rUN

time Mip-Map filtering

by Andrew Flavell

Illustration by Robert Zammarchi

Graphics programmers are constantly looking for ways to improve the realism of the graphics in games. One of the simplest techniques employed to do this is texture mapping, and while texture mapping does add considerable realism to a scene, it also adds a number of new problems. The most obvious visual problems that appear when using textures in a scene are the aliasing artifacts that are visible when texture-mapped polygons are some distance from the viewpoint. If you're moving rapidly around your virtual world, these artifacts appear as flashing or sparkling on the surface of the texture. Or, if the viewpoint is fixed, the artifacts appear as unwanted patterns within the texture after it has been mapped to a polygon. This is clearly visible in Figure 1, where the checkered texture map becomes distorted as its distance from the viewpoint increases.

Andrew Flavell is yet another out-of-work PhD grad, wondering why he spent all of those years at school studying graph-theory and Markov models. Questions regarding the article and job offers can be sent to mipmapping@weta3d.com

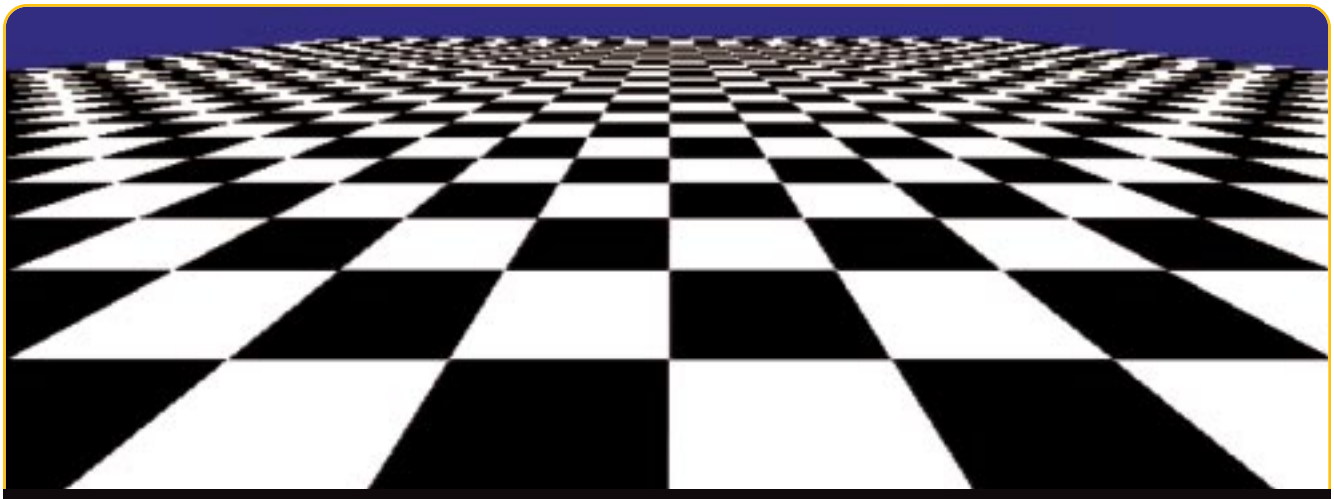


FIGURE 1. *Checkerboard with no MIP-mapping.*

MIP-mapping helps alleviate this problem. The acronym MIP comes from the Latin phrase *multum in parvo*, meaning "many things in a small place." Researchers at the New York Institute of Technology adopted this term in 1979 to describe a technique whereby many pixels are filtered and compressed into a small place, known as the MIP-map. To see how MIP-maps improve visual clarity, see Figure 2, in which MIP-mapping with bilinear filtering has been used to smooth the texture.

In order to understand what is what's causing the problems in the Figure 1, you have to look within the texture-mapping renderer and understand how the process of sampling the texture maps affects what's displayed on the screen. Look at Figure 3A, in which a sine wave is being sampled at a much higher frequency than the wave itself. As you can see, a fairly good representation of the wave can be obtained from these samples. However, if the sampling frequency drops to exactly two times the frequency of the wave, as shown if Figure 3B, then it's possible that the sampling points will coincide with the zero crossing points of the sine wave, resulting in no information recovery. Sampling frequencies of less than twice that of the sine wave being sampled, as shown in Figure 3C, causes the information within the samples to appear as a sine wave of lower frequency than the original. From these figures, we can guess that for complete recovery of a sampled signal, the sampling frequency must be at least twice that of the signal being sampled. This is known as the Nyquist limit. So, from where does the seemingly magic value of twice the signal being sampled come? In order to answer, that we'll have to digress a bit further and take a stroll into the Fourier domain.

A Stroll in the Fourier Domain

A complete discussion of Fourier theory could take up several books by itself, so for those of you who haven't suffered through a signal-processing course at college, I suggest that you take a look at the text by Bracewell that's mentioned at the end of this article. What follows is a very limited introduction to Fourier transforms and sampling, but it should be enough to demonstrate how the Nyquist limit is derived.

Figure 4A shows a plot of the function $h(t)=\text{sinc}^2 x$ and a plot of its Fourier transform, $H(f)$. It's convenient to think of $H(f)$ as being in the Fourier (or frequency) domain and of $h(t)$ as being in the time domain. (If you're wondering why I

