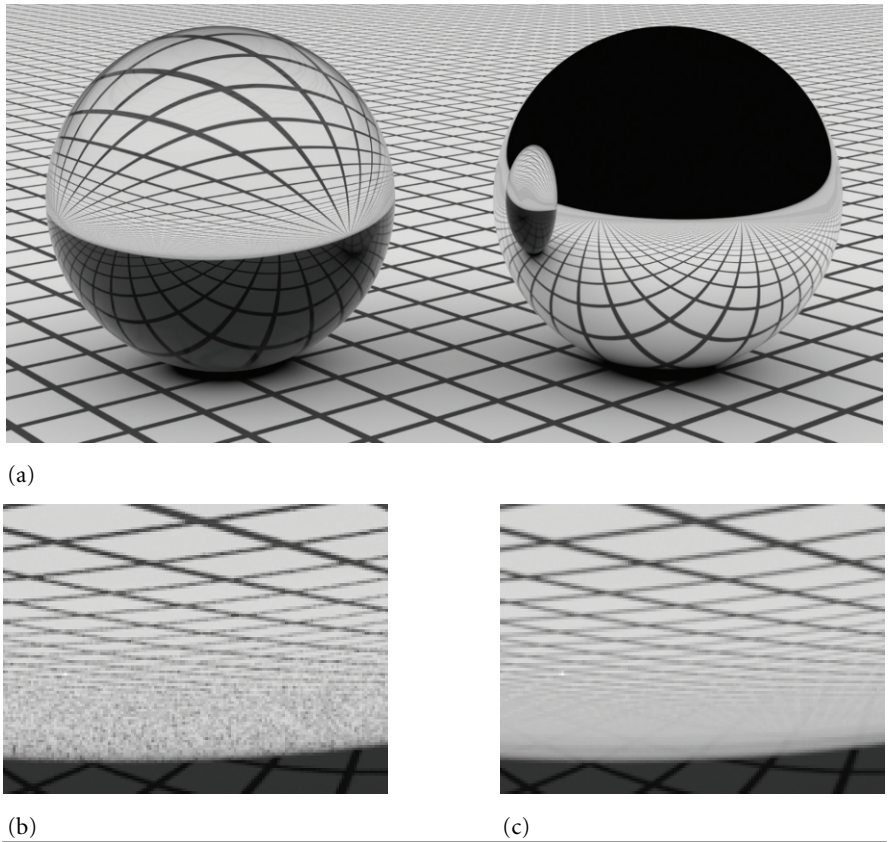
★ 10.1.3 RAY DIFFERENTIALS FOR SPECULAR REFLECTION AND TRANSMISSION

Given the effectiveness of ray differentials for finding filter regions for texture antialiasing for camera rays, it is useful to extend the method to make it possible to determine texture space sampling rates for objects that are seen indirectly via specular reflection or refraction; objects seen in mirrors, for example, should also no more have texture aliasing than directly visible objects. Igehy (1999) developed an elegant solution to the problem of how to find the appropriate differential rays for specular reflection and refraction, which is the approach used in `pbrt`.[2]

Figure 10.5 illustrates the difference that proper texture filtering for specular reflection and transmission can make. Figure 10.5(a) shows a glass ball and a mirrored ball on a plane with a texture map containing high-frequency components. Ray differentials ensure that the images of the texture seen via reflection and refraction from the balls are free of aliasing artifacts. A close-up view of the reflection in the glass ball is shown in Figure 10.5(b) and (c); Figure 10.5(b) was rendered without ray differentials for the reflected and transmitted rays, and Figure 10.5(c) was rendered with ray differentials. The aliasing errors in the left image are eliminated on the right without excessively blurring the texture.

In order to compute the reflected or transmitted ray differentials at a surface intersection point, we need an approximation to the rays that would have been traced at the intersec-

2 Igehy's formulation is slightly different than the one here—he effectively tracked the differences between the main ray and the offset rays, while we store the offset rays explicitly. The mathematics all work out to be the same in the end; we chose this alternative because we believe that it makes the algorithm's operation for camera rays easier to understand.



(a) (b) (c)

Figure 10.5: (a) Tracking ray differentials for reflected and refracted rays ensures that the image map texture seen in the balls is filtered to avoid aliasing. The left ball is glass, exhibiting reflection and refraction, and the right ball is a mirror, just showing reflection. Note that the texture is well filtered over both of the balls. (b) and (c) show a zoomed-in section of the glass ball; (b) shows the aliasing artifacts that are present if ray differentials aren't used, while (c) shows the result when they are.

tion points for the two offset rays in the ray differential that hit the surface (Figure 10.6). The new ray for the main ray is computed by the BSDF, so here we only need to compute the outgoing rays for the r_x and r_y differentials.

For both reflection and refraction, the origin of each differential ray is easily found. The `DifferentialGeometry::ComputeDifferentials()` method previously computed approximations for how much the surface position changes with respect to (x, y) position on the image plane $\partial \mathrm{p}/\partial x$ and $\partial \mathrm{p}/\partial y$. Adding these offsets to the intersection point of the main ray gives approximate origins for the new rays. If the incident ray doesn't have differentials, then it's impossible to compute reflected ray differentials and this step is skipped.

DifferentialGeometry::
ComputeDifferentials() 505

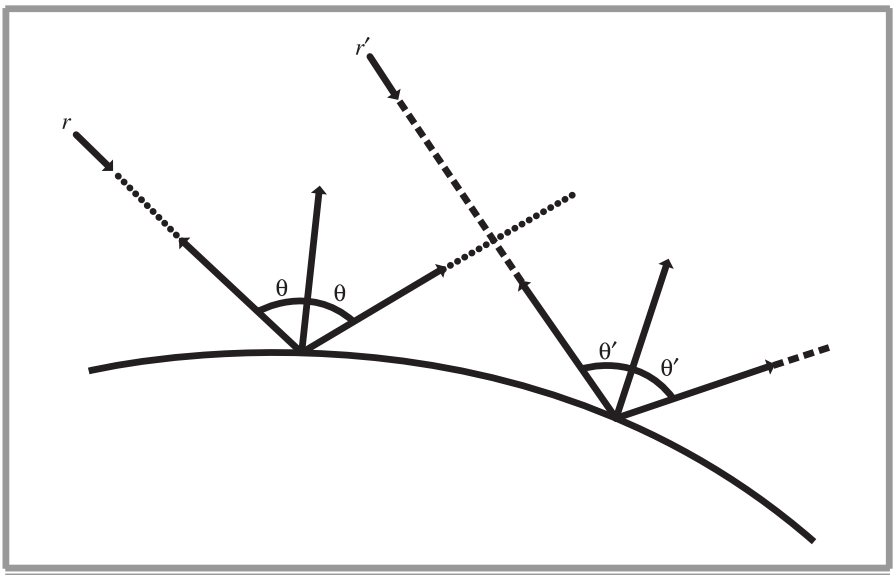


Figure 10.6: The specular reflection formula gives the direction of the reflected ray at a point on a surface. An offset ray for a ray differential (dashed line) will generally intersect the surface at a different point and be reflected in a different direction. The new direction is affected by both the different surface normal at the point as well as the offset ray's different incident direction. The computation to find the reflected direction for the offset ray in `pbrt` estimates the change in reflected direction as a function of image space position and approximates the ray differential's direction with the main ray's direction added to the estimated change in direction.

⟨*Compute ray differential* `rd` *for specular reflection*⟩ ≡ 46

```
RayDifferential rd(p, wi, ray, isect.rayEpsilon);
if (ray.hasDifferentials) {
    rd.hasDifferentials = true;
    rd.rxOrigin = p + isect.dg.dpdx;
    rd.ryOrigin = p + isect.dg.dpdy;
    ⟨Compute differential reflected directions 513⟩
}
```

Finding the directions of these rays is slightly more tricky. Igehy observed that if we know how much the reflected direction ω_i changes with respect to a shift of a pixel sample in the x and y directions on the image plane, we can use this information to approximate the direction of the offset rays:

$$\omega \approx \omega_i + \frac{\partial \omega_i}{\partial x}.$$

For a general world space surface normal and outgoing direction, the direction for perfect specular reflection is

$$\omega_i = -\omega_o + 2(\omega_o \cdot \mathbf{n})\mathbf{n}.$$

Fortunately, the partial derivatives of this expression are easily computed:

DifferentialGeometry::
dpdx 505

DifferentialGeometry::
dpdy 505

Intersection::dg 186

Intersection::rayEpsilon 186

RayDifferential 69

RayDifferential::
hasDifferentials 69

$$\frac{\partial \omega_i}{\partial x} = \frac{\partial}{\partial x}\left(-\omega_o + 2(\omega_o \cdot \mathbf{n})\mathbf{n}\right)$$
$$= -\frac{\partial \omega_o}{\partial x} + 2\left((\omega_o \cdot \mathbf{n})\frac{\partial \mathbf{n}}{\partial x} + \frac{\partial(\omega_o \cdot \mathbf{n})}{\partial x}\mathbf{n}\right).$$

Using the properties of the dot product, it can be shown that

$$\frac{\partial(\omega_o \cdot \mathbf{n})}{\partial x} = \frac{\partial \omega_o}{\partial x} \cdot \mathbf{n} + \omega_o \cdot \frac{\partial \mathbf{n}}{\partial x}.$$

The value of $\partial\omega_o/\partial x$ can be found from the difference between the direction of the ray differential's main ray and the direction of the r_x offset ray, and all of the other necessary quantities are readily available from the `DifferentialGeometry`, so the implementation of this computation for the partial derivatives in x and y is straightforward.

⟨*Compute differential reflected directions*⟩ ≡ 512

```
Normal dndx = bsdf->dgShading.dndu * bsdf->dgShading.dudx +
              bsdf->dgShading.dndv * bsdf->dgShading.dvdx;
Normal dndy = bsdf->dgShading.dndu * bsdf->dgShading.dudy +
              bsdf->dgShading.dndv * bsdf->dgShading.dvdy;
Vector dwodx = -ray.rxDirection - wo, dwody = -ray.ryDirection - wo;
float dDNdx = Dot(dwodx, n) + Dot(wo, dndx);
float dDNdy = Dot(dwody, n) + Dot(wo, dndy);
rd.rxDirection = wi - dwodx + 2 * Vector(Dot(wo, n) * dndx +
                                         dDNdx * n);
rd.ryDirection = wi - dwody + 2 * Vector(Dot(wo, n) * dndy +
                                         dDNdy * n);
```

A similar process of differentiating the equation for the direction of a specularly transmitted ray gives the equation to find the differential change in the transmitted direction. We won't include the derivation or our implementation here, but refer the interested reader to the original paper and to the source code, respectively.

10.2 TEXTURE COORDINATE GENERATION

DifferentialGeometry::
dndu 102

DifferentialGeometry::
dndv 102

DifferentialGeometry::
dudx 505

DifferentialGeometry::
dudy 505

DifferentialGeometry::
dvdx 505

DifferentialGeometry::
dvdy 505

Normal 65

TextureMapping2D 514

TextureMapping3D 519

Vector 57

Almost all of the textures in this chapter are functions that take a two-dimensional or three-dimensional coordinate and return a texture value. Sometimes there are obvious ways to choose these texture coordinates; for parametric surfaces, such as the quadrics in Chapter 3, there is a natural two-dimensional (u, v) parameterization of the surface, and for all surfaces the shading point p is a natural choice for a three-dimensional coordinate.

There is often no natural parameterization of complex surfaces, or the natural parameterization may be undesirable. For instance, the (u, v) values near the poles of spheres are severely distorted. Also, for an arbitrary subdivision surface, there is no simple, general-purpose way to assign texture values so that the entire $[0, 1]^2$ space is covered continuously and without distortion. In fact, creating smooth parameterizations of complex meshes with low distortion is an active area of research in computer graphics.

This section starts by introducing two abstract base classes—`TextureMapping2D` and `TextureMapping3D`—that provide an interface for computing these 2D and 3D texture

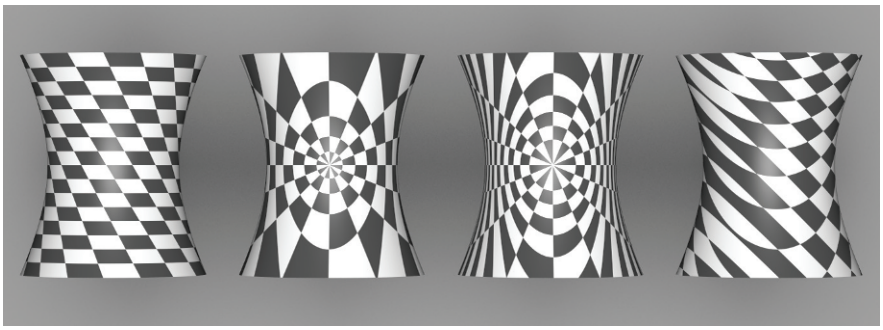


Figure 10.7: A checkerboard texture, applied to a hyperboloid with different texture coordinate generation techniques. From left to right, (u, v) mapping, spherical mapping, cylindrical mapping, and planar mapping.

coordinates. We will then implement a number of standard mappings using this interface (Figure 10.7 shows a number of them). `Texture` implementations store a pointer to a 2D or 3D mapping function as appropriate and use it to compute the texture coordinates at each point. Thus, it's easy to add new mappings to the system without having to modify all of the `Texture` implementations, and different mappings can be used for different textures associated with the same surface. In `pbrt`, we will use the convention that 2D texture coordinates are denoted by (s, t); this helps make clear the distinction between the intrinsic (u, v) parameterization of the underlying surface and the (possibly different) coordinate values used for texturing.

The `TextureMapping2D` base class has a single method, `TextureMapping2D::Map()`, which is given the `DifferentialGeometry` at the shading point and returns the (s, t) texture coordinates via `float` pointers. It also returns estimates for the change in s and t with respect to pixel x and y coordinates in the `dsdx`, `dtdx`, `dsdy`, and `dtdy` parameters so that textures that use the mapping can determine the (s, t) sampling rate and filter accordingly.

⟨*Texture Declarations*⟩ ≡
```
class TextureMapping2D {
public:
    ⟨TextureMapping2D Interface 514⟩
};
```

⟨*TextureMapping2D Interface*⟩ ≡ 514
```
virtual void Map(const DifferentialGeometry &dg,
                 float *s, float *t, float *dsdx, float *dtdx,
                 float *dsdy, float *dtdy) const = 0;
```

DifferentialGeometry 102
Texture 519
TextureMapping2D 514
TextureMapping2D::Map() 514

10.2.1 2D (u, v) MAPPING

The simplest mapping uses the 2D parametric (u, v) coordinates in the `DifferentialGeometry` to compute the texture coordinates. Their values can be offset and scaled with user-supplied values in each dimension.

⟨*Texture Declarations*⟩ +≡
```
class UVMapping2D : public TextureMapping2D {
public:
    ⟨UVMapping2D Public Methods⟩
private:
    float su, sv, du, dv;
};
```

⟨*Texture Method Definitions*⟩ ≡
```
UVMapping2D::UVMapping2D(float ssu, float ssv, float ddu, float ddv)
    : su(ssu), sv(ssv), du(ddu), dv(ddv) { }
```

The scale-and-shift computation to compute (s, t) coordinates is straightforward:

⟨*Texture Method Definitions*⟩ +≡
```
void UVMapping2D::Map(const DifferentialGeometry &dg,
                      float *s, float *t, float *dsdx, float *dtdx,
                      float *dsdy, float *dtdy) const {
    *s = su * dg.u + du;
    *t = sv * dg.v + dv;
    ⟨Compute texture differentials for 2D identity mapping 515⟩
}
```

Computing the differential change in s and t in terms of the original change in u and v and the scale amounts is also easy. Using the chain rule,

$$\frac{\partial s}{\partial x} = \frac{\partial u}{\partial x}\frac{\partial s}{\partial u} + \frac{\partial v}{\partial x}\frac{\partial s}{\partial v}$$

and similarly for the three other partial derivatives. From the mapping method,

$$s = s_u u + d_u,$$

so

$$\frac{\partial s}{\partial u} = s_u, \qquad \frac{\partial s}{\partial v} = 0,$$

and thus

$$\frac{\partial s}{\partial x} = s_u \frac{\partial u}{\partial x},$$

and so forth.

DifferentialGeometry 102
DifferentialGeometry::u 102
DifferentialGeometry::v 102
SphericalMapping2D 516
TextureMapping2D 514
UVMapping2D 515
UVMapping2D::du 515
UVMapping2D::dv 515
UVMapping2D::su 515
UVMapping2D::sv 515

⟨*Compute texture differentials for 2D identity mapping*⟩ ≡ 515
```
*dsdx = su * dg.dudx;
*dtdx = sv * dg.dvdx;
*dsdy = su * dg.dudy;
*dtdy = sv * dg.dvdy;
```

10.2.2 SPHERICAL MAPPING

Another useful mapping effectively wraps a sphere around the object. Each point is projected along the vector from the sphere's center through the point, up to the sphere's surface. There, the (u, v) mapping for the sphere shape is used. The SphericalMapping2D

stores a transformation that is applied to points before this mapping is performed; this effectively allows the mapping sphere to be arbitrarily positioned and oriented with respect to the object.

⟨*Texture Declarations*⟩ +≡
```
class SphericalMapping2D : public TextureMapping2D {
public:
    ⟨SphericalMapping2D Public Methods⟩
private:
    void sphere(const Point &P, float *s, float *t) const;
    Transform WorldToTexture;
};
```

⟨*Texture Method Definitions*⟩ +≡
```
void SphericalMapping2D::Map(const DifferentialGeometry &dg,
        float *s, float *t, float *dsdx, float *dtdx,
        float *dsdy, float *dtdy) const {
    sphere(dg.p, s, t);
    ⟨Compute texture coordinate differentials for sphere (u, v) mapping 517⟩
}
```

A short utility function computes the mapping for a single point. It will be useful to have this logic separated out for computing texture coordinate differentials.

⟨*Texture Method Definitions*⟩ +≡
```
void SphericalMapping2D::sphere(const Point &p, float *s, float *t) const {
    Vector vec = Normalize(WorldToTexture(p) - Point(0,0,0));
    float theta = SphericalTheta(vec);
    float phi = SphericalPhi(vec);
    *s = theta * INV_PI;
    *t = phi * INV_TWOPI;
}
```

We could use the chain rule again to compute the texture coordinate differentials, but will instead use a forward differencing approximation to demonstrate another way to compute these values that is useful for more complex mapping functions. Recall that the `DifferentialGeometry` stores the screen space partial derivatives $\partial \mathrm{p}/\partial x$ and $\partial \mathrm{p}/\partial y$ that give the change in position as a function of change in image sample position. Therefore, if the s coordinate is computed by some function $f_s(\mathrm{p})$, it's easy to compute approximations like

$$\frac{\partial s}{\partial x} \approx \frac{f_s(\mathrm{p} + \Delta \partial \mathrm{p}/\partial x) - f_s(\mathrm{p})}{\Delta}.$$

As the distance Δ approaches 0, this gives the actual partial derivative at p.

One other detail is that the sphere mapping has a discontinuity in the mapping formula; there is a seam at $t = 1$, where the t texture coordinate discontinuously jumps back to zero. We can detect this case by checking to see if the value computed with forward differencing is greater than 0.5 and then adjusting it appropriately.

DifferentialGeometry 102
DifferentialGeometry::p 102
INV_PI 1002
INV_TWOPI 1002
Point 63
SphericalMapping2D 516
SphericalMapping2D::sphere() 516
SphericalMapping2D::WorldToTexture 516
SphericalPhi() 292
SphericalTheta() 292
TextureMapping2D 514
Transform 76
Vector 57
Vector::Normalize() 63

⟨*Compute texture coordinate differentials for sphere* (*u*, *v*) *mapping*⟩ ≡ 516
```
float sx, tx, sy, ty;
const float delta = .1f;
sphere(dg.p + delta * dg.dpdx, &sx, &tx);
*dsdx = (sx - *s) / delta;
*dtdx = (tx - *t) / delta;
if (*dtdx > .5) *dtdx = 1.f - *dtdx;
else if (*dtdx < -.5f) *dtdx = -(*dtdx + 1);
sphere(dg.p + delta * dg.dpdy, &sy, &ty);
*dsdy = (sy - *s) / delta;
*dtdy = (ty - *t) / delta;
if (*dtdy > .5) *dtdy = 1.f - *dtdy;
else if (*dtdy < -.5f) *dtdy = -(*dtdy + 1);
```

10.2.3 CYLINDRICAL MAPPING

The cylindrical mapping effectively wraps a cylinder around the object. It also supports a transformation to orient the mapping cylinder.

⟨*Texture Declarations*⟩ +≡
```
class CylindricalMapping2D : public TextureMapping2D {
public:
    ⟨CylindricalMapping2D Public Methods⟩
private:
    ⟨CylindricalMapping2D Private Methods 517⟩
    Transform WorldToTexture;
};
```

The cylindrical mapping has the same basic structure as the sphere mapping; just the mapping function is different. Therefore, we will omit the fragment that computes texture coordinate differentials, since it is essentially the same as the spherical version.

