Chapter 5
Texturing

"All it takes is for the rendered image to look right."
–Jim Blinn

A surface's texture is its look and feel—just think of the texture of an oil painting. In computer graphics, texturing is a process that takes a surface and modifies its appearance at each location using some image, function, or other dataset. As an example, instead of precisely representing the geometry of a brick wall, a color image of a brick wall is applied to a single polygon. When the polygon is viewed, the color image appears where the polygon is located. Unless the viewer gets close to the wall, the lack of geometric detail (e.g., the fact that the image of bricks and mortar forms a smooth surface) will not be noticeable. Huge modeling, memory, and speed savings are obtained by combining images and surfaces in this way. Color image texturing also provides a way to use photographic images and animations on surfaces.

However, some textured brick walls can be unconvincing for reasons other than lack of geometry. For example, if the bricks are supposed to be shiny, whereas the mortar of course is not, the viewer will notice that the shininess is the same for both materials. To produce a more convincing experience, a specular highlighting image texture can also be applied to the surface. Instead of changing the surface's color, this sort of texture changes the wall's shininess depending on location on the surface. Now the bricks have a color from the color image texture and a shininess from this new texture.

Once the shiny texture has been applied, however, the viewer may notice that now all the bricks are shiny and the mortar is not, but each brick face appears to be flat. This does not look right, as bricks normally have some irregularity to their surfaces. By applying bump mapping, the surface normals of the bricks may be varied so that when they are rendered they do not appear to be perfectly flat. This sort of texture wobbles the direction of the polygon's original surface normal for purposes of computing lighting.

These are just three examples of the types of problems that can be solved with textures. In this chapter texturing techniques are covered in detail. First a general framework of the texturing process is presented. Next, we focus on using images to texture surfaces, since this is the most popular form of texturing used in real-time work. The various techniques for improving the appearance of image textures are detailed, and then methods of getting textures to affect the surface are explained.

5.1 Generalized Texturing

Texturing, at its simplest, is a technique for efficiently modeling the surface's properties. One way to approach texturing is to think about what happens for a single sample taken at a vertex of a polygon. As seen in the previous chapter, the color is computed by taking into account the lighting and the material, as well as the viewer's position. If present, transparency also affects the sample, and then the effect of fog is calculated. Texturing works by modifying the values used in the lighting equation. The way these values are changed is normally based on the position on the surface. So, for the brick wall example, the color at any point on the surface was replaced by a corresponding color in the image of a brick wall, based on the surface location. The specular highlight texture modified the shininess value, and the bump texture changed the direction of the normal, so each of these changed the result of the lighting equation.

Texturing can be described by a generalized texture pipeline. Much terminology will be introduced in a moment, but take heart: each piece of the pipeline will be described in detail. This full texturing process is not performed by most current real-time rendering systems, though as time goes by more parts of the pipeline will be incorporated. Once we have presented the entire process we will examine, the various simplifications and limitations of real-time texturing.

A location in space is the starting point for the texturing process. This location can be in world space, but is more often in the model's frame of reference, so that as the model moves, the texture moves along with it. Using Kershaw's terminology [203], this point in space then has a *projector* function applied to it to obtain a set of numbers, called parameter-space values, that will be used for accessing the texture. This process is called *mapping*, which leads to the phrase *texture mapping*.[1] Before these new values may be used to access the texture, one or more *corresponder* functions can be used to transform the parameter-space values to texture space. These texture-space values are used to obtain

[1] Sometimes the texture image itself is called the texture map, though this is not strictly correct.

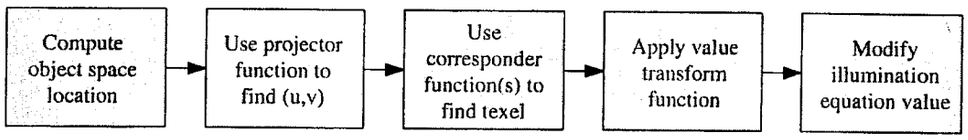

Figure 5.1. Generalized texture pipeline.

values from the texture, e.g., they may be array indices into an image texture to retrieve a pixel. The retrieved values are then potentially transformed yet again by a *value transform* function, and finally these new values are used to modify some property of the surface, such as the material or shading normal. Figure 5.1 shows this process in detail for the application of a single texture. The reason for the complexity of the pipeline is that each step provides the user with a useful control.

Using this pipeline, this is what happens when a polygon has a brick wall texture and a sample is generated on its surface (see Figure 5.2). The (x, y, z) position in the object's local frame of reference is found; say it is $(-2.3, 7.1, 88.2)$. A projector function is then applied to this position. Just as a map of the world is a projection of a three-dimensional object into two dimensions, the projector function here typically changes the (x, y, z) vector into a two-element vector (u, v). The projector function used for this example is an orthographic projection (see Section 2.3.3), acting essentially like a slide projector shining the brick wall image onto the polygon's surface. To return to the wall, a point on its plane could be transformed into a pair of values ranging from 0 to 1. Say the values obtained are $(0.32, 0.29)$. These parameter-space values are to be used to find what the color of the image is at this location. The resolution of our brick texture is, say, 256×256, so the corresponder function multiplies the (u, v) by 256 each, giving $(81.92, 74.24)$. Dropping the fractions, pixel $(81, 74)$ is found in the brick wall image, and is of color $(0.9, 0.8, 0.7)$. The original brick wall image is too dark, so a value transform function that multiplies the color by 1.1 is then applied, giving a color of $(0.99, 0.88, 0.77)$. This color modifies the surface properties by directly replacing the surface's original diffuse color, which is then used in the illumination equation.

The first step in the texture process is obtaining the surface's location and projecting it into parameter space. Projector functions include spherical, cylindrical, box, and planar projections [33, 203]. Other projector functions are not projections at all, but are an implicit part of surface formation; for example, spline surfaces have a natural set of (u, v) values as part of their definition. Non-interactive renderers often call the projector functions as part of the rendering process itself. In real-time work projector functions are usually applied at the modeling stage, and the results of the projection are stored at the vertices. This

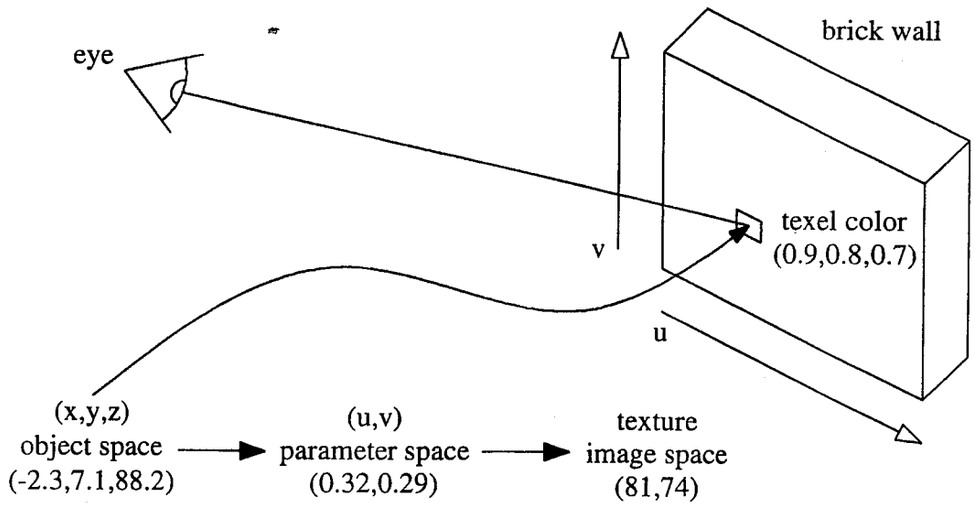

Figure 5.2. Pipeline for a brick wall.

is not always the case; for example, OpenGL's `glTexGen` routine provides a few different projector functions. Some methods, such as environment mapping (see Section 5.7.3), have specialized projector functions that are evaluated on the fly.

The typical method for using projector functions in real-time systems is by applying (u, v) texture coordinates at each vertex during modeling. These values are sometimes presented as a three-element vector, (u, v, w). Other systems use up to four coordinates, designated (s, t, r, q) [275]; q is used as the fourth value in a homogeneous coordinate (see Section A.4) and can be used for spot lighting effects [321]. To allow each separate texture map to have its own input parameters during a rendering pass, APIs allow multiple pairs of (u, v) values (see Section 5.5 on multitexturing). However the designation is accomplished, the idea is the same: these parameter values are interpolated across the surface and used to retrieve texture values. Before being interpolated, however, these parameter values are transformed by corresponder functions.

Corresponder functions convert parameter-space values to texture-space values. They provide flexibility in applying textures to surfaces. One corresponder is an optional matrix transformation. The use of a matrix is supported explicitly in OpenGL, and is simple enough to support at the application stage under any API. This transform is useful for the sorts of procedures which transforms normally do well at: it can translate, rotate, scale, and so on, the texture on the surface.[2]

[2]As discussed in Section 3.1.5, the order of transforms matters. Surprisingly, the order of transforms for textures must be the reverse of the order one would expect. This is because texture

Another class of corresponder functions controls the way an image is applied. We know that an image will appear on the surface where (u, v) are in the $[0, 1)$ range.[3] But what happens outside of this range? Corresponder functions determine the behavior.[4] Common corresponder functions are:

- **wrap**, **repeat**, or **tile** - The image repeats itself across the surface; algorithmically, the integer part of the parameter value is dropped. This function is useful for having an image of a material repeatedly cover a surface, and is often the default.

- **mirror** - The image repeats itself across the surface, but is mirrored (flipped) on every other repetition. For example, the image appears normally going from 0 to 1, then is reversed between 1 and 2, then is normal between 2 and 3, then is reversed, etc. This provides some continuity along the edges of the texture.

- **clamp** - Values outside the range $[0, 1)$ are clamped to this range. This results in the repetition of the edges of the image texture. Some APIs [122, 275] allow one parameter to be clamped while the other repeats.

- **border** - Parameter values outside $[0, 1)$ are rendered with a separately defined border color. This function is good for rendering decals onto surfaces, for example. Note that the **clamp** method can produce the same sort of effect if the border of the image texture itself is already the color you wish the rendered border to be. The border mode is useful in that it requires no such preprocessing of the image, and the border color can be modified independently.

See Figure 5.3.

For real-time work, the last corresponder is implicit, and is derived from the image's size. A texture is normally applied within the range $[0, 1)$ for u and v. As shown in the brick wall example, by multiplying parameter values in this range by the resolution of the image, one may obtain the pixel location. The pixels in the texture are often called *texels*, to differentiate them from the pixels on the screen. The advantage of being able to specify (u, v) values in a range of

transforms actually affect the space that determines where the image is seen. The image itself is not an object being transformed; the space defining the image's location is being changed.

[3]The notation "$[0, 1)$" means from 0 to 1, including 0 but not 1. A bracket '[' means include the number, parenthesis ')' means do not include the number in the range.

[4]In Direct3D the corresponder function is called the "texture addressing mode." Confusingly, Direct3D also has a feature called "texture wrapping," which is used with Blinn environment mapping. See Section 5.7.3

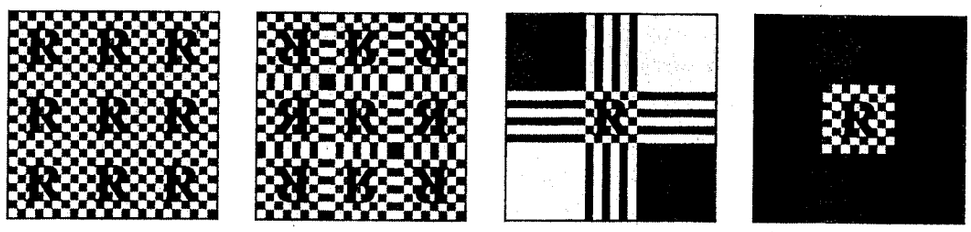

Figure 5.3. Image texture wrap, mirror, clamp, and border functions in action.

$[0, 1)$ is that image textures with different resolutions can be swapped in without having to change the values stored at the vertices of the model.

In theory, more than one corresponder function could be applied in a series (e.g., a corresponder could select a portion of the texture image, and then a border corresponder could be applied to it [203]).

The set of corresponder functions uses parameter-space values to produce texture coordinates. For image textures, the texture coordinates are used to retrieve texel information from the image. This process is dealt with extensively in Section 5.2. Two-dimensional images constitute the vast majority of texture use in real-time work, but there are other texture functions. A direct extension of image textures is three-dimensional image data which is accessed by (u, v, w) (or (s, t, r)) values. For example, medical imaging data can be generated as a three-dimensional grid; by moving a polygon through this grid, one may view two-dimensional slices of this data.

Covering an arbitrary three-dimensional surface cleanly with a two-dimensional image is often difficult or impossible [33]. As the texture is applied to some solid object, the image is stretched or compressed in places to fit the surface. Obvious mismatches may be visible as different pieces of the texture meet. A solid cone is a good example of both of these problems: the image bunches up at the tip, while the texture on the flat face of the cone does not match with the texture on the sides.

The advantage of three-dimensional textures is that they avoid the distortion and seam problems that two-dimensional texture mappings can have. A three-dimensional texture can act as a material such as wood or marble, and the model may be textured as if it were carved from this material. The texture can also be used to modify other properties, for example changing (u, v) coordinates in order to creating warping effects [245].

Three-dimensional textures can be synthesized by a variety of techniques. One of the most common is using one or more noise functions to generate values [92]. Because of the cost of evaluating the noise function, sometimes the lattice points in the three-dimensional array are precomputed and used to

interpolate texture values. There are also methods of using the accumulation buffer or color buffer blending to generate these arrays [245]. However, such arrays can be large to store and often lack sufficient detail. An alternative is to evaluate the noise function on the fly. This is normally a computationally expensive endeavor, but Barad et al. [21, 22] show how the MMX architecture can be used to speed evaluation and make it practicable.