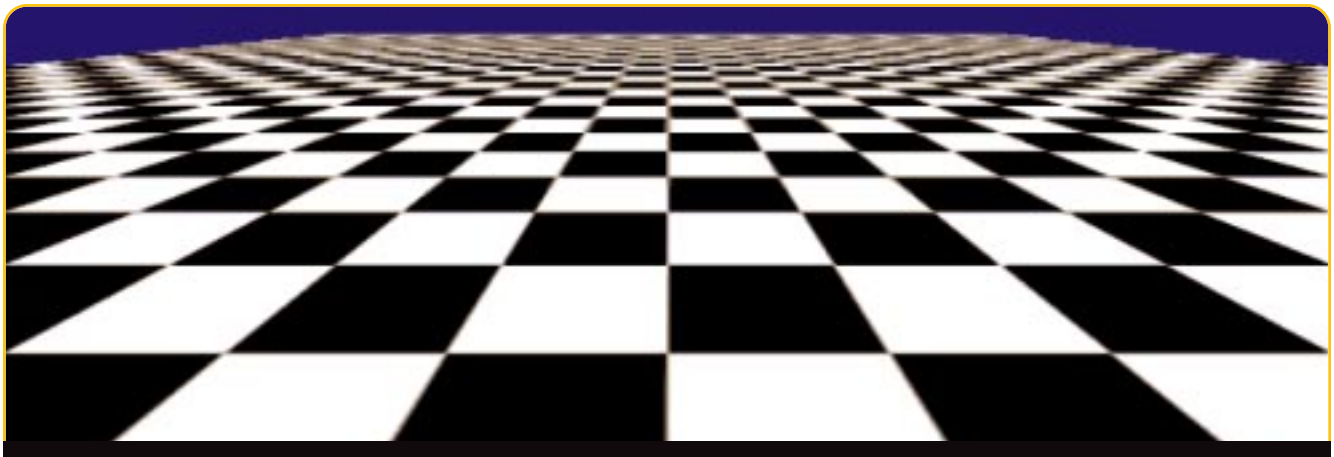


FIGURE 2. *Checkerboard with MIP-mapping and bilinear filtering.*

chose to use sinc^2x for this example, it's because it has a simple plot in the frequency domain.) To convert from the time domain to the frequency domain, the following transform is applied to $h(t)$:

$$H(f) = \int_{-\infty}^{\infty} h(t)e^{-i2\pi ft}dt$$

Eq. 1

In this form of the Fourier transform, f defines the frequencies of the sine waves making up the signal, and $i=\sqrt{-1}$ tells us that the exponential term is complex (that is, it has both real and imaginary parts). The operator $\supset$ is often used to denote "has the Fourier transform," so we can write $h(t) \supset H(f)$. Figure 4B shows the train of impulses used for sampling and the Fourier transform of the impulses. An impulse, denoted as $\delta(t)$, is a theoretical signal that is infinitely brief, infinitely powerful and has an area of one. An interesting property of an impulse train with a period of T_s is that its Fourier transform is an impulse train with a period of $1/T_s$.

$$\sum_{n=-\infty}^{\infty} \delta(t - nT_s) \supset \frac{1}{T_s}\sum \delta\left(\frac{f-n}{T_s}\right)$$

Eq. 2

The effect of sampling $h(t)$ with the sampling function $s(t)$ is shown in Figure 4C. In the time domain, the sampling can be thought of as multiplying $h(t)$ by $s(t)$, and in the frequency domain, it can be thought of as the convolution of $H(f)$ and $S(f)$.

$$h(t)s(t) \supset H(f) * S(f)$$

Eq. 3

Convolution of any two functions $f(x)$ and $g(x)$ is given by

$$f(x) * g(x) = \int_{-\infty}^{\infty} f(u)g(x-u)du$$

Eq. 4

If the thought of plugging the Fourier transforms of both $h(t)$ and $s(t)$ into Equation 4 has you wanting to skip to the current Soapbox article (p.72), just hold on a second — it isn't as bad as it looks. The convolution of a single impulse located at $t=t_0$, with $h(t)$ is just the value of $h(t)$ shifted to that location.

$$h(t) * \delta(t - t_0) = \int_{-\infty}^{\infty} h(u)\delta(t - t_0 - u)du = h(t - t_0)$$

Eq. 5

We can apply the result of Equation 5 to find the convolution of $H(f)$ and $S(f)$.

$$h(t)s(t) \supset \frac{1}{T_s}\sum H\left(\frac{f-n}{T_s}\right)$$

Eq. 6

Equation 6 simply means that the result of the convolution of $H(f)$ and $S(f)$ is such that $H(f)$ is duplicated at intervals of $1/T_s$, as can be seen in Figure 4C. The sinc^2x function is bandlimited (that is, its bandwidth is limited) to f_{max}, so the only requirement needed to ensure that there are no overlapping portions in the spectrum of the sampled signal is that $f_s > 2f_{max}$, where $f_s = 1/T_s$. So, this is from where the Nyquist limit comes. As you can see in Figure 4D, if the sampling frequency drops below $2f_{max}$, adjacent spectra overlap at higher frequencies, and these frequencies are then lost in the resulting signal. However, instead of disappearing completely, these high-frequency signals reappear at lower frequencies as aliases; this is where the term aliasing originated. To prevent aliasing from occurring, either the signal being sampled must be bandlimited to less than $2f_s$ or the sampling frequency must be set to be higher than $2f_{max}$.

MIP-Mapping Basics

Let's look at how MIP-mapping helps to reduce aliasing artifacts in our texture-mapped image. Remember that texture mapping is designed to increase the realism and detail in scenes. However, all of the fine details in the texture maps are effectively-high frequency components and they are the cause of our aliasing problems. Since we can't really modify our sampling frequency ($1/\Delta U$ and $1/\Delta V$ in the texture-mapping portion of our renderer), we have to filter the textures to remove the high-frequency details.

Although it would be possible to filter each individual texel at run time, this would require a significant amount of effort. To get around this problem, we can use MIP-maps, which are made up of a series of prefiltered and prescaled textures. The filtering of the textures either can be carried out during the startup of your game, or you can