CylindricalMapping2D 517
CylindricalMapping2D::cylinder() 517
CylindricalMapping2D::WorldToTexture 517
DifferentialGeometry 102
DifferentialGeometry::dpdx 505
DifferentialGeometry::dpdy 505
DifferentialGeometry::p 102
M_PI 1002
Point 63
TextureMapping2D 514
Transform 76
Vector 57
Vector::Normalize() 63

⟨*Texture Method Definitions*⟩ +≡
```
void CylindricalMapping2D::Map(const DifferentialGeometry &dg,
        float *s, float *t, float *dsdx, float *dtdx,
        float *dsdy, float *dtdy) const {
    cylinder(dg.p, s, t);
    ⟨Compute texture coordinate differentials for cylinder (u, v) mapping⟩
}
```

⟨*CylindricalMapping2D Private Methods*⟩ ≡ 517
```
void cylinder(const Point &p, float *s, float *t) const {
    Vector vec = Normalize(WorldToTexture(p) - Point(0,0,0));
    *s = (M_PI + atan2f(vec.y, vec.x)) / (2.f * M_PI);
    *t = vec.z;
}
```

10.2.4 PLANAR MAPPING

Another classic mapping method is planar mapping. The point is effectively projected onto a plane; a 2D parameterization of the plane then gives texture coordinates for the point. For example, a point p might be projected onto the $z = 0$ plane to yield texture coordinates given by $s = \mathrm{p}_x$ and $t = \mathrm{p}_y$.

In general, we can define such a parameterized plane with two nonparallel vectors $\mathbf{v}_s$ and $\mathbf{v}_t$ and offsets d_s and d_t. The texture coordinates are given by the coordinates of the point with respect to the plane's coordinate system, which are computed by taking the dot product of the vector from the point to the origin with each vector $\mathbf{v}_s$ and $\mathbf{v}_t$ and then adding the offset. For the example in the previous paragraph, we'd have $\mathbf{v}_s = (1, 0, 0)$, $\mathbf{v}_t = (0, 1, 0)$, and $d_s = d_t = 0$.

⟨*Texture Declarations*⟩ +≡
```
class PlanarMapping2D : public TextureMapping2D {
public:
    ⟨PlanarMapping2D Public Methods 518⟩
private:
    const Vector vs, vt;
    const float ds, dt;
};
```

⟨*PlanarMapping2D Public Methods*⟩ ≡ 518
```
PlanarMapping2D(const Vector &vv1, const Vector &vv2,
                float dds = 0, float ddt = 0)
    : vs(vv1), vt(vv2), ds(dds), dt(ddt) { }
```

The planar mapping differentials can be computed directly by finding the differentials of p in texture coordinate space.

⟨*Texture Method Definitions*⟩ +≡
```
void PlanarMapping2D::Map(const DifferentialGeometry &dg,
        float *s, float *t, float *dsdx, float *dtdx,
        float *dsdy, float *dtdy) const {
    Vector vec = dg.p - Point(0,0,0);
    *s = ds + Dot(vec, vs);
    *t = dt + Dot(vec, vt);
    *dsdx = Dot(dg.dpdx, vs);
    *dtdx = Dot(dg.dpdx, vt);
    *dsdy = Dot(dg.dpdy, vs);
    *dtdy = Dot(dg.dpdy, vt);
}
```

DifferentialGeometry 102
DifferentialGeometry::dpdx 505
DifferentialGeometry::dpdy 505
DifferentialGeometry::p 102
Dot() 60
PlanarMapping2D 518
PlanarMapping2D::ds 518
PlanarMapping2D::dt 518
PlanarMapping2D::vs 518
PlanarMapping2D::vt 518
Point 63
TextureMapping2D 514
TextureMapping3D 519
Vector 57

10.2.5 3D MAPPING

We will also define a `TextureMapping3D` class that defines the interface for generating three-dimensional texture coordinates.

⟨*Texture Declarations*⟩ +≡
```
class TextureMapping3D {
public:
    ⟨TextureMapping3D Interface 519⟩
};
```

⟨*TextureMapping3D Interface*⟩ ≡ 519
```
virtual Point Map(const DifferentialGeometry &dg,
                  Vector *dpdx, Vector *dpdy) const = 0;
```

The natural three-dimensional mapping just takes the world space coordinate of the point and applies a linear transformation to it. This will often be a transformation that takes the point back to the primitive's object space.

⟨*Texture Declarations*⟩ +≡
```
class IdentityMapping3D : public TextureMapping3D {
public:
    IdentityMapping3D(const Transform &x)
        : WorldToTexture(x) { }
    Point Map(const DifferentialGeometry &dg, Vector *dpdx,
              Vector *dpdy) const;
private:
    Transform WorldToTexture;
};
```

Because a linear mapping is used, the differential change in texture coordinates can be found by applying the same mapping to the partial derivatives of position.

⟨*Texture Method Definitions*⟩ +≡
```
Point IdentityMapping3D::Map(const DifferentialGeometry &dg,
                             Vector *dpdx, Vector *dpdy) const {
    *dpdx = WorldToTexture(dg.dpdx);
    *dpdy = WorldToTexture(dg.dpdy);
    return WorldToTexture(dg.p);
}
```

DifferentialGeometry 102
DifferentialGeometry::dpdx 505
DifferentialGeometry::p 102
IdentityMapping3D 519
IdentityMapping3D::WorldToTexture 519
Point 63
ReferenceCounted 1010
Spectrum 263
Texture 519
TextureMapping3D 519
Transform 76
Vector 57

10.3 TEXTURE INTERFACE AND BASIC TEXTURES

`Texture` is a template class parameterized by the return type of its evaluation function. This design makes it possible to reuse almost all of the texturing code between textures that return different types. `pbrt` currently uses only `float` and `Spectrum` textures.

⟨*Texture Declarations*⟩ +≡
```
template <typename T> class Texture : public ReferenceCounted {
public:
    ⟨Texture Interface 520⟩
};
```

The key to Texture's interface is its evaluation function; it returns a value of the template type T. The only information it has access to in order to evaluate its value is the DifferentialGeometry at the point being shaded. Different textures in this chapter will use different parts of this structure to drive their evaluation.

⟨*Texture Interface*⟩ ≡ 519
```
virtual T Evaluate(const DifferentialGeometry &) const = 0;
```

10.3.1 CONSTANT TEXTURE

ConstantTexture returns the same value no matter where it is evaluated. Because it represents a constant function, it can be accurately reconstructed with any sampling rate and therefore needs no antialiasing. Although this texture is trivial, it is actually quite useful. By providing this class, all parameters to all Materials can be represented as Textures, whether they are spatially varying or not. For example, a red diffuse object will have a ConstantTexture that always returns red as the diffuse color of the material. This way, the shading system always evaluates a texture to get the surface properties at a point, avoiding the need for separate textured and nontextured versions of materials. This material's implementation is in textures/constant.h and textures/constant.cpp.

⟨*ConstantTexture Declarations*⟩ ≡
```
template <typename T> class ConstantTexture : public Texture<T> {
public:
    ⟨ConstantTexture Public Methods 520⟩
private:
    T value;
};
```

⟨*ConstantTexture Public Methods*⟩ ≡ 520
```
ConstantTexture(const T &v) { value = v; }
T Evaluate(const DifferentialGeometry &) const {
    return value;
}
```

10.3.2 SCALE TEXTURE

We have defined the texture interface in a way that makes it easy to use the output of one texture function when computing another. This is useful since it lets us define generic texture operations using any of the other texture types. The ScaleTexture takes two textures and returns the product of their values when evaluated. It is defined in textures/scale.h and textures/scale.cpp.

⟨*ScaleTexture Declarations*⟩ ≡
```
template <typename T1, typename T2>
class ScaleTexture : public Texture<T2> {
public:
    ⟨ScaleTexture Public Methods 521⟩
private:
    Reference<Texture<T1> > tex1;
    Reference<Texture<T2> > tex2;
};
```

ConstantTexture 520
DifferentialGeometry 102
Material 483
Reference 1011
ScaleTexture 520
Texture 519

⟨*ScaleTexture Public Methods*⟩ ≡ 520
```
ScaleTexture(Reference<Texture<T1> > t1, Reference<Texture<T2> > t2)
    : tex1(t1), tex2(t2) { }
```

`ScaleTexture` ignores antialiasing, leaving it to its two subtextures to antialias themselves but not making an effort to antialias their product. While it is easy to show that the product of two band-limited functions is also band limited, the maximum frequency present in the product may be greater than that of either of the two terms individually. Thus, even if the scale and value textures are perfectly antialiased, the result might not be. Fortunately, the most common kind of scale texture is a constant, in which case the other texture's antialiasing is sufficient.

⟨*ScaleTexture Public Methods*⟩ +≡ 520
```
T2 Evaluate(const DifferentialGeometry &dg) const {
    return tex1->Evaluate(dg) * tex2->Evaluate(dg);
}
```

10.3.3 MIX TEXTURES

The `MixTexture` class is a more general variation of `ScaleTexture`. It takes three textures as input: two may be of any type, and the third must return a floating-point value. The floating-point texture is then used to linearly interpolate between the two other textures. Note that a `ConstantTexture` could be used for the floating-point value to achieve a uniform blend, or a more complex `Texture` to blend in a spatially nonuniform way. This texture is defined in `textures/mix.h` and `textures/mix.cpp`.

⟨*MixTexture Declarations*⟩ ≡
```
template <typename T> class MixTexture : public Texture<T> {
public:
    ⟨MixTexture Public Methods 521⟩
private:
    Reference<Texture<T> > tex1, tex2;
    Reference<Texture<float> > amount;
};
```

⟨*MixTexture Public Methods*⟩ ≡ 521
```
MixTexture(Reference<Texture<T> > t1, Reference<Texture<T> > t2,
           Reference<Texture<float> > amt)
    : tex1(t1), tex2(t2), amount(amt) { }
```

To evaluate the mixture, the three textures are evaluated and the floating-point value is used to linearly interpolate between the two. When the blend amount (`amt`) is zero, the first texture's value is returned, and when it is one the second one's value is returned. We will generally assume that `amt` will be between zero and one, but this behavior is not enforced, so extrapolation is possible as well. As with the `ScaleTexture`, antialiasing is ignored, so the introduction of aliasing here is a possibility.

ConstantTexture 520
DifferentialGeometry 102
MixTexture 521
MixTexture::amount 521
MixTexture::tex1 521
MixTexture::tex2 521
Reference 1011
ScaleTexture 520
ScaleTexture::tex1 520
ScaleTexture::tex2 520
Texture 519
Texture::Evaluate() 520

⟨*MixTexture Public Methods*⟩ +≡ 521
```
T Evaluate(const DifferentialGeometry &dg) const {
    T t1 = tex1->Evaluate(dg), t2 = tex2->Evaluate(dg);
    float amt = amount->Evaluate(dg);
    return (1.f - amt) * t1 + amt * t2;
}
```

10.3.4 BILINEAR INTERPOLATION

⟨*BilerpTexture Declarations*⟩ ≡
```
template <typename T> class BilerpTexture : public Texture<T> {
public:
    ⟨BilerpTexture Public Methods 522⟩
private:
    ⟨BilerpTexture Private Data 522⟩
};
```

The `BilerpTexture` class provides bilinear interpolation between four constant values. Values are defined at (0, 0), (1, 0), (0, 1), and (1, 1) in (s, t) parameter space. The value at a particular (s, t) position is found by interpolating between them. It is defined in the files `textures/bilerp.h` and `textures/bilerp.cpp`.

⟨*BilerpTexture Public Methods*⟩ ≡ 522
```
BilerpTexture(TextureMapping2D *m, const T &t00, const T &t01,
              const T &t10, const T &t11)
    : mapping(m), v00(t00), v01(t01), v10(t10), v11(t11) {
}
```

⟨*BilerpTexture Private Data*⟩ ≡ 522
```
TextureMapping2D *mapping;
T v00, v01, v10, v11;
```

The interpolated value of the four values at an (s, t) position can be computed by three linear interpolations. For example, we can first use s to interpolate between the values at (0, 0) and (1, 0) and store that in a temporary `tmp1`. We can then do the same for the (0, 1) and (1, 1) values and store the result in `tmp2`. Finally, we use t to interpolate between `tmp1` and `tmp2` and obtain the final result. Mathematically, this is

$$\begin{aligned}\text{tmp}_1 &= (1-s)\text{v}_{00} + s\ \text{v}_{10}\\ \text{tmp}_2 &= (1-s)\text{v}_{01} + s\ \text{v}_{11}\\ \text{result} &= (1-t)\text{tmp}_1 + t\ \text{tmp}_2.\end{aligned}$$

Rather than storing the intermediate values explicitly, some algebraic rearrangement gives us the same result from an appropriately weighted average of the four corner values:

$$\text{result} = (1-s)(1-t)\text{v}_{00} + (1-s)t\ \text{v}_{01} + s(1-t)\text{v}_{10} + s\ t\ \text{v}_{11}.$$

BilerpTexture 522
BilerpTexture::mapping 522
BilerpTexture::v00 522
BilerpTexture::v01 522
BilerpTexture::v10 522
BilerpTexture::v11 522
DifferentialGeometry 102
Texture 519
Texture::Evaluate() 520
TextureMapping2D 514

⟨*BilerpTexture Public Methods*⟩ +≡ 522

```
T Evaluate(const DifferentialGeometry &dg) const {
    float s, t, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
    return (1-s)*(1-t) * v00 + (1-s)*(  t) * v01 +
           (  s)*(1-t) * v10 + (  s)*(  t) * v11;
}
```

10.4 IMAGE TEXTURE

The `ImageTexture` class stores a 2D array of point-sampled values of a texture function. It uses these samples to reconstruct a continuous image function that can be evaluated at an arbitrary (s, t) position. These sample values are often called *texels*, since they are similar to pixels in an image but are used in the context of a texture. Image textures are the most widely used type of texture in computer graphics; digital photographs, scanned artwork, images created with image-editing programs, and images generated by renderers are all extremely useful sources of data for this particular texture representation (Figure 10.8). The term *texture map* is often used to refer to this type of texture, although this usage blurs the distinction between the mapping that computes texture coordinates and the texture function itself. The implementation of this texture is in the files `textures/imagemap.h` and `textures/imagemap.cpp`.

Figure 10.8: An Example of Image Textures. Here, a rendered image is used to modulate the diffuse color of the sphere, making the sphere look painted.

BilerpTexture::mapping 522
BilerpTexture::v00 522
BilerpTexture::v01 522
BilerpTexture::v10 522
BilerpTexture::v11 522
DifferentialGeometry 102
TextureMapping2D::Map() 514

The ImageTexture class is different from other textures in the system in that it is parameterized on both the data type of the texels it stores in memory as well as the data type of the value that it returns. Making this distinction allows us to create, for example, ImageTextures that store RGBSpectrum values in memory, but always return Spectrum values. In this way, when the system is compiled with full-spectral rendering enabled, the memory cost to store full SampledSpectrum texels doesn't necessarily need to be paid.

⟨*ImageTexture Declarations*⟩ ≡
```
template <typename Tmemory, typename Treturn>
    class ImageTexture : public Texture<Treturn> {
public:
    ⟨ImageTexture Public Methods 527⟩
private:
    ⟨ImageTexture Private Methods 526⟩
    ⟨ImageTexture Private Data 524⟩
};
```

The caller provides the ImageTexture with the filename of an image map, parameters that control the filtering of the map for antialiasing, and parameters that make it possible to scale and gamma-correct the texture values. The scale and gamma parameters will be explained later in this section, and the texture filtering parameters will be explained in Section 10.4.2. The contents of the file are used to create an instance of the MIPMap class that stores the texels in memory and handles the details of reconstruction and filtering to reduce aliasing.

For an ImageTexture that stores RGBSpectrum values in memory, its MIPMap stores the image data using three floating-point values for each sample. This can be a somewhat wasteful representation, since a single image map may have millions of texels and may not need the full 32 bits of accuracy from the floats used to store RGB values for each of them. Exercise 10.1 at the end of this chapter discusses this issue further.

⟨*ImageTexture Method Definitions*⟩ ≡
```
template <typename Tmemory, typename Treturn>
ImageTexture<Tmemory, Treturn>::ImageTexture(TextureMapping2D *m,
        const string &filename, bool doTrilinear, float maxAniso,
        ImageWrap wrapMode, float scale, float gamma) {
    mapping = m;
    mipmap = GetTexture(filename, doTrilinear, maxAniso,
                        wrapMode, scale, gamma);
}
```

⟨*ImageTexture Private Data*⟩ ≡ 524
```
MIPMap<Tmemory> *mipmap;
TextureMapping2D *mapping;
```

ImageTexture 524
ImageTexture::
GetTexture() 525
ImageTexture::mapping 524
ImageTexture::mipmap 524
ImageWrap 530
MIPMap 530
RGBSpectrum 279
SampledSpectrum 266
Spectrum 263
Texture 519
TextureMapping2D 514

10.4.1 TEXTURE CACHING

An image map may require a significant amount of memory; because the user may reuse a texture many times within a scene, pbrt maintains a table of image maps that have been loaded so far, so that they are only loaded into memory once even if they

are used in more than one ImageTexture. The ImageTexture constructor calls the static ImageTexture::GetTexture() method to get a MIPMap representation of the desired texture. If the image map does need to be loaded from disk, ReadImage() handles the low-level details of this process and returns an array of texel values.

⟨*ImageTexture Method Definitions*⟩ +≡
```
template <typename Tmemory, typename Treturn> MIPMap<Tmemory> *
ImageTexture<Tmemory, Treturn>::GetTexture(const string &filename,
        bool doTrilinear, float maxAniso, ImageWrap wrap,
        float scale, float gamma) {
    ⟨Look for texture in texture cache 525⟩
    int width, height;
    RGBSpectrum *texels = ReadImage(filename, &width, &height);
    MIPMap<Tmemory> *ret = NULL;
    if (texels) {
        ⟨Convert texels to type Tmemory and create MIPMap 526⟩
    }
    else {
        ⟨Create one-valued MIPMap 527⟩
    }
    textures[texInfo] = ret;
    return ret;
}
```

TexInfo is a simple structure that holds the image map's filename and filtering parameters; all of these must match for a MIPMap to be reused in another ImageTexture. Its definition is straightforward and won't be included here.

⟨*ImageTexture Private Data*⟩ +≡ 524
```
static std::map<TexInfo, MIPMap<Tmemory> *> textures;
```

⟨*ImageTexture Method Definitions*⟩ +≡
```
template <typename Tmemory, typename Treturn>
    std::map<TexInfo,
             MIPMap<Tmemory> *> ImageTexture<Tmemory, Treturn>::textures;
```

⟨*Look for texture in texture cache*⟩ ≡ 525
```
TexInfo texInfo(filename, doTrilinear, maxAniso, wrap, scale, gamma);
if (textures.find(texInfo) != textures.end())
    return textures[texInfo];
```

Because the image-loading routine returns an array of RGBSpectrum values for the texels, it is necessary to convert these values to the particular type Tmemory of texel that this MIPMap is storing (e.g., float) if the type of Tmemory isn't RGBSpectrum. The per-texel conversion is handled by the utility routine ImageTexture::convertIn(). This conversion

ImageTexture 524
ImageTexture::convertIn() 526
ImageTexture::GetTexture() 525
ImageTexture::textures 525
ImageWrap 530
MIPMap 530
ReadImage() 1004
RGBSpectrum 279
TexInfo 525

is wasted work in the common case where the MIPMap is storing RGBSpectrum values, but the flexibility it gives is worth this relatively small cost in efficiency.[3]

⟨*Convert texels to type* Tmemory *and create* MIPMap⟩ ≡ 525
```
Tmemory *convertedTexels = new Tmemory[width*height];
for (int i = 0; i < width*height; ++i)
    convertIn(texels[i], &convertedTexels[i], scale, gamma);
ret = new MIPMap<Tmemory>(width, height, convertedTexels, doTrilinear,
                          maxAniso, wrap);
delete[] texels;
delete[] convertedTexels;
```

Per-texel conversion is done using C++ function overloading. For every type to which we would like to be able to convert these values, a separate ImageTexture::convertIn() function must be provided. In the loop over texels earlier, C++'s function overloading mechanism will select the appropriate instance of ImageTexture::convertIn() based on the destination type. Unfortunately, it is not possible to return the converted value from the function, since C++ doesn't support overloading by return type.

In addition to converting types, these functions optionally scale and gamma correct the texel values to map them to a desired range. Gamma correction is particularly important to handle carefully: computer displays have the property that most of them don't exhibit a linear relationship between the pixel values to be displayed and the radiance that they emit. Thus, an artist may create a texture map where, as seen on an LCD display, one part of the image appears twice as bright as another. However, the corresponding pixel values won't in fact have a 2:1 relationship. (Conversely, pixels that do have a 2:1 relationship don't lead to pixels with a 2:1 brightness ratio.)

