texture coordinates themselves. This is an extremely powerful technique called *indirect texturing*. The first texture lookup forms a "table lookup," or "indirection," that generates a new texture coordinate for the second texture lookup.

Indirect texturing is an example of a more general case of texturing in which evaluating a texture sample generates a "value" other than a color. Clearly, not all texture lookups are used as colors. However, for ease of understanding in the following discussion, we will assume that the texture image's values represent the most common case—colors.

9.7 Evaluating the Fragment Shader

Armed with the current shader uniform values and interpolated per-vertex attributes at the fragment center, we are ready to compute the fragment's color by evaluating (or "running") the fragment shader. All of the source values are in place. Note, however, that we have not yet evaluated the texture lookups in the shader. Some of the earliest shading languages required that the textures be addressed only by per-vertex attributes, and in some cases, actually computed the texture lookups before even invoking the fragment shader. However, as discussed above, modern shaders allow for texture coordinates to be computed in the fragment shader itself, perhaps even as the result of a texture lookup. Also, conditionals and varying loop iterations in a shader may cause texture lookups to be skipped for some fragments. As a result, we will consider the rasterization of textures to be a part of the fragment shader itself.

In fact, while the mathematical computations that are done inside of the fragment shader are interesting, the most (mathematically) complex part of an isolated fragment shader evaluation is the computation of the texture lookups. The texture lookups are, as we shall see, far more than merely grabbing and returning the closest texel to the fragment center. The wide range of mappings of textures onto geometry and then geometry into fragments requires a much larger set of techniques to avoid glaring visual artifacts. The next section will describe in detail these complexities.

9.8 Rasterizing Textures

The previous section described how to interpolate per-vertex texture coordinate-attributes for use in a fragment shader, but this is only the first step in evaluating a texture lookup in a fragment shader. Having computed or interpolated the texture coordinate for a given fragment, the texture coordinate must be mapped into the texture image itself to produce a color.

9.8.1 Texture Coordinate Review

We will be using a number of different forms of coordinates throughout our discussion of rasterizing textures. This includes the application-level, normalized, texel-independent *texture coordinates* (u, v), as well as the texture size-dependent texel coordinates (u_{texel}, v_{texel}), both of which are considered real values. We used these coordinates in our introduction to texturing.

A final form of texture coordinate is the *integer texel coordinate*, or *texel address*. These represent direct indexing into the texture image array. Unlike the other two forms of coordinates, these are (as the name implies) integral values. The mapping from texel coordinates to integer texel coordinates is not universal and is dependent upon the texture filtering mode, which will be discussed below.

9.8.2 Mapping a Coordinate to a Texel

When rasterizing textures, we will find that — due to the nature of perspective projection, the shape of geometric objects, and the way texture coordinates are generated — fragments will rarely correspond directly and exactly to texels in a one-to-one mapping. Any rasterizer that supports texturing needs to handle a wide range of texel-to-fragment mappings. In the initial discussions of texturing in Chapter 7, we noted that texel coordinates generally include precision (via either floating-point or fixed-point numbers) that is much more fine-grained than the per-texel values that would seem to be required. As we shall see, in several cases we will use this so-called subtexel precision to improve the quality of rendered images in a process known as *texture filtering*.

Texture filtering (in its numerous forms) performs the mapping from real-valued texel coordinates to final texture image values or colors through a mixture of texel coordinate mapping and combinations of the values of the resulting texel or texels. We will break down our discussion of texture filtering into two major cases: one in which a single texel maps to an area that is the size of multiple fragments (magnification), and one in which a number of texels map into an area covered by a single fragment (minification), as they are handled quite differently.

Magnifying a Texture

Source Code Demo TextureFiltering

Our initial texturing discussion stated that one common method of mapping these subtexel precise coordinates to texture image colors was simply to select

the texel containing the fragment center point and use its color directly. This method, called *nearest-neighbor texturing*, is very simple to compute. For any (u_{texel}, v_{texel}) texel coordinate, the integer texel coordinate (u_{int}, v_{int}) is the nearest integer texel center, computed via rounding:

$$(u_{int}, v_{int}) = (\lfloor u_{texel} + 0.5 \rfloor, \lfloor v_{texel} + 0.5 \rfloor)$$

Having computed this integer texel coordinate, we simply use the *Image*() function to look up the value of the texel. The returned color is passed to the fragment shader for the current fragment. While this method is easy and fast to compute, it has a significant drawback when the texture is mapped in such a way that a single texel covers more than one pixel. In such a case, the texture is said to be "magnified," as a quadrilateral block of multiple fragments on the screen is entirely covered by a single texel in the texture, as can be seen in Figure 9.12.

FIGURE 9.12 Nearest-neighbor magnification.

With nearest-neighbor texturing, all (u_{texel}, v_{texel}) texel coordinates in the square

$$i_{int} - 0.5 \leq u_{texel} < i_{int} + 0.5$$

$$j_{int} - 0.5 \leq v_{texel} < j_{int} + 0.5$$

will map to the integer texel coordinates (i_{int}, j_{int}) and thus produce a constant fragment shader value. This is a square of height and width 1 in texel space, centered at the texel center. This results in obvious squares of constant color, which tends to draw attention to the fact that a low-resolution image has been mapped onto the surface. See Figure 9.12 for an example of a nearest-neighbor filtered texture used with a fragment shader that returns the texture as the final output color directly. In most cases, this blocky result is not the desired visual impression.

The problem lies with the fact that nearest-neighbor texturing represents the texture image as a piecewise constant function of (u, v). The resulting fragment shader attribute is constant across all fragments in a triangle until either u_{int} or v_{int} changes. Since the *floor* operation is discontinuous at integer values, this leads to sharp edges in the function represented by the texture over the surface of the triangle.