FIGURE 3. *Sampling a sine wave with varying sampling intervals.*

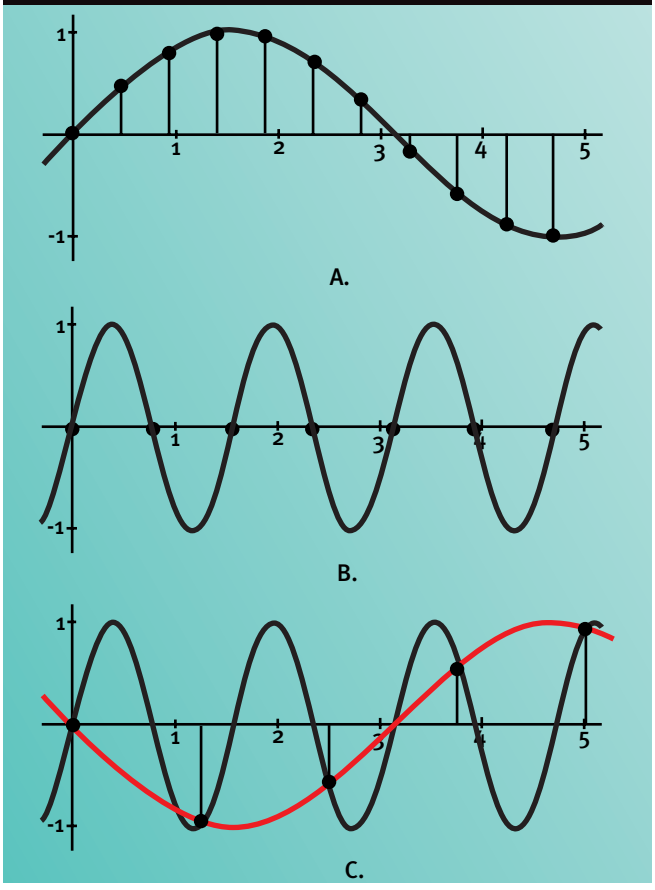
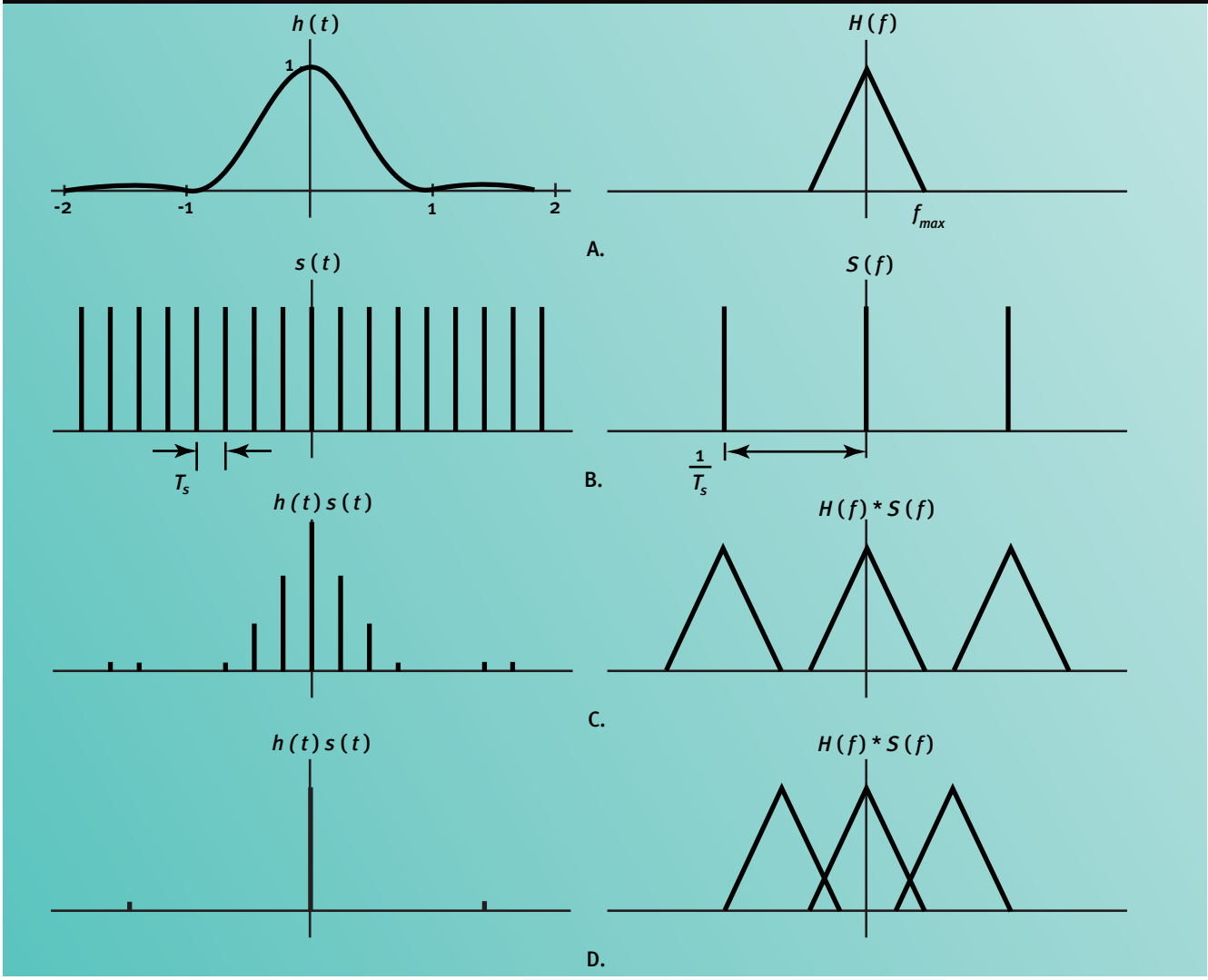


FIGURE 4. *Fourier analysis of sampling.*



prefilter all of your textures during development. Another alternative with some graphics cards, such as those using the Nvidia RIVA 128 accelerator, is to have the card automatically generate MIP-maps for you when textures are uploaded into video memory. Figure 5 illustrates the pyramid-like structure formed by the MIP-map for a 64×64 pixel texture. As you can see in the figure, the level of detail (LOD) decreases as the MIP-map level increases. Once textures have been filtered, all you have to do at run time to achieve basic per-polygon MIP-mapping is to select the correct MIP-map level (or LOD) for the desired texture and pass this to the renderer.

