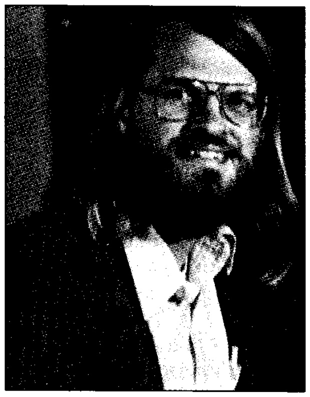
Jim Blinn's Corner

The Truth About Texture Mapping

James F. Blinn, Caltech



Texture mapping is a good, cheap way to make a picture look more realistic than it really is. Most of my texture mapping has been devoted to making pictures of the outer planets. Astrologers tell us that the planets control our destinies. That may not be true for everyone, but the unique configuration of the planet Uranus has had its effect on me. It influenced me to take a close look at how the layout of a texture map in memory affects the performance of the rendering algorithm.

A trip to the planets

My planet rendering program basically calculates, for each occupied pixel on the screen, the latitude and longitude visible at that pixel. It uses this latitude and longitude to index into a texture map to get a surface color. The actual texture color comes from bilinearly interpolating the texture colors at the four texture map pixels that surround that latitude and longitude. This, along with the shading calculations, gives the net color of the pixel.

I use texture maps that are 512 pixels wide (east to west) by 256 pixels tall (north to south). So to look up a value in the map the longitude is scaled to the range 0 to 512 and the latitude to the range 0 to 255. The integer parts of these numbers, call them I_u and I_v, give the coordinates of the upper left of the 2×2 pixel region that must be fetched for interpolation. The interpolation amount comes from the fractional parts of the scaled u and v values.

So how do we lay the map out in memory? The most obvious thing is to lay out the map just as you would for a 2D matrix: by rows. Given a 9-bit I_u value and an 8-

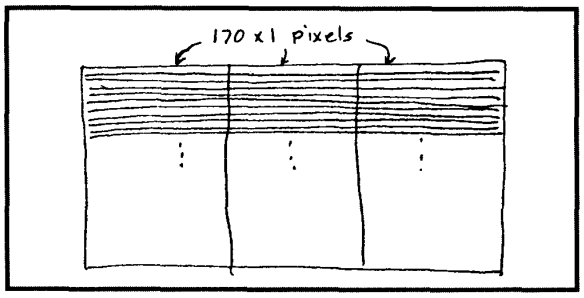


Figure 1. Row-ordered map.

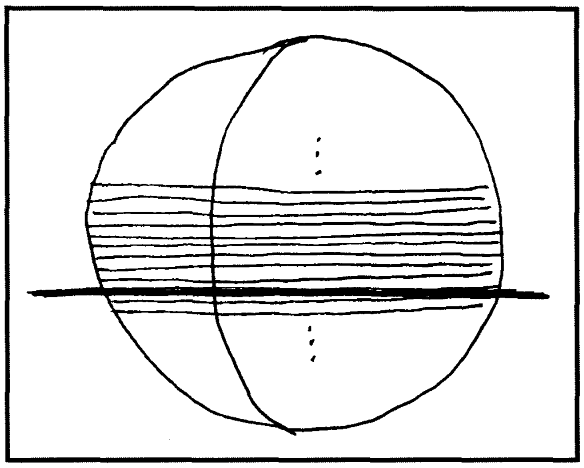


Figure 2. Geometry of row-ordered map, side view.

bit I_v value, the index of a map pixel can be found simply by concatenating the two values to give a 17-bit number

8 bits I_v	9 bits I_u

Originally each pixel of the map consisted of 1 byte that encoded a brightness and a saturation (white to orange). Nowadays I use 3-byte pixels for full color, so I must multiply this index by 3 to get the desired address in the texture map array.

Virtual memory

Now if you have gobs of memory you can read the whole map into RAM and all accesses go lickety-split. RAM means just that; access to one random location is just as fast as access to some other random location. (Of course, for images, RAM is not truly random. You can always access two pixels side by side in consecutive memory locations faster by some sort of block move or autoincrement addressing mode.)

But suppose you don't have enough memory for the whole pattern. Or suppose you have enough memory for one or two maps, but you want to have *lots* of maps applied to the scene at once. (You can usually depend on an art director to want more texture maps than is convenient.) What do you do? You use virtual memory.