Two-dimensional texture functions can also be used to generate textures, but here the major advantages are some storage savings (and bandwidth savings from not having to send down the corresponding image texture) and the fact that such textures have essentially infinite resolution and potentially no repeatability.

It is also worth noting that one-dimensional texture images and functions have their uses. For example, these include contour lines [149, 275] and coloration determined by altitude (e.g., the lowlands are green, the mountain peaks are white).

The texture is accessed and a set of values is retrieved from it. The most straightforward data to return is an RGB triplet which is used to replace or modify the surface color; similarly, a single gray-scale value could be returned. Another type of data to return is RGBα, as described in Section 4.5. The α (alpha) value is normally the opacity of the color, which determines the extent to which the color may affect the pixel. There are certainly other types of data that can be stored in image textures, as will be seen when bump-mapping is discussed in detail (Section 5.7.4).

Once the texture values have been retrieved, they may be used directly or further transformed, and then used to modify one or more surface attributes. The modified surface attribute may then also be modified by some other operation. Recall, however, that almost all real-time systems use Gouraud shading, meaning that only certain values are interpolated across a surface. We cannot modify much beyond the RGB result of the lighting equation, since this equation was already evaluated once per vertex and we interpolate the results. Most real-time systems let us pick one of a number of methods for modifying the surface. The methods, called *combine functions* or *texture blending operations*, for gluing an image texture onto a surface include:

- **replace** - Simply replace the original surface color with the texture color. Note that this removes any lighting computed for the surface, unless the texture itself includes it.

- **modulate** - Multiply the surface color by the texture color. If the surface material is originally white, then the lighting applied to it will modify the color texture, giving a shaded, textured surface.

These two are the most common methods for simple color texture mapping. Using **replace** for texturing in an illuminated environment is sometimes called using a *glow texture*, since the texture's color always appears the same, regardless of changing light conditions. There are other property modifiers, which will be discussed as other texture techniques are introduced.

Revisiting the brick wall texture example, here is what happens in a typical real-time system. A modeler sets the (u, v) parameter values once in advance for the wall model vertices. The texture is read into the renderer, and the wall polygons are sent down the rendering pipeline. A white material is used in computing the illumination at each vertex. This color and (u, v) values are interpolated across the surface. At each pixel the proper brick image's texel is retrieved and modulated (multiplied) by the illumination color and displayed. In our original example this texture was multiplied by 1.1 at this point to make it brighter; in practice this color boost would probably be performed on the texture itself in the modeling stage. In the end, a lit, textured brick wall is displayed.

5.2 Image Texturing

In image texturing a two-dimensional image is effectively glued onto the surface of a polygon and rendered. We have walked through the process with respect to the polygon; now we will address the issues surrounding the image itself and its application to the surface. For the rest of this chapter the image texture will be referred to simply as the texture. In addition, when we refer to a pixel's cell here, we mean the screen grid cell surrounding that pixel. As mentioned in Section 4.4, a pixel is actually a displayed color value which can (and should, for better quality) be affected by samples outside of its grid cell.

The texture image size used in hardware accelerators is restricted to $2^m \times 2^n$ texels, or sometimes even $2^m \times 2^m$ square, where m and n are non-negative integers. Some graphics accelerators have an upper limit on texture size. Also, OpenGL has a lower limit of 64×64 for textures [275].

Assume that we have an image of size 256×256 pixels and that we want to use it as a texture on a square. As long as the projected square on the screen is roughly the same size as the texture, the texture on the square looks almost like the original image. But what happens if the projected square covers 10 times as many pixels as the original image contains (called *magnification*), or if the projected square covers only a fraction of the pixels (*minification*)? The

answer is that it depends on what kind of filtering methods you use for these two separate cases.

5.2.1 Magnification

In Figure 5.4, a texture of size 32×64 texels is textured onto a rectangle, and the rectangle is viewed rather closely with respect to the texture size, so the underlying graphics system had to magnify the texture. The most common filtering techniques for magnification are *nearest neighbor* and *bilinear interpolation.*[5]

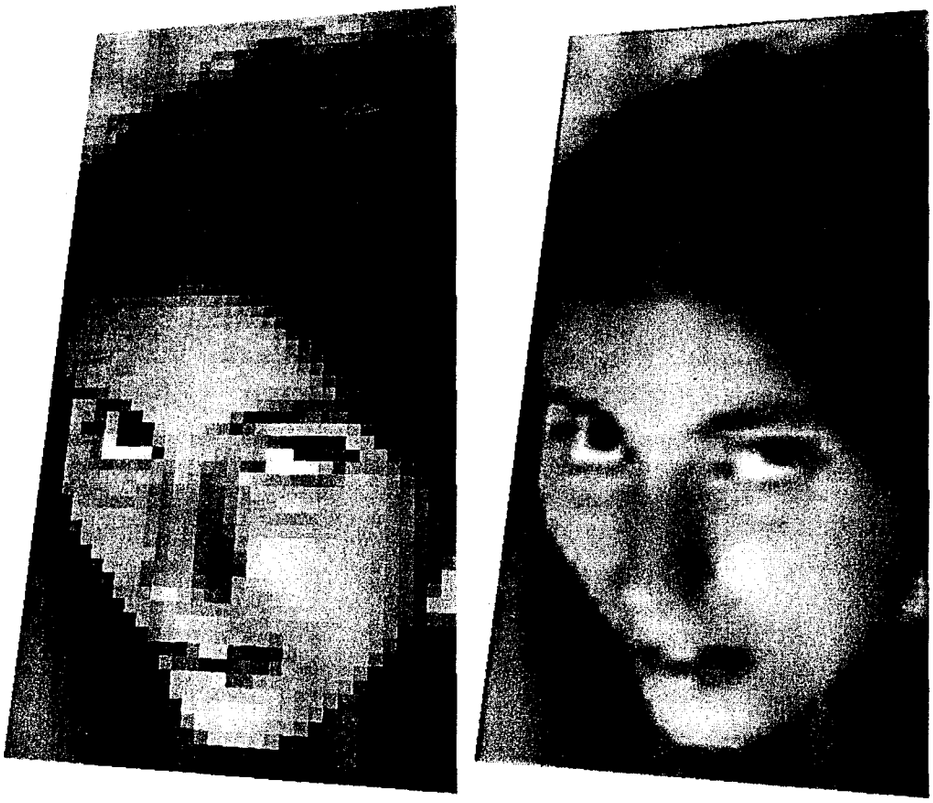

Figure 5.4. Texture magnification. Here, a texture of size 32×64 was applied to a rectangle, which was viewed very closely (with respect to texture size). Therefore, the texture had to be magnified. On the left, the nearest neighbor filter is used, which simply selects the nearest texel to each pixel. Bilinear interpolation is used on the rectangle on the right. Here, each pixel is computed from a bilinear interpolation of the closest four neighbor texels.

[5]There is also *cubic convolution*, which uses the weighted sum of a 4×4 array of texels, but it is currently not commonly available.

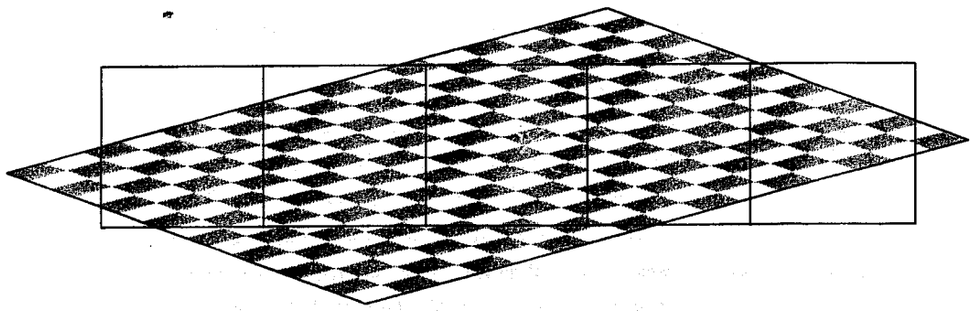

Figure 5.5. A view of a checkerboard-textured polygon through a row of pixel cells, showing how a number of texels affect each pixel.

In the left part of Figure 5.4, the nearest neighbor method was used, and the characteristic of this magnification technique is that the individual texels may become apparent. This effect is called *pixelation*. This is because this method takes the value of the nearest texel to each pixel center when magnifying, resulting in a blocky appearance. While the quality of this method is sometimes poor, it requires only one texel to be fetched per pixel.

In the right part of the same figure, bilinear interpolation (sometimes called *linear interpolation*) is used. For each pixel, this kind of filtering finds the four neighboring texels and linearly interpolates in two dimensions to find a blended value for the pixel. The result is blurrier, and much of the jaggedness from using the nearest neighbor method has disappeared.[6] Which filter is best typically depends on the desired result. The nearest neighbor can give a crisper feel when little magnification is occurring, but bilinear interpolation is usually a safer (though sometimes slower) choice in most situations.

5.2.2 Minification

When a texture is minimized, several texels may cover a pixel's cell, as shown in Figure 5.5. To get a correct color value for each pixel, you should integrate the effect of the texels influencing the pixel. However, it is difficult to determine precisely the exact influence of all texels near a particular pixel, and it is effectively impossible to do so perfectly in real-time.

Because of this limitation, a number of different methods are used in real-time work. One method is to use the nearest neighbor, which works exactly

[6]Looking at these images with eyes squinted has approximately the same effect as a low-pass filter, and reveals the face a bit more.

as the corresponding magnification filter does, i.e., it selects the texel closest to the center of the pixel's cell. This filter may cause severe aliasing problems. In Figure 5.6, nearest neighbor is used in the top figure. Towards the horizon, artifacts appear because only one of the many texels influencing a pixel is chosen to represent the surface. Such artifacts are even more noticeable as the surface moves with respect to the viewer, and are one manifestation of what is called *temporal aliasing*.

Another filter often available is bilinear interpolation, again working exactly as in the magnification filter. This filter is only slightly better than the nearest neighbor approach for minification. It blends four texels instead of using just one, but when a pixel is influenced by more than four texels, the filter soon fails and produces aliasing.

There are a few algorithms that are used to solve the problem for real-time rendering. Before delving into these, some filtering and sampling theory (beyond that presented in Section 4.4) is useful.

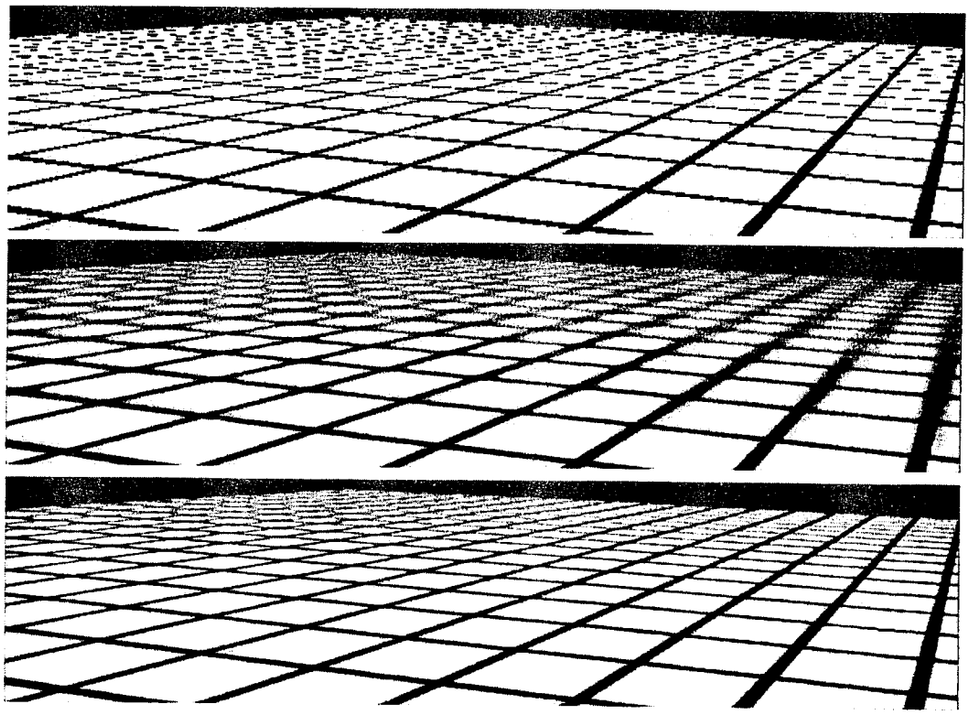

Figure 5.6. The top image was rendered with point sampling (nearest neighbor), the center with mipmapping, and the bottom with summed area tables.

Sampling and Filtering

The generation of an image is the process of sampling a three-dimensional scene in order to obtain color values for each pixel in the image. The rendering of images is inherently a sampling task, since the output device consists of an array of discrete pixels. Whenever sampling is done, aliasing may occur. This is an unwanted artifact, and we need to battle aliasing in order to generate pleasing images. Aliasing occurs when a signal is being sampled at too low a frequency. The sampled signal then appears to be a signal of lower frequency than the original. This is illustrated in Figure 5.7. For a signal to be sampled properly (i.e., so that it is possible to reconstruct the original signal from the samples), the sampling frequency has to be at least twice the maximum frequency of the signal to be sampled. This sampling frequency is called the *Nyquist rate* [293, 380] or *Nyquist limit* [369]. These concepts extend naturally to two dimensions as well, and so can be used when handling two-dimensional images.

If we knew in advance the maximum frequency in the three-dimensional scene we were about to view, then we would also know what sampling frequency would suffice in order to render an image without aliasing artifacts. However, often the maximum frequency is infinite. This is so because an edge contains infinite frequencies (because at an edge the intensity changes abruptly), and edges appear very often in computer-generated imagery. Even if the frequency is finite, the sampling rate per pixel might still have to be extremely high.