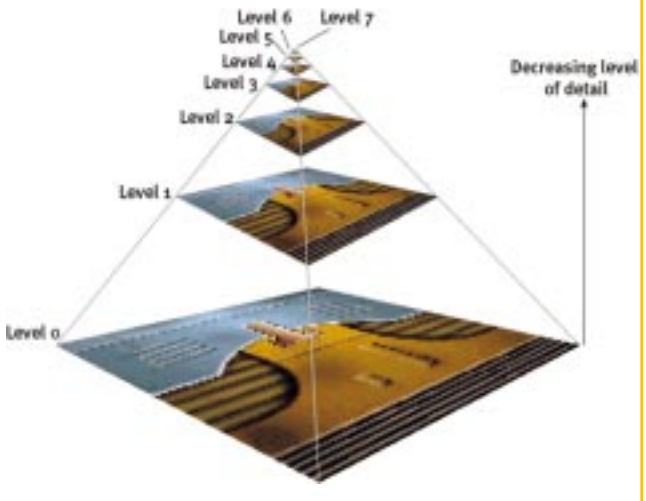
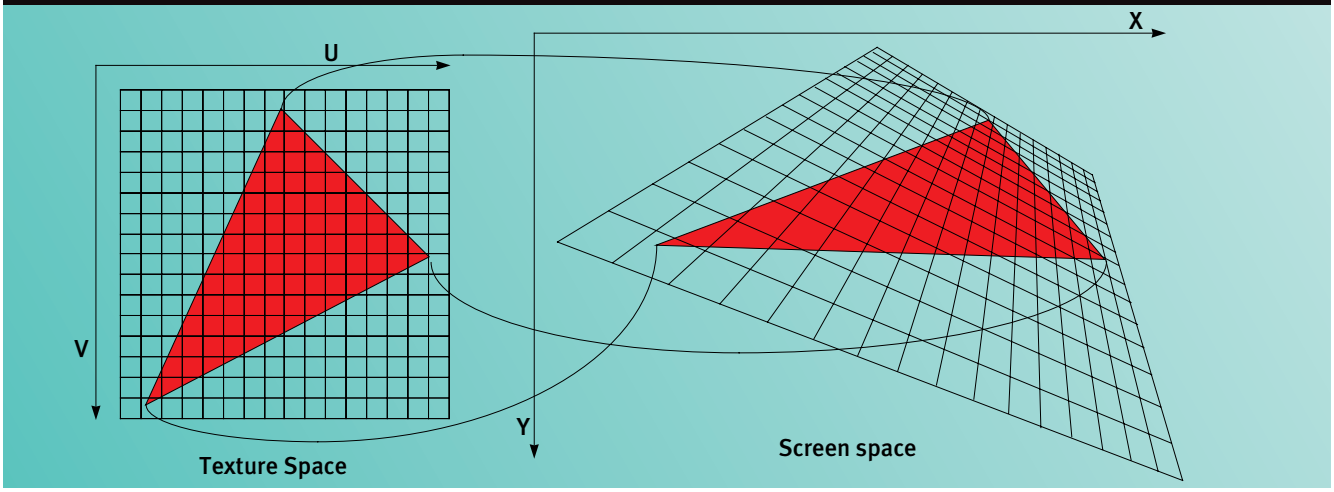


FIGURE 5. *A MIP-map pyramid.*

Generating MIP-Maps

There are a number of ways to generate MIP-maps. One option is simply to prebuild them using a graphics processing tool such as Photoshop. The alternative is to generate your MIP-maps on the fly. Prebuilding MIP-maps requires about 30 percent more storage space for your textures when you ship

FIGURE 6. *Texture distortion after perspective projection.*



your game, but it gives you finer control over their generation and it lets you add effects and details to different MIP levels. Regardless of which method you choose, MIP-maps require 30 percent more storage space at run-time than the original textures, so they can have a significant effect on your game's memory requirements.

Let's begin by generating a MIP-map for an 8-bit texture. Generating a MIP-map is a fairly simple process and although there are many possible filtering techniques that could be applied during MIP-map generation, a standard box filter usually suffices. The first level of the MIP-map is generated by taking the raw input texture and copying it directly into the MIP-map data structure shown in Listing 1. [In the interest of conserving editorial space, code listings are available for download from *Game Developer*'s web site. -Ed.]

Creating the rest of the MIP levels of a texture is an iterative process. Each successive level of the MIP-map is generated using the preceding, larger, MIP-map level. As each level of the MIP-map is created, it's stored consecutively in memory, and a pointer to the starting memory address of the MIP level is stored as well, so that the game engine can quickly access the correct LOD during rendering.

The first step in generating a new pixel value is to calculate the average color value from the four corresponding pixels in the preceding level, as shown in Listing 2. As there is a palette associated with the texture in this example, once the new color value has been calculated, we need to search the palette associated with this texture to find the entry that most closely matches the desired color. This process is shown in Listing 3. The color search process is quite simple, but it can be time-consuming, as we need to search the palette for a color match for every pixel in each level of the MIP-map. Thankfully, this step is only required during the initialization of the MIP-map, so it's not much of a problem. However, if you want to perform other effects during rendering (such as bilinear or trilinear filtering), the search process will be too slow.

In this case, we'll need to use 16- or 24-bit textures. Because most graphics cards currently support 16-bit screen depths, we'll use 16-bit textures here. The process of building MIP-maps for 16-bit textures is very similar to that used for 8-bit textures, as you can see in Listing 4. Because 16-bit textures don't require a palette, averaging the color values from the four corresponding pixels in the preceding level directly gives each new pixel value. One problem that can occur as a result of repeatedly averaging the color values for each LOD is that the texture map will become darker at each successive LOD. You can compensate for this effect by adding a small amount to each color component at each LOD, but this compensation usually isn't necessary, as the loss of color during the entire process is very small.

Applying MIP-maps at Run Time

Figure 6 shows some of the problems you can encounter when selecting which LOD to apply at run time. In the figure, the rectangular texture that's mapped onto the triangle in texture space is transformed into a quadrilateral in screen space, and the perspective projection of the texture causes the

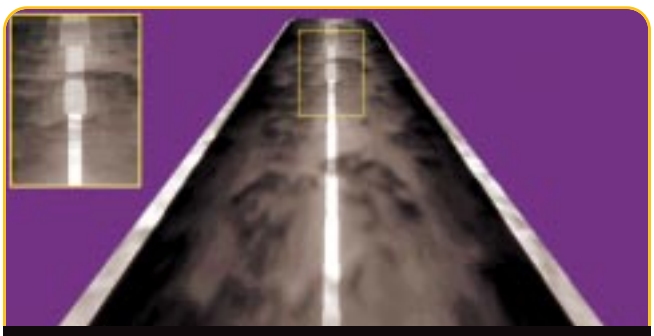


FIGURE 7. *Texture mapping with incorrect LOD.*

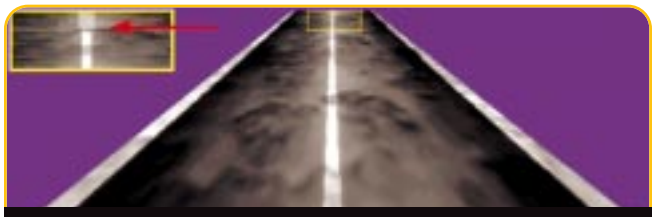


FIGURE 8. *Road rendered with per-polygon MIP-mapping.*

individual texels to become quadrilaterals of varying sizes. In a case such as this, where the orientation of a polygon is skewed in screen space, determining the best LOD to apply to a polygon is especially crucial if you want to produce good visual results. If the chosen LOD is too high (the texture dimensions are too large), aliasing will occur in the texture. If the LOD is too low (the dimensions of the texture are too small), then the image will appear blurred. For example, the LOD chosen for the texture in Figure 7 is much too low, as can be seen by the large texels visible in the inset zoomed image. Many different methods can be used for LOD selection, all of which have advantages and disadvantages. The two well-known methods that we'll examine here are the selection of the LOD based on the area of the texture in screen space, and the selection of the LOD using the projected *u* and *v* vectors.