With virtual memory you store the whole database on a disk file and read only parts of it into real memory as needed. The memory is divided, both on the disk and in RAM, into medium-size chunks called pages. You have fewer real RAM pages than disk pages. Each time a virtual memory location is accessed, some process looks at the high bits of the address (which select the page) and either translates the virtual memory location into the location in real memory where the data is stored or returns an indication that it is not there and must be read from disk. The latter event is called a page fault. When this occurs, one of the existing pages in real memory is recycled, usually the least recently used one, and the desired page of data is read into it. The tragedy is when you may later have to reread the page you threw out now. But it can't be helped.

On big computers and workstations this is all done with a combination of special CPU hardware and operating system code. On my small computer I have implemented a simple virtual memory simulator just for the texture memory.

With a virtual memory system (either hardware assisted or software simulated) you might think that you can just pretend you have zigabytes of real memory and everything will work out OK. Sorry, you lose. If you happen to access the memory in a truly random way, or even a particularly unfortunately chosen ordered way, you can give the virtual memory system fits. A classic example is the problem of zeroing out a very large matrix with two nested loops. If the matrix is laid out by rows and your loops are nested so that they access the memory by columns, you will likely get a page fault on every memory access. Bad idea.

The trick to minimizing the page fault rate is to keep your memory accesses within pages that are already in memory. You do this by keeping the memory accesses localized as much as possible. So a vital aspect of a texture map layout scheme is the addressing pattern it implies. Let's look at the memory access pattern for a specific example of texture mapping.

Recall that we are storing the map by rows. I use a page size of 512 bytes, so each row takes three pages. I store the three RGB values consecutively, so the first 170⅔ pixels are in the first page, etc. Figure 1 shows a schematic of the map with each rectangle standing for one memory page.

Let me digress for a moment and talk about this drawing. If I were to draw it completely to scale there should be 256 rectangles vertically by 3 horizontally. It would look like a solid black blob. So the diagram is merely suggestive of the actual map. Now if I were to use my computer-enhanced drawing mechanism to draw the schematic diagram with perfectly regular lines, I think it would give the wrong impression about its literal accuracy. A schematic diagram should look sketchy to help you realize that it is indeed sketchy and should not be taken literally. For this reason I have

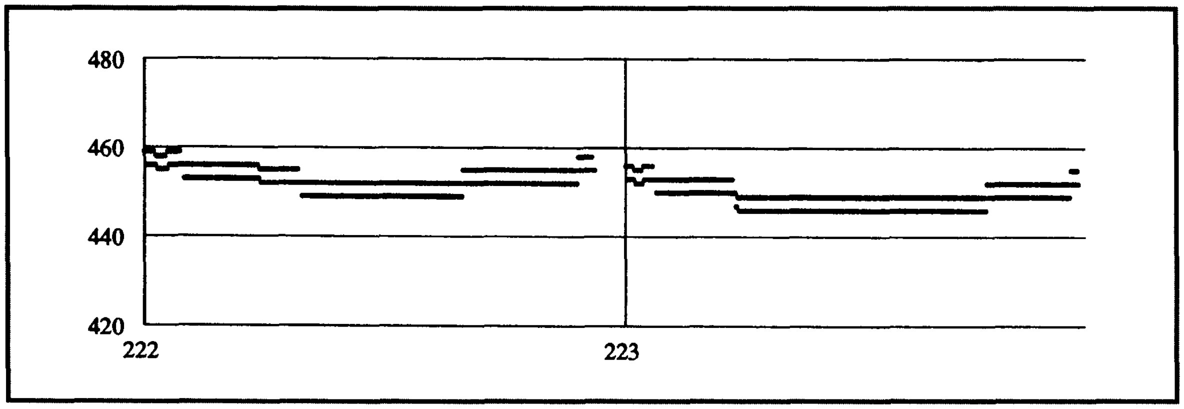


Figure 3. Addressing pattern: row-ordered map, side view.