As discussed in Section 4.4, one antialiasing method is to use supersampling to distribute a number of samples evenly over each pixel cell. The color of the pixel is then some weighted mean of the samples. Note, however, that super-

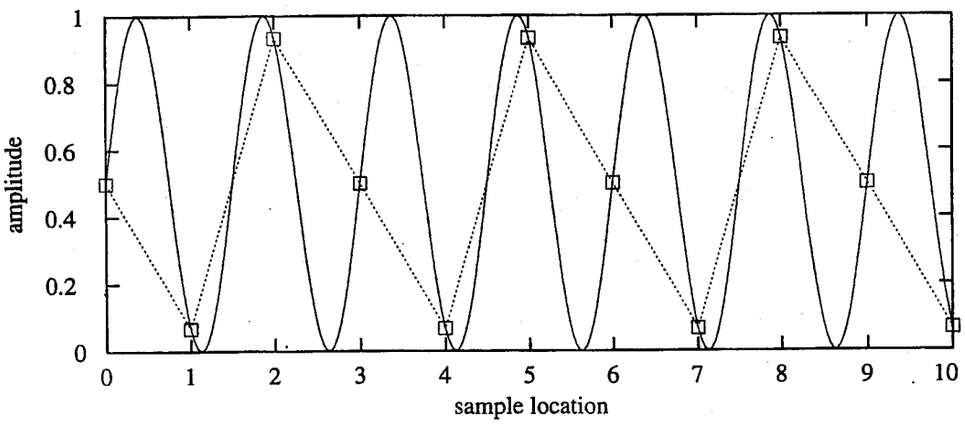

Figure 5.7. The solid line is the original signal. Sampling this signal at too low a rate gives a reconstructed signal (the dotted line) that has a frequency half that of the original.

sampling only increases the sampling rate, and thus can never eliminate— only reduce—aliasing.

Another solution to edge aliasing is filtering the edges, blurring them so that the high frequencies are eliminated or reduced. This same approach is used in solving texture aliasing. The goal is to gather the effects of the texels near the pixel, which is effectively equivalent to blurring the texture to eliminate high frequencies. The signal frequency of a texture depends upon how closely spaced its texels are on the screen. Due to the Nyquist limit, we need to make sure that the texture's signal frequency is no greater than half the sample frequency. In other words, we want to make sure that, along any one dimension, there is at most one texel for every two pixels. This is the theory, at least; in practice, systems tend to default to having one texel per pixel. The basic intent of texture antialiasing algorithms is the same: to preprocess the texture and create data structures that will allow quick approximation of the effect of a set of texels on a pixel.

Mipmapping

The most popular method of antialiasing for textures is called *mipmapping* [375]. It is implemented in some form on even the most modest graphics accelerators now produced. "Mip" stands for *multum in parvo*, Latin for "many things in a small place"—a good name for a process in which the original texture is filtered down repeatedly into smaller images.

When the mipmapping minimization filter is used, the original texture is augmented with a set of smaller versions of the texture before the actual rendering takes place. The texture (level zero) is downsampled to a quarter of the original area, with each new texel value typically computed as the average of the four neighbor texels. The new, level-one texture is called a *subtexture* of the original texture. The reduction is performed recursively until one or both of the dimensions of the texture equals 1 texel. This process is illustrated in Figure 5.8.

The basic process of accessing this structure while texturing is straightforward. A screen pixel encloses an area on the texture itself. When the pixel's area is projected onto the texture (Figure 5.9), it covers a number of texels. There are two common measures used to compute a coordinate called d (which OpenGL calls λ, and which is also known as the level of detail). One is to use the longer edge of the quadrilateral formed by the pixel's cell to approximate the pixel's coverage [375]; the other is to use as a measure the largest absolute value of the four differentials $\partial u/\partial x$, $\partial v/\partial x$, $\partial u/\partial y$, and $\partial v/\partial y$ [207]. Each differential is a measure of the amount of change in the texture coordinate with respect to a screen axis. For example, $\partial u/\partial x$ is the amount of change in the u texture value along the x-screen-axis for one pixel. Because mipmapping is now stan-

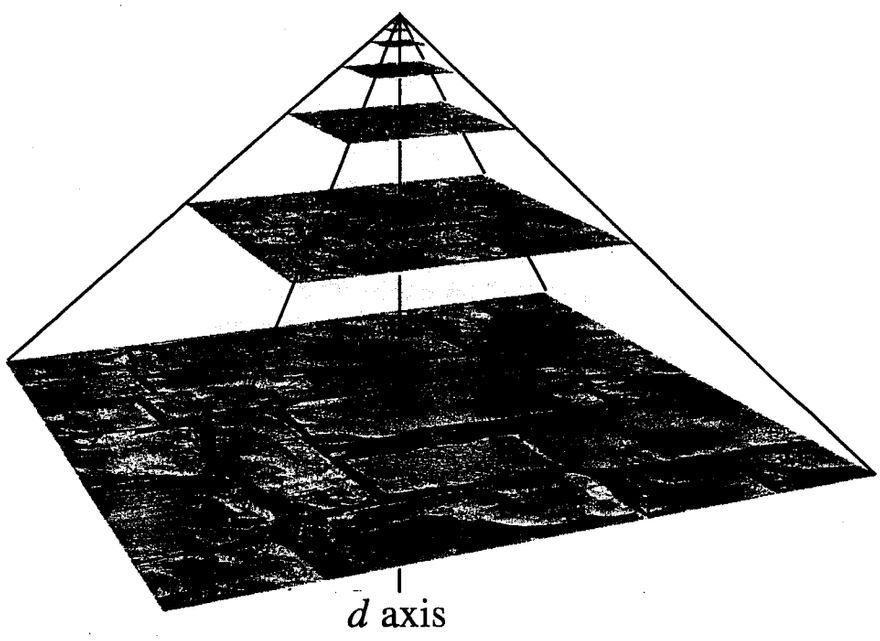

Figure 5.8. A mipmap is formed by taking the original image (level 0), at the base of the pyramid, and averaging each 2×2 area into a texel value on the next level up. The vertical axis is the third texture coordinate, d. In this figure, d is not linear; it is a measure of which two texture levels a sample uses for interpolation.

dard in hardware, the precise computations are not covered in depth here. See Williams's original article [375] or Flavell's presentation [107] for more about these equations. McCormack et al. [244] discuss the introduction of aliasing by the largest absolute value method, and they present an alternate formula.

The intent of computing the coordinate d is to determine where to sample along the mipmap's pyramid axis (see Figure 5.8). The goal is a pixel-to-texel ratio of at least 1:1 and preferably 2:1 [1], in order to achieve the Nyquist rate. The important principle here is that as the pixel cell comes to include more texels, a smaller, blurrier version of the texture is accessed. The (u, v, d) triplet is used to access the mipmap. The value d is analogous to a texture level, but instead of an integer value, d has the fractional value of the distance between levels. The texture level above and the level below the d location are sampled. The (u, v) location is used to retrieve a bilinearly interpolated sample from each of these two texture levels. The resulting sample is then linearly interpolated, depending on the distance from each texture level to d. This entire process

is called trilinear interpolation and is performed per pixel.[7] Some hardware performs weaker versions of this algorithm, e.g., per-polygon, nearest neighbor, bilinear interpolation on the closest texture level, dithered per-pixel between two bilinear samples, or other combinations. Because trilinear interpolation uses two mipmap accesses (versus bilinear, where a single subtexture is accessed), it is twice as expensive to perform on some hardware.

One user control on the d coordinate is the *level of detail bias (LOD bias)* [122]. This is a value added to d, and so it affects the relative perceived sharpness of a texture. If we move further up the pyramid to start (increasing d), the texture will look blurrier. A good LOD bias for any given texture will vary with the image type and with the way it is used. For example, images that are somewhat blurry to begin with could use a negative bias, while poorly filtered (aliased) synthetic images used for texturing could use a positive bias. When a textured surface is moving, it often requires a higher bias to avoid temporal aliasing problems.

The result of mipmapping is that, instead of trying to sum all the texels which affect a pixel individually, precombined sets of texels are accessed and interpolated. This process takes a fixed amount of time, no matter what the level of minification. However, mipmapping has a number of flaws [107]. A major one is *overblurring*. Imagine a pixel cell that covers a large number of texels in the u direction and only a few in the v direction. This case commonly occurs when a viewer looks along a textured surface nearly edge-on. The effect of

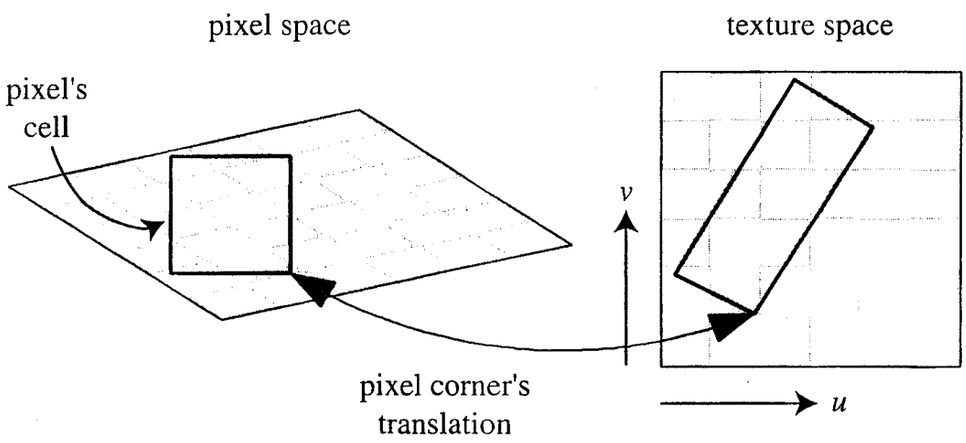

Figure 5.9. On the left is a square pixel cell and its view of a texture. On the right is the projection of the pixel cell onto the texture itself.

[7] Mipmapping and filtering in general can also be applied to three-dimensional image textures, in which case an additional level of interpolation is used.

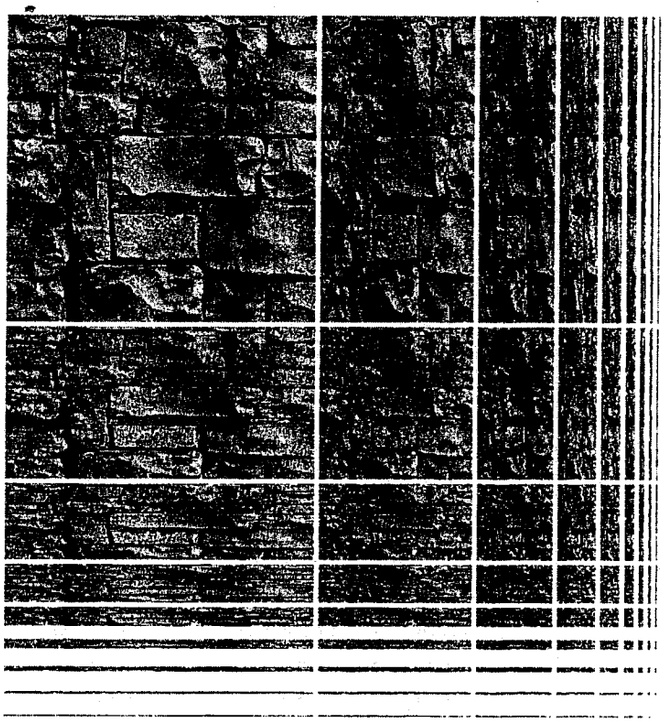

Figure 5.10. The ripmap structure. Note that the images along the diagonal from the upper left to the lower right are the mipmap subtextures.

accessing the mipmap is that square areas on the texture are retrieved; retrieving rectangular areas is not possible. To avoid aliasing, we choose the largest measure of the approximate coverage of the pixel cell on the texture. This results in the retrieved sample often being relatively blurry. This effect can be seen in the mipmap image in Figure 5.6. The lines moving into the distance on the right show overblurring.

There are a number of techniques to avoid some or all of this overblurring. One method is the *ripmap*. The idea is to extend the mipmap to include downsampled rectangular areas as subtextures that can be accessed. Figure 5.10 shows the ripmap subtexture array. Four coordinates are used to access this structure: the usual two (u, v) values to access each subtexture, and two for a location in the ripmap array; these indicate the four subtextures among which to interpolate. These last two coordinates are computed using the pixel cell's u and v extents on the texture: the more texels included along an axis, the more downsampled the map that is used [245].

Another method is the *summed-area table* [73]. To use this method one first creates an array that is the size of the texture but contains more bits of precision

for the color stored (e.g., 16 bits or more for each of red, green, and blue). At each location in this array, one must compute and store the sum of all the corresponding texture's texels in the rectangle formed by this location and texel $(0, 0)$ (the origin). During texturing, the pixel cell's projection onto the texture is bound by a rectangle. The summed-area table is then accessed to determine the average color of this rectangle, which is passed back as the texture's color for the pixel. The average is computed using the texture coordinates of the rectangle shown in Figure 5.11. This is done using the formula given in Equation 5.1.

$$\mathbf{c} = \frac{\mathbf{s}[x_{ur}, y_{ur}] - \mathbf{s}[x_{ur}, y_{ll}] - \mathbf{s}[x_{ll}, y_{ur}] + \mathbf{s}[x_{ll}, y_{ll}])}{(x_{ur} - x_{ll})(y_{ur} - y_{ll})} \tag{5.1}$$

Here, x and y are the texel coordinates of the rectangle and $\mathbf{s}[x, y]$ is the summed-area value for that texel. This equation works by taking the sum of the entire area from the upper right corner to the origin, then subtracting off areas A and B by subtracting the neighboring corners' contributions. Area C has been subtracted twice, so it is added back in by the lower left corner.