This discrepancy is a problem for a renderer using such an image as a texture map, since the renderer usually expects a linear relationship between texel values and the quantity that they represent. In practice, the relationship between pixel values and display brightness is well modeled with a power curve with the exponent of a value gamma. (Gamma values between 1.5 and 2.2 are typical.) *Gamma correction*, as performed in convertIn(), can be used to restore an approximately linear relationship between texel values and the values they are intended to represent. See the "Further Reading" section for more discussion of gamma correction.

⟨*ImageTexture Private Methods*⟩ ≡ 524
```
static void convertIn(const RGBSpectrum &from, RGBSpectrum *to,
                      float scale, float gamma) {
    *to = Pow(scale * from, gamma);
}
static void convertIn(const RGBSpectrum &from, float *to,
                      float scale, float gamma) {
    *to = powf(scale * from.y(), gamma);
}
```

ImageTexture::
  convertIn() 526
MIPMap 530
RGBSpectrum 279
Spectrum 263
Spectrum::Pow() 265
Spectrum::y() 273

3 Additional C++ template trickery could ensure that this step was skipped when T is type Spectrum if the cost of this unnecessary work was unacceptable.

If the texture file wasn't found or was unreadable, an image map with a single sample with a value of one is created so that the renderer can continue to generate an image of the scene without needing to abort execution. The `ReadImage()` function will issue a warning message in this case.

⟨*Create one-valued* `MIPMap`⟩ ≡ 525
```
Tmemory *oneVal = new Tmemory[1];
oneVal[0] = powf(scale, gamma);
ret = new MIPMap<Tmemory>(1, 1, oneVal);
delete[] oneVal;
```

After the image is rendered and the system is cleaning up, the `ClearCache()` method is called to free the memory for the entries in the texture cache.

⟨*ImageTexture Public Methods*⟩ ≡ 524
```
static void ClearCache() {
    typename std::map<TexInfo, MIPMap<Tmemory> *>::iterator iter;
    iter = textures.begin();
    while (iter != textures.end()) {
        delete iter->second;
        ++iter;
    }
    textures.erase(textures.begin(), textures.end());
}
```

The `ImageTexture::Evaluate()` routine does the usual texture coordinate computation and then hands the image map lookup to the `MIPMap`, which does the image filtering work for antialiasing. The returned value is still of type `Tmemory`; another conversion step similar to `ImageTexture::convertIn()` above converts to the returned type `Treturn`.

⟨*ImageTexture Method Definitions*⟩ +≡
```
template <typename Tmemory, typename Treturn> Treturn
ImageTexture<Tmemory,
             Treturn>::Evaluate(const DifferentialGeometry &dg) const {
    float s, t, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
    Tmemory mem = mipmap->Lookup(s, t, dsdx, dtdx, dsdy, dtdy);
    Treturn ret;
    convertOut(mem, &ret);
    return ret;
}
```

⟨*ImageTexture Private Methods*⟩ +≡ 524
```
static void convertOut(const RGBSpectrum &from, Spectrum *to) {
    float rgb[3];
    from.ToRGB(rgb);
    *to = Spectrum::FromRGB(rgb);
}
static void convertOut(float from, float *to) {
    *to = from;
}
```

DifferentialGeometry 102
ImageTexture 524
ImageTexture::convertIn() 526
ImageTexture::Evaluate() 527
ImageTexture::mapping 524
ImageTexture::mipmap 524
ImageTexture::textures 525
MIPMap 530
MIPMap::Lookup() 540
ReadImage() 1004
RGBSpectrum 279
Spectrum 263
Spectrum::FromRGB() 277
Spectrum::ToRGB() 275
TexInfo 525
TextureMapping2D::Map() 514

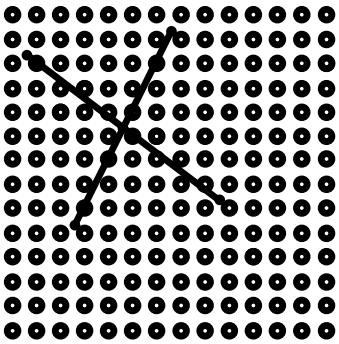


Figure 10.9: Given a point at which to perform an image map (denoted by the solid point in the center) and estimates of the texture space sampling rate (denoted by adjacent solid points), it may be necessary to filter the contributions of a large number of texels in the image map (denoted by open points).

10.4.2 MIP MAPS

As always, if the image function has higher-frequency detail than can be represented by the texture sampling rate, aliasing will be present in the final image. Any frequencies higher than the Nyquist limit must be removed by prefiltering before the function is evaluated. Figure 10.9 shows the basic problem we face: an image texture has texels that are samples of some image function at a fixed frequency. The filter region for the lookup is given by its (s, t) center point and offsets to the estimated texture coordinate locations for the adjacent image samples. Because these offsets are estimates of the texture sampling rate, we must remove any frequencies higher than twice the distance to the adjacent samples in order to satisfy the Nyquist criterion.

The texture sampling and reconstruction process has a few key differences from the image sampling process discussed in Chapter 7. These differences make it possible to address the antialiasing problem with more effective and less computationally expensive techniques. For example, here it is inexpensive to get the value of a sample—only an array lookup is necessary (as opposed to having to trace a ray). Further, because the texture image function is fully defined by the set of samples and there is no mystery about what its highest frequency could be, there is no uncertainty related to the function's behavior between samples. These differences make it possible to remove detail from the texture before sampling, thus eliminating aliasing.

However, the texture sampling rate will typically change from pixel to pixel. The sampling rate is determined by scene geometry and its orientation, the texture coordinate mapping function, and the camera projection and image sampling rate. Because the sampling rate is not fixed, texture filtering algorithms need to be able to filter over arbitrary regions of texture samples efficiently.

MIPMap 530

The `MIPMap` class implements two methods for efficient texture filtering with spatially varying filter widths. The first, trilinear interpolation, is fast and easy to implement and

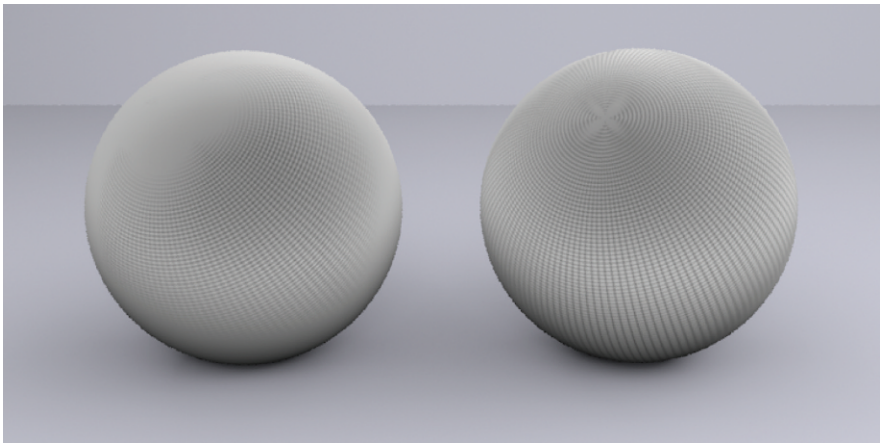


Figure 10.10: Filtering the image map properly substantially improves the image. On the left, trilinear interpolation was used; on the right, the EWA algorithm. Both of these approaches give a much better image than the unfiltered image map in Figure 11.1.Trilinear interpolation is less able to handle strongly anisotropic filter footprints than EWA, which is why the edges of the sphere on the left are a uniform gray color (the overall average value of the texture), while the edges of the sphere on the right are much better able to continue to resolve detail from the image map before fading to gray.

has been widely used for texture filtering in graphics hardware. The second, elliptically weighted averaging, is slower and more complex, but returns extremely high-quality results. Figure 10.1 shows the aliasing errors that result from ignoring texture filtering and just bilinearly interpolating texels from the most detailed level of the image map. Figure 10.10 shows the improvement from using the triangle filter and the EWA algorithm instead.

To limit the potential number of texels that need to be accessed, both of these filtering methods use an *image pyramid* of increasingly lower-resolution prefiltered versions of the original image to accelerate their operation.[4] The original image texels are at the bottom level of the pyramid, and the image at each level is half the resolution of the previous level, up to the top level, which has a single texel representing the average of all of the texels in the original image. This collection of images needs at most 1/3 more memory than storing the most detailed level alone and can be used to quickly find filtered values over large regions of the original image. The basic idea behind the pyramid is that if a large area of texels needs to be filtered a reasonable approximation is to use a higher level of the pyramid and do the filtering over the same area there, accessing many fewer texels.

`MIPMap` is a template class and is parameterized by the data type of the image texels. `pbrt` creates `MIPMaps` of both `RGBSpectrum` and `float` images; `float` MIP maps are used for representing directional distributions of intensity from goniometric light sources, for example. The `MIPMap` implementation requires that the type `T` support just a few basic operations, including addition and multiplication by a scalar.

`MIPMap` 530
`RGBSpectrum` 279

4 The name "MIP map" comes from the Latin *multum in parvo*, which means "much in little," a nod to the image pyramid.

The ImageWrap enumerant, passed to the MIPMap constructor, specifies the desired behavior when the supplied texture coordinates are not in the legal [0, 1] range.

⟨*MIPMap Declarations*⟩ ≡
```
typedef enum {
    TEXTURE_REPEAT,
    TEXTURE_BLACK,
    TEXTURE_CLAMP
} ImageWrap;
```

⟨*MIPMap Declarations*⟩ +≡
```
template <typename T> class MIPMap {
public:
    ⟨MIPMap Public Methods 535⟩
private:
    ⟨MIPMap Private Methods 532⟩
    ⟨MIPMap Private Data 530⟩
};
```

In the constructor, the MIPMap copies the image data provided by the caller, resizes the image if necessary to ensure that its resolution is a power of two in each direction, and initializes a lookup table used by the elliptically weighted average filtering method in Section 10.4.4. It also records the desired behavior for texture coordinates that fall outside of the legal range in the wrapmode argument.

⟨*MIPMap Method Definitions*⟩ ≡
```
template <typename T>
MIPMap<T>::MIPMap(uint32_t sres, uint32_t tres, const T *img, bool doTri,
                  float maxAniso, ImageWrap wm) {
    doTrilinear = doTri;
    maxAnisotropy = maxAniso;
    wrapMode = wm;
    T *resampledImage = NULL;
    if (!IsPowerOf2(sres) || !IsPowerOf2(tres)) {
        ⟨Resample image to power-of-two resolution 531⟩
    }
    width = sres;
    height = tres;
    ⟨Initialize levels of MIPMap from image 534⟩
    if (resampledImage) delete[] resampledImage;
    ⟨Initialize EWA filter weights if needed 544⟩
}
```

⟨*MIPMap Private Data*⟩ ≡ 530
```
bool doTrilinear;
float maxAnisotropy;
ImageWrap wrapMode;
```

ImageWrap 530
IsPowerOf2() 1001
MIPMap 530

Implementation of an image pyramid is somewhat easier if the resolution of the original image is an exact power of two in each direction; this ensures that there is a straightfor-

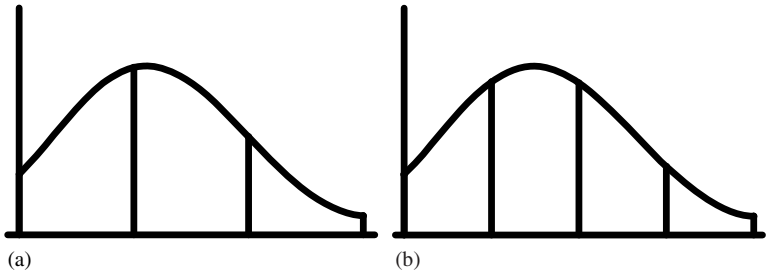


Figure 10.11: To increase an image's resolution to be a power of two, the `MIPMap` performs two 1D resampling steps with a separable reconstruction filter. (a) A 1D function reconstructed from four samples, denoted by dots. (b) To represent the same image function with more samples, we just need to reconstruct the continuous function and evaluate it at the new positions.

ward relationship between the level of the pyramid and the number of texels at that level. If the user has provided an image where the resolution in one or both of the dimensions is not a power of two, then the `MIPMap` constructor starts by resizing the image up to the next power-of-two resolution greater than the original resolution before constructing the pyramid. Exercise 10.5 at the end of the chapter describes an approach to building image pyramids with non-power-of-two resolutions.

Image resizing here involves more application of the sampling and reconstruction theory from Chapter 7: we have an image function that has been sampled at one sampling rate, and we'd like to reconstruct a continuous image function from the original samples to resample at a new set of sample positions. Because this represents an increase in the sampling rate from the original rate, we don't have to worry about introducing aliasing due to undersampling high-frequency components in this step; we only need to reconstruct and directly resample the new function. Figure 10.11 illustrates this task in 1D.

The `MIPMap` uses a separable reconstruction filter for this task; recall from Section 7.7 that separable filters can be written as the product of one-dimensional filters: $f(x, y) = f(x)f(y)$. One advantage of using a separable filter is that if we are using one to resample an image from one resolution (s, t) to another (s', t'), then we can implement the resampling as two one-dimensional resampling steps, first resampling in s to create an image of resolution (s', t) and then resampling that image to create the final image of resolution (s', t'). Resampling the image via two 1D steps in this manner simplifies implementation and makes the number of texels accessed for each texel in the final image a linear function of the filter width, rather than a quadratic one.

⟨*Resample image to power-of-two resolution*⟩ ≡ 530

```
uint32_t sPow2 = RoundUpPow2(sres), tPow2 = RoundUpPow2(tres);
⟨Resample image in s direction 532⟩
⟨Resample image in t direction⟩
sres = sPow2;
tres = tPow2;
```

MIPMap 530
RoundUpPow2() 1002

Reconstructing the original image function and sampling it at a new texel's position is mathematically equivalent to centering the reconstruction filter kernel at the new texel's position and weighting the nearby texels in the original image appropriately. Thus, each new texel is a weighted average of a small number of texels in the original image.

The `MIPMap::resampleWeights()` method determines which original texels contribute to each new texel and what the values are of the contribution weights for each new texel. It returns the values in an array of `ResampleWeight` structures for all of the texels in a 1D row or column of the image. Because this information is the same for all rows of the image when resampling in s and all columns when resampling in t, it's more efficient to compute it once for each of the two passes and then reuse it many times for each one. Given these weights, the image is first magnified in the s direction, turning the original image with resolution `(sres,tres)` into an image with resolution `(sPow2,tres)`, which is stored in `resampledImage`. The implementation here allocates enough space in `resampledImage` to hold the final zoomed image with resolution `(sPow2,tPow2)`, so two large allocations can be avoided.

⟨*Resample image in s direction*⟩ ≡ 531
```
ResampleWeight *sWeights = resampleWeights(sres, sPow2);
resampledImage = new T[sPow2 * tPow2];
⟨Apply sWeights to zoom in s direction 534⟩
delete[] sWeights;
```

For the reconstruction filter used here, no more than four of the original texels will contribute to each new texel after zooming, so `ResampleWeight` only needs to hold four weights. Because the four texels are contiguous, we only store the offset to the first one.

⟨*MIPMap Private Data*⟩ +≡ 530
```
struct ResampleWeight {
    int firstTexel;
    float weight[4];
};
```

⟨*MIPMap Private Methods*⟩ ≡ 530
```
ResampleWeight *resampleWeights(uint32_t oldres, uint32_t newres) {
    Assert(newres >= oldres);
    ResampleWeight *wt = new ResampleWeight[newres];
    float filterwidth = 2.f;
    for (uint32_t i = 0; i < newres; ++i) {
        ⟨Compute image resampling weights for ith texel 533⟩
    }
    return wt;
}
```

Just as it was important to distinguish between discrete and continuous pixel coordinates in Chapter 7, the same issues need to be addressed with texel coordinates here. We will use the same conventions as described in Section 7.1.7. For each new texel, this function starts by computing its continuous coordinates in terms of the old texel coordinates. This value is stored in `center`, because it is the center of the reconstruction filter for the new texel. Next, it is necessary to find the offset to the first texel that contributes to the new texel.

Assert() 1005
MIPMap::
resampleWeights() 532
ResampleWeight 532

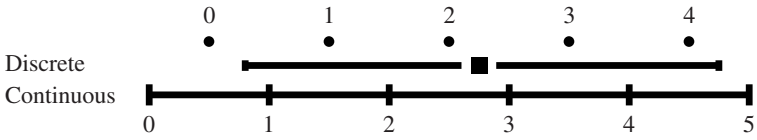


Figure 10.12: The computation to find the first texel inside a reconstruction filter's support is slightly tricky. Consider a filter centered around continuous coordinate 2.75 with width 2, as shown here. The filter's support covers the range [0.75, 4.75], although texel zero is outside the filter's support: adding 0.5 to the lower end before taking the floor to find the discrete texel gives the correct starting texel, number one.

This is a slightly tricky calculation—after subtracting the filter width to find the start of the filter's nonzero range, it is necessary to add an extra 0.5 offset to the continuous coordinate before taking the floor to find the discrete coordinate. Figure 10.12 illustrates why this offset is needed.

Starting from this first contributing texel, this function loops over four texels, computing each one's offset to the center of the filter kernel and the corresponding filter weight. The reconstruction filter function used to compute the weights, `Lanczos()`, is the same as the one in `LanczosSincFilter::Sinc1D()`.

⟨*Compute image resampling weights for* `i`*th texel*⟩ ≡ 532
```
float center = (i + .5f) * oldres / newres;
wt[i].firstTexel = Floor2Int((center - filterwidth) + 0.5f);
for (int j = 0; j < 4; ++j) {
    float pos = wt[i].firstTexel + j + .5f;
    wt[i].weight[j] = Lanczos((pos - center) / filterwidth);
}
```
⟨*Normalize filter weights for texel resampling* 533⟩

Depending on the filter function used, the four filter weights may not sum to one. Therefore, to ensure that the resampled image won't be any brighter or darker than the original image, the weights are normalized here.

⟨*Normalize filter weights for texel resampling*⟩ ≡ 533
```
float invSumWts = 1.f / (wt[i].weight[0] + wt[i].weight[1] +
                         wt[i].weight[2] + wt[i].weight[3]);
for (uint32_t j = 0; j < 4; ++j)
    wt[i].weight[j] *= invSumWts;
```

Floor2Int() 1002
Lanczos() 533
LanczosSincFilter::
  Sinc1D() 402
ResampleWeight::
  firstTexel 532
ResampleWeight::weight 532

⟨*Texture Declarations*⟩ +≡
```
float Lanczos(float, float tau=2);
```

Once the weights have been computed, it's easy to apply them to compute the zoomed texels. For each of the `tres` horizontal scan lines in the original image, a pass is made across the `sPow2` texels in the s-zoomed image using the precomputed weights to compute their values.

⟨*Apply* sWeights *to zoom in s direction*⟩ ≡ 532
```
for (uint32_t t = 0; t < tres; ++t) {
    for (uint32_t s = 0; s < sPow2; ++s) {
        ⟨Compute texel (s, t) in s-zoomed image 534⟩
    }
}
```

The ImageWrap parameter to the MIPMap constructor determines the convention to be used for out-of-bounds texel coordinates. It either remaps them to valid values with a modulus or clamp calculation or uses a black texel value.

⟨*Compute texel* (*s*, *t*) *in s-zoomed image*⟩ ≡ 534
```
resampledImage[t*sPow2+s] = 0.;
for (int j = 0; j < 4; ++j) {
    int origS = sWeights[s].firstTexel + j;
    if (wrapMode == TEXTURE_REPEAT)
        origS = Mod(origS, sres);
    else if (wrapMode == TEXTURE_CLAMP)
        origS = Clamp(origS, 0, sres-1);
    if (origS >= 0 && origS < (int)sres)
        resampledImage[t*sPow2+s] += sWeights[s].weight[j] *
                                     img[t*sres + origS];
}
```

The process for resampling in the t direction is almost the same as for s, so we won't include the implementation here. Once we have an image with resolutions that are powers of two, the levels of the MIP map can be initialized, starting from the bottom (finest) level. Each higher level is found by filtering the texels from the previous level.

Because image maps use a fair amount of memory, and because 8 to 20 texels are typically used per image texture lookup to compute a filtered value, it's worth carefully considering how the texels are laid out in memory, since reducing cache misses while accessing the texture map can noticeably improve the renderer's performance. Because both of the texture filtering methods implemented in this section access a set of texels in a rectangular region of the image map each time a lookup is performed, the MIPMap uses the BlockedArray template class to store the 2D arrays of texel values, rather than using a standard C++ array. The BlockedArray reorders the array values in memory in a way that improves cache coherence when the values are accessed with these kinds of rectangular patterns; it is described in Section A.5.5 in Appendix A.