The common solution to the issue of discontinuous colors at texel boundaries is to treat the texture image values as specifying a different kind of function. Rather than creating a piecewise constant function from the discrete texture image values, we create a piecewise smooth color function. While there are many ways to create a smooth function from a set of discrete values, the most common method in rasterization hardware is linearly interpolating between the colors at each texel center in two dimensions. The method first computes the maximum integer texel coordinate (u_{int}, v_{int}) that is less than (u_{texel}, v_{texel}), the texel coordinate (i.e., the floor of the texel coordinates):

$$(u_{int}, v_{int}) = (\lfloor u_{texel} \rfloor, \lfloor v_{texel} \rfloor)$$

In other words, (u_{int}, v_{int}) defines the minimum (lower left in texture image space) corner of a square of four adjacent texels that "bound" the texel coordinate (Figure 9.13). Having found this square, we can also compute a *fractional texel coordinate* $0.0 \leq u_{frac}, v_{frac} < 1.0$ that defines the position of the texel coordinate within the 4-texel square (Figure 9.14).

$$(u_{frac}, v_{frac}) = (u_{texel} - u_{int}, v_{texel} - v_{int})$$

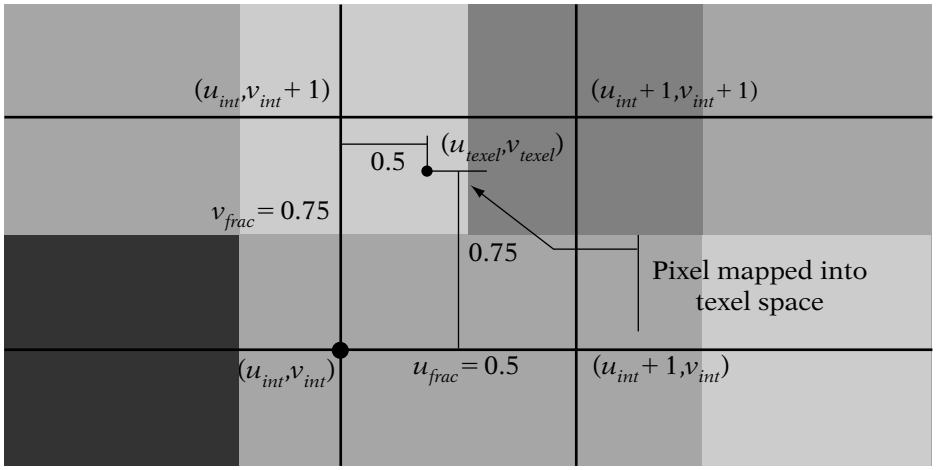


FIGURE 9.13 Finding the four texels that "bound" a pixel center and the fractional position of the pixel.

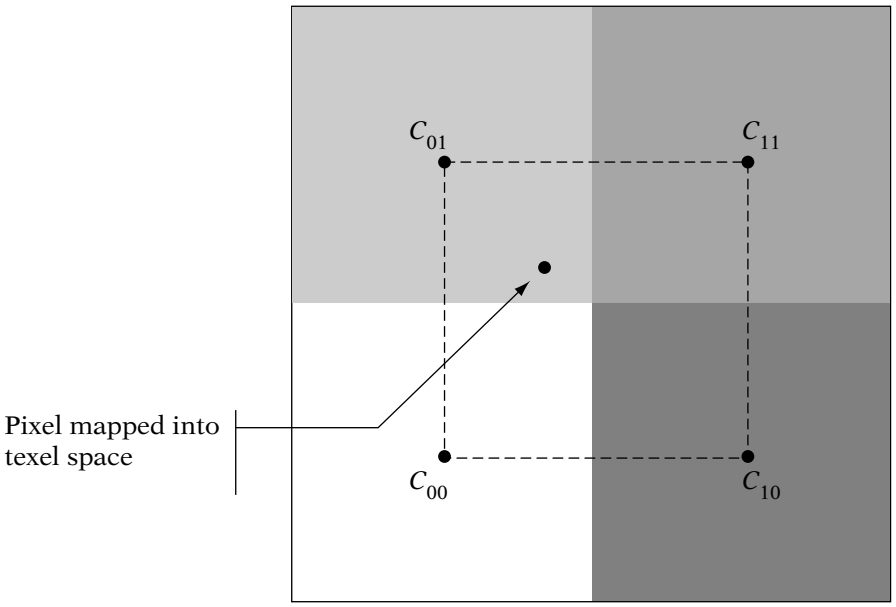


FIGURE 9.14 The four corners of the texel-space bounding square around the pixel center.

We use *Image*() to look up the texel colors at the four corners of the square. For ease of notation, we define the following shorthand for the color of the texture at each of the four corners of the square (Figure 9.14):

$$C_{00} = Image(u_{int}, v_{int})$$
$$C_{10} = Image(u_{int} + 1, v_{int})$$
$$C_{01} = Image(u_{int}, v_{int} + 1)$$
$$C_{11} = Image(u_{int} + 1, v_{int} + 1)$$

Then, we define a smooth interpolation of the four texels surrounding the texel coordinate. We define the smooth mapping in two stages, as shown in Figure 9.15. First, we linearly interpolate between the colors along the minimum-v edge of the square, based on the fractional u coordinate:

$$C_{MinV} = C_{00}(1 - u_{frac}) + C_{10}u_{frac}$$

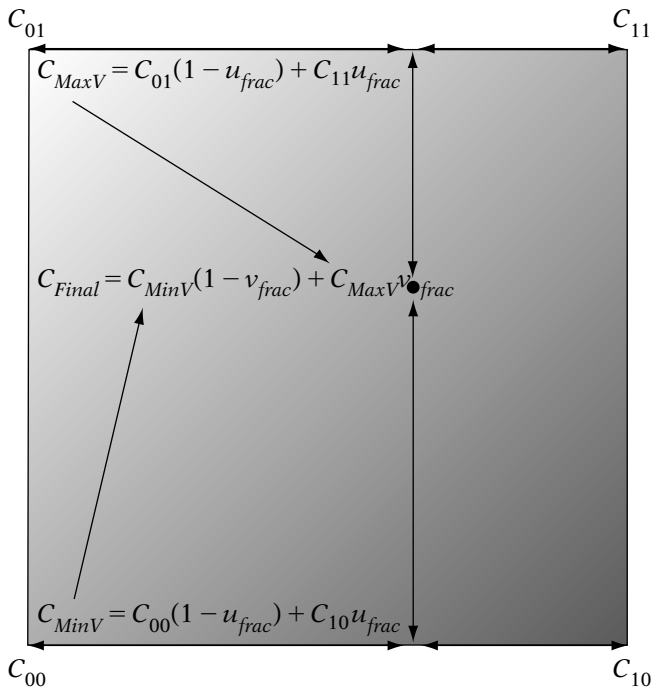


FIGURE 9.15 Bilinear filtering.

and similarly along the maximum-v edge:

$$C_{MaxV} = C_{01}(1 - u_{frac}) + C_{11}u_{frac}$$

Finally, we linearly interpolate between these two values using the fractional v coordinate:

$$C_{Final} = C_{MinV}(1 - v_{frac}) + C_{MaxV}v_{frac}$$

See Figure 9.15 for a graphical representation of these two steps. Substituting these into a single, direct formula, we get

$$\begin{aligned} C_{Final} = {} & C_{00}(1 - u_{frac})(1 - v_{frac}) + C_{10}u_{frac}(1 - v_{frac}) \\ & + C_{01}(1 - u_{frac})v_{frac} + C_{11}u_{frac}v_{frac} \end{aligned}$$

This is known as *bilinear texture filtering* because the interpolation involves linear interpolation in two dimensions to generate a smooth function from four neighboring texture image values. It is extremely popular in hardware 3D graphics systems. The fact that we interpolated along u first and then interpolated along v does not affect the result (other than by potential precision issues). A quick substitution shows that the results are the same either way. However, note that this is *not* an affine mapping. Affine mappings in 2D are uniquely defined by three distinct points. The fourth source point of our bilinear texture mapping may not fit the mapping defined by the other three points.

Using bilinear filtering, the colors across the entire texture domain are continuous. An example of the visual difference between nearest-neighbor and bilinear filtering is shown in Figure 9.16. While bilinear filtering can

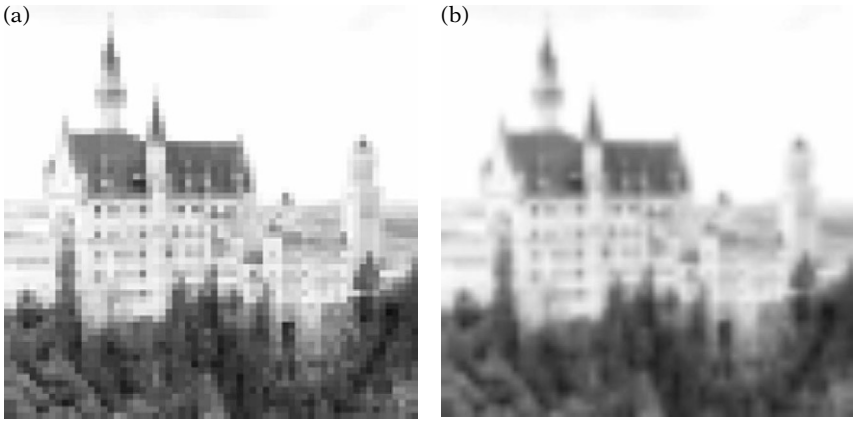


Figure 9.16 Extreme magnification of a texture (a) using nearest-neighbor filtering and (b) using bilinear filtering.

greatly improve the image quality of magnified textures by reducing the visual "blockiness," it will not add new detail to a texture. If a texture is magnified considerably (i.e., one texel maps to many pixels), the image will look blurry due to this lack of detail. The texture shown in Figure 9.16 is highly magnified, leading to obvious blockiness in the left image (a) and blurriness in the right image (b).

Texture Magnification in Practice

The `Iv` APIs use the `IvTexture` function `SetMagFiltering` to control texture magnification. `Iv` supports both bilinear filtering and nearest-neighbor selection. They are each set as follows:

```
IvTexture* texture;

// ...

{
   // Nearest-neighbor
   texture->SetMagFiltering(kNearestTexMagFilter);

   // Bilinear interpolation
   texture->SetMagFiltering(kBilerpTexMagFilter);

   // ...
```

Minifying a Texture

Throughout the course of our discussions of rasterization so far, we have mainly referred to fragments by their centers — infinitesimal points located at the center of a square fragment (continuing to assume only complete fragments for now). However, fragments have nonzero area. This difference between the area of a fragment and the point sample representing it becomes very obvious in a common case of texturing.

As an example, imagine an object that is distant from the camera. Objects in a scene are generally textured at high detail. This is done to avoid the blurriness (such as the blurriness we saw in Figure 9.16(b)) that can occur when an object that is close to the camera has a low-resolution texture applied to it. As that same object and texture is moved into the distance (a common situation in a dynamic scene), this same, detailed texture will be mapped to smaller and smaller regions of the screen due to perspective scaling of the object. This is known as *minification* of a texture, as it is the inverse of magnification. This results in the same object and texture covering fewer and fewer fragments.

In an extreme (but actually quite common) case, the entire high-detail texture could be mapped in such a way that it maps to only a few fragments. Figure 9.17 provides such an example; in this case, note that if the object moves even slightly (even less than a pixel), the exact texel covering the fragment's center point can change drastically. In fact, such a point sample is almost random in the texture and can lead to the point-sampled color of the texture used for the fragment changing wildly from frame to frame as the object moves in tiny, subpixel amounts on the screen. This can lead to flickering over time, a distracting artifact in an animated, rendered image.