The results of using a summed-area table are shown in Figure 5.6. The lines going to the horizon are sharper near the right edge, but the diagonally crossing lines in the middle are still overblurred. Similar problems occur with the ripmap scheme. The problem is that when a texture is viewed along its diagonal, a large rectangle is generated with many of the texels situated nowhere near the pixel being added in. For example, imagine a long, thin rectangle

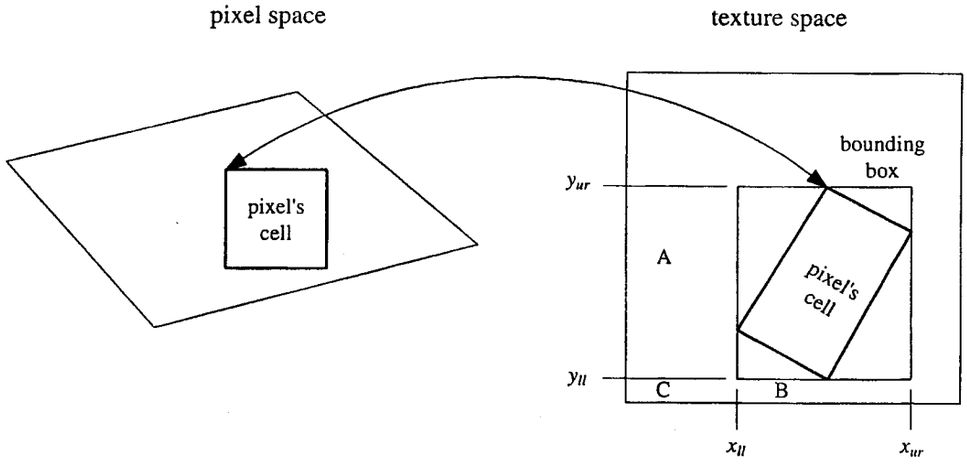

Figure 5.11. The pixel cell is back-projected onto the texture, bound by a rectangle, and the four corners of the rectangle are used to access the summed-area table.

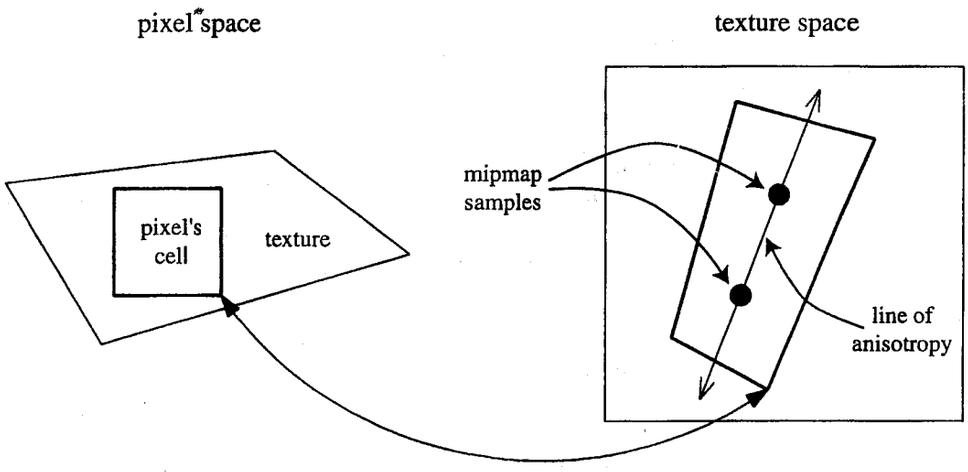

Figure 5.12. Talisman anisotropic filtering. The back-projection of the pixel cell creates a quadrilateral. A line of anisotropy is formed parallel to the longer sides.

representing the pixel cell's back-projection lying diagonally across the entire texture in Figure 5.11. The whole texture rectangle's average will be returned, rather than just the average within the pixel cell.

Ripmaps and summed-area tables are examples of what are called *anisotropic filtering* algorithms [164]. Such algorithms are schemes that can retrieve texel values over areas which are not square. However, they are able to do this most effectively in primarily horizontal and vertical directions. Ripmaps were used in high-end Hewlett-Packard graphics accelerators in the early 1990s. To our knowledge, summed-area tables have not been implemented in any specialized hardware. Both schemes are memory intensive. While a mipmap's subtextures take only an additional third of the memory of the original texture, a ripmap's take an additional three times as much as the original. Summed-area tables take at least two times as much memory for textures of size 256×256 or less, with more precision needed for larger textures. Texture memory is a relatively precious commodity, not to be squandered.

A relatively new solution to the problem uses multiple sampling of the mipmap structure to obtain a better result. This anisotropic filtering algorithm is a part of the Talisman architecture [354], and is explained in depth by Barkans [28]. The basic idea is that the pixel cell is back-projected, and this quadrilateral (quad) on the texture is then sampled a number of times, and the samples are combined. As noted above, each mipmap sample has a location and a squarish area associated with it. Instead of using a single mipmap sample to approximate this quad's coverage, the algorithm uses a number of squares to cover the quad.

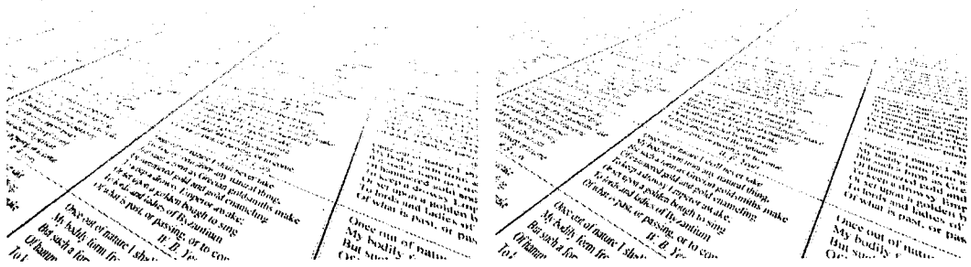

Figure 5.13. Mipmap versus anisotropic filtering of text. Trilinear mipmapping has been done on the left, 2:1 anisotropic filtering on the right, both at 640 × 480 resolution. *(Example pictures generated by software. Courtesy of NVIDIA Inc.)*

The shorter side of the quad is used to determine d (unlike in mipmapping, where the longer side is often used); this makes the averaged area smaller (and so less blurred) for each mipmap sample. The quad's longer side is used to create a *line of anisotropy* parallel to the longer side and through the middle of the quad. When the amount of anisotropy is between 1:1 and 2:1, two samples are taken along this line (see Figure 5.12). At higher ratios of anisotropy more samples are taken along the axis, with the two end-point samples weighted half as much as the rest of the samples.

This scheme allows the line of anisotropy to run in any direction, and so does not have the limitations that ripmaps and summed-area tables had. It also requires no more texture memory than mipmaps do, since it uses the mipmap algorithm to do its sampling. For example, a graphics accelerator can have a dual-pipe architecture that allows it to obtain and combine two mipmap samples in parallel, thus allowing it to perform anisotropic filtering up to a ratio of 2:1. We expect more parallelism to appear in chip sets as time goes on, thereby increasing the anisotropic ratio that can be used. An example of an isotropic filtering is shown in Figure 5.13.

Other minification filters are possible [164], but the mipmapping scheme and its variants are the ones that have made it into graphics hardware.

5.3 Texture Caching and Compression

A complex application may require a considerable number of textures. The amount of fast texture memory varies from system to system, but the general rule is that it is never enough. There are various techniques for *texture caching*,

where a balance is sought between speed and minimizing the number of textures (or parts of textures) in memory at one time. For example, for textured polygons that are initially far away, the application may load only the smaller subtextures in a mipmap, since these are the only levels that will be accessed [1]. Each system has its own texture management tools and its own performance characteristics (e.g., 256×256 textures are optimized on some machines [83]). Some general advice is to keep the textures small—no larger than is necessary to avoid magnification problems—and to try to keep polygons grouped by their use of texture. Even if all textures are always in memory, such precautions may improve the processor's cache performance. Another method, called *tiling* or *mosaicing*, involves combining a few smaller (non-repeating) textures into a single larger texture image in order to speed access [83, 245]. However, if mipmapping is used then care must be taken to avoid having the images bleed into each other at higher subtexture levels.

For flight simulators and geographical information systems, the image datasets can be huge. The traditional approach is to break these images into smaller tiles that hardware can handle. Tanner et al. [262, 347] present an improved data structure called the *clipmap*. The idea is that the entire dataset is treated as a mipmap, but only a small part of the lower levels of the mipmap is required for any particular view. For example, if the viewer is flying above terrain and looking off into the distance, a large amount of the entire texture may be seen. However, the texture level 0 image is needed for only a small portion of this picture. Since minification will take place, a different portion of the level 1 image is needed further out, level 2 beyond that, and so on. By clipping the mipmap structure by the view, one can identify which parts of the mipmap are needed. An example of this technique is shown in Plate VIII (following p. 194).

One solution that directly attacks the memory problem is fixed-rate texture compression. By having hardware decode compressed textures on the fly, a texture can require less texture memory. Another benefit is a reduction in the bandwidth cost of transferring textures. There are a variety of image compression methods [248, 265], but it is costly to implement decoding for these in hardware. S3 came up with a scheme call S3 Texture Compression (S3TC) [304], which was chosen as a standard for DirectX 6.0. It has the advantages of creating a compressed image which is fixed in size, has independently encoded pieces, and is simple (and therefore fast) to decode. Given a particular image type and resolution, we know in advance how much space is needed for the compressed version. This is useful for texture caching, as we know that any image of the given resolution can reuse this image's memory. Each compressed part of the image can be dealt with independently from the others; there are no shared look-up tables or other dependencies, which simplifies decoding.

The image compression scheme is relatively simple. First the image is broken up into 4×4 pixel blocks, called *tiles*. For opaque images (i.e., those with no alpha channel), each 16-pixel block is encoded by storing two colors and 16 2-bit values. The two colors are represented by 16 bits (5 bits red, 6 green, 5 blue) and are chosen to bound the color range of the pixel block. Given these two colors, the encoding and decoding processes derive two other colors that are evenly spaced between them. This gives four colors to choose from, and so for each pixel a 2-bit value is stored as a selection of one of these four colors. Thus, a 16-pixel block is represented by a total of 64 bits, or an average of 4 bits per pixel. Given that an image is often originally stored with 16 or 24 bits per pixel, the scheme results in a 4:1 or 6:1 texture compression ratio. Similar schemes are used for images with 1-bit or 8-bit alpha channels [242, 304].

The main drawback of the compression scheme is that it is *lossy*. That is, the original image cannot always be retrieved from the compressed version. Only 4 different color values are used to represent 16 pixels, so if a tile has more than 4 colors in it there will be some loss. In practice, the compression scheme gives generally acceptable image fidelity. It offers the advantage of using higher-resolution compressed textures in place of uncompressed textures, for the same cost in memory.

5.4 Multipass Rendering

In computer graphics theory, all illumination equation factors are evaluated at once and a sample color is generated. In practice, the various parts of the lighting equation can be evaluated in separate passes, with each successive pass modifying the previous results. This technique is called *multipass rendering* [82, 262].

Hardware accelerators are able to significantly improve performance over software renderers for various operations; however, they do not have the flexibility to provide arbitrarily complex lighting models in a single pass. As a simple example, consider the diffuse and specular parts of the lighting equation. Say you wish to have the diffuse color modulated (multiplied) by a texture, but do not want the specular highlight to be modified by the texture. This is possible using two passes. In the first pass, you would compute and interpolate the diffuse illumination contribution and modulate it by the texture. You would then compute and interpolate the specular part and render the scene again, this time without the texture. The result would then be added to the existing diffusely lit, textured image. Since diffuse and specular components can be treated independently (see Equation 4.14 in chapter 4), it is perfectly acceptable to compute the lighting in this way.

In addition to the texture operations **replace** and **modulate** discussed on page 105, two other combine functions are commonly used in multipass rendering:

- **add** - Add the surface color and the texture color. There are also methods which subtract a bias of 0.5 in order to bring the result back within a useful range. Subtracting the texture color is also a possible function.

- **blend** - As explained in Section 4.5, two color values can be blended by an alpha value. This alpha value can come from a variety of sources: from vertices (in which each vertex is given an alpha value and the alpha is interpolated across the surface), from the image texture itself, from a constant alpha set externally, or from previous computations.

As will be seen, the alpha channel can be used for a variety of effects beyond transparency and antialiasing.

In multipass rendering the same geometry is often reused in each pass. If the transformation stage (and possibly the lighting stage) will yield identical results, it is a waste to have to recompute these results on each pass (and, as importantly, to resend the vertex data on each pass). Direct3D has support for *vertex buffers* and OpenGL for *vertex arrays*, which allow the intermediate results to be computed once and reused.

Some care has to be taken with multipass rendering when it is used with other effects, such as fog. If fog is enabled normally, multiple passes can result in fogging's being applied a number of times. Various blend modes can be used to apply fog properly [122, 187].

Multipass rendering is useful when one wants to enable a rendering system to work on a variety of hardware. For example, the game *Quake III* uses 10 passes. From Hook's notes [70]:

- (passes 1-4: accumulate bump map)
- pass 5: diffuse lighting
- pass 6: base texture (with specular component)
- (pass 7: specular lighting)
- (pass 8: emissive lighting)
- (pass 9: volumetric/atmospheric effects)
- (pass 10: screen flashes)

On the fastest machine up to 10 passes are done to render a single frame. However, if the graphics accelerator[8] cannot maintain a reasonable frame rate, various passes (those in parentheses) can be eliminated. For example, removing the bump-mapping passes reduces the model to 6 passes. The quality of the image is lower, but the real-time experience is maintained. The rendering system is said to be *scalable* if it has this ability to work in some reduced form on various platforms.