One further point to consider here is that it's possible that a different number of texels map to each pixel in screen space. As a result, correct LOD selection requires calulating the LOD for each pixel. Calculating which LOD to use can be quite slow; consequently most software renderers (and quite a few older hardware accelerators) calculate the LOD on a per-polygon or per-triangle basis. An added advantage of per-polygon MIP-map selection, especially for software-based renderers, is that you can use smaller versions of textures for distant (smaller) polygons, helping to reduce the amount of processor cache that's required during texturing operations. However, per-pixel LOD selection lets you do a number of other things with MIP-mapping, including point sampling, bilinear filtering within a single LOD, or trilinear filtering between the two closest LODs.

Per-Polygon MIP-Mapping

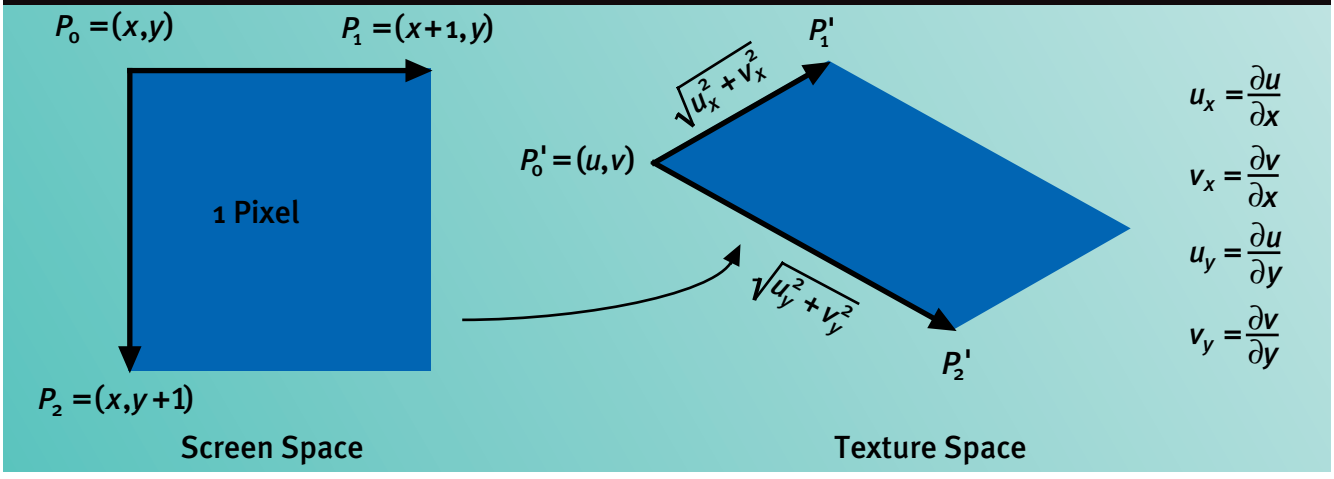
Per-polygon MIP-map selection is the least expensive method from a computational standpoint, becuse you only do MIP-map selection once per polygon. There are, however, a couple of drawbacks to this approach. One problem is that adjacent polygons that share a texture may be drawn using differing LODs; this will be appear as a discontinuity in the texture when displayed on the screen (this is called MIP-banding). Figure 8 shows a small amount of MIP-banding that is occurring due to the use of per-triangle MIP-mapping. Another problem is that visible popping may occur as a texture's LOD is changed due to movement of the viewpoint (or the polygon).

AREA-BASED LOD SELECTION. Area-based LOD selection complements per-polygon MIP-mapping techniques. In this method, you select the LOD by determining the amount of texture compression that will be applied to the texture in screen space. To determine the proper texture compression, you calculate the area of the polygon in screen space and the area, in texture space, of texture that is mapped onto the polygon. As shown in Listing 5, you can determine the ratio of texels to pixels and then determine which LOD to use. The *u* and *v* dimensions of each successive LOD are one-half the size of the preceding LOD, so each successive LOD has one-quarter the area of the preceding level. During LOD selection, we step up one level in the MIP-map pyramid for each multiple of four that the texel area is greater than the pixel area. For example, if the texel-to-pixel ratio is 3:1, we would select MIP-map level zero, or, if the texel-to-pixel ratio is 7:1, we would select MIP-map level one. Once the LOD has been selected, we can pass a pointer to the correct LOD, along with the LOD's dimensions, to our normal texture-mapping routines. One problem with any approach that uses the projected area of the polygon and the texture area as the basis for LOD selection is that aliasing will tend to occur whenever a projected polygon is very thin, due to the anisotropic nature of the texture compression (that is, the texture is compressed more in one dimension than the other).

Per-Pixel MIP-Mapping

Per-pixel MIP-mapping offers far better control of LOD selection than per-polygon MIP-mapping, and it also permits additional texture filtering — but at some additional cost. All of the per-pixel methods require storage of the entire MIP-mapped texture in memory, and adding LOD selection to the inner loop of a renderer's texture-mapping routine can significantly reduce rendering performance. Fortunately, most of today's 3D chips support per-pixel MIP-mapping with bilinear filtering (a few of the latest devices even support trilinear filtering), so we'll look at what it takes

FIGURE 9. *A single pixel back-projected into texture space.*



to implement sophisticated per-pixel MIP-mapping. Although we could use area-based LOD selection here also (we'd need to calculate the texture area underneath each pixel rather than for the entire polygon), we'll look at an all-together more accurate method.