The problem lies in the fact that most of the texels in the texture have an almost equal "claim" to the fragment, as all of them are projected within the rectangular area of the fragment. The overall color of the fragment's texture sample *should* represent all of the texels that fall inside of it. One way of thinking of this is to map the square of a complete fragment on the projection plane onto the plane of the triangle, giving a (possibly skewed) quadrilateral, as seen in Figure 9.18. In order to evaluate the color of the texture for that

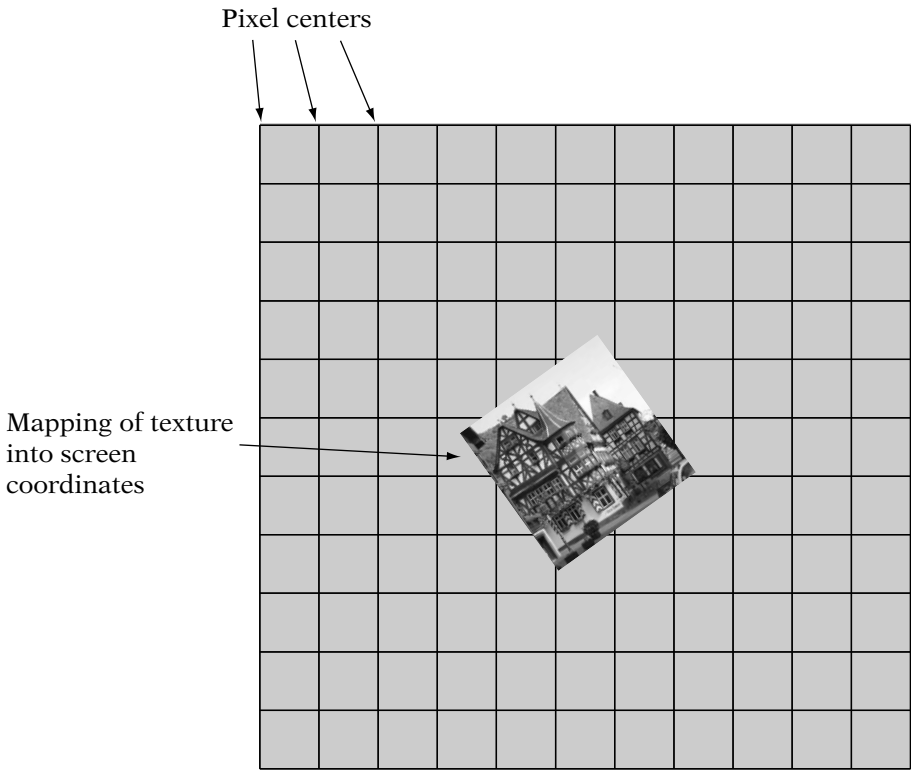


Figure 9.17 Extreme minification of a texture.

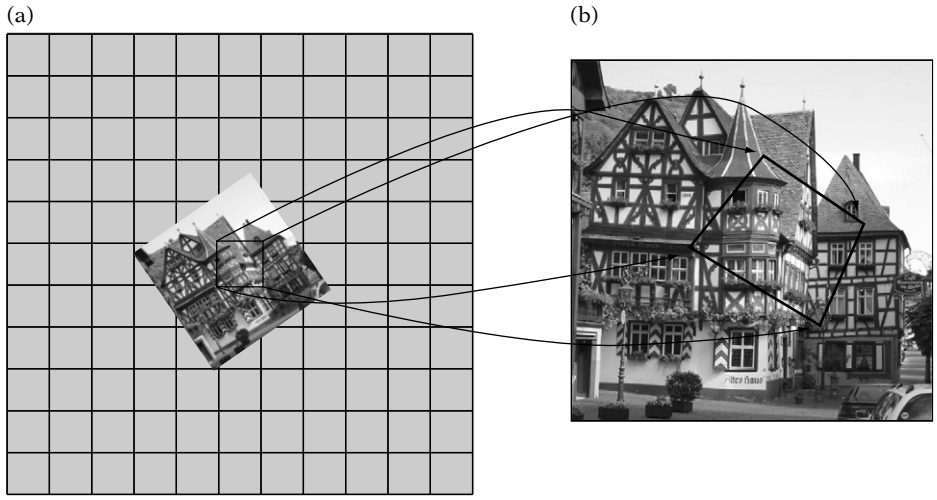


FIGURE 9.18 Mapping the square screen-space area of a pixel back into texel space: (a) screen space with pixel of interest highlighted and (b) texel-space back-projection of pixel area.

fragment fairly, we need to compute a weighted average of the colors of all of the texels in this quadrilateral, based on the relative area of the quadrilateral covered by each texel. The more of the fragment that is covered by a given texel, the greater the contribution of that texel's color to the final color of the fragment's texture sample.

While an exact area-weighted-average method would give a correct fragment color and would avoid the issues seen with point sampling, in reality this is not an algorithm that is best suited for real-time rasterization. Depending on how the texture is mapped, a fragment could cover an almost unbounded number of texels. Finding and summing these texels on a per-fragment basis would require a potentially unbounded amount of per-fragment computation, which is well beyond the means of even hardware rasterization systems. A faster (preferably constant-time) method of approximating this texel averaging algorithm is required. For most modern graphics systems, a method known as *mipmapping* satisfies these requirements.

9.8.3 MIPMAPPING

SOURCE CODE DEMO Mipmapping

Mipmapping [120] is a texture-filtering method that avoids the per-fragment expense of computing the average of a large number of texels. It does so by precomputing and storing additional information with each texture, requiring some additional memory over standard texturing. Mipmapping is a constant-time operation per texture sample and requires a fixed amount

of extra storage per texture (in fact, it increases the number of texels that must be stored by approximately one-third). Mipmapping is a popular filtering algorithm in both hardware and software rasterizers and is relatively simple conceptually.