5.5 Multitexturing

Obviously, the more passes a renderer must take, the lower its overall performance. To reduce the number of passes, some graphics accelerators support *multitexturing*, in which two or more textures are accessed during the same pass [83, 322]. To combine the results of these texture accesses, a *texture blending cascade* (a pipeline) is defined that is made up of a series of *texture stages* [83], also called texture units [322]. The first texture stage combines two texture (or interpolated vertex) values, typically RGB and perhaps α (alpha), and this result is then passed on to the next texture stage. Second and successive stages then blend another texture's or interpolant's values with the previous result. This is illustrated in Figure 5.14. Another way to think of this process is that the texture units form a series like a pipeline. The triangle's interpolated vertex values enter the pipeline. Then, each texture in turn is applied to the set of values. The final output set is applied to the stored values (RGB and possibly α) in the frame buffer [322].

In addition to saving rendering passes, multitexturing actually allows more complex shading models than does the application of a single texture per pass. For example, say you want to use a lighting model with expression $AB + CD$, where each variable represents a different color texture's value. This expression is impossible to evaluate without multitexturing. A multipass algorithm could combine AB in two passes and add C in the next, but it could not fold D in because C would already have been added to AB. There is no place to keep C separate from AB, since only one color can be stored in the color buffer. With multitexturing, the first pass could compute AB. Then the second pass would compute CD and add this result to the AB in the color buffer [255].

As will be seen in the example at the end of Section 5.7.3, multitexturing can also help avoid the allocation of an alpha channel in the frame buffer.

[8] *Quake III* requires an accelerator.

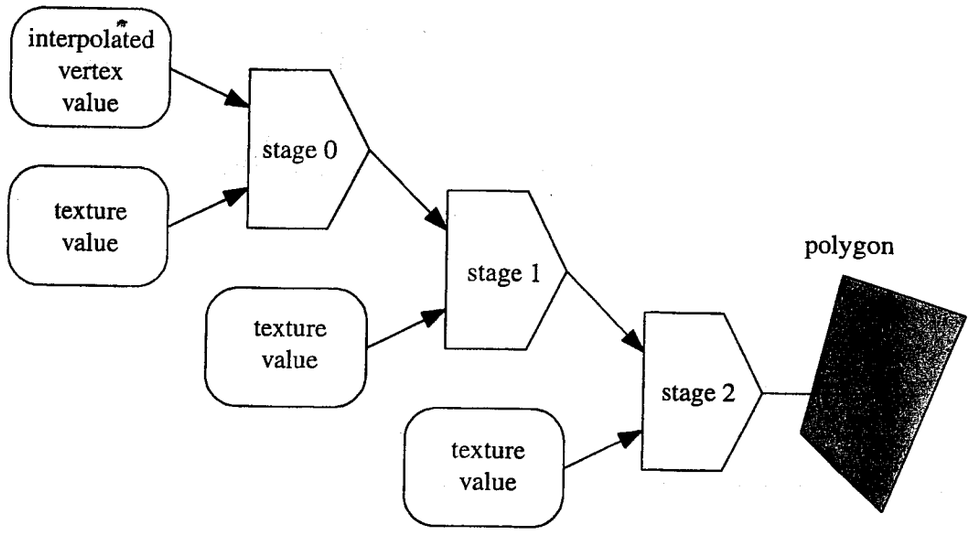

Figure 5.14. The result from a texture is combined with an interpolated value in stage 0, and the result of that combination is then combined with another texture in stage 1. This result of that combination is combined with yet another texture in stage 2 and then applied to the polygon.

5.6 Texture Animation

The image applied to a surface does not have to be static. For example, a video source can be used as a texture that changes from frame to frame.

The texture coordinates need not be static, either. In fact, for environment mapping, they usually change with each frame because of the way they are computed (see section 5.7.3). The application designer can also explicitly change the texture coordinates from frame to frame. Imagine that a waterfall has been modeled and that it has been textured with an image that looks like falling water. Say the v coordinate is the direction of flow. To make the water move, one must subtract an amount from the v coordinates on each successive frame. Subtraction from the texture coordinates has the effect of making the texture itself appear to move forward.

More elaborate effects can be created by modifying the texture coordinates. Linear transformations such as zoom, rotation, and shearing are possible [245]. Image warping and morphing transforms are also possible [380], as are generalized projections [149].

By using texture blending techniques, one can realize other animated effects. For example, by starting with a marble texture and fading in a flesh texture, one can make a statue come to life [254].

5.7 Texturing Methods

With the basic theory in place, we will now cover various other forms of texturing beyond gluing simple color images onto surfaces. The rest of this chapter will discuss the use of alpha blending in texturing, reflections via environment mapping, and rough surface simulation using bump mapping, among others. Chapter 6, on special effects, will also frequently draw upon texturing methods.

5.7.1 Alpha Mapping

The alpha value can be used for many interesting effects. One texture-related effect is decaling. As an example, say you wish to put a picture of a flower on a teapot. You do not want the whole picture, but just the parts where the flower is present. By assigning an alpha of 0 to a texel you make it transparent so that it has no effect. So, by properly setting the decal texture's alpha, you can replace or blend the underlying surface with the decal. Typically, a clamp or border corresponder function is used with a transparent border to apply the decal just once to the surface.

Another application of alpha is in making cutouts. Say you make a decal image of a tree, but you do not want the background of this image to affect the scene at all. That is, if an alpha is found to be fully transparent, then the textured surface itself does not affect that pixel. In this way you can render an object with a complex silhouette using a single polygon.

In the case of the tree, if you rotate the viewer around it, the illusion fails since the tree has no thickness. One answer is to copy this tree polygon and rotate it 90 degrees along the trunk. The two polygons form an inexpensive three-dimensional tree, and the illusion is fairly effective when viewed from ground level. See Figure 5.15. In Section 6.2.2, we discuss a method called billboarding, which is used to reduce such rendering to a single polygon.

Providing this texture effect means a slight extension of the rendering pipeline. Before testing and replacing the Z-buffer value, there needs to be a test of the alpha value. If alpha is 0, then nothing further is done for the pixel. If this additional test is not done, the tree's background color will not affect the pixel's color, but will potentially affect the z-value, which can lead to rendering errors.

Combining alpha blending and texture animation can produce convincing special effects, such as flickering torches, plant growth, explosions, atmospheric effects, etc.

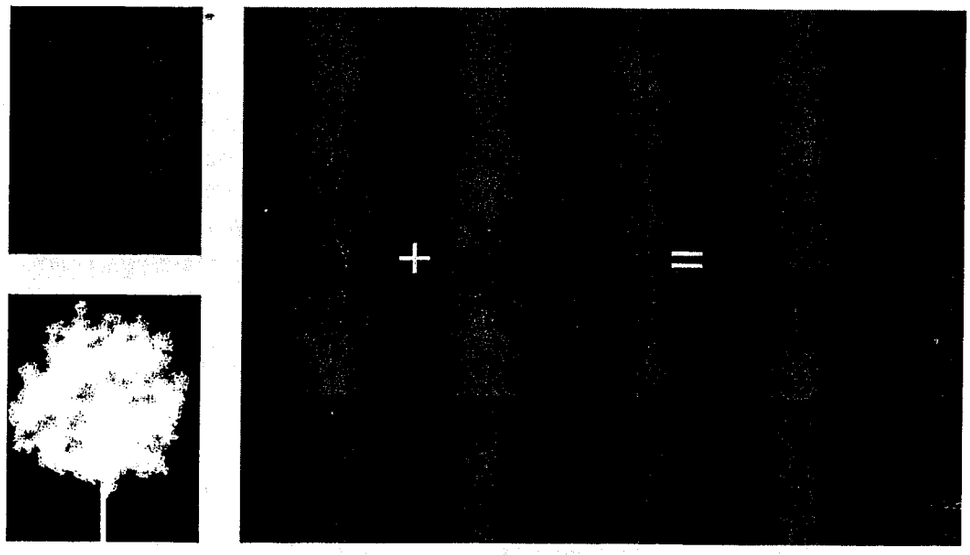

Figure 5.15. On the left, the tree texture map and the 1-bit alpha channel map below it. On the right, the tree rendered on a single polygon; by adding a second copy of the polygon rotated 90 degrees we form an inexpensive three-dimensional tree.

5.7.2 Light Mapping

In section 4.3 Phong shading was presented. This technique yields more precise lighting evaluation by computing the illumination at each pixel. But since Gouraud shading is the norm in graphics hardware, short of meshing surfaces finely in order to capture the lighting's effect more finely, it appears that general Phong shading is not possible. However, for static lighting in an environment, the diffuse component on any surface remains the same from any angle. Because of this view-independence, the contribution of light to a surface could be captured in a texture structure attached to a surface. An elaborate version of this concept was first implemented by Arvo [15] and later by Heckbert [165] in order to capture global illumination information, i.e., light bouncing around the environment.

Carmack realized that this sort of method could be applied to real-time work in lieu of using lighting equations during rendering [1]. By using a separate, precomputed texture that captured the lighting contributions, and multiplying it with the underlying surface, one could achieve Phong-like shading (see Plate IV, following p. 194). As Hook points out, this technique is more accurately termed "dark mapping," because the original surface actually decreases in intensity [187]. To make light mapping a bit easier to use and to maintain brightness,

Direct3D provides texture blending operations which boost the output intensity by a factor of two or four. Values above 1.0 are clamped to 1.0. An example is shown in Plate V (following p. 194). While multiplying the two textures has a physical meaning associated with it, for a different look another lighting model could be used, such as adding or blending the two textures together.

The light texture can simply be multiplied by the surface's material texture during the modeling stage, and the single resulting texture can be used. However, a number of advantages accrue by using light mapping in a separate stage. First, the light texture can generally be low-resolution. In many cases lighting changes slowly across a surface, so the texture representing the light can be quite small compared to the surface. Minimal processing is needed to modify or swap such light textures on the fly. Lighting situations can be reused with a variety of environments, and color textures can be tiled and have different lighting on each tile (see Plate V, following p. 194). Using texture coordinate animation techniques (Section 5.6), light textures can be made to move on walls–e.g., a moving spot light effect can be created. Also, light maps can be recalculated on the fly for dynamic (changing) lighting. In summary, the flexibility of using separate light textures often outweighs the cost of a separate pass. With multitexturing support, even this cost disappears.

Using textures to represent shadows is related to the idea of light mapping. See Section 6.6.1 for more about this method.

An interesting extension to image texture light maps is three-dimensional texture light maps, where actual beams of light are stored throughout the volume [245]. Though memory-intensive (as are all three-dimensional textures), this method has the advantage that the light is actually there in space. Moving the texture around moves the light's effect in a simple and natural way. Projective textures [245, 321] are also related to light maps, providing a more flexible method of creating lighting such as that made by a slide projector.

5.7.3 Gloss Mapping

Image light maps are typically used on diffuse surfaces, and so are sometimes called diffuse light maps. The specular component can also be affected by light mapping, but here the effect is sometimes a little more involved.

The specular component is computed using the eye direction and the light direction. In real-time work it is typically calculated at polygon vertices and interpolated. Just as a texture can be used to modulate the diffuse lighting on a surface to produce a brick wall effect, so can a different texture produce varying specularity. Such a texture is typically a monochrome (gray-scale) texture, where 1.0 means the full specular component is to be used, and 0.0 means none is used.

This technique is called *gloss mapping* [255]. The pear in Plate IX (following p. 194) shows this effect.

It is important to realize that for Gouraud-shaded objects the underlying specular component is still computed only on a per-vertex basis and interpolated. For example, imagine a brick wall where the bricks are shiny and the mortar is not. This shininess would form a gloss texture where the bricks are white and the mortar is black. Applying this texture to a single polygon with the brick wall texture will change the specular component's contribution. So if the light source is, say, a spot light, its effect will not be captured. For example, if we situated a spot light so as to give the effect shown in Plate IV (following p. 194), the light's computed intensity at the corners of the textured polygon would be zero, and it would therefore provide no specular highlighting at all across the surface.

To account for effects such as spot lights and cones of influence, the specular highlight component of the lighting equation can be replaced with a *specular light map*. This is simply a light map that portrays the spot-light or cone effect desired. A diffuse light map directly modulates a surface texture. A specular light map modulates the computed highlight contribution, and this result modulates the gloss map. In the brick wall example, the specular highlight is computed without the effect of spot lighting or cones, and so it will no longer be zero. This interpolated highlight color is multiplied by the specular light map, giving a spot-light effect to the specular contribution. Finally the gloss map is multiplied into this result, and the bricks appear shiny while the mortar does not. This whole process computes the specular contribution.

The diffuse calculations could also use a light map, but this light map is likely to have a different texture than the specular light map. This is because the diffuse light map includes both the spot-light effect and the fall-off in intensity due to the diffuse surface's angle to the light. To summarize in an equation:

$$\mathbf{o} = \mathbf{m}_{diff} \otimes \mathbf{t}_{diff} + (\mathbf{i}_{spec} \otimes \mathbf{m}_{spec}) t_{gloss} \tag{5.2}$$

where $\mathbf{m}_{diff}$ is the diffuse light map, $\mathbf{t}_{diff}$ is the diffuse texture map, $\mathbf{i}_{spec}$ is the interpolated highlight color, $\mathbf{m}_{spec}$ is the specular spot-light effect light map, and t_{gloss} is the single-value gloss texture. This is just one possible lighting equation. For example, the light maps could have a single intensity value instead of a color. They could be replaced by interpolated diffuse lighting in the diffuse term and be left out entirely of the specular term. The interpolated specular color could be replaced by an interpolated intensity value. Lighting equations can always be changed or extended in order to create a desired effect or to increase speed.

While often convincing, this procedure has some limitations. Specular highlights are often blurry and animate poorly, as they are captured only at each vertex

instead of at each pixel. In the brick wall example, if the brightest part of the highlight fell in the middle of the wall, it would not be captured by interpolation from the corners. It would be much better to have the lights' contributions per pixel, or better yet the contributions of the lights and all objects reflecting light. These contributions could then be modulated by the gloss texture. To create such reflections we can use environment mapping, the next topic.