⟨*Initialize levels of MIPMap from image*⟩ ≡ 530
```
nLevels = 1 + Log2Int(float(max(sres, tres)));
pyramid = new BlockedArray<T> *[nLevels];
⟨Initialize most detailed level of MIPMap 535⟩
for (uint32_t i = 1; i < nLevels; ++i) {
    ⟨Initialize ith MIPMap level from i − 1st level 536⟩
}
```

BlockedArray 1017
Clamp() 1000
ImageWrap 530
Log2Int() 1001
MIPMap 530
Mod() 1001
ResampleWeight::
firstTexel 532
ResampleWeight::weight 532
TEXTURE_CLAMP 530
TEXTURE_REPEAT 530

⟨*MIPMap Private Data*⟩ +≡ 530
```
BlockedArray<T> **pyramid;
uint32_t width, height, nLevels;
```

⟨*MIPMap Public Methods*⟩ ≡ 530
```
uint32_t Width() const { return width; }
uint32_t Height() const { return height; }
uint32_t Levels() const { return nLevels; }
```

The base level of the MIP map, which holds the original data (or the resampled data, if it didn't originally have power-of-two resolutions), is initialized by the default `BlockedArray` constructor.

⟨*Initialize most detailed level of MIPMap*⟩ ≡ 534
```
pyramid[0] = new BlockedArray<T>(sres, tres, img);
```

Before showing how the rest of the levels are initialized, we will first define a texel access function that will be used during that process. `MIPMap::Texel()` returns a reference to the texel value for the given discrete integer-valued texel position. As described earlier, if an out-of-range texel coordinate is passed in, this method effectively repeats the texture over the entire 2D texture coordinate domain by taking the modulus of the coordinate with respect to the texture size, clamps the coordinates to the valid range so that the border pixels are used, or returns a black texel for out-of-bounds coordinates.

⟨*MIPMap Method Definitions*⟩ +≡
```
template <typename T>
const T &MIPMap<T>::Texel(uint32_t level, int s, int t) const {
    const BlockedArray<T> &l = *pyramid[level];
    ⟨Compute texel (s, t) accounting for boundary conditions 535⟩
    return l(s, t);
}
```

⟨*Compute texel* (*s*, *t*) *accounting for boundary conditions*⟩ ≡ 535
```
switch (wrapMode) {
    case TEXTURE_REPEAT:
        s = Mod(s, l.uSize());
        t = Mod(t, l.vSize());
        break;
    case TEXTURE_CLAMP:
        s = Clamp(s, 0, l.uSize() - 1);
        t = Clamp(t, 0, l.vSize() - 1);
        break;
    case TEXTURE_BLACK: {
        static const T black = 0.f;
        if (s < 0 || s >= (int)l.uSize() ||
            t < 0 || t >= (int)l.vSize())
            return black;
        break;
    }
}
```

BlockedArray 1017
BlockedArray::uSize() 1019
BlockedArray::vSize() 1019
MIPMap 530
MIPMap::pyramid 535
MIPMap::Texel() 535
TEXTURE_BLACK 530
TEXTURE_CLAMP 530
TEXTURE_REPEAT 530

For nonsquare images, the resolution in one direction must be clamped to one for the upper levels of the image pyramid, where there is still downsampling to do in the larger of the two resolutions. This is handled by the following `max()` calls:

⟨*Initialize i th MIPMap level from i − 1st level*⟩ ≡ 534
```
uint32_t sRes = max(1u, pyramid[i-1]->uSize()/2);
uint32_t tRes = max(1u, pyramid[i-1]->vSize()/2);
pyramid[i] = new BlockedArray<T>(sRes, tRes);
⟨Filter four texels from finer level of pyramid 536⟩
```

The `MIPMap` uses a simple box filter to average four texels from the previous level to find the value at the current texel. The Lanczos filter here would give a slightly better result for this computation, although this modification is left as an exercise at the end of the chapter.

⟨*Filter four texels from finer level of pyramid*⟩ ≡ 536
```
for (uint32_t t = 0; t < tRes; ++t)
    for (uint32_t s = 0; s < sRes; ++s)
        (*pyramid[i])(s, t) = .25f *
           (Texel(i-1, 2*s, 2*t)   + Texel(i-1, 2*s+1, 2*t) +
            Texel(i-1, 2*s, 2*t+1) + Texel(i-1, 2*s+1, 2*t+1));
```

10.4.3 ISOTROPIC TRIANGLE FILTER

The first of the two `MIPMap::Lookup()` methods uses a triangle filter over the texture samples to remove high frequencies. Although this filter function does not give high-quality results, it can be implemented very efficiently. In addition to the (s, t) coordinates of the evaluation point, the caller passes this method a filter width for the lookup, giving the extent of the region of the texture to filter across. This method filters over a square region in texture space, so the width should be conservatively chosen to avoid aliasing in both the s and t directions. Filtering techniques like this one that do not support a filter extent that is nonsquare or non-axis-aligned are known as *isotropic*. The primary disadvantage of isotropic filtering algorithms is that textures viewed at an oblique angle will appear blurry, since the required sampling rate along one axis will be very different from the sampling rate along the other in this case.

Because filtering over many texels for wide filter widths would be inefficient, this method chooses a MIP map level from the pyramid such that the filter region at that level would cover four texels at that level. Figure 10.13 illustrates this idea.

⟨*MIPMap Method Definitions*⟩ +≡
```
template <typename T>
T MIPMap<T>::Lookup(float s, float t, float width) const {
    ⟨Compute MIPMap level for trilinear filtering 537⟩
    ⟨Perform trilinear interpolation at appropriate MIPMap level 537⟩
}
```

BlockedArray 1017
BlockedArray::uSize() 1019
BlockedArray::vSize() 1019
MIPMap 530
MIPMap::Lookup() 540
MIPMap::Texel() 535

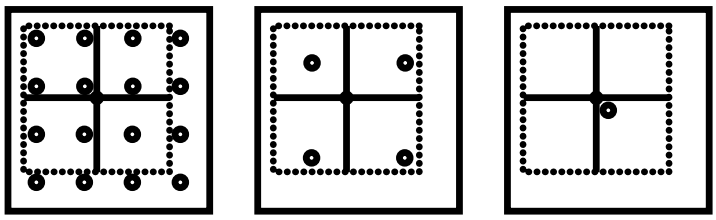


Figure 10.13: Choosing a MIP Map Level for the Triangle Filter. The `MIPMap` chooses a level such that the filter covers four texels.

Since the resolutions of the levels of the pyramid are all powers of two, the resolution of level l is $2^{\text{nLevels}-1-l}$. Therefore, to find the level with a texel spacing width w requires solving

$$\frac{1}{w} = 2^{\text{nLevels}-1-l}$$

for l. In general, this will be a floating-point value between two MIP map levels.

⟨*Compute MIPMap level for trilinear filtering*⟩ ≡ 536
```
float level = nLevels - 1 + Log2(max(width, 1e-8f));
```

As shown by Figure 10.13, applying a triangle filter to the four texels around the sample point will either filter over too small a region or too large a region (except for very carefully selected filter widths). The implementation here applies the triangle filter at both of these levels and blends between them according to how close `level` is to each of them. This helps hide the transitions from one MIP map level to the next at nearby pixels in the final image. While applying a triangle filter to four texels at two levels in this manner doesn't give exactly the same result as applying it to the original highest-resolution texels, the difference isn't too bad in practice and the efficiency of this approach is worth this penalty. In any case, the elliptically weighted average filtering in the next section should be used when texture quality is important.

⟨*Perform trilinear interpolation at appropriate MIPMap level*⟩ ≡ 536
```
if (level < 0)
    return triangle(0, s, t);
else if (level >= nLevels - 1)
    return Texel(nLevels-1, 0, 0);
else {
    uint32_t iLevel = Floor2Int(level);
    float delta = level - iLevel;
    return (1.f-delta) * triangle(iLevel, s, t) +
           delta * triangle(iLevel+1, s, t);
}
```

Floor2Int() 1002
Log2() 1001
MIPMap::nLevels 535
MIPMap::Texel() 535
MIPMap::triangle() 539

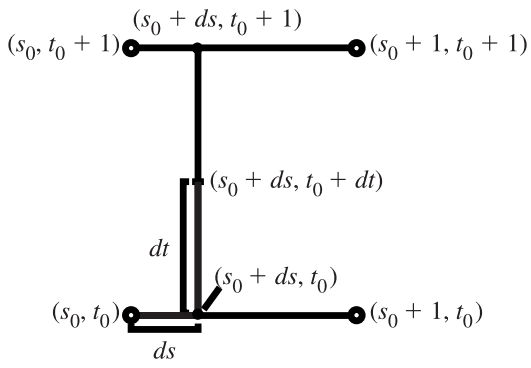


Figure 10.14: To compute the value of the image texture function at an arbitrary (s, t) position, `MIPMap::triangle()` finds the four texels around (s, t) and weights them according to a triangle filter based on their distance to (s, t). One way to implement this is as a series of linear interpolations, as shown here: First, the two texels below (s, t) are linearly interpolated to find a value at $(s, 0)$, and the two texels above it are interpolated to find $(s, 1)$. Then, $(s, 0)$ and $(s, 1)$ are linearly interpolated again to find the value at (s, t).

Given floating-point texture coordinates in $[0, 1]^2$, the `MIPMap::triangle()` routine uses a triangle filter to interpolate between the four texels that surround the sample point, as shown in Figure 10.14. This method first scales the coordinates by the texture resolution at the given MIP map level in each direction, turning them into continuous texel coordinates. Because these are continuous coordinates, but the texels in the image map are defined at discrete texture coordinates, it's important to carefully convert into a common representation. Here, we will do all of our work in discrete coordinates, mapping the continuous texture coordinates to discrete space.

For example, consider the 1D case with a continuous texture coordinate of 2.4: this coordinate is a distance of 0.1 below the discrete texel coordinate 2 (which corresponds to a continuous coordinate of 2.5), and is 0.9 above the discrete coordinate 1 (continuous coordinate 1.5). Thus, if we subtract 0.5 from the continuous coordinate 2.4, giving 1.9, we can correctly compute the correct distances to the discrete coordinates 1 and 2 by subtracting coordinates.

After computing the distances in s and t to the texel at the lower left of the given coordinates, `ds` and `dt`, `MIPMap::triangle()` determines weights for the four texels and computes the filtered value. Recall that the triangle filter is

$$f(x, y) = (1 - |x|)(1 - |y|);$$

BilerpTexture::
Evaluate() 523

MIPMap::triangle() 539

the appropriate weights follow directly. Notice the similarity between this computation and `BilerpTexture::Evaluate()`.

⟨*MIPMap Method Definitions*⟩ +≡

```
template <typename T>
T MIPMap<T>::triangle(uint32_t level, float s, float t) const {
    level = Clamp(level, 0, nLevels-1);
    s = s * pyramid[level]->uSize() - 0.5f;
    t = t * pyramid[level]->vSize() - 0.5f;
    int s0 = Floor2Int(s), t0 = Floor2Int(t);
    float ds = s - s0, dt = t - t0;
    return (1.f-ds) * (1.f-dt) * Texel(level, s0, t0) +
           (1.f-ds) * dt       * Texel(level, s0, t0+1) +
           ds       * (1.f-dt) * Texel(level, s0+1, t0) +
           ds       * dt       * Texel(level, s0+1, t0+1);
}
```

★ 10.4.4 ELLIPTICALLY WEIGHTED AVERAGE

The elliptically weighted average (EWA) algorithm fits an ellipse to the two axes in texture space given by the texture coordinate differentials and then filters the texture with a Gaussian filter function (Figure 10.15). It is widely regarded as one of the best texture filtering algorithms in graphics and has been carefully derived from the basic principles of sampling theory. Unlike the triangle filter in the previous section, it can filter over arbitrarily oriented regions of the texture, with some flexibility of having different filter extents in different directions. This type of filter is known as *anisotropic*. This capability greatly improves the quality of its results, since it can properly adapt to different sampling rates along the two image axes.

We won't show the full derivation of this filter here, although we do note that it is distinguished by being a *unified resampling filter*: it simultaneously computes the result of a Gaussian filtered texture function convolved with a Gaussian reconstruction filter in image space. This is in contrast to many other texture filtering methods that ignore the effect of the image space filter or equivalently assume that it is a box. Even if a

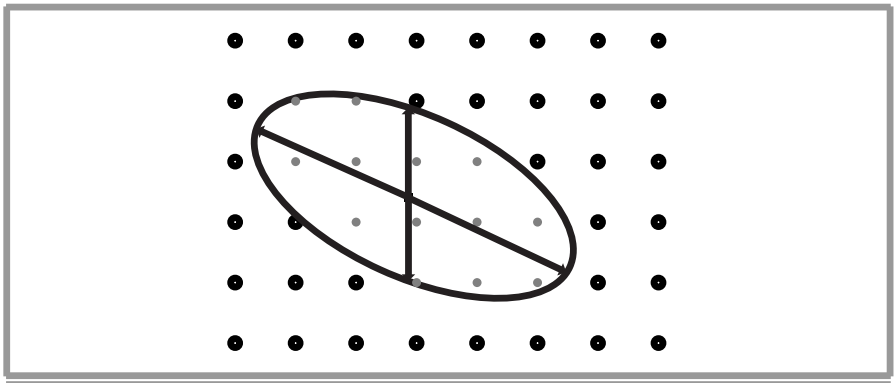


Figure 10.15: The EWA filter applies a Gaussian filter to the texels in an elliptical area around the evaluation point. The extent of the ellipse is such that its edge passes through the positions of the adjacent texture samples as estimated by the texture coordinate partial derivatives.

BlockedArray::uSize() 1019
BlockedArray::vSize() 1019
Clamp() 1000
Floor2Int() 1002
MIPMap 530
MIPMap::nLevels 535
MIPMap::Texel() 535

Gaussian isn't being used for filtering the samples for the image being rendered, taking some account of the spatial variation of the image filter improves the results, assuming that the filter being used is somewhat similar in shape to the Gaussian, as the Mitchell and windowed sinc filters are.

⟨*MIPMap Method Definitions*⟩ +≡
```
template <typename T>
T MIPMap<T>::Lookup(float s, float t, float ds0, float dt0,
                    float ds1, float dt1) const {
    if (doTrilinear) {
        T val = Lookup(s, t,
                       2.f * max(max(fabsf(ds0), fabsf(dt0)),
                                 max(fabsf(ds1), fabsf(dt1))));
        return val;
    }
    ⟨Compute ellipse minor and major axes 540⟩
    ⟨Clamp ellipse eccentricity if too large 540⟩
    ⟨Choose level of detail for EWA lookup and perform EWA filtering 541⟩
}
```

The screen space partial derivatives of the texture coordinates define the axes of the ellipse. The lookup method starts out by determining which of the two axes is the major axis (the longer of the two) and which is the minor, swapping them if needed so that `(ds0,dt0)` is the major axis. The length of the minor axis will be used shortly to select a MIP map level.

⟨*Compute ellipse minor and major axes*⟩ ≡ 540
```
if (ds0*ds0 + dt0*dt0 < ds1*ds1 + dt1*dt1) {
    swap(ds0, ds1);
    swap(dt0, dt1);
}
float majorLength = sqrtf(ds0*ds0 + dt0*dt0);
float minorLength = sqrtf(ds1*ds1 + dt1*dt1);
```

Next the *eccentricity* of the ellipse is computed—the ratio of the length of the major axis to the length of the minor axis. A large eccentricity indicates a very long and skinny ellipse. Because this method filters texels from a MIP map level chosen based on the length of the minor axis, highly eccentric ellipses mean that a large number of texels need to be filtered. To avoid this expense (and to ensure that any EWA lookup takes a bounded amount of time), the length of the minor axis may be increased to limit the eccentricity. The result may be an increase in blurring, although this effect usually isn't noticeable in practice.

⟨*Clamp ellipse eccentricity if too large*⟩ ≡ 540
```
if (minorLength * maxAnisotropy < majorLength && minorLength > 0.f) {
    float scale = majorLength / (minorLength * maxAnisotropy);
    ds1 *= scale;
    dt1 *= scale;
    minorLength *= scale;
}
```

MIPMap 530
MIPMap::doTrilinear 530
MIPMap::maxAnisotropy 530

```
if (minorLength == 0.f) {
    T val = triangle(0, s, t);
    return val;
}
```

Like the triangle filter, the EWA filter uses the image pyramid to reduce the number of texels to be filtered for a particular texture lookup, choosing a MIP map level based on the length of the minor axis. Given the limited eccentricity of the ellipse due to the clamping above, the total number of texels used is thus bounded. Given the length of the minor axis, the computation to find the appropriate pyramid level is the same as was used for the triangle filter. Similarly, the implementation here blends between the filtered results at the two levels around the computed level of detail, again to reduce artifacts from transitions from one level to another.

⟨*Choose level of detail for EWA lookup and perform EWA filtering*⟩ ≡ 540

```
float lod = max(0.f, nLevels - 1.f + Log2(minorLength));
uint32_t ilod = Floor2Int(lod);
float d = lod - ilod;
T val = (1.f - d) * EWA(ilod,   s, t, ds0, dt0, ds1, dt1) +
                d  * EWA(ilod+1, s, t, ds0, dt0, ds1, dt1);
return val;
```

The `MIPMap::EWA()` method actually applies the filter at a particular level.

⟨*MIPMap Method Definitions*⟩ +≡

```
template <typename T>
T MIPMap<T>::EWA(uint32_t level, float s, float t, float ds0, float dt0,
                 float ds1, float dt1) const {
    if (level >= nLevels) return Texel(nLevels-1, 0, 0);
    ⟨Convert EWA coordinates to appropriate scale for level 541⟩
    ⟨Compute ellipse coefficients to bound EWA filter region 542⟩
    ⟨Compute the ellipse's (s, t) bounding box in texture space 542⟩
    ⟨Scan over ellipse bound and compute quadratic equation 543⟩
}
```

This method first converts from texture coordinates in [0, 1] to coordinates and differentials in terms of the resolution of the chosen MIP map level. It also subtracts 0.5 from the continuous position coordinate to align the sample point with the discrete texel coordinates, as was done in `MIPMap::triangle()`.

⟨*Convert EWA coordinates to appropriate scale for level*⟩ ≡ 541

```
s = s * pyramid[level]->uSize() - 0.5f;
t = t * pyramid[level]->vSize() - 0.5f;
ds0 *= pyramid[level]->uSize();
dt0 *= pyramid[level]->vSize();
ds1 *= pyramid[level]->uSize();
dt1 *= pyramid[level]->vSize();
```

It next computes the coefficients of the implicit equation for the ellipse with axes `(ds0,dt0)` and `(ds1,dt1)` and centered at the origin. Placing the ellipse at the origin

BlockedArray::uSize() 1019
BlockedArray::vSize() 1019
Floor2Int() 1002
Log2() 1001
MIPMap 530
MIPMap::EWA() 541
MIPMap::nLevels 535
MIPMap::Texel() 535
MIPMap::triangle() 539

rather than at (s, t) simplifies the implicit equation and the computation of its coefficients, and can be easily corrected for when the equation is evaluated later. The general form of the implicit equation for all points (s, t) inside such an ellipse is

$$e(s, t) = As^2 + Bst + Ct^2 < F,$$

although it is more computationally efficient to divide through by F and express this as

$$e(s, t) = \frac{A}{F}s^2 + \frac{B}{F}st + \frac{C}{F}t^2 = A's^2 + B'st + C't^2 < 1.$$

We will not derive the equations that give the values of the coefficients, although the interested reader can easily verify their correctness.[5]

⟨*Compute ellipse coefficients to bound EWA filter region*⟩ ≡ 541

```
float A = dt0*dt0 + dt1*dt1 + 1;
float B = -2.f * (ds0*dt0 + ds1*dt1);
float C = ds0*ds0 + ds1*ds1 + 1;
float invF = 1.f / (A*C - B*B*0.25f);
A *= invF;
B *= invF;
C *= invF;
```

The next step is to find the axis-aligned bounding box in discrete integer texel coordinates of the texels that are potentially inside the ellipse. The EWA algorithm loops over all of these candidate texels, filtering the contributions of those that are in fact inside the ellipse. The bounding box is found by determining the minimum and maximum values that the ellipse takes in the s and t directions. These extrema can be calculated by finding the partial derivatives $\partial e/\partial s$ and $\partial e/\partial t$, finding their solutions for $s = 0$ and $t = 0$, and adding the offset to the ellipse center. For brevity, we will not include the derivation for these expressions here.