To understand the basic concept behind mipmapping, imagine a 2×2–texel texture. If we look at a case where the entire texture is mapped to a single fragment, we could replace the 2×2 texture with a 1×1 texture (a single color). The appropriate color would be the mean of the four texels in the 2×2 texture. We could use this new texture directly. If we precompute the 1×1–texel texture at load-time of our application, we can simply choose between the two textures as needed (Figure 9.19).

When the given fragment maps in such a way that it only covers one of the four texels in the original 2×2–texel texture, we simply use a magnification method and the original 2×2 texture to determine the color. If the fragment covers the entire texture, we would use the 1×1 texture directly, again applying the magnification algorithm to it (although with a 1×1 texture, this is just the single texel color). The 1×1 texture adequately represents the overall color of the 2×2 texture in a single texel, but it does not include the detail of the

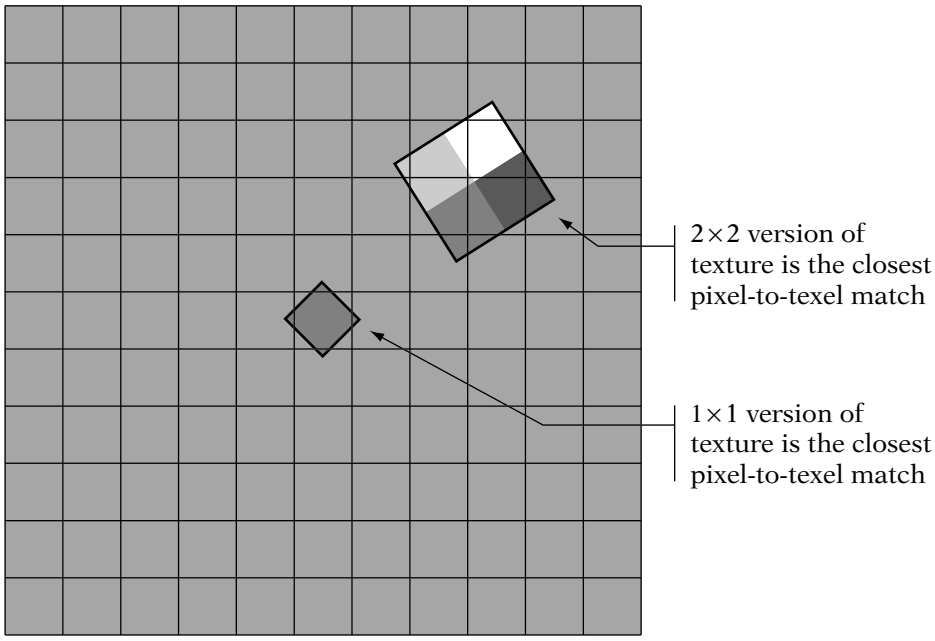


Figure 9.19 Choosing between two sizes of a texture.

original 2×2 texel texture. Each of these two versions of the texture has a useful feature that the other does not.

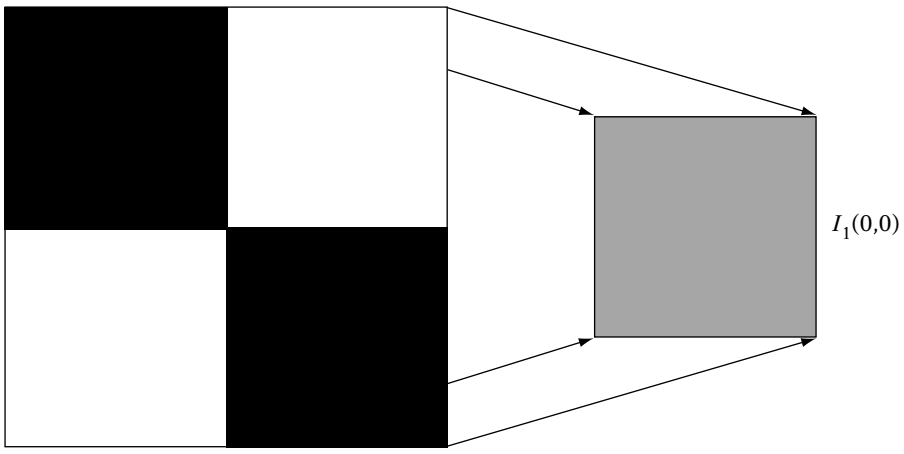
Mipmapping takes this method and generalizes it to any texture with power-of-two dimensions. For the purposes of this discussion, we assume that textures are square (the algorithm does not require this, as we shall see later in our discussion of mipmapping in practice). Mipmapping takes the initial texture image $Image_0$ (abbreviated I_0) of dimension $w_{texture} = h_{texture} = 2^L$ and generates a new version of the texture by averaging each square of four adjacent texels into a single texel. This generates a texture image $Image_1$ of size

$$\frac{1}{2}w_{texture} = \frac{1}{2}h_{texture} = 2^{L-1}$$

as follows:

$$Image_1(i, j) = \frac{I_0(2i, 2j) + I_0(2i+1, 2j) + I_0(2i, 2j+1) + I_0(2i+1, 2j+1)}{4}$$

where $0 \leq i, j < \frac{1}{2}w_{texture}$. Each of the texels in $Image_1$ represents the overall color of a block of the corresponding four texels in $Image_0$ (Figure 9.20).



$$I_1(0,0) = \frac{I_0(0,0) + I_0(1,0) + I_0(0,1) + I_0(1,1)}{4}$$

$$I_1(0,0) = \frac{(1,1,1) + (0,0,0) + (0,0,0) + (1,1,1)}{4} = (\tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{2})$$

FIGURE 9.20 Texel block to texel mapping between mipmap levels.