5.7.4 Environment Mapping

Environment mapping (EM), also called reflection mapping, is a simple yet powerful method of generating approximations of reflections in curved surfaces. This technique was introduced by Blinn and Newell [36]. All EM methods start with a ray from the viewer to a point on the reflector. This ray is then reflected with respect to the normal at that point. Instead of finding the intersection with the closest surface, as ray tracing does [116], EM uses the direction of the reflection vector as an index to an image containing the environment. This is conceptualized in Figure 5.16.

The environment mapping approximation assumes that the objects and lights being reflected with EM are far away, and that the reflector will not reflect itself. If these assumptions hold, then the environment around the reflector can

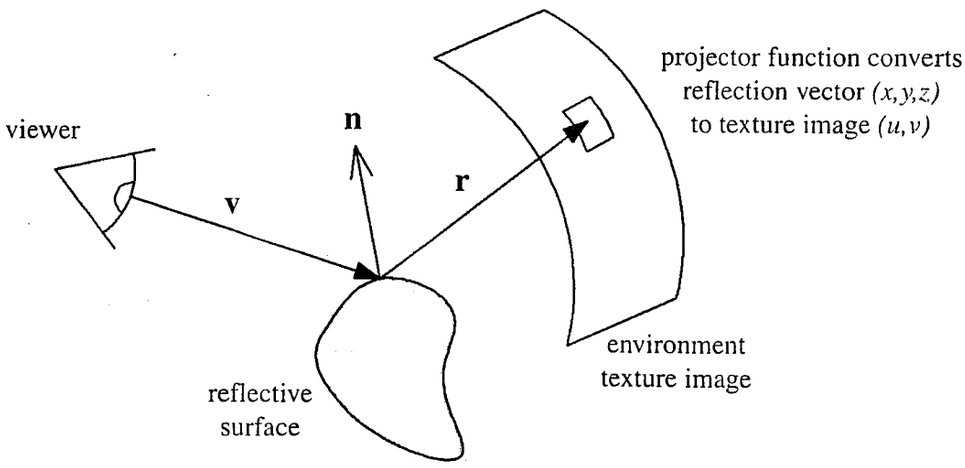

Figure 5.16. Environment mapping. The viewer sees an object, and the reflection vector $\mathbf{r}$ is computed from $\mathbf{v}$ and $\mathbf{n}$. The reflection vector accesses the environment's representation. The access information is computed by using some projector function to convert the reflection vector's (x, y, z) to (typically) a (u, v) value, which is used to retrieve texture data.

be treated as a two-dimensional projection surrounding it.

The steps of an EM algorithm are:

- Generate a two-dimensional image of the environment (this is the environment map).
- For each pixel that contains a reflective object, compute the normal at the location on the surface of the object (if per-pixel EM is not available, then the normal is computed at triangle vertices).
- Compute the reflection vector from the view vector and the normal.
- Use the reflection vector to compute an index into the environment map that represents the objects in the reflection direction.
- Use the texel data from the environment map to color the current pixel.

There are a variety of projector functions that map the reflection vector into one or more textures. Blinn and Newell's algorithm and Greene's cubic environment mapping technique are classic mapping methods, and so are covered first. However, these two methods are only just beginning to see support in commercial hardware. The sphere map technique is presented next. It is the projector function most commonly used for EM in graphics accelerators. Finally, a new method from Heidrich and Seidel is explained, as it shows great promise for future hardware support.

Blinn and Newell's Method

In 1976, Blinn and Newell [36] developed the first environment mapping algorithm. For each environment-mapped pixel, they compute the reflection vector and then transform it into spherical coordinates (ρ, ϕ). Here ϕ, called longitude, varies from 0 to 2π radians, and ρ, called latitude, varies from 0 to π radians. (ρ, ϕ) is computed from Equation 5.3, where $\mathbf{r} = (r_x, r_y, r_z)$ is the normalized reflection vector.

$$\begin{aligned} \rho &= \arccos(-r_z) \\ \phi &= \begin{cases} \arccos(r_x / \sin\rho) & \text{if } r_y \geq 0 \\ 2\pi - \arccos(r_x / \sin\rho) & \text{otherwise} \end{cases} \end{aligned} \tag{5.3}$$

The viewer's reflection vector is computed similarly to the light's reflection vector (see Section 4.3.2):

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n} - \mathbf{v} \tag{5.4}$$

where $\mathbf{v}$ is the normalized vector from the eye to the surface location, and $\mathbf{n}$ is the surface normal at that location.

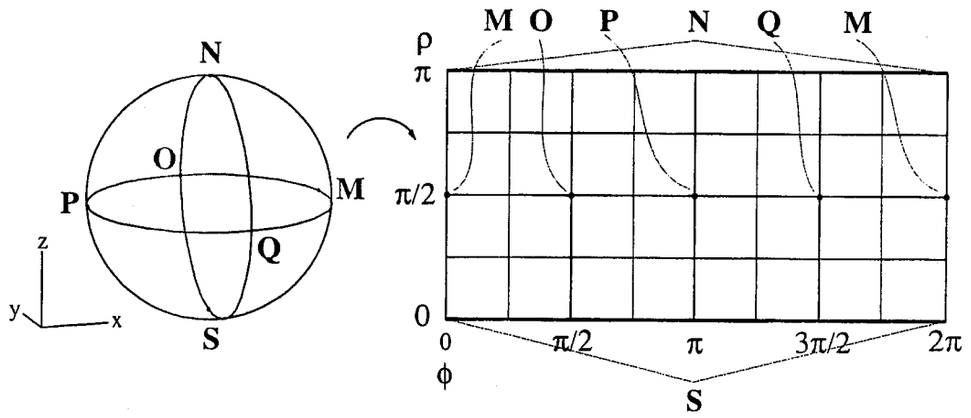

Figure 5.17. Illustration of Blinn and Newell's environment map. The sphere on the left is unfolded into the rectangle (the environment map) on the right. The key points **N**, **S**, **M**, **O**, **P**, and **Q**, mentioned in the text, are also shown.

The spherical coordinates (ρ, ϕ) are then transformed to the range $[0, 1)$ and used as (u, v) coordinates to access the environment texture, producing a reflection color. Since the reflection vector is transformed into spherical coordinates, the environment texture is an image of an "unfolded" sphere. Essentially, the texture covers a sphere that surrounds the reflection point. This projector function is sometimes called a latitude-longitude mapping, since v corresponds to latitude and u to longitude. This mapping is shown in Figure 5.17.

Some key points are displayed in this table:

name	*coordinate*	*angles*
N (north pole)	$(0, 0, 1)$	$\rho = \pi, \phi$ is undefined
S (south pole)	$(0, 0, -1)$	$\rho = 0, \phi$ is undefined
M	$(1, 0, 0)$	$\rho = \pi/2, \phi = 0$
O	$(0, 1, 0)$	$\rho = \pi/2, \phi = \pi/2$
P	$(-1, 0, 0)$	$\rho = \pi/2, \phi = \pi$
Q	$(0, -1, 0)$	$\rho = \pi/2, \phi = 3\pi/2$

Although easy to implement, this method has some disadvantages. First, there is a border at $\phi = 0$, and second, the map converges at the poles. An image used in environment mapping must match at the seam along its vertical edges (i.e., must tile seamlessly) and should avoid distortion problems near each horizontal edge.

This method computes an index into the environment map for each visible point on the objects that should have reflections. Its computations are therefore

on a per-pixel basis, which implies that it is not normally feasible for real-time graphics. This is because lighting equations are normally evaluated at the vertices, not per pixel.

For real-time work, Equation 5.3 can be used to compute indices into the environment map at the vertices, and then these coordinates can be interpolated across the triangle. However, errors will occur if the vertices of a triangle have indices to the environment map that cross the poles. As will be discussed in Section 9.2.1, texture coordinate problems at the poles are difficult to avoid when using triangles for interpolation.

Errors can also occur if the endpoints span the seam where the vertical edges of the environment textures meet. For example, imagine a short line that has one u coordinate at 0.97 and the other u coordinate at 0.02. An error results if we interpolate from 0.97 to 0.02 without paying attention to the seam, as the interpolation would travel through 0.96 on down. Interpolation should go up to 0.98, 0.99, then wrap to 0.0, 0.01, 0.02. One solution to this problem is to find the absolute value of the difference between the coordinates ($0.95 = 0.97 - 0.02$ for the example). If this value is greater than 0.5, then 1.0 is added to the smaller coordinate and the texture is repeated. For the example, the range would then be 0.97 to 1.02. Some APIs have direct support to avoid seam problems. For example, Direct3D supports what they call "texture wrapping" (not to be confused with their "wrap texture address mode"; see Section 5.1).

Cubic Environment Mapping

In 1986, Greene [140] introduced another EM technique. His environment map is obtained by placing the camera in the center of the environment and then projecting the environment onto the sides of a cube positioned with its center at the camera's location. The images of the cube are then used as the environment map. In practice, the scene is rendered six times (one for each cube face) with the camera at the center of the cube, looking at each cube face with a 90-degree view angle.

This type of environment map is shown in Figure 5.18. In Figure 5.19 is a typical cubic environment map.

Greene's method is used in many photorealistic systems today, and we expect to see it in real-time rendering APIs. Its strength is that environment maps can be generated by the system itself relatively easily (vs. Blinn and Newell's method, which uses a spherical projection), and could even be generated on the fly. It also has more uniform sampling characteristics. Blinn and Newell's method has an excessive number of texels near the poles as compared to the equator.

The direction of the normalized reflection vector determines which face of the cube to use. The reflection vector coordinate with the largest magnitude selects the corresponding face (e.g., the vector $(-0.2, 0.5, -0.84)$ selects the $-Z$ face).

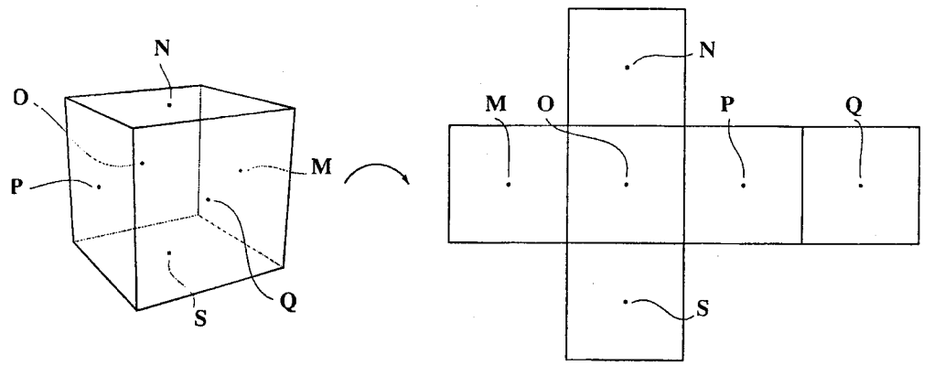

Figure 5.18. Illustration of Greene's environment map, with key points shown. The cube on the left is unfolded into the environment map on the right.

The remaining two coordinates range from -1 to 1 and are simply remapped to $[0, 1]$ in order to compute the texture coordinates (e.g., the coordinates $(-0.2, 0.5)$ are mapped to $(0.4, 0.75)$). In the same manner as Blinn and Newell's method, this technique is per-pixel based. If two vertices are found to be on different cube faces, it is difficult to interpolate correctly between them. One method is to subdivide the problematic polygon along the cube edge [149]. Another method that can help with smaller polygons is to make the environment map faces larger than 90 degrees in view angle, then see if any face fully contains the polygon.

Voorhies and Foran [361] have developed a possible hardware solution for Greene's EM method, but as of this writing it has not been implemented in any commercial hardware.

A high-performance real-time graphics system called *PixelFlow* [259] has shaders that can be programmed to perform Greene's environment mapping method in real-time.

Sphere Mapping

First mentioned by Williams [375], this is the standard projector function used on SGI machines, and so is supported by OpenGL. It is also supported indirectly in Direct3D, since it involves accessing a single texture. A texture representing a circle is used for the environment texture (see Plate VII, following p. 194). The image is derived from the appearance of the environment as viewed orthographically in a perfectly reflective sphere, so this texture is called a sphere map. Sphere map textures can be generated using ray tracing or by warping the images generated for a cubic environment map [245].