⟨*Compute the ellipse's* (s, t) *bounding box in texture space*⟩ ≡ 541

```
float det = -B*B + 4.f*A*C;
float invDet = 1.f / det;
float uSqrt = sqrtf(det * C), vSqrt = sqrtf(A * det);
int s0 = Ceil2Int (s - 2.f * invDet * uSqrt);
int s1 = Floor2Int(s + 2.f * invDet * uSqrt);
int t0 = Ceil2Int (t - 2.f * invDet * vSqrt);
int t1 = Floor2Int(t + 2.f * invDet * vSqrt);
```

Now that the bounding box is known, the EWA algorithm loops over the texels, transforming each one to the coordinate system where the texture lookup point (s, t) is at the origin with a translation. It then evaluates the ellipse equation to see if the texel is inside the ellipse (Figure 10.16). The value of the implicit ellipse equation $e(s, t)$ gives the squared ratio of the distance from the texel to the ellipse center to the distance from

5 Heckbert's thesis has the original derivation (Heckbert 1989, p. 80). A and C have an extra term of 1 added to them so the ellipse is a minimum of one texel separation wide. This ensures that the ellipse will not fall between the texels when magnifying at the most detailed level.

Ceil2Int() 1002
Floor2Int() 1002

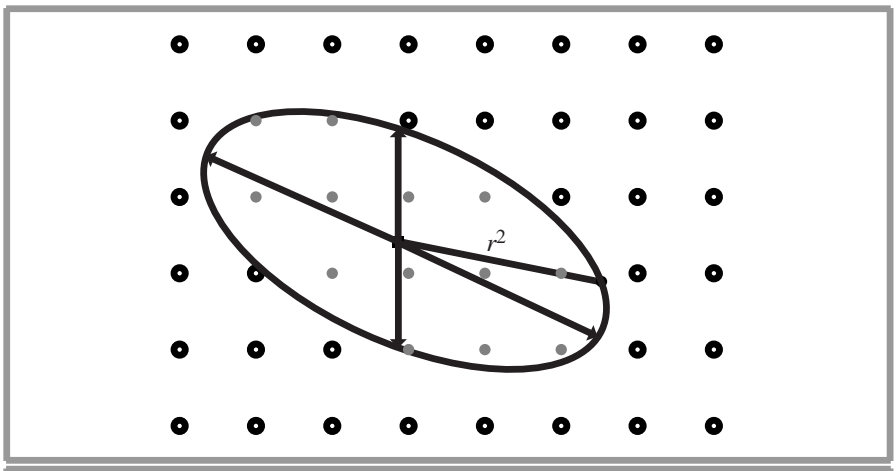


Figure 10.16: Finding the r^2 Ellipse Value for the EWA Filter Table Lookup.

the ellipse edge to the center along the line through the texel. If it is inside, the weight of the texel is computed with a Gaussian centered at the middle of the ellipse. The final filtered value returned is a weighted sum over texels (s', t') inside the ellipse, where f is the Gaussian filter function:

$$\frac{\sum f(s'-s, t'-t)t(s', t')}{\sum f(s'-s, t'-t)}.$$

⟨*Scan over ellipse bound and compute quadratic equation*⟩ ≡ 541

```
T sum(0.);
float sumWts = 0.f;
for (int it = t0; it <= t1; ++it) {
    float tt = it - t;
    for (int is = s0; is <= s1; ++is) {
        float ss = is - s;
        ⟨Compute squared radius and filter texel if inside ellipse 544⟩
    }
}
return sum / sumWts;
```

A nice feature of the implicit equation $e(s, t)$ is that its value at a particular texel is the squared ratio of the distance from the center of the ellipse to the texel to the distance from the center of the ellipse to the ellipse boundary along the line through that texel (Figure 10.16). This value is used to index into a precomputed lookup table of Gaussian filter function values.

⟨*Compute squared radius and filter texel if inside ellipse*⟩ ≡ 543
```
float r2 = A*ss*ss + B*ss*tt + C*tt*tt;
if (r2 < 1.) {
    float weight = weightLut[min(Float2Int(r2 * WEIGHT_LUT_SIZE),
                                 WEIGHT_LUT_SIZE-1)];
    sum += Texel(level, is, it) * weight;
    sumWts += weight;
}
```

The lookup table is initialized the first time a `MIPMap` is constructed. Because it will be indexed with squared distances from the filter center r^2, each entry stores a value $e^{-\alpha r}$, rather than $e^{-\alpha r^2}$.

⟨*MIPMap Private Data*⟩ +≡ 530
```
#define WEIGHT_LUT_SIZE 128
static float *weightLut;
```

So that the filter function goes to zero at the end of its extent rather than having an abrupt step, `expf(-alpha)` is subtracted from the filter values here.

⟨*Initialize EWA filter weights if needed*⟩ ≡ 530
```
if (!weightLut) {
    weightLut = AllocAligned<float>(WEIGHT_LUT_SIZE);
    for (int i = 0; i < WEIGHT_LUT_SIZE; ++i) {
        float alpha = 2;
        float r2 = float(i) / float(WEIGHT_LUT_SIZE - 1);
        weightLut[i] = expf(-alpha * r2) - expf(-alpha);
    }
}
```

10.5 SOLID AND PROCEDURAL TEXTURING

Once one starts to think of the (s, t) texture coordinates used by 2D texture functions as quantities that can be computed by arbitrary functions and not just from the parametric coordinates of the surface, it is natural to generalize texture functions to be defined over three-dimensional domains (often called *solid textures*) rather than just 2D (s, t). One reason solid textures are particularly convenient is that all objects have a natural three-dimensional texture mapping—object space position. This is a substantial advantage for texturing objects that don't have a natural two-dimensional parameterization (e.g., triangle meshes and implicit surfaces), and for objects that have a distorted parameterization (e.g., near the poles of a sphere). In preparation for this idea, Section 10.2.5 defined a general `TextureMapping3D` interface to compute 3D texture coordinates as well as an `IdentityMapping3D` implementation.

Solid textures introduce a new problem, however: texture representation. A three-dimensional image map takes up a fair amount of storage space and is much harder to acquire than a two-dimensional texture map, which can be extracted from photographs

AllocAligned() 1013
Float2Int() 1002
IdentityMapping3D 519
MIPMap 530
MIPMap::Texel() 535
MIPMap::weightLut 544
MIPMap::WEIGHT_LUT_SIZE 544
TextureMapping3D 519

or painted by an artist. Therefore, procedural texturing—the idea that programs could be executed to generate texture values at arbitrary positions on surfaces in the scene—came into use at the same time that solid texturing was developed. A simple example of procedural texturing is a procedural sine wave. If we wanted to use a sine wave for bump mapping (for example, to simulate waves in water), it would be inefficient and potentially inaccurate to precompute values of the function at a grid of points and then store them in an image map. Instead, it makes much more sense to evaluate the `sin()` function at points on the surface as needed.

If we can find a three-dimensional function that describes the colors of the grain in a solid block of wood, for instance, then we can generate images of complex objects that appear to be carved from wood. Over the years, procedural texturing has grown in application considerably as techniques have been developed to describe more and more complex surfaces procedurally.

Procedural texturing has a number of interesting implications. First, it can be used to reduce memory requirements for rendering, by reducing the need for the storage of large, high-resolution texture maps. In addition, procedural shading gives the promise of potentially infinite detail; as the viewer approaches an object, the texturing function is evaluated at the points being shaded, which naturally leads to the right amount of detail being visible. In contrast, image texture maps become blurry when the viewer is too close to them. On the other hand, subtle details of the appearance of procedural textures can be much more difficult to control than image maps.

Another difficulty with procedural textures is antialiasing. Procedural textures are often expensive to evaluate, and sets of point samples that fully characterize their behavior aren't available as they are for with image maps. Because we would like to remove high-frequency information in the texture function before we take samples from it, we need to be aware of the frequency content of the various steps we take along the way so we can avoid introducing high frequencies. Although this sounds daunting, there are a handful of techniques that work well to handle this issue.

10.5.1 UV TEXTURE

Our first procedural texture converts the surface's (u, v) coordinates into the red and green components of a `Spectrum` (Figure 10.17). It is especially useful when debugging the parameterization of a new `Shape`, for example. It is defined in `textures/uv.h` and `textures/uv.cpp`.

```
⟨UVTexture Declarations⟩ ≡
    class UVTexture : public Texture<Spectrum> {
    public:
        ⟨UVTexture Public Methods 546⟩
    private:
        TextureMapping2D *mapping;
    };
```

Shape 108
Spectrum 263
Texture 519
TextureMapping2D 514

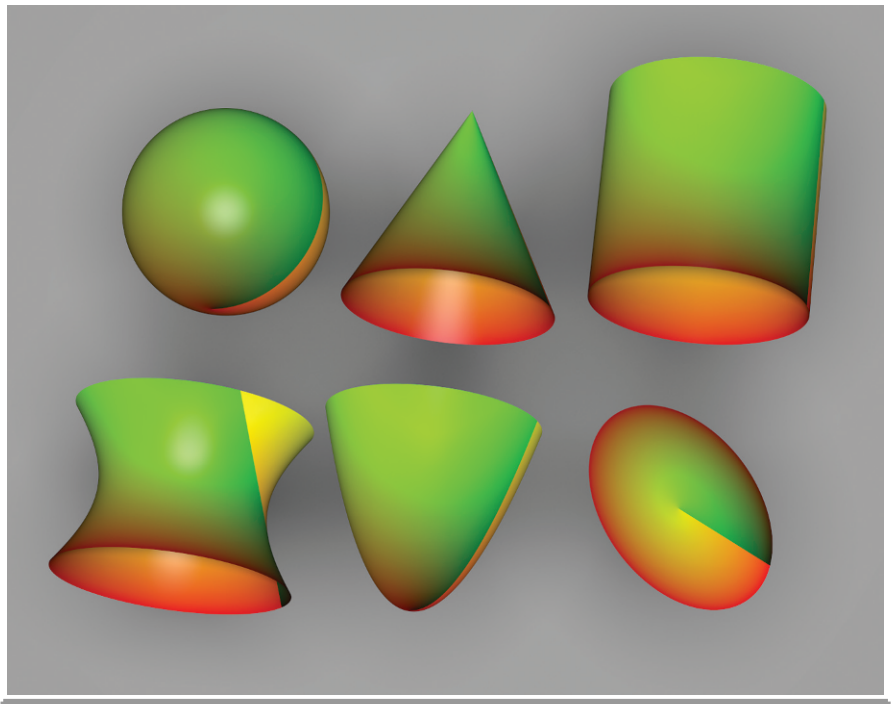


Figure 10.17: The UV Texture Applied to All of pbrt's Quadric Shapes. The u parameter is mapped to the red channel, and the v parameter is mapped to green.

⟨*UVTexture Public Methods*⟩ ≡ 545

```
Spectrum Evaluate(const DifferentialGeometry &dg) const {
    float s, t, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
    float rgb[3] = { s - Floor2Int(s), t - Floor2Int(t), 0.f };
    return Spectrum::FromRGB(rgb);
}
```

10.5.2 CHECKERBOARD

The checkerboard is the canonical procedural texture (Figure 10.18). The (s, t) texture coordinates are used to break up parameter space into square regions that are shaded with alternating patterns. Rather than just supporting checkerboards that switch between two fixed colors, the implementation here allows the user to pass in two textures to color the alternating regions. The traditional black-and-white checkerboard is obtained by passing two `ConstantTextures`. Its implementation is in the files `textures/checkerboard.h` and `textures/checkerboard.cpp`.

ConstantTexture 520
DifferentialGeometry 102
Floor2Int() 1002
Spectrum 263
Spectrum::FromRGB() 277
TextureMapping2D::Map() 514
UVTexture::mapping 545

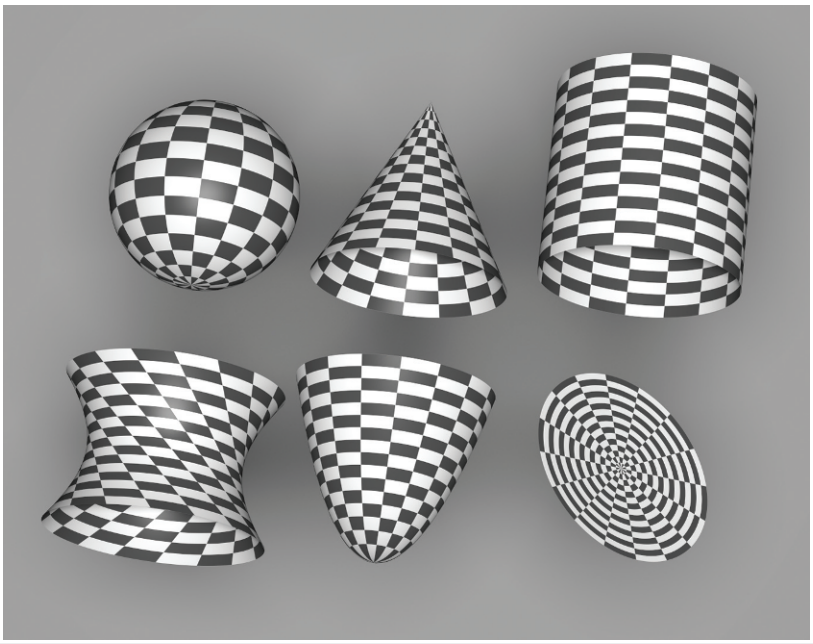


Figure 10.18: The Checkerboard Texture Applied to All of pbrt's Quadric Shapes.

⟨*CheckerboardTexture Declarations*⟩ ≡
```
template <typename T> class Checkerboard2DTexture : public Texture<T> {
public:
    ⟨Checkerboard2DTexture Public Methods 547⟩
private:
    ⟨Checkerboard2DTexture Private Data 547⟩
};
```

For simplicity, the frequency of the check function is 1 in (s, t) space: checks are one unit wide in each direction. The effective frequency can always be changed by the `TextureMapping2D` class with an appropriate scale of the (s, t) coordinates.

⟨*Checkerboard2DTexture Public Methods*⟩ ≡ 547
```
Checkerboard2DTexture(TextureMapping2D *m, Reference<Texture<T> > c1,
                      Reference<Texture<T> > c2, const string &aa)
    : mapping(m), tex1(c1), tex2(c2) {
    ⟨Select antialiasing method for Checkerboard2DTexture 548⟩
}
```

⟨*Checkerboard2DTexture Private Data*⟩ ≡ 547
```
TextureMapping2D *mapping;
Reference<Texture<T> > tex1, tex2;
```

Checkerboard2DTexture 547
Checkerboard2DTexture::mapping 547
Checkerboard2DTexture::tex1 547
Checkerboard2DTexture::tex2 547
Reference 1011
Texture 519
TextureMapping2D 514

The checkerboard is good for demonstrating trade-offs between various antialiasing approaches for procedural textures. The implementation here supports both simple point sampling (no antialiasing) and a closed-form box filter evaluated over the filter region. The image sequence in Figure 10.22 at the end of this section shows the results of these approaches.

⟨*Select antialiasing method for* Checkerboard2DTexture⟩ ≡ 547
```
if (aa == "none")             aaMethod = NONE;
else if (aa == "closedform")  aaMethod = CLOSEDFORM;
else {
    Warning("Antialiasing mode \"%s\" not understood by "
            "Checkerboard2DTexture; using \"closedform\"", aa.c_str());
    aaMethod = CLOSEDFORM;
}
```

⟨*Checkerboard2DTexture Private Data*⟩ +≡ 547
```
enum { NONE, CLOSEDFORM } aaMethod;
```

The evaluation routine does the usual texture coordinate and differential computation and then uses the appropriate fragment to compute an antialiased checkerboard value (or not antialiased, if point sampling has been selected).

⟨*Checkerboard2DTexture Public Methods*⟩ +≡ 547
```
T Evaluate(const DifferentialGeometry &dg) const {
    float s, t, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
    if (aaMethod == NONE) {
        ⟨Point sample Checkerboard2DTexture 548⟩
    }
    else {
        ⟨Compute closed-form box-filtered Checkerboard2DTexture value 549⟩
    }
}
```

The simplest case is to ignore antialiasing and just point-sample the checkerboard texture at the point. For this case, after getting the (s, t) texture coordinates from the TextureMapping2D, the integer checkerboard coordinates for that (s, t) position are computed, added together, and checked for odd or even parity to determine which of the two textures to evaluate.

⟨*Point sample* Checkerboard2DTexture⟩ ≡ 548, 550
```
if ((Floor2Int(s) + Floor2Int(t)) % 2 == 0)
    return tex1->Evaluate(dg);
return tex2->Evaluate(dg);
```

Given how bad aliasing can be in a point-sampled checkerboard texture, we will invest some effort to antialias it properly. The easiest case happens when the entire filter region lies inside a single check (Figure 10.19). In this case, we simply need to determine which of the check types we are inside and evaluate that one. As long as the Texture inside that check does appropriate antialiasing itself, the result for this case will be antialiased.

Checkerboard2DTexture::
aaMethod 548
Checkerboard2DTexture::
CLOSEDFORM 548
Checkerboard2DTexture::
mapping 547
Checkerboard2DTexture::
NONE 548
Checkerboard2DTexture::
tex1 547
Checkerboard2DTexture::
tex2 547
DifferentialGeometry 102
Floor2Int() 1002
Texture 519
Texture::Evaluate() 520
TextureMapping2D 514
TextureMapping2D::Map() 514
Warning() 1005

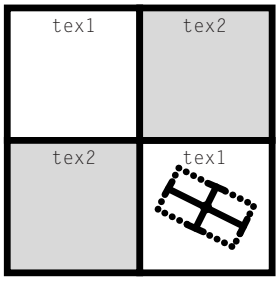


Figure 10.19: The Easy Case for Filtering the Checkerboard. If the filter region around the lookup point is entirely in one check, the checkerboard texture doesn't need to worry about antialiasing and can just evaluate the texture for that check.

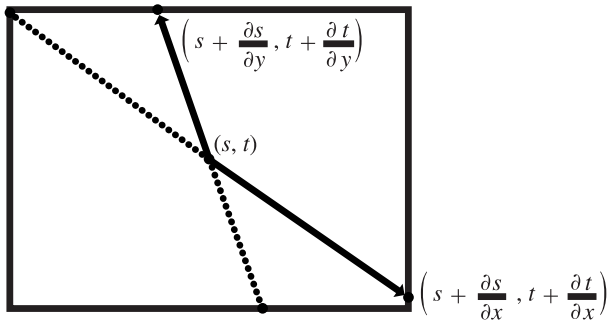


Figure 10.20: It is often convenient to use the axis-aligned bounding box around the texture evaluation point and the offsets from its partial derivatives as the region to filter over. Here, it's easy to see that the lengths of sides of the box are $2\max(|\partial s/\partial x|, |\partial s/\partial y|)$ and $2\max(|\partial t/\partial x|, |\partial t/\partial y|)$.

⟨*Compute closed-form box-filtered* `Checkerboard2DTexture` *value*⟩ ≡ 548
⟨*Evaluate single check if filter is entirely inside one of them* 550⟩
⟨*Apply box filter to checkerboard region* 551⟩

It's straightforward to check if the entire filter region is inside a single check by computing its bounding box and seeing if its extent lies inside the same check. For the remainder of this section, we will use the axis-aligned bounding box of the filter region given by the partial derivatives $\partial s/\partial x$, $\partial s/\partial y$, and so on, as the area to filter over, rather than trying to filter over the ellipse defined by the partial derivatives as the EWA filter did (Figure 10.20). This simplifies the implementation here, although somewhat increases the blurriness of the filtered values. The variables `ds` and `dt` in the following hold half the filter width in each direction, so the total area filtered over ranges from `(s-ds, t-dt)` to `(s+ds, t+dt)`.

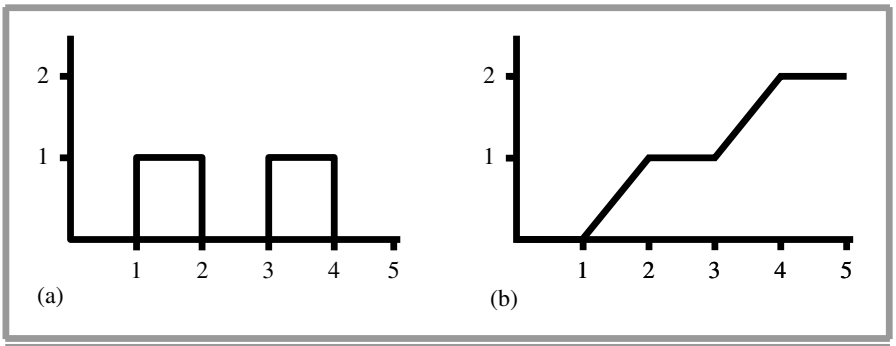


Figure 10.21: Integrating the Checkerboard Step Function. (a) The 1D step function that defines the checkerboard texture function, $c(x)$. (b) A graph of the value of the integral $\int_0^x c(x)\mathrm{d}x$.