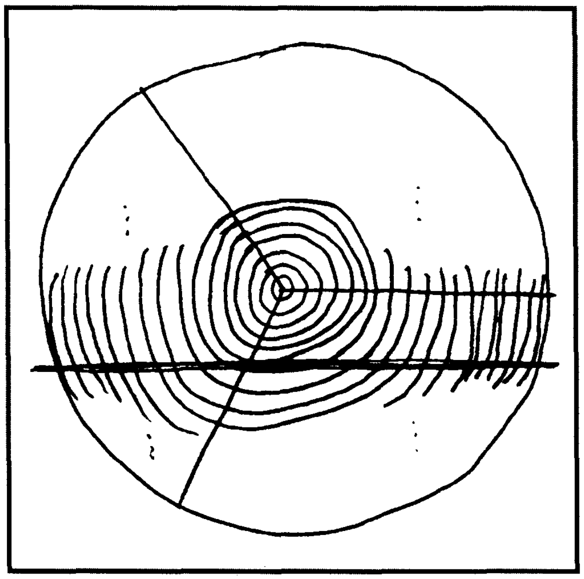


Figure 4. Geometry of row-ordered map, end view.

drawn it and all other such diagrams by hand using sketchy lines. I think this is a good idea in general for diagrams that are not meant to be geometrically precise. Take note, all you visualizers out there. End of digression.

Now, the first two planets I had occasion to draw were Jupiter and Saturn (and their moons). If we make a side view of the planet, as is usually the case with Jupiter and Saturn, the memory-page pattern of Figure 1 wraps around the sphere as sketched in Figure 2. The scan lines more or less follow the horizontal stripes of the texture map. To see exactly how well this worked, I tricked up my planet renderer to dump out the page numbers for each texture access. Figure 3 shows the addressing pattern for two consecutive representative scan lines at $y = 222$ and $y = 223$ (indicated by the horizontal line in Figure 2). The vertical scale shows memory-page numbers and the horizontal scale shows the time sequence of the accesses. Seven different pages are accessed on the first scan line and eight on the next, with five of them common to both lines.

For the whole image, there was a total of 16,000 texture map accesses, each requiring a 2×2 array of pixels. The total number of virtual memory accesses turns out to be 33,028. (It's not 64,000 because I have in my code a test to see whether two texture pixels at I_u and $I_u + 1$ are in consecutive memory locations, and make only one call to the virtual memory routines if so.) The 33,028 virtual memory accesses generated 446 page faults. The whole image took 24.8 seconds to render. My profiler tells me that only 3.9 seconds of it were spent in the disk I/O for the page faults. Not too shabby. (Profilers are my favorite programming toy. I can spend days fiddling with code after examining the results of a profile test. If you haven't played with one, you are missing out on one of the great joys of programming.)

A problem with Uranus

In 1980 and 1981 the two Voyager spacecraft went past Saturn, and it was time to start drawing pictures of the next planet, Uranus. Uranus, as it happens, spins with its axis pointing almost directly toward the sun. The views from the spacecraft, and for my animations, required looking down at the pole. Now one scan line of the image could cover all the latitudes from the equator to the pole. If each latitude of the texture map came from a different memory page, it required accessing half the map for each scan line! This is very reminiscent of the zero-the-matrix problem done the wrong way.

For our explicit example I took Figure 2 and rendered it with a 90-degree rotation of the planet. Figure 4 is a drawing of the sphere with texture pages sketched in. Figure 5 is a plot of the addressing pattern; note the change of vertical scale. Out of the 16,000 map