Note that if we use the same original texture coordinates for both versions of the texture, $Image_1$ simply appears as a blurry version of $Image_0$ (with half the detail of $Image_0$). If a block of about four adjacent texels in $Image_0$ covers a fragment, then we can simply use $Image_1$ when texturing. But what about more extreme cases of minification? The algorithm can be continued recursively. For each image $Image_i$ whose dimensions are greater than 1, we can define $Image_{i+1}$, whose dimensions are half of $Image_i$, and average texels of $Image_i$ into $Image_{i+1}$. This generates an entire set of $L + 1$ versions of the original texture, where the dimensions of $Image_i$ are equal to

$$\frac{w_{texture}}{2^i}$$

This forms a pyramid of images, each one-half the dimensions (and containing one-quarter the texels) of the previous image in the pyramid. Figure 9.21 provides an example of such a pyramid. We compute this pyramid for each texture in our scene once at load-time or as an offline preprocess and store each entire pyramid in memory. This simple method of computing the mipmap images is known as *box filtering* (as we are averaging a 2×2 "box" of texels into a single texel). Box filtering is not the sole method for generating the mipmap pyramid, nor is it the highest quality. Other, more complex methods are often used to filter each mipmap level down to the next lower level. These methods can avoid some of the visual issues that can crop up from the simple box filter. See Foley et al. [38] and Wohlberg [122] for details of other image-filtering methods.

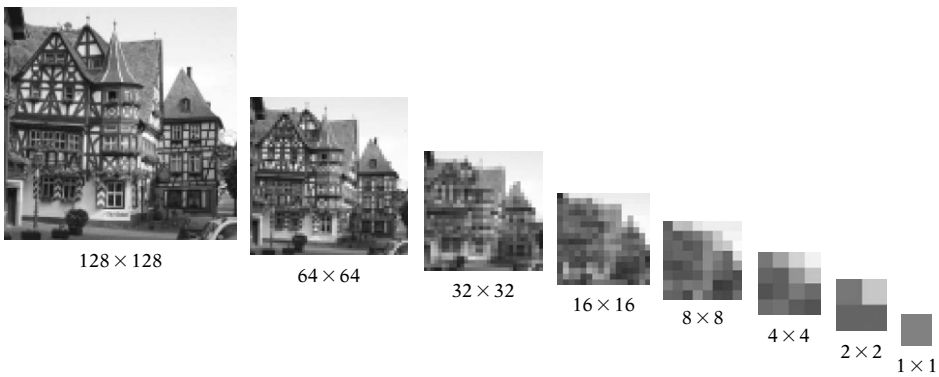


FIGURE 9.21 Mipmap level size progression.

Texturing a Fragment with a Mipmap

The most simple, general algorithm for texturing a fragment with a mipmap can be summarized as follows:

1. Determine the mapping of the fragment in screen space back into a quadrilateral in texture space by determining the texture coordinates at the corners of the fragment.
2. Having mapped the fragment square into a quadrilateral in texture space, select whichever mipmap level comes closest to exactly mapping the quadrilateral to a single texel.
3. Texture the fragment with the "best match" mipmap level selected in the previous step, using the desired magnification algorithm.

There are numerous common ways of determining the "best match" mipmap level, and there are numerous methods of filtering this mipmap level into a final fragment texture value. We would like to avoid having to explicitly map the fragment's corners back into texture space, as this is expensive to compute. We can take advantage of information that other rasterization stages already need. As a part of rasterization, it is common to compute the difference between the texel coordinates at a given fragment center and those of the fragment to the right and below the given fragment. Such differences are used to step the texture coordinates from one fragment to the adjacent fragment, one pixel away. These differences are written as derivatives. The listing that follows is designed to assign intuitive values to each of these four partial derivatives. For those unfamiliar with ∂, it is the symbol for a partial derivative, a basic concept of multivariable calculus. The ∂ operator represents how much one component of the output of a vector-valued function changes when you change one of the input components.

$$\frac{\partial u_{texel}}{\partial x_s} = \text{Change in } u_{texel} \text{ per horizontal pixel step}$$

$$\frac{\partial u_{texel}}{\partial y_s} = \text{Change in } u_{texel} \text{ per vertical pixel step}$$

$$\frac{\partial v_{texel}}{\partial x_s} = \text{Change in } v_{texel} \text{ per horizontal pixel step}$$

$$\frac{\partial v_{texel}}{\partial y_s} = \text{Change in } v_{texel} \text{ per vertical pixel step}$$

If a fragment maps to about one texel, then

$$\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial x_s}\right)^2 \approx 1, \text{ and } \left(\frac{\partial u_{texel}}{\partial y_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial y_s}\right)^2 \approx 1$$

In other words, even if the texture is rotated, if the fragment is about the same size as the texel mapped to it, then the overall change in texture coordinates over a single fragment has a length of about one texel. Note that all four of these differences are independent. These partials are dependent upon u_{texel} and v_{texel}, which are in turn dependent upon texture size. In fact, for each of these differentials, moving from $Image_i$ to $Image_{i+1}$ causes the differential to be halved. As we shall see, this is a useful property when computing mipmapping values.

A common formula that is used to turn these differentials into a metric of pixel-texel size ratio is described in Heckbert [55], which defines a formula for the *radius* of a pixel as mapped back into texture space. Note that this is actually the maximum of two radii, the radius of the pixel in u_{texel} and the radius in v_{texel}:

$$size = \max\left(\sqrt{\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial x_s}\right)^2}, \sqrt{\left(\frac{\partial u_{texel}}{\partial y_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial y_s}\right)^2}\right)$$

We can see (by substituting for the ∂) that this value is halved each time we move from $Image_i$ to $Image_{i+1}$ (as all of the ∂ values will halve). So, in order to find a mipmap level at which we map one texel to the complete fragment, we must compute the L such that

$$\frac{size}{2^L} \approx 1$$

where $size$ is computed using the texel coordinates for $Image_0$. Solving for L,

$$L = \log_2 size$$

This value of L is the mipmap level index we should use. Note that if we plug in partials that correspond to an exact one-to-one texture-to-screen mapping,

$$\frac{\partial u_{texel}}{\partial x_s} = 1, \frac{\partial v_{texel}}{\partial x_s} = 0, \frac{\partial u_{texel}}{\partial y_s} = 0, \frac{\partial v_{texel}}{\partial y_s} = 1$$

we get $size = 1$, which leads to $L = 0$, which corresponds to the original texture image as expected.