⟨*Evaluate single check if filter is entirely inside one of them*⟩ ≡ 549

```
float ds = max(fabsf(dsdx), fabsf(dsdy));
float dt = max(fabsf(dtdx), fabsf(dtdy));
float s0 = s - ds, s1 = s + ds;
float t0 = t - dt, t1 = t + dt;
if (Floor2Int(s0) == Floor2Int(s1) && Floor2Int(t0) == Floor2Int(t1)) {
    ⟨Point sample Checkerboard2DTexture 548⟩
}
```

Otherwise, the lookup method approximates the filtered value by first computing a floating-point value that indicates what fraction of the filter region covers each of the two check types. This is equivalent to computing the average of the 2D step function that takes on the value 0 when we are in `tex1` and 1 when we are in `tex2`, over the filter region. Figure 10.21(a) shows a graph of the checkerboard function $c(x)$, defined as

$$c(x) = \begin{cases} 0 & \lfloor x \rfloor \text{ is odd} \\ 1 & \text{otherwise.} \end{cases}$$

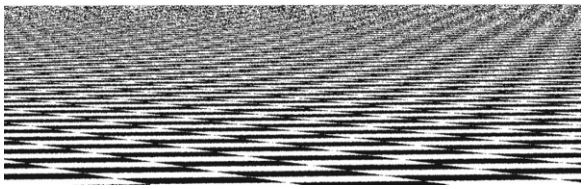
Given the average value, we can blend between the two subtextures, according to what fraction of the filter region each one is visible for.

The integral of the 1D checkerboard function $c(x)$ can be used to compute the average value of the function over some extent. Inspection of the graph reveals that

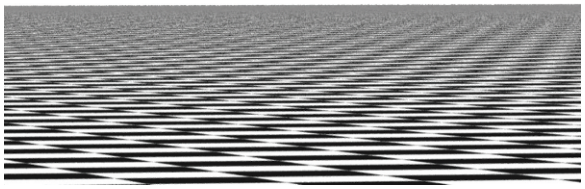
$$\int_0^x c(x)\,\mathrm{d}x = \lfloor x/2 \rfloor + 2\max(x/2 - \lfloor x/2 \rfloor - .5, 0).$$

To compute the average value of the step function in two dimensions, we separately compute the integral of the checkerboard in each 1D direction in order to compute its average value over the filter region.

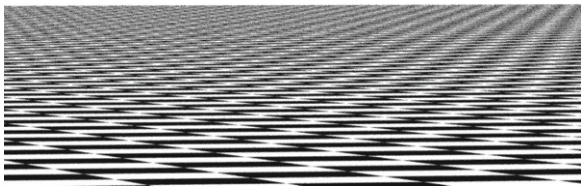
`Floor2Int()` 1002



(a)



(b)



(c)

Figure 10.22: Comparisons of three approaches for antialiasing procedural textures, applied to the checkerboard texture. (a) No effort has been made to remove high-frequency variation from the texture function, so there are severe artifacts in the image, rendered with one sample per pixel. (b) This image shows the approach based on computing the filter region in texture space and averaging the texture function over that area, also rendered with one sample per pixel. (c) Here the checkerboard function was effectively supersampled by taking 16 samples per pixel and then point-sampling the texture. Both the area-averaging and the supersampling approaches give substantially better results than the first approach. In this example, supersampling gives the best results, since the averaging approach has blurred out the checkerboard pattern sooner than was needed because it approximates the filter region with its axis-aligned box.

⟨*Apply box filter to checkerboard region*⟩ ≡ 549

```
#define BUMPINT(x) \
    (Floor2Int((x)/2) + \
     2.f * max((x/2)-Floor2Int(x/2) - .5f, 0.f))
float sint = (BUMPINT(s1) - BUMPINT(s0)) / (2.f * ds);
float tint = (BUMPINT(t1) - BUMPINT(t0)) / (2.f * dt);
float area2 = sint + tint - 2.f * sint * tint;
if (ds > 1.f || dt > 1.f)
    area2 = .5f;
return (1.f - area2) * tex1->Evaluate(dg) +
       area2         * tex2->Evaluate(dg);
```

Checkerboard2DTexture::
tex1 547

Texture::Evaluate() 520

Figure 10.23: The Dragon Model, Textured with the `Checkerboard3DTexture` Procedural Texture. Notice how the model appears to be carved out of 3D checks, rather than having them pasted on its surface.

10.5.3 SOLID CHECKERBOARD

The `Checkerboard2DTexture` class from the previous section wraps a checkerboard pattern *around* the object in parameter space. We can also define a solid checkerboard pattern based on three-dimensional texture coordinates so that the object appears carved out of 3D checker cubes (Figure 10.23). Like the 2D variant, this implementation chooses between texture functions based on the lookup position. Note that these two textures need not be solid textures themselves; the `Checkerboard3DTexture` merely chooses between them based on the 3D position of the point.

⟨*CheckerboardTexture Declarations*⟩ +≡

```
template <typename T> class Checkerboard3DTexture : public Texture<T> {
public:
    ⟨Checkerboard3DTexture Public Methods 552⟩
private:
    ⟨Checkerboard3DTexture Private Data 553⟩
};
```

⟨*Checkerboard3DTexture Public Methods*⟩ ≡ 552

```
Checkerboard3DTexture(TextureMapping3D *m, Reference<Texture<T> > c1,
                      Reference<Texture<T> > c2)
    : mapping(m), tex1(c1), tex2(c2) {
}
```

Checkerboard2DTexture 547
Checkerboard3DTexture 552
Checkerboard3DTexture::mapping 553
Checkerboard3DTexture::tex1 553
Checkerboard3DTexture::tex2 553
Reference 1011
Texture 519
TextureMapping3D 519

⟨*Checkerboard3DTexture Private Data*⟩ ≡ 552
```
TextureMapping3D *mapping;
Reference<Texture<T> > tex1, tex2;
```

Ignoring antialiasing, the basic computation to see if a point is inside a 3D checker region is

```
((Floor2Int(P.x) + Floor2Int(P.y) + Floor2Int(P.z)) % 2 == 0).
```

The `Checkerboard3DTexture` doesn't have any built-in support for antialiasing.

⟨*Checkerboard3DTexture Public Methods*⟩ +≡ 552
```
T Evaluate(const DifferentialGeometry &dg) const {
    Vector dpdx, dpdy;
    Point p = mapping->Map(dg, &dpdx, &dpdy);
    if ((Floor2Int(p.x) + Floor2Int(p.y) + Floor2Int(p.z)) % 2 == 0)
        return tex1->Evaluate(dg);
    else
        return tex2->Evaluate(dg);
}
```

10.6 NOISE

In order to write solid textures that describe complex surface appearances, it is helpful to be able to introduce some controlled variation to the process. Consider a wood floor made of individual planks; each plank's color is likely to be slightly different than the others. Or consider a windswept lake; we might want to have waves of similar amplitude across the entire lake, but we don't want them to be homogeneous over all parts of the lake (as they might be if they were constructed from a sum of sine waves, for example). Modeling this sort of variation in a texture helps make the final result look more realistic.

One difficulty in developing textures like these is that the renderer evaluates the surface's texture functions at an irregularly distributed set of points, where each evaluation is completely independent of the others. As such, procedural textures must *implicitly* define a complex pattern by answering queries about what the pattern's value is at all of these points. In contrast, the *explicit* pattern description approach is embodied by the PostScript® language, for example, which describes graphics on a page with a series of drawing commands. One difficulty that the implicit approach introduces is that the texture can't just call `RNG::RandomFloat()` at each point at which it is evaluated to introduce randomness: because each point would have a completely different random value than its neighbors, no coherence would be possible in the generated pattern.

An elegant way to address this issue of introducing controlled randomness to procedural textures in graphics is the application of what is known as a *noise function*. In general, noise functions used in graphics are smoothly varying functions taking $\mathbb{R}^n \rightarrow [-1, 1]$, for at least $n = 1, 2, 3$, without obvious repetition. One of the most crucial properties of a practical noise function is that it be band limited with a known maximum frequency. This makes it possible to control the frequency content added to a texture due to the

Checkerboard3DTexture::mapping 553
Checkerboard3DTexture::tex1 553
Checkerboard3DTexture::tex2 553
DifferentialGeometry 102
Floor2Int() 1002
Point 63
Reference 1011
RNG::RandomFloat() 1003
Texture 519
Texture::Evaluate() 520
TextureMapping3D 519
TextureMapping3D::Map() 519
Vector 57

noise function so that frequencies higher than those allowed by the Nyquist limit aren't introduced.

Many of the noise functions that have been developed are built on the idea of an integer lattice over $\mathbb{R}^3$. First, a value is associated with each integer (x, y, z) position in space. Then, given an arbitrary position in space, the eight surrounding lattice values are found. These lattice values are then interpolated to compute the noise value at the particular point. This idea can be generalized or restricted to more or fewer dimensions d, where the number of lattice points is 2^d. A simple example of this approach is *value noise*, where pseudo-random numbers between -1 and 1 are associated with each lattice point, and actual noise values are computed with trilinear interpolation or with a more complex spline interpolant, which can give a smoother result by avoiding derivative discontinuities when moving from one lattice cell to another.

For such a noise function, given an integer (x, y, z) lattice point, it must be possible to efficiently compute its parameter value in a way that always associates the same value with each lattice point. Because it is infeasible to store values for all possible (x, y, z) points, some compact representation is needed. One option is to use a hash function, where the coordinates are hashed and then used to look up parameters from a fixed-size table of precomputed pseudo-random parameter values.

10.6.1 PERLIN NOISE

In `pbrt` we will implement a noise function introduced by Ken Perlin (1985a, 2002); as such, it is known as *Perlin noise*. It has a value of zero at all (x, y, z) integer lattice points. Its variation comes from gradient vectors at each lattice point that guide the interpolation of a smooth function in between the points (Figure 10.24). This noise function has many

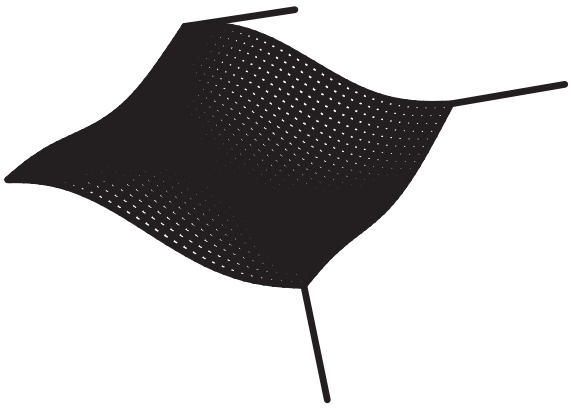


Figure 10.24: The Perlin noise function is computed by generating a smooth function that is zero but with a given derivative at integer lattice points. The derivatives are used to compute a smooth interpolating surface. Here, a 2D slice of the noise function is shown with four gradient vectors.

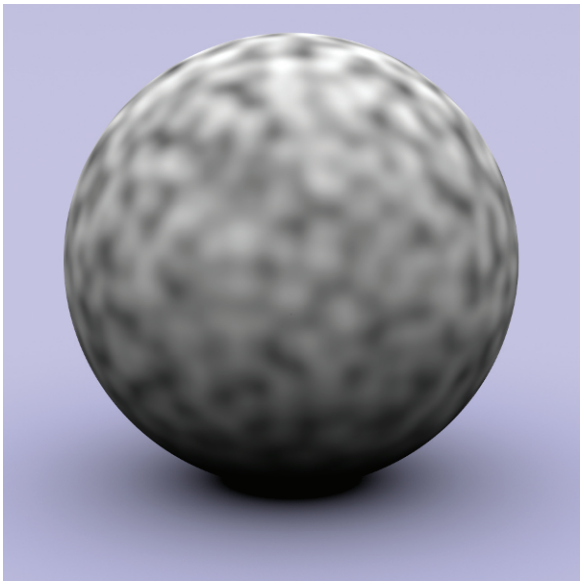


Figure 10.25: Perlin's Noise Function Modulating the Diffuse Color of a Sphere.

of the desired characteristics of a noise function described above, is computationally efficient, and is easy to implement. Figure 10.25 shows its value rendered on a sphere.

⟨*Texture Method Definitions*⟩ +≡
```
float Noise(float x, float y, float z) {
    ⟨Compute noise cell coordinates and offsets 555⟩
    ⟨Compute gradient weights 556⟩
    ⟨Compute trilinear interpolation of weights 558⟩
}
```

For convenience, there is also a variant of `Noise()` that takes a `Point` directly:

⟨*Texture Method Definitions*⟩ +≡
```
float Noise(const Point &P) { return Noise(P.x, P.y, P.z); }
```

The implementation first computes the integer coordinates of the cell that contains the given point and the fractional offsets of the point from the lower cell corner:

⟨*Compute noise cell coordinates and offsets*⟩ ≡ 555
```
int ix = Floor2Int(x), iy = Floor2Int(y), iz = Floor2Int(z);
float dx = x - ix, dy = y - iy, dz = z - iz;
```

Floor2Int() 1002
Noise() 555
Point 63

It next computes eight weight values, one for each corner of the cell that the point lies inside. Each integer lattice point has a gradient vector associated with it. The influence of the gradient vector for any point inside the cell is obtained by computing the dot product of the vector from the gradient's corner to the lookup point and the gradient

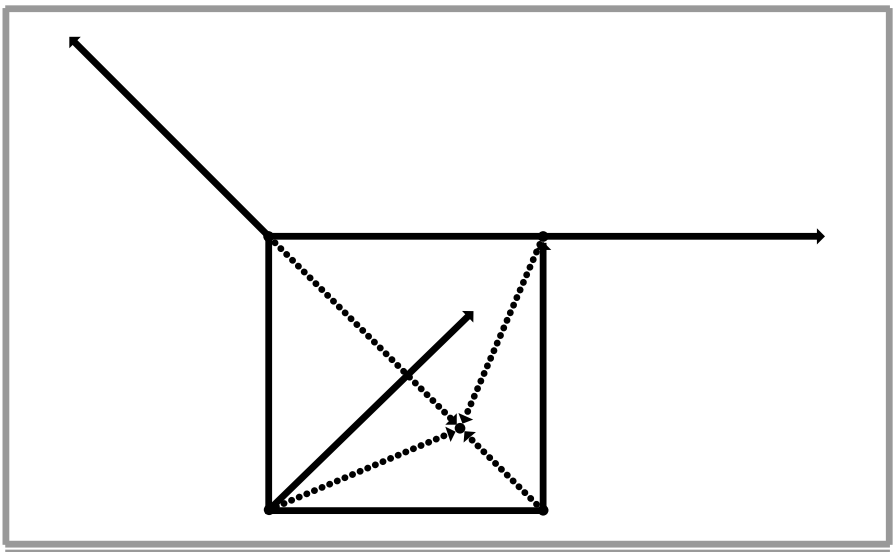


Figure 10.26: The dot product of the vector from the corners of the cell to the lookup point (dotted lines) with each of the gradient vectors (solid lines) gives the influence of each gradient to the noise value at the point.

vector (Figure 10.26); this is handled by the `Grad()` function. Note that the vectors to the corners other than the lower-left one can be easily computed incrementally based on that vector.

⟨*Compute gradient weights*⟩ ≡ 555

```
ix &= (NOISE_PERM_SIZE-1);
iy &= (NOISE_PERM_SIZE-1);
iz &= (NOISE_PERM_SIZE-1);
float w000 = Grad(ix,   iy,   iz,   dx,   dy,   dz);
float w100 = Grad(ix+1, iy,   iz,   dx-1, dy,   dz);
float w010 = Grad(ix,   iy+1, iz,   dx,   dy-1, dz);
float w110 = Grad(ix+1, iy+1, iz,   dx-1, dy-1, dz);
float w001 = Grad(ix,   iy,   iz+1, dx,   dy,   dz-1);
float w101 = Grad(ix+1, iy,   iz+1, dx-1, dy,   dz-1);
float w011 = Grad(ix,   iy+1, iz+1, dx,   dy-1, dz-1);
float w111 = Grad(ix+1, iy+1, iz+1, dx-1, dy-1, dz-1);
```

The gradient vector for a particular integer lattice point is found by indexing into a precomputed table of integer values, `NoisePerm`. The four low-order bits of the table's value for the lattice point determine which of 16 gradient vectors is associated with it. In a preprocessing step, this table of size `NOISE_PERM_SIZE` was filled with numbers from 0 to `NOISE_PERM_SIZE-1` and then randomly permuted. These values were then duplicated, making an array of size `2*NOISE_PERM_SIZE` that holds the table twice in succession. The second copy of the table makes lookups in the following code slightly more efficient.

`Grad()` 557
`NOISE_PERM_SIZE` 557

Given a particular (ix,iy,iz) lattice point, a series of table lookups gives a value from the random-number table:

```
NoisePerm[NoisePerm[NoisePerm[ix]+iy]+iz];
```

By doing three nested permutations in this way, rather than NoisePerm[ix+iy+iz], for example, the final result is more irregular. For example, the first approach usually doesn't generally return the same value if ix and iy are interchanged, as the second does. Furthermore, since the table was replicated to be twice the original length, the lookups can be done as described above, eliminating the need for modulus operations in code along the lines of

```
(NoisePerm[ix]+iy) % NOISE_PERM_SIZE
```

Given a final value from the permutation table that determines the gradient number, the dot product with the corresponding gradient vector must be computed. However, the gradient vectors do not need to be represented explicitly. All of the gradients use only −1, 0, or 1 in their coordinates, so that the dot products reduce to addition of some (possibly negated) components of the vector.[6] The final implementation is the following:

⟨*Texture Method Definitions*⟩ +≡

```
inline float Grad(int x, int y, int z, float dx, float dy, float dz) {
    int h = NoisePerm[NoisePerm[NoisePerm[x]+y]+z];
    h &= 15;
    float u = h<8 || h==12 || h==13 ? dx : dy;
    float v = h<4 || h==12 || h==13 ? dy : dz;
    return ((h&1) ? -u : u) + ((h&2) ? -v : v);
}
```

⟨*Perlin Noise Data*⟩ ≡

```
#define NOISE_PERM_SIZE 256
static int NoisePerm[2 * NOISE_PERM_SIZE] = {
    151, 160, 137, 91, 90, 15, 131, 13, 201, 95, 96,
    53, 194, 233, 7, 225, 140, 36, 103, 30, 69, 142,
    ⟨Remainder of the noise permutation table⟩
};
```

Given these eight contributions from the gradients, the next step is to trilinearly interpolate between them at the point. Rather than interpolating with dx, dy, and dz directly, though, each of these values is passed through a smoothing function. This ensures that the noise function has first- and second-derivative continuity as lookup points move between lattice cells.

NoisePerm 557

NOISE_PERM_SIZE 557

6 The original formulation of Perlin noise also had a precomputed table of pseudo-random gradient directions, although Perlin has more recently suggested that the randomness from the permutation table is enough to remove regularity from the noise function.

⟨*Texture Method Definitions*⟩ +≡

```
inline float NoiseWeight(float t) {
    float t3 = t*t*t;
    float t4 = t3*t;
    return 6.f*t4*t - 15.f*t4 + 10.f*t3;
}
```

⟨*Compute trilinear interpolation of weights*⟩ ≡ 555

```
float wx = NoiseWeight(dx), wy = NoiseWeight(dy), wz = NoiseWeight(dz);
float x00 = Lerp(wx, w000, w100);
float x10 = Lerp(wx, w010, w110);
float x01 = Lerp(wx, w001, w101);
float x11 = Lerp(wx, w011, w111);
float y0 = Lerp(wy, x00, x10);
float y1 = Lerp(wy, x01, x11);
return Lerp(wz, y0, y1);
```

10.6.2 RANDOM POLKA DOTS

A basic use of the noise function is as part of a polka dot texture that divides (s, t) texture space into rectangular cells (Figure 10.27). Each cell has a 50% chance of having a dot inside of it, and the dots are randomly placed inside their cells. `DotsTexture` takes the

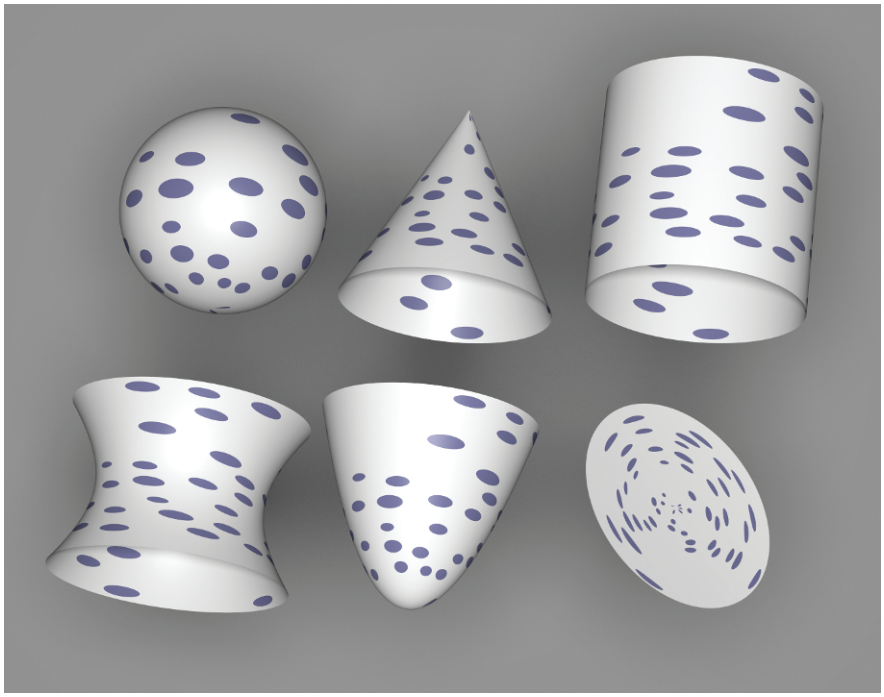


Figure 10.27: The Polka Dot Texture Applied to All of `pbrt`'s Quadric Shapes.