EDGE COMPRESSION-BASED LOD SELECTION. In 1983, Paul Heckbert probably examined more LOD calculation techniques than he'd care to remember before he decided that techniques based on the compression that a texture suffers along the edge of a pixel seem to work best. Figure 9 shows a single pixel in screen space and the corresponding parallelogram in texture space. To prevent aliasing from occurring, we want to select the LOD based on the maximum compression of an edge in texture space. This corresponds to the maximum length of a side in texture space, which is given by Equation 7.

$$\max\left(\sqrt{u_x^2+v_x^2},\sqrt{u_y^2+v_y^2}\right)$$

Eq. 7

The values of u_x, u_y, v_x, and v_y are given by four partial derivatives. Because we already know how to calculate the u and v values for any pixel on the screen, we can use this knowledge to determine the partial derivatives. We know that, given the u/z, v/z, and $1/z$ gradients in x and y, and the starting u/z, v/z, and $1/z$ values at the screen origin, the u and v values for the texture at any pixel can be found using Equations 8 and 9. The notation in Equations 8 through 19 is derived from Chris Hecker's series on perspective texture mapping, which can be found on his web site (see "Acknowledgements" for the URL).

$$u=\frac{\mathrm{dUOverZdX}*x+\mathrm{dUOverZdY}*y+\mathrm{UOverZ_0}}{\mathrm{dOneOverZdX}*x+\mathrm{dOneOverZdY}*y+\mathrm{OneOverZ_0}}=\frac{\mathrm{UOverZ}}{\mathrm{Z}}$$

Eq. 8

$$v=\frac{\mathrm{dVOverZdX}*x+\mathrm{dVOverZdY}*y+\mathrm{VOverZ_0}}{\mathrm{dOneOverZdX}*x+\mathrm{dOneOverZdY}*y+\mathrm{OneOverZ_0}}=\frac{\mathrm{VOverZ}}{\mathrm{Z}}$$

Eq. 9

We can use these results to find the partial derivatives, as shown in Equations 10 through 13.

$$u_x=\frac{\mathrm{Z}*\mathrm{dUOverZdX}-\mathrm{UOverZ}*\mathrm{dOneOverZdX}}{\mathrm{Z}^2}=\frac{c+ay}{\mathrm{Z}^2}$$

Eq. 10

$$v_x=\frac{\mathrm{Z}*\mathrm{dVOverZdX}-\mathrm{VOverZ}*\mathrm{dOneOverZdX}}{\mathrm{Z}^2}=\frac{d+by}{\mathrm{Z}^2}$$

Eq. 11

$$u_y=\frac{\mathrm{Z}*\mathrm{dUOverZdY}-\mathrm{UOverZ}*\mathrm{dOneOverZdY}}{\mathrm{Z}^2}=\frac{e+ay}{\mathrm{Z}^2}$$

Eq. 12

$$v_y=\frac{\mathrm{Z}*\mathrm{dVOverZdY}-\mathrm{VOverZ}*\mathrm{dOneOverZdY}}{\mathrm{Z}^2}=\frac{f+by}{\mathrm{Z}^2}$$

Eq. 13

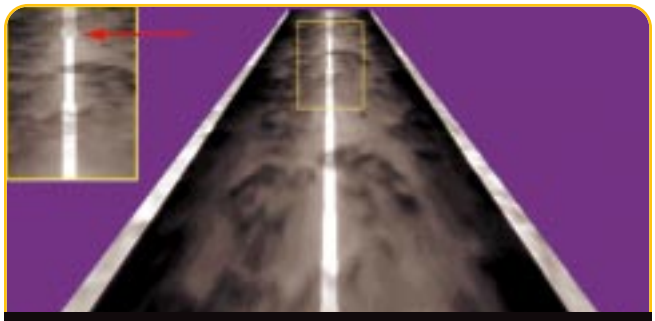


FIGURE 10. *Road rendered with per-pixel MIP-mapping and point sampling.*

Where a, b, c, d, e, and f are given by Equations 14 through 19.

$$a=\mathrm{dUOverZdX}*\mathrm{dOneOverZdY}-\mathrm{dOneOverZdX}*\mathrm{dUOverZdY}$$

Eq. 14

$$b=\mathrm{dVOverZdX}*\mathrm{dOneOverZdY}-\mathrm{dOneOverZdX}*\mathrm{dVOverZdY}$$

Eq. 15

$$c=\mathrm{dUOverZdX}*\mathrm{OneOverZ_0}-\mathrm{dOneOverZdX}*\mathrm{UOverZ_0}$$

Eq. 16

$$d=\mathrm{dVOverZdX}*\mathrm{OneOverZ_0}-\mathrm{dOneOverZdX}*\mathrm{VOverZ_0}$$

Eq. 17

$$e=\mathrm{dUOverZdY}*\mathrm{OneOverZ_0}-\mathrm{dOneOverZdY}*\mathrm{UOverZ_0}$$

Eq. 18

$$f=\mathrm{dVOverZdY}*\mathrm{OneOverZ_0}-\mathrm{dOneOverZdY}*\mathrm{VOverZ_0}$$

Eq. 19

An important point to note here is that the numerators of the partial derivatives u_x and v_x are functions of y only, and the numerators of the partial derivatives u_y and v_y are functions of x only. The values of a, b, c, d, e, and f are calculated once per polygon, along with the usual texture gradients, as shown in Listing 6. Finally, the formula for finding the maximum edge compression is given by Equation 20.