This gives us a closed-form method that can convert existing partials (used to interpolate the texture coordinates across a scan line) to a specific mipmap level L. The final formula is

$$
\begin{aligned}
L &= \log_2\left(\max\left(\sqrt{\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2+\left(\frac{\partial v_{texel}}{\partial x_s}\right)^2}, \sqrt{\left(\frac{\partial u_{texel}}{\partial y_s}\right)^2+\left(\frac{\partial v_{texel}}{\partial y_s}\right)^2}\right)\right) \\
&= \log_2\left(\sqrt{\max\left(\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2+\left(\frac{\partial v_{texel}}{\partial x_s}\right)^2, \left(\frac{\partial u_{texel}}{\partial y_s}\right)^2+\left(\frac{\partial v_{texel}}{\partial y_s}\right)^2\right)}\right) \\
&= \frac{1}{2}\log_2\left(\max\left(\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2+\left(\frac{\partial v_{texel}}{\partial x_s}\right)^2, \left(\frac{\partial u_{texel}}{\partial y_s}\right)^2+\left(\frac{\partial v_{texel}}{\partial y_s}\right)^2\right)\right)
\end{aligned}
$$

Note that the value of L is real, not integer (we will discuss the methods of mapping this value into a discrete mipmap pyramid later). The preceding function is only one possible option for computing the mipmap level L. Graphics systems use numerous simplifications and approximations of this value (which is itself an approximation) or even other functions to determine the correct mipmap level. In fact, the particular approximations of L used by some hardware devices are so distinct that some experienced users of 3D hardware can actually recognize a particular piece of display hardware by looking at rendered, mipmapped images. Other pieces of 3D hardware allow the developer (or even the end user) to bias the L values used, as some users prefer "crisp" images (biasing L in the negative direction, selecting a larger, more detailed mipmap level and more texels per fragment) while others prefer "smooth" images (biasing L in the positive direction, tending toward a less detailed mipmap level and fewer texels per fragment). For a detailed derivation of one case of mipmap level selection, see page 106 of Eberly [25].

Another method that has been used to lower the per-fragment expense of mipmapping is to select an L value and thus an single mipmap level per triangle in each frame and rasterize the entire triangle using that mipmap level. While this method does not require any per-fragment calculations of L, it can lead to serious visual artifacts, especially at the edges of triangles, where the mipmap level may change sharply. Software rasterizers that support mipmapping often use this method, known as *per-triangle mipmapping*.

Note that by its very nature, mipmapping tends to use smaller textures on distant objects. When used with software rasterizers, this means that mipmapping can actually *increase* performance, because the smaller mipmap levels are more likely to fit in the processor's cache than the full-detail texture. Most software rasterizers that support texturing are performance bound to some degree by the memory bandwidth of reading textures. Keeping a texture in the cache can decrease these bandwidth requirements significantly. Furthermore,

if point sampling is used with a nonmipmapped texture, adjacent pixels may require reading widely separated parts of the texture. These large per-pixel strides through a texture can result in horrible cache behavior and can impede the performance of nonmipmapped rasterizers severely. These cache miss stalls make the cost of computing mipmapping information (at least on a per-triangle basis) worthwhile, independent of the significant increase in visual quality. In fact, many hardware platforms also see performance increases when using mipmapping, owing to the small, on-chip texture cache memories used to hold recently accessed texture image regions.

Texture Filtering and Mipmaps

The methods described above work on the concept that there will be a single, "best" mipmap level for a given fragment. However, since each mipmap level is twice the size of the next mipmap level in each dimension, the "closest" mipmap level may not be an exact fragment-to-texel mapping. Rather than selecting a given mipmap level as the best, linear mipmap filtering uses a method similar to (bi)linear texture filtering. Basically, mipmap filtering uses the real-valued L to find the pair of adjacent mipmap levels that bound the given fragment-to-texel ratio, $\lfloor L \rfloor$ and $\lceil L \rceil$. The remaining fractional component $(L - \lfloor L \rfloor)$ is used to blend between texture colors found in the two mipmap levels.

Put together, there are now two independent filtering axes, each with two possible filtering modes, leading to four possible mipmap filtering modes as shown in Table 9.1. Of these methods, the most popular is *linear-bilinear*, which is also known as *trilinear interpolation* filtering, or *trilerp*, as it is the

TABLE 9.1 Mipmap filtering modes

Mipmap Filter	***Texture Filter***	***Result***
Nearest	Nearest	Select "best" mipmap level and then select closest texel from it
Nearest	Bilinear	Select "best" mipmap level and then interpolate four texels from it
Linear	Nearest	Select two "bounding" mipmap levels, select closest texel in each, and then interpolate between the two texels
Linear	Bilinear	Select two "bounding" mipmap levels, interpolate four texels from each, and then interpolate between the two results; also called trilerp

exact 3D analog to bilinear interpolation. It is the most expensive of these mipmap filtering operations, requiring the lookup of eight texels per fragment, as well as seven linear interpolations (three per each of the two mipmap levels, and one additional to interpolate between the levels), but it also produces the smoothest results. Filtering between mipmap levels also increases the amount of texture memory bandwidth used, as the two mipmap levels must be accessed per sample. Thus, multilevel mipmap filtering often counteracts the aforementioned performance benefits of mipmapping on hardware graphics devices.