DotsTexture 559
Lerp() 1000
NoiseWeight() 558

usual 2D mapping function, as well as two Textures, one for the regions of the surface outside of the dots and one for the regions inside. It is defined in the files textures/dots.h and textures/dots.cpp.

⟨*DotsTexture Declarations*⟩ ≡
```
template <typename T> class DotsTexture : public Texture<T> {
public:
    ⟨DotsTexture Public Methods 559⟩
private:
    ⟨DotsTexture Private Data 559⟩
};
```

⟨*DotsTexture Public Methods*⟩ ≡ 559
```
DotsTexture(TextureMapping2D *m, Reference<Texture<T> > t1,
            Reference<Texture<T> > t2)
    : mapping(m), outsideDot(t1), insideDot(t2) {
}
```

⟨*DotsTexture Private Data*⟩ ≡ 559
```
TextureMapping2D *mapping;
Reference<Texture<T> > outsideDot, insideDot;
```

The evaluation function starts by taking the (s, t) texture coordinates and computing integer sCell and tCell values, which give the coordinates of the cell they are inside. We will not consider antialiasing of the polka dots texture here; an exercise at the end of the chapter outlines how this might be done.

⟨*DotsTexture Public Methods*⟩ +≡ 559
```
T Evaluate(const DifferentialGeometry &dg) const {
    ⟨Compute cell indices for dots 559⟩
    ⟨Return insideDot result if point is inside dot 560⟩
    return outsideDot->Evaluate(dg);
}
```

⟨*Compute cell indices for dots*⟩ ≡ 559
```
float s, t, dsdx, dtdx, dsdy, dtdy;
mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
int sCell = Floor2Int(s + .5f), tCell = Floor2Int(t + .5f);
```

DifferentialGeometry 102
DotsTexture 559
DotsTexture::insideDot 559
DotsTexture::mapping 559
DotsTexture::outsideDot 559
Floor2Int() 1002
Reference 1011
Texture 519
Texture::Evaluate() 520
TextureMapping2D 514
TextureMapping2D::Map() 514

Once the cell coordinate is known, it's necessary to decide if there is a polka dot in the cell. Obviously, this computation needs to be consistent so that for each time this routine runs for points in a particular cell it returns the same result. Yet we'd like the result not to be completely regular (e.g., with a dot in every other cell). Noise solves this problem: by evaluating the noise function at a position that is the same for all points inside this cell—sCell+.5, tCell+.5—we can compute an irregularly varying but consistent value for each cell.[7] If this value is greater than zero, a dot is placed in the cell.

7 Recall that the noise function always returns zero at integer (x, y, z) coordinates, so we don't want to just evaluate it at (sCell, tCell). Although the 3D noise function would actually be evaluating noise at sCell, tCell, .5, slices through noise with integer values for any of the coordinates are not as well-distributed as with all of them offset.

If there is a dot in the cell, the noise function is used again to randomly shift the center of the dot within the cell. The points at which the noise function is evaluated for the center shift are offset by arbitrary constant amounts, however, so that noise values from different noise cells are used from them, eliminating a possible source of correlation with the noise value used to determine the presence of a dot in the first place. (But the dot's radius needs to be small enough so that it doesn't spill over the cell's boundary after being shifted; in that case, points where the texture was being evaluated would also need to consider the dots based in neighboring cells as potentially affecting their value.)

Given the dot center and radius, the texture needs to decide if the (s, t) coordinates are within the radius of the shifted center. It does this by computing their squared distance to the center and comparing it to the squared radius.

⟨*Return* `insideDot` *result if point is inside dot*⟩ ≡ 559

```
if (Noise(sCell+.5f, tCell+.5f) > 0) {
    float radius = .35f;
    float maxShift = 0.5f - radius;
    float sCenter = sCell + maxShift *
        Noise(sCell + 1.5f, tCell + 2.8f);
    float tCenter = tCell + maxShift *
        Noise(sCell + 4.5f, tCell + 9.8f);
    float ds = s - sCenter, dt = t - tCenter;
    if (ds*ds + dt*dt < radius*radius)
        return insideDot->Evaluate(dg);
}
```

10.6.3 NOISE IDIOMS AND SPECTRAL SYNTHESIS

The fact that noise is a band-limited function means that its frequency content can be adjusted by scaling the domain over which it is evaluated. For example, if `Noise(p)` has some known frequency content, then the frequency content of `Noise(2*p)` will be twice as high. This is just like the relationship between the frequency content of $\sin(x)$ and $\sin(2x)$. This technique can be used to create a noise function with a desired rate of variation.

For many applications in procedural texturing, it's useful to have variation over multiple scales—for example, to add finer variations to the base noise function. One effective way to do this with noise is to compute patterns via *spectral synthesis*, where a complex function $f_s(s)$ is defined by a sum of contributions from another function $f(x)$:

$$f_s(x) = \sum_i w_i f(s_i x),$$

for a set of weight values w_i and parameter scale values s_i. If the base function $f(x)$ has a well-defined frequency content (e.g., is a sine or cosine function, or a noise function), then each term $f(s_i x)$ also has a well-defined frequency content as described earlier. Because each term of the sum is weighted by a weight value w_i, the result is a sum of contributions of various frequencies, with different frequency ranges weighted differently.

Typically, the scales s_i are chosen in a geometric progression such that $s_i = 2s_{i-1}$ and the weights are $w_i = w_{i-1}/2$. The result is that as higher-frequency variation is added to

DotsTexture::insideDot 559
Noise() 555
Texture::Evaluate() 520

the function, it has relatively less influence on the overall shape of $f_s(x)$. Each additional term is called an *octave* of noise, since it has twice the frequency content of the previous one. When this scheme is used with Perlin noise, the result is often referred to as "fractional Brownian motion" (fBm), after a particular type of random process that varies in a similar manner.

Fractional Brownian motion is a useful building block for procedural textures because it gives a function with more complex variation than plain noise, while still being easy to compute and still having well-defined frequency content. The utility function `FBm()` implements the fractional Brownian motion function. Figure 10.28 shows two graphs of it.

In addition to the point at which to evaluate the function and the function's partial derivatives at that point, the function takes an `omega` parameter, which ranges from zero to one and affects the smoothness of the pattern by controlling the falloff of contributions at higher frequencies (values around 0.5 work well), and `maxOctaves`, which gives the maximum number of octaves of noise that should be used in computing the sum.

⟨*Texture Method Definitions*⟩ +≡

```
float FBm(const Point &P, const Vector &dpdx, const Vector &dpdy,
          float omega, int maxOctaves) {
    ⟨Compute number of octaves for antialiased FBm 562⟩
    ⟨Compute sum of octaves of noise for FBm 563⟩
    return sum;
}
```

Antialiasing the fBm function is based on a technique called *clamping* (Norton, Rockwood, and Skolmoski 1982). The idea is that when we are computing a value based on a sum of components, each with known frequency content, we should stop adding in components that would have frequencies beyond the Nyquist limit and instead add their average values to the sum. Because the average value of `Noise()` is zero, all that needs to be done is to compute the number of octaves such that none of the terms has excessively high frequencies and not evaluate the noise function for any higher octaves.

`Noise()` (and thus the first term of $f_s(x)$ as well) has a maximum frequency content of roughly $\omega = 1$. Each subsequent term represents a doubling of frequency content. Therefore, we would like to find the appropriate number of terms n such that if the sampling rate in noise space is s, we have

$$2^n s = 2\omega = 2.$$

This condition guarantees that there are no frequencies that can't be represented at the sampling rate. Thus, we have

$$2^{n-1} = \frac{1}{s}$$

$$n - 1 = \log\left(\frac{1}{s}\right)$$

$$n = 1 - \frac{1}{2}\log(s^2).$$

FBm() 561
Noise() 555
Point 63
Vector 57

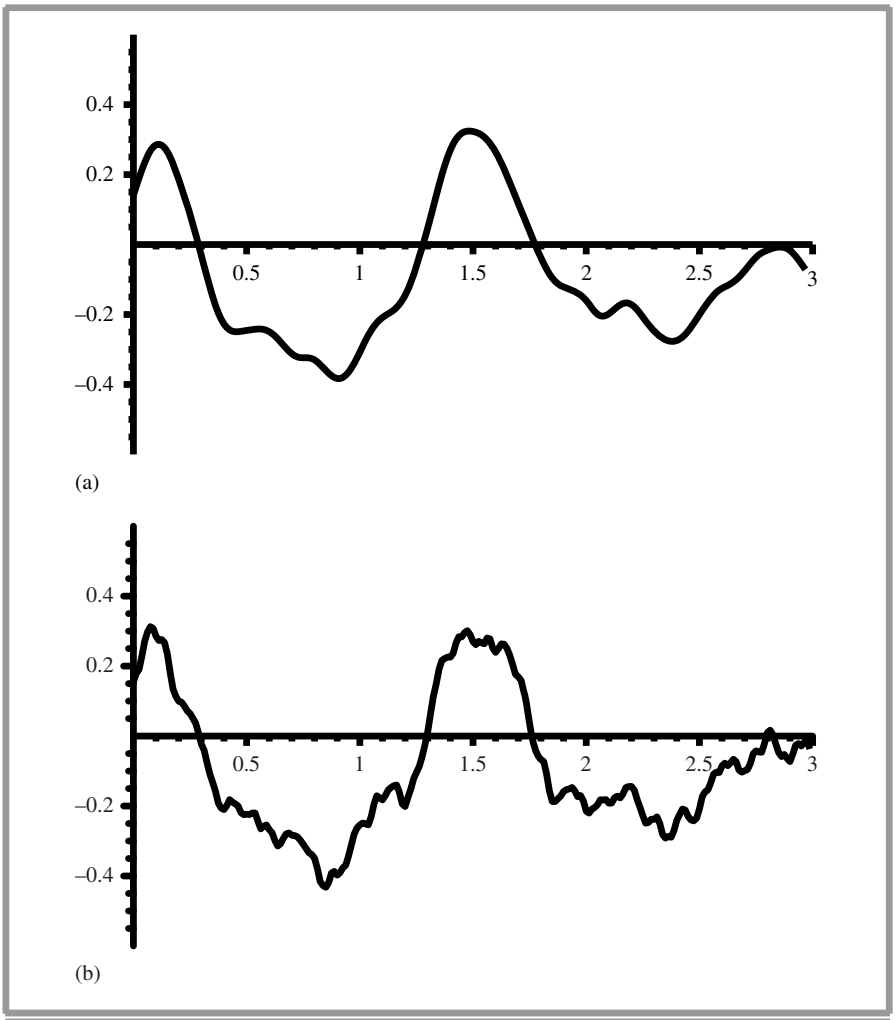


Figure 10.28: Graphs of the `FBm()` Function. (a) 2 and (b) 6 octaves of noise. Notice how as more levels of noise are added, the graph has progressively more detail, although its overall shape remains roughly the same.

The squared sampling rate s^2 can be computed by finding the maximum of the length of the differentials $\partial \mathrm{p}/\partial x$ and $\partial \mathrm{p}/\partial y$.

⟨*Compute number of octaves for antialiased FBm*⟩ ≡ **561, 564**

```
float s2 = max(dpdx.LengthSquared(), dpdy.LengthSquared());
float foctaves = min((float)maxOctaves, 1.f - .5f * Log2(s2));
int octaves = Floor2Int(foctaves);
```

FBm() 561
Floor2Int() 1002
Log2() 1001
Vector::LengthSquared() 62

Finally, the integral number of octaves up to the Nyquist limit are added together and the last octave is faded in, according to the fractional part of `foctaves`. This ensures that

successive octaves of noise fade in gradually, rather than appearing abruptly, which can cause visually noticeable artifacts at the transitions. The implementation here actually increases the frequency between octaves by 1.99, rather than by a factor of 2, in order to reduce the impact of the fact that the noise function is zero at integer lattice points. This breaks up that regularity across sums of octaves of noise, which can also lead to subtle visual artifacts.

⟨*Compute sum of octaves of noise for FBm*⟩ ≡ 561
```
float sum = 0., lambda = 1., o = 1.;
for (int i = 0; i < octaves; ++i) {
    sum += o * Noise(lambda * P);
    lambda *= 1.99f;
    o *= omega;
}
float partialOctave = foctaves - octaves;
sum += o * SmoothStep(.3f, .7f, partialOctave) * Noise(lambda * P);
```

The `SmoothStep()` function takes a minimum and maximum value and a point at which to evaluate a smooth interpolating function. If the point is below the minimum zero is returned, and if it's above the maximum one is returned. Otherwise, it smoothly interpolates between zero and one using a cubic Hermite spline.

⟨*Texture Inline Functions*⟩ ≡
```
inline float SmoothStep(float min, float max, float value) {
    float v = Clamp((value - min) / (max - min), 0.f, 1.f);
    return v * v * (-2.f * v  + 3.f);
}
```

Closely related to the `FBm()` function is the `Turbulence()` function. It also computes a sum of terms of the noise function, but takes the absolute value of each one:

$$f_s(x) = \sum_i w_i |f(s_i x)|.$$

Taking the absolute value introduces first-derivative discontinuities in the synthesized function and thus the turbulence function has infinite frequency content. Nevertheless, the visual characteristics of this function can be quite useful for procedural textures. Figure 10.29 shows two graphs of the turbulence function.

The `Turbulence()` implementation here tries to antialias itself in the same way that `FBm()` did. As described earlier, however, the first-derivative discontinuities in the function introduce infinitely high-frequency content, so these efforts can't hope to be perfectly successful. The `Turbulence()` antialiasing here at least eliminates some of the worst of the artifacts; otherwise, increasing the pixel sampling rate is the best recourse. In practice, this function doesn't alias too terribly when used in procedural textures, particularly compared to the aliasing from infinitely high frequencies from geometric and shadow edges.

Clamp() 1000
FBm() 561
Noise() 555
SmoothStep() 563
Turbulence() 564

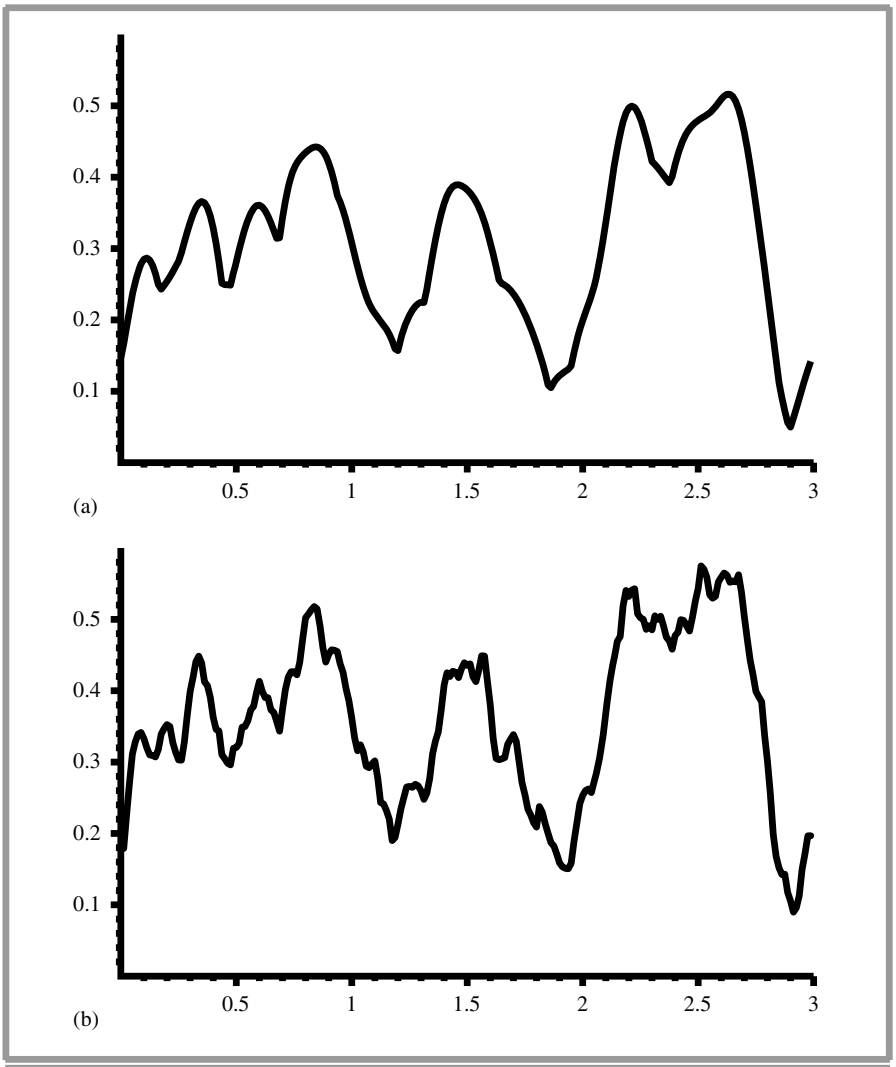


Figure 10.29: Graphs of the `Turbulence()` function for (a) 2 and (b) 6 octaves of noise. Note that the first derivative discontinuities introduced by taking the absolute value of the noise function make this function substantially more rough than FBm.

⟨*Texture Method Definitions*⟩ +≡

```
float Turbulence(const Point &P, const Vector &dpdx, const Vector &dpdy,
                 float omega, int maxOctaves) {
    ⟨Compute number of octaves for antialiased FBm 562⟩
    ⟨Compute sum of octaves of noise for turbulence 565⟩
    return sum;
}
```

Point 63
Turbulence() 564
Vector 57

⟨*Compute sum of octaves of noise for turbulence*⟩ ≡ 564
```
float sum = 0., lambda = 1., o = 1.;
for (int i = 0; i < octaves; ++i) {
    sum += o * fabsf(Noise(lambda * P));
    lambda *= 1.99f;
    o *= omega;
}
float partialOctave = foctaves - octaves;
sum += o * SmoothStep(.3f, .7f, partialOctave) *
       fabsf(Noise(lambda * P));
```

10.6.4 BUMPY AND WRINKLED TEXTURES

The fBm and turbulence functions are particularly useful as a source of random variation for bump mapping. The `FBmTexture` is a `float`-valued texture that uses `FBm()` to compute offsets, and `WrinkledTexture` uses `Turbulence()` to do so. They are demonstrated in Figures 10.30 and 10.31 and are implemented in `textures/fbm.h`, `textures/fbm.cpp`, `textures/wrinkled.h`, and `textures/wrinkled.cpp`.

⟨*FBmTexture Declarations*⟩ ≡
```
template <typename T> class FBmTexture : public Texture<T> {
public:
    ⟨FBmTexture Public Methods 566⟩
private:
    ⟨FBmTexture Private Data 566⟩
};
```

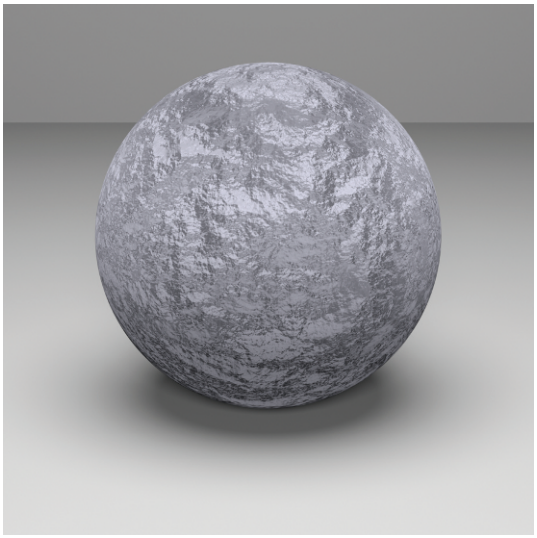


Figure 10.30: Sphere with `FBmTexture` Used for Bump Mapping.