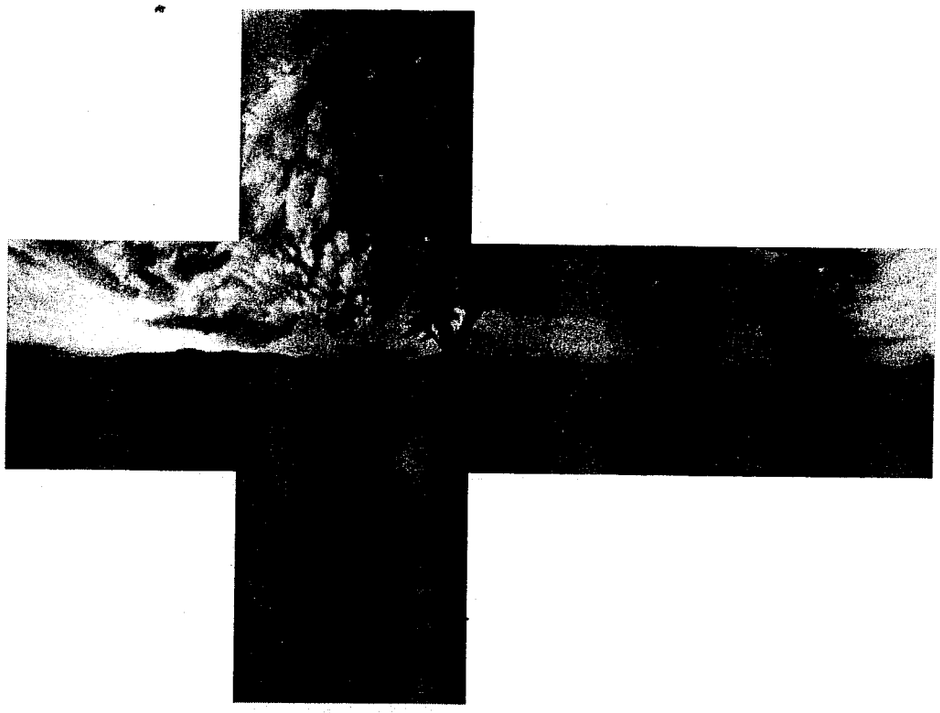

Figure 5.19. A typical cubic environment map. *(Courtesy of Ned Greene/NYIT)*

The sphere map has a basis (see Appendix A) which is the frame of reference in which the texture was generated. That is, the image is viewed along some axis $-\mathbf{e}$ in world space, with $\mathbf{u}$ as the up vector for the image and $\mathbf{r}$ going to the right (and all are normalized). This gives a basis matrix:

$$\begin{pmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ -e_x & -e_y & -e_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.5}$$

To access the sphere map, first transform the surface normal, $\mathbf{n}$, and the vector from the current eye position to the vertex $\mathbf{e}$, using this matrix. This yields $\mathbf{n}'$ and $\mathbf{e}'$ in the sphere map's space. The reflection vector is then computed to access the sphere map texture:

$$\mathbf{r} = 2(\mathbf{n}' \cdot \mathbf{e}')\mathbf{n}' - \mathbf{e}' \tag{5.6}$$

with $\mathbf{r}$ being the resulting reflection vector, in the sphere map's space. From the reflection vector the parameter-space values are calculated as follows:

$$m = \sqrt{r_x^2 + r_y^2 + (r_z + 1)^2} \tag{5.7}$$

$$u = \frac{r_x}{2m} + 0.5 \tag{5.8}$$

$$v = \frac{r_y}{2m} + 0.5 \tag{5.9}$$

The value m is computed in part by adding 1 to r_z, which effectively brings the hemisphere behind the reflective sphere to the front. The texture coordinates are normalized by dividing by m, then remapped from the range $[-1, 1]$ to $[0, 1]$ by the rest of the formulae.

Sphere mapping is an improvement over Blinn and Newell's method in a number of ways. The simplicity of implementing its projector function in hardware is its great strength. There is no texture seam to interpolate across and only one singularity, located around the edge of the sphere map. However, moving between two points on the sphere map is not a linear affair. As in Blinn and Newell's method, linear interpolation on the texture is just an approximation, and errors become more serious as the viewer leaves the view axis used to generate the sphere map texture. To improve quality, sphere maps can be regenerated from cubic maps on the fly [245]. If mipmapping is used, then the entire mipmap has to be generated, too. Note that OpenGL support for sphere mapping is in view space by default, so even if the view changes direction, the same mapping gets used. In other words, if you were to look at a reflective sphere and then move elsewhere and look at the same sphere, the reflection would not change but be represented by the same sphere map. A slower but more accurate approach is to compute the EM texture coordinates in the application stage for each frame.

See Plate VI (following p. 194) for an example of environment mapping done with sphere maps.

Paraboloid Mapping

Heidrich and Seidel [171, 174] propose using two environment textures, in what we call here paraboloid mapping. The idea is similar to that of sphere mapping, but instead of generating the texture by recording the reflection of the environment off a sphere, two paraboloids are used. Each paraboloid creates a circular texture similar to a sphere map, with each covering an environment hemisphere. See Figure 5.20.

As with sphere mapping, the reflection ray is computed with Equation 5.6 in the map's basis, choosing one of the two textures as the front-facing texture. The sign of the z-component of the reflection vector is used to decide which

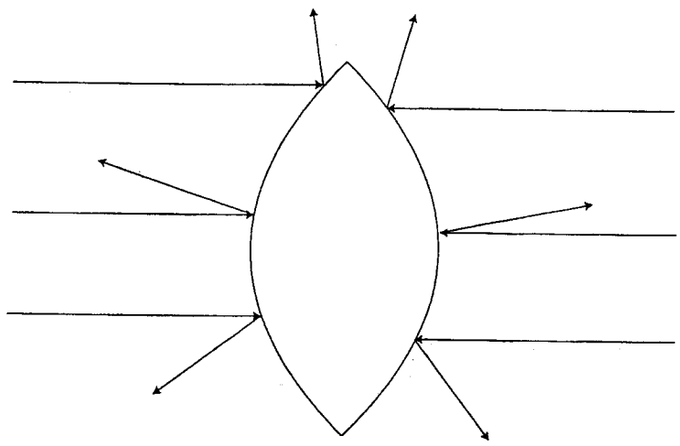

Figure 5.20. Two paraboloid mirrors that capture an environment using diametrically opposite views.

texture to access. Then the access function is simply:

$$u = \frac{r_x}{2(1+r_z)} + 0.5 \tag{5.10}$$

$$v = \frac{r_y}{2(1+r_z)} + 0.5 \tag{5.11}$$

for the front image, and the same with sign reversals for r_z for the back image.

The authors present an OpenGL implementation for this method. The problem of interpolating across the seam between the two textures is handled by accessing both paraboloid textures. If a sample is not on one texture it is black, and each sample will be on one and only one of the textures. Summing the results (one of which is always zero) gives the environment's contribution.

This method is superior to the sphere map scheme in that it has no singularity. Also, the paraboloid has more uniform texel sampling of the environment compared to the sphere map and even the cubical map. The paraboloid map's uniformity means that it is generally good from any view and so does not need to be regenerated. In contrast, to maintain image quality, some applications regenerate the sphere map when the viewer moves. Paraboloid mapping is a possible future candidate for hardware implementation.

Extensions and Limitations

It is worth pointing out an important use of EM techniques: specular reflections and refractions [254, 375]. As discussed in Section 4.3, Gouraud shading can miss highlights (reflections of lights) because a light's effect is assessed only at the vertices. Environment mapping can solve this problem by representing the

lights in the texture. By doing so, we can simulate highlighting on a per-pixel basis (see Plate VI, following p. 194). Furthermore, to simulate a different degree of roughness, the environment's representation in the texture can be modified. By blurring the texture, we can present a rougher-appearing surface. In theory, such blurring should be done in a non-linear fashion; that is, different parts of the texture should be blurred differently. In practice, the eye tends to be fairly forgiving, because the general effect of the reflection is usually more important than the exact reflection itself.

Other advantages to using environment mapping instead of specular highlighting include speed and versatility. The speed of accessing the environment map is constant, regardless of how many lights are represented in the texture. Environment mapping is also flexible, as reflected light sources can be linear or area lights, and more than just lights can be represented in the texture. That said, the limitations are that generating and storing EM textures can be costly in time, effort, and resources. Such textures are also often limited to a certain volume of space. For example, if the reflective object moves from one room to another, it is extremely difficult to transition smoothly from one environment texture to another. Methods such as Greene's cubic EM allows on-the-fly regeneration of environment textures, but this is a costly process.

Another potential stumbling block of EM is worth mentioning. Flat surfaces usually do not work well when environment mapping is used. The problem with a flat surface is that the rays that reflect off of it usually do not vary by more than a few degrees. This results in a small part of the EM texture's being mapped onto a relatively large surface. Normally, the individual texels of the texture become visible, unless bilinear interpolation is used; even then, the results do not look good, as a small part of the texture is extremely magnified. We have also been assuming that perspective projection is being used. If orthographic viewing is used, the situation is much worse for flat surfaces. In this case, all the reflection vectors are the same, and so the surface will get a constant color from some single texel. Other real-time techniques such as planar reflections (Section 6.5.1) may be of more use for flat surfaces.

EXAMPLE: DIFFUSE, GLOSS, AND ENVIRONMENT MAPPING Various texturing effects can be combined. One example is using environment mapping with gloss and diffuse color texturing [255]. The pear shown in Plate IX (following p. 194) is rendered using this lighting model, which looks like this:

$$\mathbf{o} = \mathbf{t}_{diff} \otimes \mathbf{i}_{diff} + t_{gloss}\mathbf{e}_{spec} \tag{5.12}$$

where $\mathbf{t}_{diff}$ is the RGB texture, $\mathbf{i}_{diff}$ the interpolated diffuse lighting, t_{gloss} the monochrome gloss texture, and $\mathbf{e}_{spec}$ the environment map's specular color.

This model can be computed in two passes. The first texture applied to the pear is the specular highlight texture, $\mathbf{e}_{spec}$. Doing so gives a black pear with a bright white highlight. The second texture is set up so that the RGB values are the diffuse texture, $\mathbf{t}_{diff}$, and the alpha value is actually the gloss intensity, t_{gloss}.

To clarify: the alpha channel is not used in the traditional way here; it does not affect the opacity of the RGB values. In the second rendering pass, this texture is applied to the surface. The RGB values are multiplied by the diffuse illumination interpolated from the vertices, $\mathbf{i}_{diff}$. The shading mode is set so that blending is done by multiplying the second texture's alpha value (the gloss) by the previous color (the specular environment map result from the first pass, now in the frame buffer). This product is added to the first term, the illuminated RGB texture. Essentially, the alpha channel is used as a parallel data stream, computing the effect of gloss mapping on the specular highlight while the RGB texture is multiplied by the diffuse lighting.

Using multitexturing here eliminates the need to allocate a separate alpha channel in the frame buffer. Without multitexturing, the algorithm would have to be restructured, and the t_{gloss} alpha value from the second texture would have to be stored in the frame buffer's alpha channel so that it could modulate the specular environment map in a later pass. □

5.7.5 Bump Mapping

Introduced by Blinn in 1978 [38], *bump mapping* is a technique that makes a surface appear uneven in some manner: bumpy, wrinkled, wavy, etc. The basic idea is that, instead of changing the color components in the illumination equation, we modify the surface normal by accessing a texture. The geometric normal of the surface remains the same; we merely modify the normal used in the lighting equation. This operation has no physical equivalent; we perform changes on the surface normal, but the surface itself remains smooth in the geometric sense. Just as having a normal per vertex gives the illusion that the surface is smooth between polygons, modifying the normal per pixel changes the perception of the polygon surface itself.

There are two basic methods of modifying the normal with a bump map. One bump texturing technique uses two signed values, b_u and b_v, at each point. These two values correspond to the amount along the $\mathbf{u}$ and $\mathbf{v}$ image axes by which to vary the normal. That is, these texture values, which typically are bilinearly interpolated, are used to scale two vectors that are perpendicular to the normal. These two vectors are added to the normal to change its direction.

These two values essentially represent how steep the surface is and which way it faces at the point. See Figure 5.21.

The second method uses a *height field* to modify the surface normal's direction. That is, each monochrome texture value represents a height, so white is a high area and black a low one (or vice versa; a signed value is used to scale or negate the values in bump maps in still-image rendering systems). The height field is used to derive u and v signed values similar to those used in the first method. This is done by taking the differences between neighboring columns to get the slopes for u, and between neighboring rows for v [317]. Figure 5.22 shows an example.

Per-pixel bump mapping is extremely convincing, and offers an inexpensive way to add the effect of geometric detail. One place the illusion breaks down is around the silhouettes of objects. At these edges the viewer notices that there are no real bumps but just smooth outlines. Also note that mipmapping does not work with bump textures, as it has the effect of making the texture slopes disappear. Another artifact of using bump mapping is that shadows are not produced by the bumps, which can look unrealistic.

In classical bump mapping, the normal is varied per pixel and used in an illumination equation, which is beyond the capabilities of almost all real-time systems. However, there are a number of techniques that can be used for real-time bump mapping.

One algorithm that is conceptually simple is the dot product method. Instead of storing heights or slopes, the actual new normals for the surface are stored as

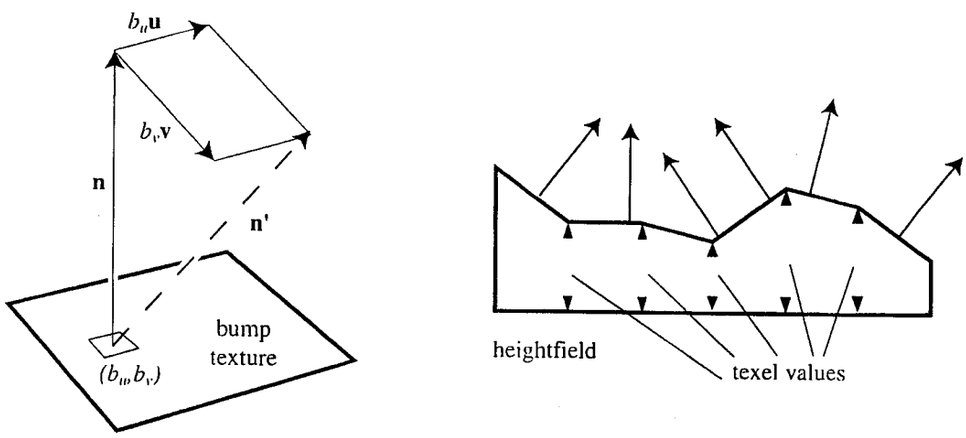

Figure 5.21. On the left, a normal vector **n** is modified in the **u** and **v** directions by the (b_u, b_v) values taken from the bump texture, giving $\mathbf{n}'$ (which is unnormalized). On the right, a height field and its effect on shading normals is shown.