$$\mathrm{Compression}=\frac{1}{\mathrm{Z}^2}\max\left(\mathrm{x}_\ell,y_\ell\right)$$

where

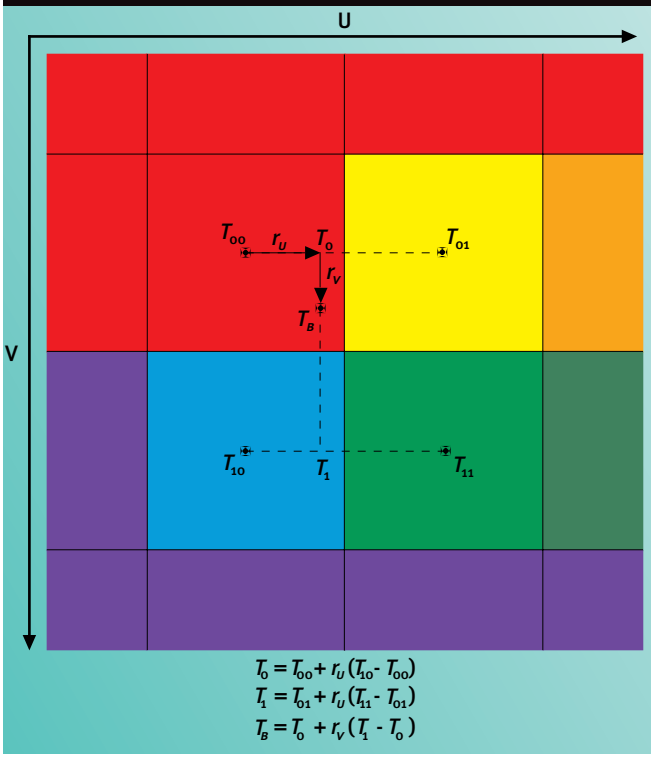
$$x_\ell=\left(\sqrt{(c=ay)^2+(d+by)^2}\right)$$

$$y_\ell=\left(\sqrt{(e=ax)^2(f-bx)^2}\right)$$

Eq. 20

At first glance, it would seem that we would need to carry out a square-root function at each pixel. However, if you look closely, you'll see that we only need to compute y_ℓ once for the polygon's range of x values. Furthermore, we only need to compute x_ℓ once per scan line. Listing 7 shows how we precompute the y_ℓ values for a polygon's range of x values during the normal set-up for texture mapping, and also that we only calculate x_ℓ once per scan line. We also don't have to worry

FIGURE 11. *Bilinear filtering calculation.*



about the divide required for the denominator, because it's already required for standard texture lookup. So the overhead for the compression calculation within the texture-mapping inner loop is just two multiplies and a compare. Now that we know how to calculate the edge compression, let's apply per-pixel LOD selection to our texture-mapping routines using point sampling, bilinear filtering, and trilinear filtering.

POINT-SAMPLED PER-PIXEL MIP-MAPPING. Point sampling is the simplest form of per-pixel MIP-mapping, and as you can see in Listing 8, there isn't much difference between our normal texture-mapping loop and one that uses point sampling. Once we've found the amount of edge compression for the current pixel, we need to determine the correct LOD. The raw compression value ranges from a zero to one, but we need to scale it by the texture dimensions to get a meaningful height in our MIP-map pyramid. Once we have the height, we determine the correct LOD by stepping up one level in the pyramid for each power of two that the height is greater than one. We then use our fast LOD lookup table to get a pointer to our texture and access the correct texel as usual. Figure 10 shows the same object that we used to generate Figure 8, but this time we're applying point-sampled MIP-mapping. As you can see in the figure, the main problem with point-sampled MIP-mapping is that MIP-banding is clearly visible at the points where transitions between different LODs occur. This is because adjacent pixels can have different LODs, so a discontinuity appears as we switch between LODs.

BILINEARLY FILTERED PER-PIXEL MIP-MAPPING. Bilinear filtering attempts to further reduce any aliasing errors present in a scene by averaging the values of the four pixels that are closest to the real *u* and *v* texture values for each pixel. As you can see in Figure 11 and Listing 9, bilinear interpolation can

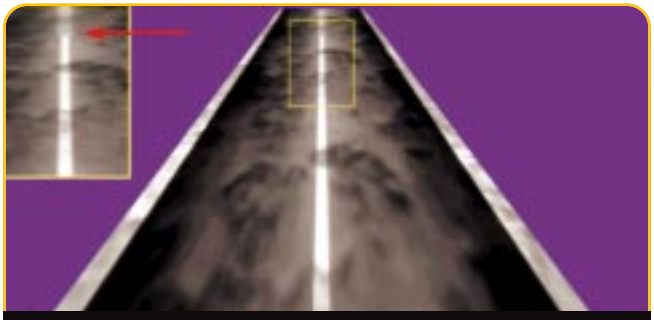


FIGURE 12. *Road rendered with per-pixel MIP-mapping and bilinear filtering.*

be implemented using three linear interpolations. We calculate the correct LOD and retrieve the pointer to our texture in exactly the same way that we did with point sampling. However, we then retrieve four texture values and apply bilinear interpolation to each color component to generate the new pixel value. Figure 12 shows our road after MIP-mapping and bilinear filtering. Although Figure 12 is an improvement over Figure 10, you can still make out the MIP-banding. Nothing has been done to remove the discontinuities that occur when we switch between LODs.

TRILINEARLY FILTERED PER-PIXEL MIP-MAPPING. The current state-of-the-art for 3D hardware-accelerated MIP-mapping is trilinear filtering. Trilinear filtering attempts to remove the problems associated with MIP-banding by smoothly blending between differing LODs. As you can see in Listing 10, we once again calculate the correct LOD in exactly the same way that we did it for point sampling, then retrieve pointers to the calculated LOD and the next lower LOD (the next level up in the pyramid). Trilinear interpolation is implemented using eight linear interpolations. We begin by carrying out bilinear interpolation separately for each of the selected LODs, then finish off by linearly interpolating between the two LODs. As you can seen in Figure 13, trilinear interpolation does result in a smooth transition between LODs (though the overall scene appears somewhat blurred). Unfortunately, this feature comes at a considerable cost: the straightforward implementation of trilinearly filtered MIP-mapping presented here requires eight texture accesses for each pixel and a considerable amount of computation. Although it's possible to cut down on the number of texture look-ups by saving texel values between loop