FBm() 561
FBmTexture 565
Noise() 555
SmoothStep() 563
Texture 519
Turbulence() 564
WrinkledTexture 566

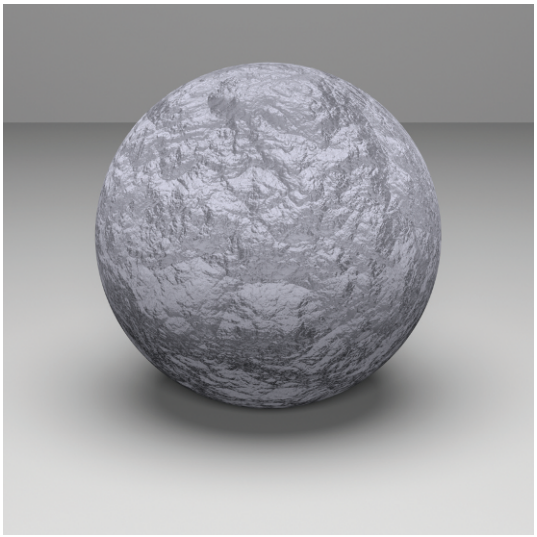


Figure 10.31: `WrinkledTexture` Used as Bump Mapping Function for Sphere.

⟨*FBmTexture Public Methods*⟩ ≡ 565
```
FBmTexture(int oct, float roughness, TextureMapping3D *map)
    : omega(roughness), octaves(oct), mapping(map) { }
```

⟨*FBmTexture Private Data*⟩ ≡ 565
```
float omega;
int octaves;
TextureMapping3D *mapping;
```

⟨*FBmTexture Public Methods*⟩ +≡ 565
```
T Evaluate(const DifferentialGeometry &dg) const {
    Vector dpdx, dpdy;
    Point P = mapping->Map(dg, &dpdx, &dpdy);
    return FBm(P, dpdx, dpdy, omega, octaves);
}
```

The implementation of `WrinkledTexture` is almost identical to `FBmTexture`, save for a call to `Turbulence()` instead of `FBm()`. As such, it isn't included here.

10.6.5 WINDY WAVES

Application of fBm can give a reasonably convincing representation of waves (Ebert et al. 2003). Figures 1.11, 4.1, and 7.36 use this texture for the water in those scenes. This `Texture` is based on two observations. First, across the surface of a wind-swept lake (for example), some areas are relatively smooth and some are more choppy; this effect comes from the natural variation of the wind's strength from area to area. Second, the overall

DifferentialGeometry 102
FBm() 561
FBmTexture 565
FBmTexture::mapping 566
FBmTexture::octaves 566
FBmTexture::omega 566
Point 63
Texture 519
TextureMapping3D 519
TextureMapping3D::Map() 519
Turbulence() 564
Vector 57
WrinkledTexture 566

form of individual waves on the surface can be described well by the fBm-based wave pattern scaled by the wind strength. This texture is implemented in `textures/windy.h` and `textures/windy.cpp`.

⟨*WindyTexture Declarations*⟩ ≡
```
template <typename T> class WindyTexture : public Texture<T> {
public:
    ⟨WindyTexture Public Methods 567⟩
private:
    ⟨WindyTexture Private Data 567⟩
};
```

⟨*WindyTexture Public Methods*⟩ ≡ 567
```
WindyTexture(TextureMapping3D *map) : mapping(map) { }
```

⟨*WindyTexture Private Data*⟩ ≡ 567
```
TextureMapping3D *mapping;
```

The evaluation function uses two calls to the `FBm()` function. The first scales down the point `P` by a factor of 10; as a result, the first call to `FBm()` returns relatively low-frequency variation over the surface of the object being shaded. This value is used to determine the local strength of the wind. The second call determines the amplitude of the wave at the particular point, independent of the amount of wind there. The product of these two values gives the actual wave offset for the particular location.

⟨*WindyTexture Public Methods*⟩ +≡ 567
```
T Evaluate(const DifferentialGeometry &dg) const {
    Vector dpdx, dpdy;
    Point P = mapping->Map(dg, &dpdx, &dpdy);
    float windStrength = FBm(.1f * P, .1f * dpdx, .1f * dpdy, .5f, 3);
    float waveHeight = FBm(P, dpdx, dpdy, .5f, 6);
    return fabsf(windStrength) * waveHeight;
}
```

DifferentialGeometry 102
FBm() 561
MarbleTexture 568
Point 63
Texture 519
TextureMapping3D 519
TextureMapping3D::Map() 519
Vector 57
WindyTexture 567
WindyTexture::mapping 567

10.6.6 MARBLE

Another classic use of the noise function is to perturb texture coordinates before using another texture or lookup table. For example, a facsimile of marble can be made by modeling the marble material as a series of layered strata and then using noise to perturb the coordinate used for finding a value among the strata. The `MarbleTexture` in this section implements this approach. Figure 10.32 illustrates the idea behind this texture. On the left, the layers of marble are indexed directly using the y coordinate of the point on the sphere. On the right, fBm has been used to perturb the y value, introducing variation. This texture is implemented in `textures/marble.h` and `textures/marble.cpp`.

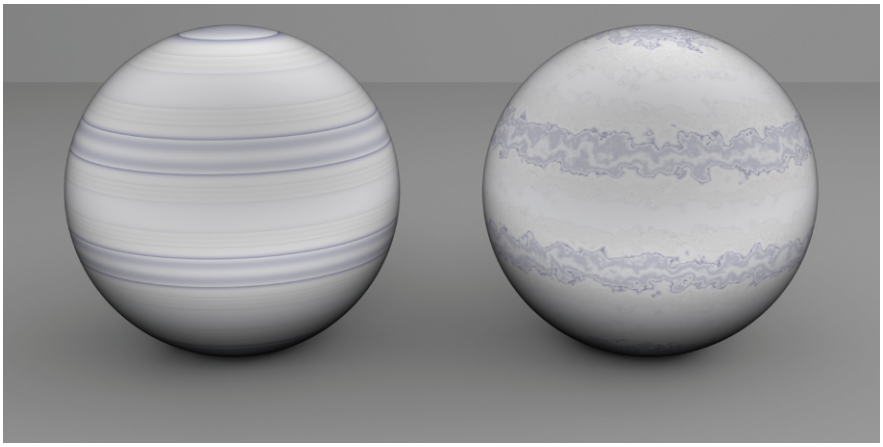


Figure 10.32: Marble. The `MarbleTexture` perturbs the coordinate used to index into a one-dimensional table of colors using FBm, giving a plausible marble appearance.

⟨*MarbleTexture Declarations*⟩ ≡
```
class MarbleTexture : public Texture<Spectrum> {
public:
    ⟨MarbleTexture Public Methods 568⟩
private:
    ⟨MarbleTexture Private Data 568⟩
};
```

The texture takes the usual set of parameters to control the `FBm()` function that will be used to perturb the lookup coordinate. The `variation` parameter modulates the magnitude of the perturbation.

⟨*MarbleTexture Public Methods*⟩ ≡ 568
```
MarbleTexture(int oct, float roughness, float sc, float var,
              TextureMapping3D *map)
    : octaves(oct), omega(roughness), scale(sc), variation(var),
      mapping(map) { }
```

⟨*MarbleTexture Private Data*⟩ ≡ 568
```
int octaves;
float omega, scale, variation;
TextureMapping3D *mapping;
```

An offset into the marble layers is computed by adding the variation to the point's y component and using the sine function to remap its value into the range $[0, 1]$. The ⟨*Evaluate marble spline at* `t`⟩ fragment uses the t value as the evaluation point for a cubic spline through a series of colors that are similar to those of real marble.

FBm() 561
MarbleTexture 568
MarbleTexture::mapping 568
MarbleTexture::octaves 568
MarbleTexture::omega 568
MarbleTexture::scale 568
MarbleTexture::variation 568
Spectrum 263
Texture 519
TextureMapping3D 519

⟨*MarbleTexture Public Methods*⟩ +≡ 568

```
Spectrum Evaluate(const DifferentialGeometry &dg) const {
    Vector dpdx, dpdy;
    Point P = mapping->Map(dg, &dpdx, &dpdy);
    P *= scale;
    float marble = P.y + variation *
                   FBm(P, scale * dpdx, scale * dpdy, omega, octaves);
    float t = .5f + .5f * sinf(marble);
    ⟨Evaluate marble spline at t⟩
}
```

FURTHER READING

The cone-tracing method of Amanatides (1984) was one of the first techniques for automatically estimating filter footprints for ray tracing. The beam-tracing algorithm of Heckbert and Hanrahan (1984) was another early extension of ray tracing to incorporate an area associated with each image sample, rather than just an infinitesimal ray. The pencil-tracing method of Shinya, Takahashi, and Naito (1987) is another approach to this problem. Other related work on the topic of associating areas or footprints with rays includes Mitchell and Hanrahan's paper on rendering caustics (Mitchell and Hanrahan 1992) and Turkowski's technical report (Turkowski 1993).

Collins (1994) estimated the ray footprint by keeping a tree of all rays traced from a given camera ray, examining corresponding rays at the same level and position. Also, Worley's chapter in *Texturing and Modeling* (Ebert et al. 2003) on computing differentials for filter regions presents an approach similar to ours. The ray differentials used in `pbrt` are based on Igehy's (1999) formulation, which was extended by Suykens and Willems (2001) to handle glossy reflection in addition to perfect specular reflection.

Two-dimensional texture mapping with images was first introduced to graphics by Blinn and Newell (1976). Ever since Crow (1977) identified aliasing as the source of many errors in images in graphics, quite a bit of work has been done to find efficient and effective ways of antialiasing image maps. Dungan, Stenger, and Sutty (1978) were the first to suggest creating a pyramid of prefiltered texture images; they used the nearest texture sample at the appropriate level when looking up texture values, using supersampling in screen space to antialias the result. Feibush, Levoy, and Cook (1980) investigated a spatially varying filter function, rather than a simple box filter. (Blinn and Newell were aware of Crow's results and used a box filter for their textures.)

Williams (1983) used a MIP map image pyramid for texture filtering with trilinear interpolation. Shortly thereafter, Crow (1984) introduced summed area tables, which make it possible to efficiently filter over axis-aligned rectangular regions of texture space. Summed area tables handle anisotropy better than Williams's method, although only for primarily axis-aligned filter regions. Heckbert (1986) wrote a good general survey of texture mapping algorithms through the mid-1980s.

Greene and Heckbert (1986) originally developed the elliptically weighted average technique, and Heckbert's master's thesis put the method on a solid theoretical footing

DifferentialGeometry 102
FBm() 561
MarbleTexture::mapping 568
Point 63
Spectrum 263
TextureMapping3D::Map() 519
Vector 57

(Heckbert 1989). Fournier and Fiume (1988) developed an even higher-quality texture filtering method that focuses on using a bounded amount of computation per lookup. Nonetheless, their method appears to be less efficient than EWA overall. Lansdale's master's thesis has an extensive description of EWA and Fournier and Fiume's method, including implementation details (Lansdale 1991).

More recently, a number of researchers have investigated generalizing Williams's original method using a series of trilinear MIP map samples in an effort to increase quality without having to pay the price for the general EWA algorithm. By taking multiple samples from the MIP map, anisotropy is handled well while preserving the computational efficiency. Examples include Barkans's (1997) description of texture filtering in the Talisman architecture, McCormack et al.'s (1999) Feline method, and Cant and Shrubsole's (2000) technique. Chen et al. (2004) summarized the state of the art in texture filtering and introduced a method that addresses some of the shortcomings of existing techniques.

Gamma correction has a long history in computer graphics. Poynton (2002a, 2002b) has written comprehensive FAQs on issues related to color representation and gamma correction. Even though there is no physical reason for LCD displays to have a nonlinear response, Fairchild and Wyble (1998) measured the response of a widely used LCD and found that gamma correction is also necessary with that display. See also Gibson and Fairchild (2000), where similar results are found for other LCDs. See Gritz and d'Eon (2008) for a detailed discussion of the implications of gamma correction for rendering and how to correctly account for it in rendering systems.

Smith's Web site and document on audio resampling gives a good overview of resampling signals in one dimension (Smith 2002). Heckbert's `zoom` source code is the canonical reference for image resampling (Heckbert 1989). His implementation carefully avoids feedback without using auxiliary storage, unlike ours in this chapter, which allocates additional temporary buffer space to do so.

Three-dimensional solid texturing was originally developed by Gardner (1984, 1985), Perlin (1985a), and Peachey (1985). Norton, Rockwood, and Skolmoski (1982) developed the *clamping* method that is widely used for antialiasing textures based on solid texturing. The general idea of procedural texturing was introduced by Cook (1984), Perlin (1985a), and Peachey (1985).

Peachey's chapter in *Texturing and Modeling* (Ebert et al. 2003) has a thorough summary of approaches to noise functions. After Perlin's original noise function, both Lewis (1989) and van Wijk (1991) developed alternatives that made different time/quality trade-offs. Worley (1996) has developed a quite different noise function for procedural texturing that is well suited for cellular and organic patterns. Perlin (2002) revised his noise function to correct a number of subtle shortcomings.

Noise functions have received additional attention from the research community in recent years. Building on Lewis's observation that individual bands of Perlin's noise function actually have frequency content over a fairly wide range (Lewis 1989), Cook and DeRose (2005) also identified the problem that 2D slices through 3D noise functions aren't in general band limited, even if the original 3D noise function is. They proposed a new noise function that addresses both of these issues. Goldberg et al. (2008) developed a noise function that makes efficient anisotropic filtering possible, leading to higher-

quality results than just applying the clamping approach for antialiasing. Their method is also well suited to programmable graphics hardware. Kensler et al. (2008) suggested a number of improvements to Perlin's revised noise function. Finally, Lagae et al. (2009) have developed a noise function that has good frequency control and can be mapped well to surfaces even without a surface parameterization.

The first languages and systems that supported the idea of user-supplied procedural shaders were developed by Cook (1984) and Perlin (1985a). (The texture composition model in this chapter is similar to Cook's shade trees.) The RenderMan shading language, described in a paper by Hanrahan and Lawson (1990), remains the classic shading language in graphics. See Ebert et al. (2003) and Apodaca and Gritz (2000) for techniques for writing procedural shaders; both of those have excellent discussions of issues related to antialiasing in procedural shaders. More recently, Van Horn and Turk (2008) developed an approach to automatically generate MIP maps of reflection functions that represent the characteristics of shaders over finite areas in order to antialias them.

Many creative methods for computing texture on surfaces have been developed. A sampling of our favorites includes reaction diffusion, which simulates growth processes based on a model of chemical interactions over surfaces and was simultaneously introduced by Turk (1991) and Witkin and Kass (1991); Sims's genetic algorithm-based approach, which finds programs that generate interesting textures through random mutations from which a user selects their favorites (Sims 1991); Fleischer et al.'s cellular texturing algorithms that generate geometrically accurate scales and spike features on surfaces (Fleischer et al. 1995); and Dorsey et al.'s flow simulations that model the effect of weathering on buildings and encode the results in image maps that stored the relative wetness, dirtiness, and so on, at points on the surfaces of structures (Dorsey et al. 1996). Porumbescu et al. (2005) developed *shell maps*, which make it possible to map geometric objects onto a surface in the manner of texture mapping.

A variety of *texture synthesis* algorithms have been developed in the last decade; these approaches take an example texture image and then synthesize larger texture maps that appear similar to the original texture while not being exactly the same. The survey article by Wei et al. (2009) describes recent work in this area as well as the main approaches that have been developed so far.

EXERCISES

❷ 10.1 Many image file formats don't store floating-point color values but instead use eight bits for each color component, mapping the values to the range [0, 1]. For images originally stored in this format, the `ImageTexture` uses four times more memory than strictly necessary by using `floats` in `RGBSpectrum` objects to store these colors. Modify the image reading routines to support an image file format with eight-bit components and return an indication of when an image is read from such a file. Then, modify the `ImageTexture` so that it keeps the data for such textures in an eight-bit representation and modify the `MIPMap` so that it can filter data stored in this format. How much memory is saved for image texture-heavy scenes? How is `pbrt`'s performance affected? Can you explain the causes of any performance differences?

`ImageTexture` 524
`MIPMap` 530
`RGBSpectrum` 279

10.2 For scenes with many image textures where reading them all into memory simultaneously has a prohibitive memory cost, an effective approach can be to allocate a fixed amount of memory for image maps (a *texture cache*), load textures into that memory on demand, and discard the image maps that haven't been accessed recently when the memory fills up (Peachey 1990). To enable good performance with small texture caches, image maps should be stored in a *tiled* format that makes it possible to load in small square regions of the texture independently of each other. Tiling techniques like these are used in graphics hardware to improve the performance of their texture memory caches (Hakura and Gupta 1997; Igehy et al. 1998; Igehy et al. 1999). Implement a texture cache in `pbrt`. Write a conversion program that converts images in other formats to a tiled format. (You may want to investigate OpenEXR's tiled image support.) How small can you make the texture cache and still see good performance?

10.3 The Feline texture filtering technique is a middle ground between trilinear interpolation and EWA filtering. It gives results nearly as good as EWA by doing trilinear filtering at a series of positions along the longer filtering axis in texture space. Read the paper that describes Feline (McCormack et al. 1999) and implement this method in `pbrt`. How do its performance and quality compare to EWA?

10.4 Improve the filtering algorithm used for resampling image maps to initialize the MIP map levels using the Lanczos filter instead of the box filter. How do the spheres test images in the file `scenes/sphere-ewa-vs-trilerp.pbrt` and Figure 10.10 change after your improvements?

10.5 It is possible to use MIP-mapping with textures that have non-power-of-two resolutions—the details are explained by Guthe and Heckbert (2005). Implementing this approach can save a substantial amount of memory: in the worst case, the resampling that `pbrt`'s `MIPMap` implementation performs can increase memory requirements by a factor of four. (Consider a 513×513 texture that is resampled to be 1024×1024.) Implement this approach in `pbrt` and compare the amount of memory used to store texture data for a variety of texture-heavy scenes.

10.6 Some of the light transport algorithms in Chapter 15 require a large number of samples to be taken per pixel for good results. (Examples of such algorithms include path tracing as implemented by the `PathIntegrator`.) If hundreds or thousands of samples are taken in each pixel, then the computational expense of high-quality texture filtering isn't worthwhile; the high pixel sampling rate serves well to antialias texture functions with high frequencies. Modify the `MIPMap` implementation so that it optionally just returns a bilinearly interpolated value from the finest level of the pyramid, even if a filter footprint is provided. Compare rendering time and image quality with this approach when rendering an image using many samples per pixel and a scene that has image maps that would otherwise exhibit aliasing at lower pixel sampling rates.

`MIPMap` 530

`PathIntegrator` 766

10.7 An additional advantage of properly antialiased image map lookups is that they improve cache performance. Consider, for example, the situation of undersampling a high-resolution image map: nearby samples on the screen will access

widely separated parts of the image map, such that there is low probability that texels fetched from main memory for one texture lookup will already be in the cache for texture lookups at adjacent pixel samples. Modify `pbrt` so that it always does image texture lookups from the finest level of the `MIPMap`, being careful to ensure that the same number of texels are still being accessed. How does performance change? What do cache-profiling tools report about the overall change in effectiveness of the CPU cache?

● 10.8 Read Worley's paper that describes a new noise function with substantially different visual characteristics than Perlin noise (Worley 1996). Implement this cellular noise function and add `Textures` to `pbrt` that are based on it.

● 10.9 Implement one of the improved noise functions, such as the ones introduced by Cook and DeRose (2005), Goldberg et al. (2008), or Lagae et al. (2009). Compare image quality and rendering time for scenes that make substantial use of noise functions to the current implementation in `pbrt`.

● 10.10 The implementation of the `DotsTexture` texture in this chapter does not make any effort to avoid aliasing in the results that it computes. Modify this texture to do some form of antialiasing. The `Checkerboard2DTexture` offers a guide as to how this might be done, although this case is more complicated, both because the polka dots are not present in every grid cell and because they are irregularly positioned.

At the two extremes of a filter region that is within a single cell and a filter region that spans a large number of cells, the task is easier. If the filter is entirely within a single cell and is entirely inside or outside the polka dot in that cell (if present), then it is only necessary to evaluate one of the two subtextures as appropriate. If the filter is within a single cell but overlaps both the dot and the base texture, then it is possible to compute how much of the filter area is inside the dot and how much is outside and blend between the two. At the other extreme, if the filter area is extremely large, it is possible to blend between the two textures according to the overall average of how much area is covered by dots and how much is not. (Note that this approach potentially makes the same error as was made in the checkerboard, where the subtextures aren't aware that part of their area is occluded by another texture. Ignore this issue for this exercise.)

Implement these approaches and then consider the intermediate cases, where the filter region spans a small number of cells. What approaches work well for antialiasing in this case?

● 10.11 Write a general-purpose `Texture` that stores a reference to another texture and supersamples that texture when the evaluation method is called, thus making it possible to apply supersampling to any `Texture`. Use your implementation to compare the effectiveness and quality of the built-in antialiasing done by various procedural textures. Also compare the run time efficiency of texture supersampling versus increased pixel sampling.

Checkerboard2DTexture 547
DotsTexture 559
Texture 519

■ 10.12 Modify `pbrt` to support a shading language to allow user-written programs to compute texture values.