A final, newer form of mipmap filtering is known as *anisotropic* filtering. The mipmap filtering methods discussed thus far implicitly assume that the pixel, when mapped into texture space, produces a quadrilateral that is fit quite closely by some circle. In other words, cases in which the quadrilateral in texture space is basically square. In practice, this is generally not the case. With polygons in extreme perspective, a complete fragment often maps to a very long, thin quadrilateral in texture space. The standard *isotropic* filtering modes can tend to look too blurry (having selected the mipmap level based on the long axis of the quad) or too sharp (having selected the mipmap level based on the short axis of the quad). Anisotropic texture filtering takes the aspect ratio of the texture-space quadrilateral into account when sampling the mipmap and is capable of filtering nonsquare regions in the mipmap to generate a result that accurately represents the tilted polygon's texturing.

Mipmapping in Practice

The individual levels of a mipmap pyramid may be specified manually in the `Iv` interfaces through the use of the `IvTexture` functions `BeginLoadData` and `EndLoadData`. These functions were briefly described in the introduction to texturing (Chapter 7). However, in the case of mipmaps, we use the argument to these functions, `unsigned int level` (previously defaulted to 0), which specifies the mipmap level. The mipmap level of the highest resolution image is 0. Each subsequent level number (1, 2, 3 . . .) represents the mipmap pyramid image with half the dimensions of the previous level. Some APIs (such as OpenGL) require that a "full" pyramid (all the way down to a 1×1 texel) be specified for mipmapping to work correctly. In practice, it is a good idea to provide a full pyramid for all mipmapped textures. The number of mipmap levels in a full pyramid is equal to

$$Levels = \log_2(max(w_{texture}, h_{texture})) + 1$$

Note that the number of mipmap levels is based on the larger dimension of the texture. Once a dimension falls to one texel, it stays at one texel while

TABLE 9.2 Mipmap level size progression

Level	*Width*	*Height*
0	32	8
1	16	4
2	8	2
3	4	1
4	2	1
5	1	1

the larger dimension continues to decrease. So, for a 32×8–texel texture, the mipmap levels are shown in Table 9.2.

Note that the texels of the mipmap level images set in the array returned by `BeginLoadData` must be computed by the application. `Iv` simply accepts these images as the mipmap levels and uses them directly. Once all of the mipmap levels for a texture are specified, the texture may be used for mipmapped rendering by attaching the texture sampler as a shader uniform. An example of specifying an entire pyramid follows.

```
IvTexture* texture;

// ...

{

   for (unsigned int level = 0;
        level < texture->GetLevels();
        level++) {
      unsigned int width = texture->GetWidth(level);
      unsigned int height = texture->GetHeight(level);
      IvTexColorRGBA* texels
         = (IvTexColorRGBA*)texture->BeginLoadData(level);

      for (unsigned int y = 0; y < height; y++) {
         for (unsigned int x = 0; x < width; x++) {
            IvTexColorRGBA& texel = texels[x + y * width];

            // Set the texel color, based on
            // filtering the previous level...
         }
      }
```

```
        texture->EndLoadData(level);
    }

    // ...
```

As a convenience, APIs such as `Iv` support automatic box filtering and creation of mipmap pyramids from a single image. In `Iv`, an application may provide the top-level image via the methods above and then automatically generate the remaining levels via the `IvTexture` function `GenerateMipmapPyramid`. The preceding code could be completely replaced with the following automatic mipmap generation.

```
IvTexture* texture;

// ...

{

    unsigned int width = texture->GetWidth();
    unsigned int height = texture->GetHeight();
    IvTexColorRGBA* texels
        = (IvTexColorRGBA*)texture->BeginLoadData();

    for (unsigned int y = 0; y < height; y++) {
        for (unsigned int x = 0; x < width; x++) {
            IvTexColorRGBA& texel = texels[x + y * width];

                // Set the texel color
        }
    }

    texture->EndLoadData(0);

    texture->GenerateMipmapPyramid();
    }

    // ...
```

In order to set the minification filter, the `IvTexture` function `SetMinFiltering` is used. `Iv` supports both nonmipmapped modes (bilinear filtering and nearest-neighbor selection), as well as all four mipmapped modes. The most

common mipmapped mode (as described previously) is trilinear filtering, which is set using

```
IvTexture* texture;

// ...

   texture->SetMinFiltering(kBilerpMipmapLerpTexMinFilter);

   // ...
```

9.9 From Fragments to Pixels

Thus far, this chapter has discussed generating fragments, computing the per-fragment source values for a fragment's shader, and some details of the more complex aspects of evaluating a fragment's shader (texture lookups). However, the first few sections of the chapter outlined the real goal of all of this per-fragment work: to generate the final color of a pixel in a rendered view of a scene. Recall that pixels are the destination values that make up the rectangular gridded screen (or framebuffer). The pixels are "bins" into which we place pieces of surface that impinge upon the area of that pixel. Fragments represent these pixel-sized pieces of surface. In the end, we must take all of the fragments that fall into a given pixel's bin and convert them into a single color and depth for that pixel. We have made two important simplifying assumptions in the chapter so far:

- All fragments are complete; that is, a fragment covers the entire pixel.
- All fragments are opaque; that is, near fragments obscure more distant ones.

Put together, these two assumptions lead to an important overall simplification: the nearest fragment at a given pixel completely determines the color of that pixel. In such a system, all we need do is find the nearest fragment at a pixel, shade that fragment, and write the result to the framebuffer. This was a useful simplifying assumption when discussing visible surface determination and texturing. However, it limits the ability to represent some common types of surface materials. It can also cause jagged visual artifacts at the edges of objects on the screen. As a result, two additional features in modern graphics systems have removed these simplifying assumptions: pixel blending allows fragments to be partially transparent, and antialiasing handles