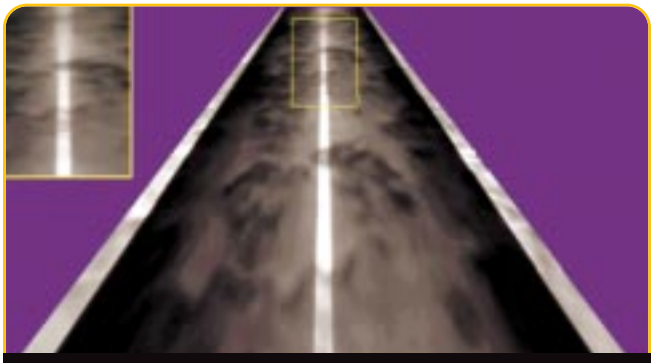


FIGURE 13. *Road rendered with per-pixel MIP-mapping and trilinear filtering.*

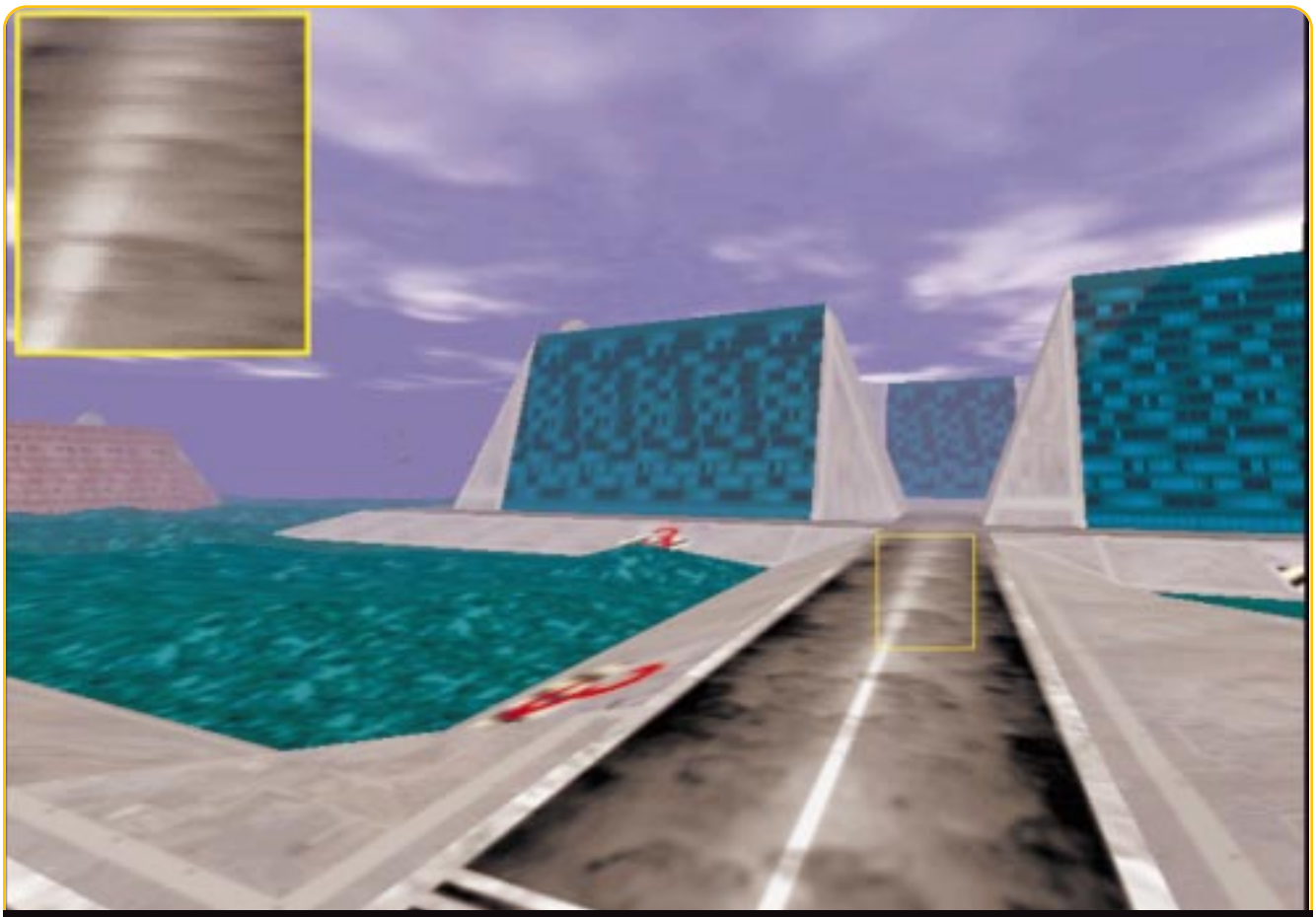


FIGURE 14. *Screen shot of CHAOSVR rendered using trilinear filtering on a Voodoo2.*

iterations, the interpolations themselves need to be performed for each loop, so achieving acceptable frame rates with software-based trilinear filtering is very difficult.

Closing Remarks

We've covered a lot of ground for one article, and although the output of our renderer using trilinear MIP-mapping is significantly better than plain old texture mapping, it still isn't perfect. The biggest defect remaining in our filtering is that, as I mentioned earlier, we've ignored the fact that the texture compression is anisotropic. We're selecting LODs using the maximum compression along one edge, but what if there's a significant difference in the amount of compression between each edge? In this case, the LOD selected will be too low for the least compressed edge, and our scene will appear blurred. You can clearly see this effect in Figure 14, which is a screen shot from the CHAOSVR demo that was rendered using a card based on 3Dfx's Voodoo2 chipset. This problem will occur with any 3D accelerator that uses methods similar to the ones that we've developed here for calculating the LOD — not just the Voodoo2 card that I'm using. Clearly, the next step to improve rendering accuracy will be to adopt some form of anisotropic filtering. I'm sure that it won't be long before this capability appears on high-end accelerators. ■

FOR FURTHER INFO

Bracewell, R. N., *The Fourier Transform and its applications*, McGraw-Hill Book Co., New York, 1986.

Williams, L., "Pyramidal Parametrics," Computer Graphics, vol. 17, no. 3, (Proc. SIGGRAPH 1983).

Heckbert, P., "Texture Mapping Polygons in Perspective," NYIT Tech. Memo No. 13, 1983

Acknowledgements

Thanks go out to Chris Hecker who kindly allowed me to plug my MIP-mapping into his texture mapping routines, saving me a lot of time. Check out Chris's home page, http://www.d6.com/users/checker, for more information on texture mapping and his old columns from *Game Developer*.

I'd also like to thank Paul Heckbert for taking the trouble to send me one of the first publications to ever discuss MIP-mapping. You can also find a lot of information about texture mapping and myriad other graphics techniques on Paul's home page http://www.cs.cmu.edu/afs/cs/user/ph/www/index.html. Finally, I'd like to thank Peter Laufenberg for allowing me to use a screen shot from Virtually Unlimited's CHAOSVR demo. You can find out more about the demo at http://www.virtually3d.com.