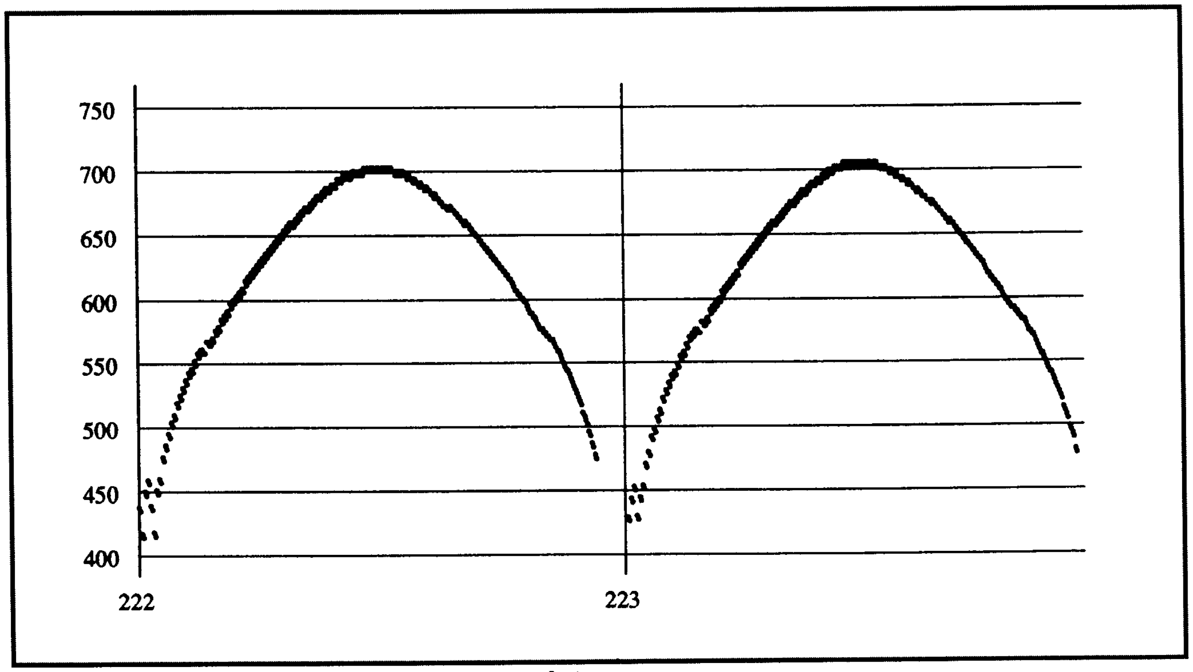


Figure 5. Addressing pattern: row-ordered map, end view.

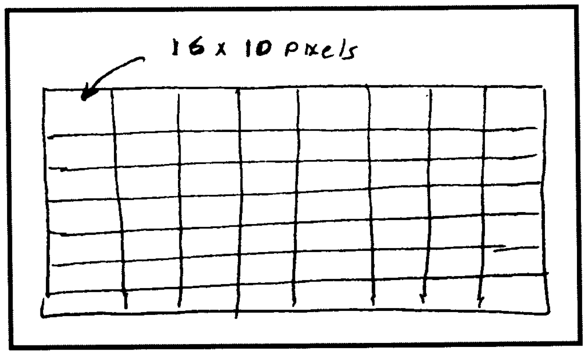


Figure 6. Tiled map.

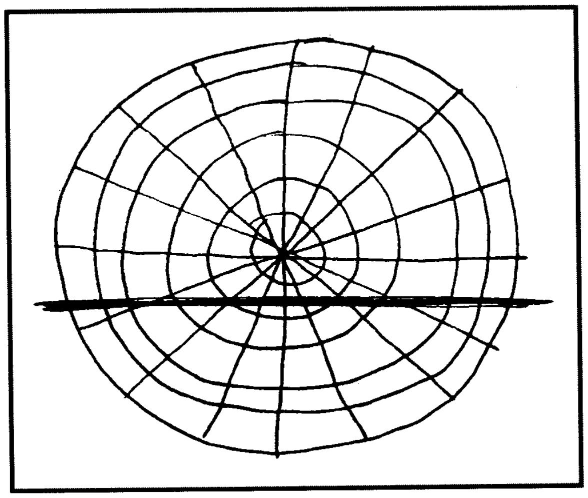


Figure 7. Geometry of tiled map, end view.

accesses and 32,992 virtual memory accesses, there were 15,207 page faults! Ouch! The total run time was 144.4 seconds, with fully 119.9 seconds devoted to paging. And that's using a RAM disk, so the actual I/O for disk accesses is negligible. (If I've got enough room for a RAM disk, you may ask, why am I using this virtual memory mechanism? The answer is flexibility. If I need more maps than will fit on the RAM disk, I can easily switch to getting the pages from a real disk.)

This might seem to be an extreme case, but a very similar thing happens when texture mapping any shape where the "grain" of the texture map is rotated 90 degrees to the direction of the inner loop of the rendering program.