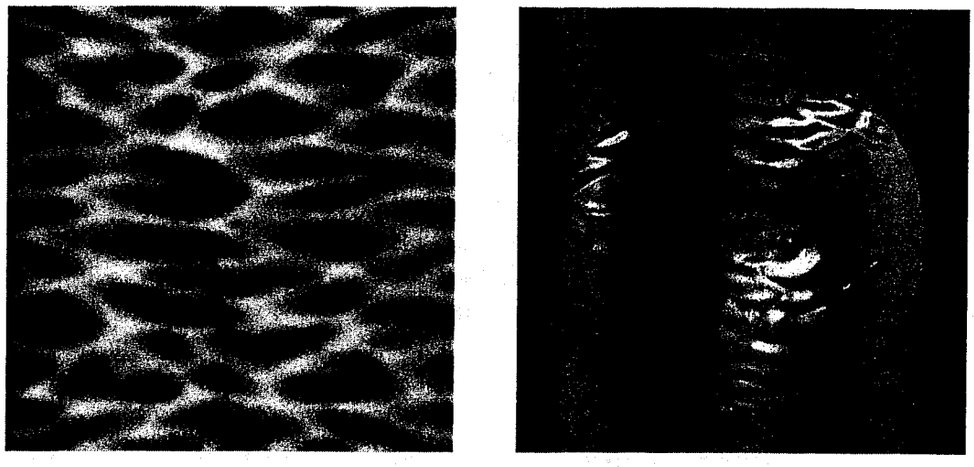

Figure 5.22. A wavy height field bump image and its use on a sphere, rendered with per-pixel illumination.

(x, y, z) vectors in a *normal map* [61]. To compute the effect of a light source, the positional light source's location is first transformed to the surface's basis (similarly to Equation 5.5). The surface's basis is formed by its geometric normal and the $\mathbf{u}$ and $\mathbf{v}$ axes of the texture. With the light now oriented with respect to the surface, the vector from each vertex to the light is computed in this basis and then normalized. The coordinates of each vertex's normal are interpolated across the surface. In other words, instead of a color or depth value, a vector to the light is interpolated.[9] The bump texture, which consists of normals, is then combined with the interpolated light vector at each pixel. These are combined by taking their dot product, which is a special texture-blending function provided for precisely this purpose. An example is shown in Plate XI (following p. 194). Computing the dot product of the normal and the light vector is exactly how the diffuse component is calculated (see Section 4.3.1), so this results in a bumpy-looking surface that will change appearance as it moves with respect to the light. For directional lights the technique is simpler still, as the vector from the polygon to the light is constant. Direct3D supports this specialized texture-blending operation as `D3DTOP_DOTPRODUCT3`, but that does not mean there is widespread hardware support for it. For this reason, another algorithm is more popular, as it uses commonly available capabilities, though at the cost of additional rendering passes.

This other method for Gouraud bump mapping employs a technique borrowed from two-dimensional image processing. Called *embossing* [317], it is

[9] The light vector that is interpolated will actually become shorter than a length of 1, but the error from this is usually unnoticeable.

Figure 5.23. Embossing. The original image is on the left; the center shows the image embossed as if lit from the upper left; the right shows it as if lit from the upper right.

a way to give a chiselled look to an image. To obtain an embossed effect, the height field image is copied, shifted slightly, and subtracted from its original. See Figure 5.23.

This same embossing technique can be used with three-dimensional surfaces. The basic algorithm is as follows [245]:

1. Render the surface with the height field applied as a diffuse monochrome texture.

2. Shift all the vertex (u, v) coordinates in the direction of the light.

3. Render this surface with the height field again applied as a diffuse texture, subtracting from the first-pass result. This gives the emboss effect.

4. Render the surface again with no height field, simply illuminated and Gouraud-shaded. Add this shaded image to the result.

The involved part of the procedure is determining how much to shift the vertex (u, v) values in step 2. What we need to do is add to these vertex values the light's direction relative to the surface. This process is shown in Figure 5.24.

To find the light's direction relative to the surface, we need to form a coordinate system at the vertex and transform the light into this system. First retrieve the normal $\mathbf{n}$ at the vertex. Then find a surface vector $\mathbf{t}$ that follows one of the two texture coordinate axes $\mathbf{u}$ or $\mathbf{v}$; $\mathbf{t}$ is tangent to (i.e., travels along) the surface. Finally create a third vector $\mathbf{b}$, the binormal, which is mutually perpendicular to $\mathbf{n}$ and $\mathbf{t}$ and runs in the direction of the other texture coordinate axis (the one not used when forming $\mathbf{t}$). This is done by simply computing the cross product:

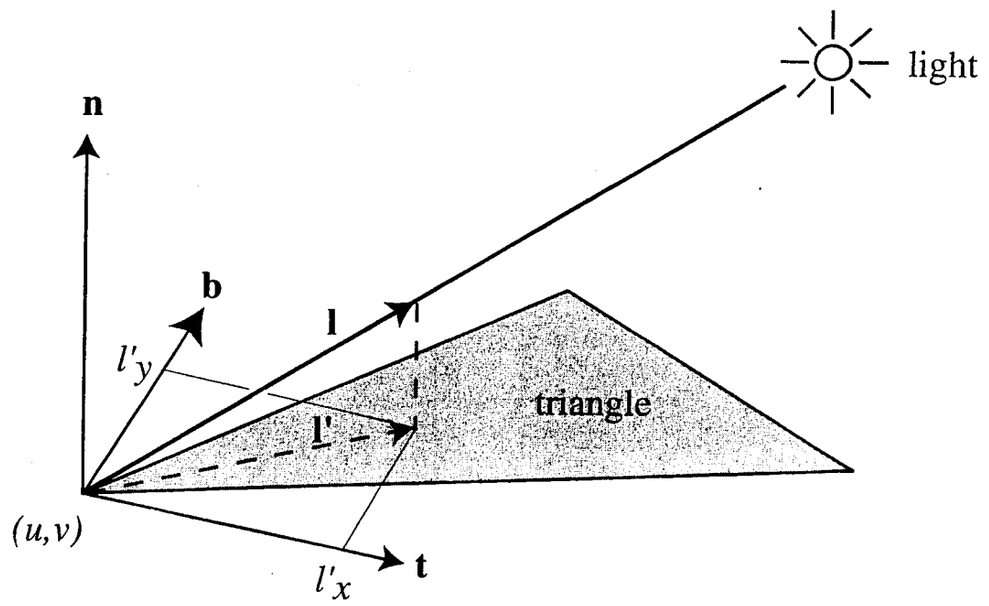

Figure 5.24. The light's direction l is cast upon the triangle's plane, and the resulting vector l′ is added to the vertex's texture coordinates.

$\mathbf{b} = \mathbf{n} \times \mathbf{t}$. Normalize all these vectors and form a basis matrix:

$$\begin{pmatrix} t_x & t_y & t_z & 0 \\ b_x & b_y & b_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.13}$$

This matrix is used to transform l, the normalized vector from the vertex to the light. Note that for curved surfaces and for flat surfaces where the light vector noticeably changes, this basis matrix creation and shift computation must be performed separately at each vertex. The resulting vector l′ gives the direction in which to shift (u, v). For example, if t follows the u axis, then l'_x is multiplied by some value and added to texture coordinate u, and l'_y is multiplied by the same value and added to v.[10]

The value used to multiply l'_x and l'_y varies depending on the height field's characteristics and the effect desired. A good start is to resize the vector l' to be the width of a texel. That is, normalize l' and divide by r, where r is the bump texture's resolution [245].

[10]In practice, the vector n does not affect the calculations, and so it does not have to be placed in the basis matrix 5.13. All that is actually needed is the upper left 2 × 3 matrix.

When the light is nearly directly over the surface, it is often worthwhile to make the bumps have less of an effect. This is done by shifting less when the light is overhead, which occurs when the length of l' is small. To normalize l' we normally divide it by its length. If the length is found to be less than, say, $\frac{1}{16}$, then divide l' by $\frac{1}{16}$ (i.e., multiply it by 16) instead [160]. This softens the effect of a large change in the direction of l'_x and l'_y as a light passes overhead during animation.

A little more needs to be said about the blending operations in practice. The second pass's subtraction operation can bring the range of values from $[0, 1]$ to $[-1, 1]$. Since negative values cannot be stored by an unsigned byte, something else must be done. One solution is part of the following example.

EXAMPLE: EMBOSS BUMP MAPPING STAGES The goal is to bump-map a lit, diffuse, color-textured surface. One texture is used, and is set up so that the RGB channels have the color texture, and the alpha channel contains the bump height field. The alpha channel will not be used for transparency; it is just a place where the height field can be stored. In the first stage, the color texture is multiplied with the interpolated diffuse illumination computed per vertex. At the same time, the height-field data is placed in the alpha in the texture stage. Each texture stage can store and pass on RGB and alpha to the next stage.

In the second stage, the shifted texture coordinates are used. The RGB channels (containing the diffusely lit texture from the first stage) are left untouched, but the height-field alpha value is inverted (i.e., $a' = 1 - a$), then added signed to the original height-field value with a bias of -0.5. This puts the embossed effect into the alpha component of the second stage:

$$\begin{aligned} o &= d + (1 - s) - 0.5 \\ o &= d - s + 0.5 \end{aligned} \tag{5.14}$$

where d is the destination (the first-pass height-field value rendered) and s is the source (the new value being subtracted). This gives a height field value that in fact could span the range $[-0.5, 1.5]$, but the values are clamped to $[0, 1]$. For most height-field textures the values outside $[0, 1]$ are rare and unimportant.

Finally, the shading mode is set such that after these two texture stages are performed the final alpha result (the emboss effect) is used to multiply the color result. The emboss effect modulates the diffusely lit texture and produces bump mapping [255]. □

More about the emboss bump mapping method, as well as an extension to perform specular bump mapping, can be found in the Advanced OpenGL

course notes [245]. An image made with emboss bump mapping is shown in Figure 5.25.

There are other methods that can be used for Gouraud-shaded bump mapping. For example, one method is to perturb (u, v) environment-mapping coordinates by u and v differentials found in the bump texture. This gives the effect of wobbling the reflection vector, thereby wobbling the look of the reflective surface. A 2×2 matrix is also needed in order to rotate and scale the differentials before they are added [83]. This technique is a little tricky to control and can be expensive to implement in hardware.[11] Its advantage is that it requires only one additional pass and can support any number of lights, since it uses an environment map. Figure 5.25 shows an image made using this technique, as does Plate XII (following p. 194).

Hardware-supported bump mapping is an active area of research. Miller et al. [249] discuss ways of combining current hardware with table lookups and parameter caching to increase performance and functionality. Ernst et al. [96], Schilling et al. [315], Peercy et al. [285], and Ikedo & Ma [196] all propose new schemes for bump mapping and hardware support. Ernst et al. [97] give a summary of many different bump mapping techniques and propose their own Gouraud-based method. Heidrich et al. [173, 174] show how normal maps can be combined with spherical and paraboloid environment maps.

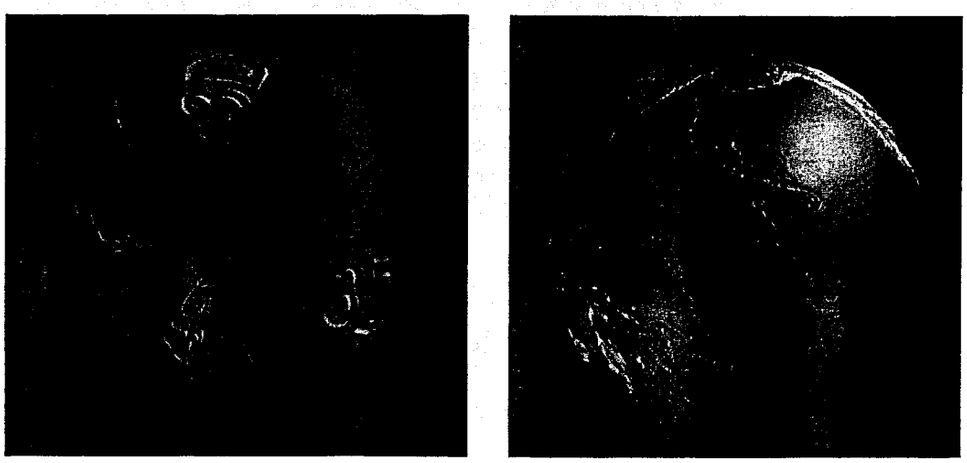

Figure 5.25. On the left is an example of bump mapping via embossing. On the right is an image created using environment-mapped bump mapping, along with a color texture. The illusion of bumpiness is created by using an environment map for the lighting and having the surface bump map vary how this environment map is accessed. *(Image on left courtesy of 3dfx Interactive, Inc. Image on right courtesy of Microsoft Inc.)*

[11] The Matrox G400 is the first chip we know of that supports this technique.

5.7.6 Other Texturing Techniques

This chapter has covered some of the uses of texturing. There are many others potentially useful in real-time work, including:

- Detail textures, which are used in, for example, flight simulators to add visual detail to highly magnified ground textures [245].
- Cutaway views, in which some layers of an object are peeled away [245].
- Antialiased lines and text rendered as rectangles with smooth edges by using alpha-blended transparency [149].
- Cylinder mapping, a form of environment mapping which reflects horizontal tube lights, useful in curved body visualization and anomaly detection [277].
- Anisotropic reflections (such as those off of brushed metal surfaces or compact discs) can be created by encoding tangent vectors into the texture coordinates and using a modified light map [172].
- Volume rendering, by rendering a set of image texture slices with alpha values in back-to-front order and blending these [149, 245]. See Section 6.8.
- Line integral convolution, a technique for visualizing vector fields [173].

Undoubtedly, there are more applications of textures that remain to be discovered.

Further Reading and Resources

Heckbert has written a good survey of texture mapping [163] and a more in-depth report on the topic [164]; both are available on the web, and the URLs appear in this book's bibliography. Wolberg's book [380] is another good work on image textures, particularly in the areas of sampling and filtering. Watt's books [367, 368, 369] and Rogers' *Procedural Elements* book [300] are general texts on computer graphics which have good coverage of texturing.

The SIGGRAPH OpenGL Advanced Techniques course notes [245] have extensive coverage of texturing algorithms and are available on the web. While RenderMan is meant for high-quality still images, the *RenderMan Companion* [357] has some good material on texture mapping and on procedural textures.

For extensive coverage of three-dimensional procedural textures, see *Texturing and Modeling: A Procedural Approach* [92].

Visit this book's website, `http://www.acm.org/tog/resources/RTR/`, for other resources.