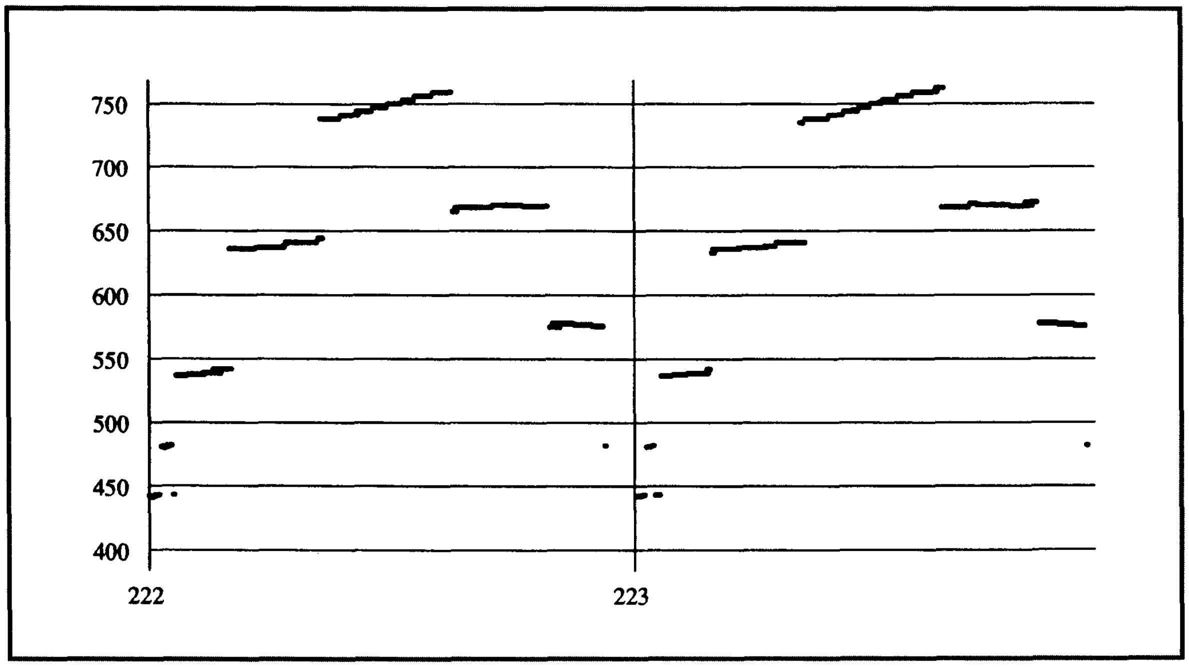


Figure 8. Addressing pattern: tiled map, end view.

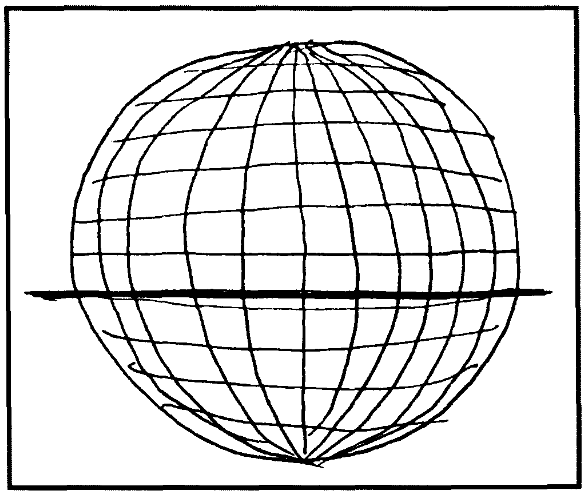


Figure 9. Geometry of tiled map, side view.

Tiles

We can give our poor disk some time off by using a better memory layout. With the by-rows layout two pixels that are vertically adjacent are *always* in different memory pages. Let us instead try to store geometrically close texture pixels close together in address space. Let's break the map into a series of 16 × 32-pixel tiles. (For the productions I have done to date I have used 32 × 32-pixel tiles, but while writing this column I realized that the above works better.) Each tile requires three memory pages (16 × 32 × 3 bytes = 3 × 512 bytes). Within a tile the pixels are stored by rows, so each memory page covers a squarish region 16 pixels wide by a little more than 10 pixels high. Finally, we store the tiles themselves by rows. The addressing of a pixel in the map is performed by interleaving the low bits of the I_u and I_v values with their high bits. The address is constructed as follows:

Hi 3 bits I_v	Hi 5 bits I_u	Lo 5 bits I_v	Lo 4 bits I_u

Multiply this by 3 to get the net memory index.

This arrangement gives the memory-page to map-region correspondence sketched in Figure 6. Figure 7 is a sketch of the planet with this wrapped around it. You can actually see that a lot fewer pages intersect a given scan line. Figure 8 is a plot of the addressing pattern. Tiling the map cut the total number of page faults down to 1,492. The total run time was 33.7 seconds (over four times faster!) with 11.8 seconds of it being paging.

In fact, tiling even improved things for the side view. This is because, with the row layout, lots of pixels of the map from the back half of the planet were read in unnecessarily, since they happened to be in the same memory page as pixels from the front of the planet.

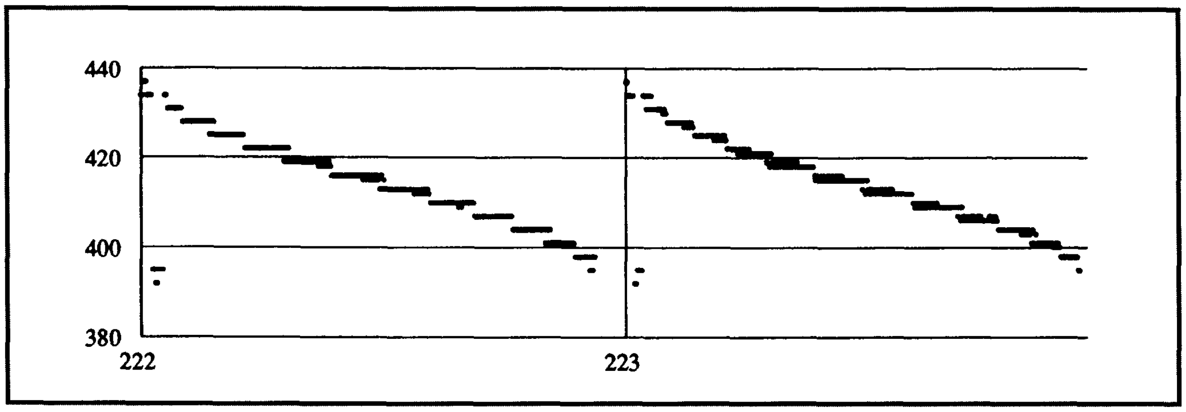


Figure 10. Addressing pattern: tiled map, side view.

With tiling we read in only pixels we have some chance of actually using. Figure 9 is a sketch of the geometry, and the addressing pattern is in Figure 10. The total number of page faults dropped to 356, and the total runtime was slightly improved—22.8 seconds, with 2.6 seconds of it being paging.

Address generation

The address-bit shuffling can itself take up a bunch of time. (Address calculation for a row-ordered map is admittedly much faster, but remember you may just be generating page faults faster.) I have speeded address calculation for tiled maps by using a table lookup. Set up a table for I_u whose entries contain

3 bits zero	Hi 5 bits I_u	5 bits zero	Lo 4 bits I_u

and a table for I_v whose entries contain

Hi 3 bits I_v	5 bits zero	Lo 5 bits I_v	4 bits zero

Since we are ultimately going to need to multiply by 3, we can build this into the table, too. If you are using tables you build as much arithmetic into the table as possible, so I also added a needed constant to the I_v table entries to skip automatically over the header in the texture file. When doing texture accessing you only need to add the table values to get the net address:

$$Address = UTable(I_u) + VTable(I_v)$$

Some analysis

There are two related things going on that improve matters when you lay out your pattern in tiles. First, you minimize the total number of pages that need to be accessed for a given scan line. The tile layout does this by effectively not reading in those parts of the pattern we won't be using for this scan line. For the end view and the two scan lines I monitored, this reduced the number of needed pages from about 158 to 33.

Secondly, the tile layout increases our chances that the set of pages we read in for one scan line will be almost the same set that we need for the next scan line. And if you have enough real memory to hold one scan line's worth of texture, you will be less likely to need to read a particlular page more than once. The advantages are substantial. The minimum number of page-ins possible happens when you read in each necessary page only once. (A necessary page is one that contains some visible bit of the texture.) Because of perspective, this will be a bit less than half of the whole map, a bit less than 384 for our test image. Compare this with 446 and 15,207 for the row-ordered map, side and end views. For the latter this meant that each page had to be reread about 40 times. No wonder it was so slow. Now compare with 356 and 1,492 for the tiled map, side and end views.

Summary

I've described this whole thing using very specific numbers for page sizes, RAM size, etc. But how does this work with hardware virtual memory environments? What if there are lots of RAM pages available, but you also have lots of simultaneous texture maps? I don't have the means to test this. If anyone out there has some statistics, let me know.

Tiling the map can hardly help but improve matters, but your mileage, as they say, may vary. ■