Chapter 69

Surface Caching and Quake's Triangle Models

Chapter 69

Probing Hardware-Assisted Surfaces and Fast Model Animation Without Sprites

In the late '70s, I spent a summer doing contract programming at a government-funded installation called the Northeast Solar Energy Center (NESEC). Those were heady times for solar energy, what with the oil shortages, and there was lots of money being thrown at places like NESEC, which was growing fast.

NESEC was across the street from MIT, which made for good access to resources. Unfortunately, it also meant that NESEC was in a severely parking-impaired part of the world, what with the student population and Boston's chronic parking shortage. The NESEC building did have its own parking lot, but it wasn't nearly big enough, because students parked in it at every opportunity. The lot was posted, and cars periodically got towed, but King Canute stood a better chance against the tide than NESEC did against the student hordes, and late arrivals to work often had to park blocks away and hike to work, to their considerable displeasure.

Back then, I drove an aging Volvo sedan that was sorely in need of a ring job. It ran fine but burned a quart of oil every 250 miles, so I carried a case of oil in the trunk, and checked the level frequently. One day, walking to the computer center a couple of blocks away, I cut through the parking lot and checked the oil in my car. It was low, so I topped it off, left the empty oil can next to the car so I would see it and remember to pick it up to throw out on my way back, and headed toward the computer center.

I'd gone only a few hundred feet when I heard footsteps and shouting behind me, and a wild-eyed man in a business suit came running up to me, screaming. "It's bad enough you park in our lot, but now you're leaving your garbage lying around!" he yelled. "Don't you people have any sense of decency?" I told him I worked at NESEC and was going to pick up the can on my way back, and he shouted, "Don't give me that!" I repeated my statements, calmly, and told him who I worked for and where my office was, and he said, "Don't give me that" again, but with a little less certainty. I kept adding detail until it was obvious that I was telling the truth, and he suddenly said, "Oh, my God," turned red, and started to apologize profusely. A few days later, we passed in the hallway, and he didn't look me in the eye.

The interesting point is that there was really no useful outcome that could have resulted from his outburst. Suppose I had been a student—what would he have accomplished by yelling at me? He let his emotions overrule his common sense, and as a result, did something he later wished he hadn't. I've seen many programmers do the same thing, especially when they're working long hours and not feeling adequately appreciated. For example, some time back I got mail from a programmer who complained bitterly that although he was critical to his company's success, management didn't appreciate his hard work and talent, and asked if I could help him find a better job. I suggested several ways that he might look for another job, but also asked if he had tried working his problems out with his employers; if he really was that valuable, what did he have to lose? He admitted he hadn't, and recently he wrote back and said that he had talked to his boss, and now he was getting paid a lot more money, was getting credit for his work, and was just flat-out happy.

We programmers think of ourselves as rational creatures, but most of us get angry at times, and when we do, like everyone else, we tend to be driven by our emotions instead of our minds. It's my experience that thinking rationally under those circumstances can be difficult, but produces better long-term results every time—so if you find yourself in that situation, stay cool and think your way through it, and odds are you'll be happier down the road.

Of course, most of the time programmers really *are* rational creatures, and the more information we have, the better. In that spirit, let's look at more of the stuff that makes Quake tick, starting with what I've recently learned about surface caching.

Surface Caching with Hardware Assistance

In Chapter 68, I discussed in detail the surface caching technique that Quake uses to do detailed, high-quality lighting without lots of polygons. Since writing that chapter, I've gone further, and spent a considerable amount of time working on the port of Quake to Rendition's Verite 3-D accelerator chip. So let me start off this chapter by discussing what I've learned about using surface caching in conjunction with hardware.

As you'll recall, the key to surface caching is that lighting information and polygon detail are stored separately, with lighting not tied to polygon vertices, then com-

bined on demand into what I call *surfaces*: lit, textured rectangles that are used as the input to the texture mapper. Building surfaces takes time, so performance is enhanced by caching the surfaces from one frame to the next. As I pointed out in Chapter 68, 3-D hardware accelerators are designed to optimize Gouraud shading, but surface caching can also work on hardware accelerators, with some significant quality advantages.

The surface-caching architecture of the Verite version of Quake (which we call VQuake) is essentially the same as in the software-only version of Quake: The CPU builds surfaces on demand, which are then downloaded to the accelerator's memory and cached there. There are a couple of key differences, however: the need to download surfaces, and the requirement that the surfaces be in 16-bit-per-pixel (bpp) format.

Downloading surfaces to the accelerator is a performance hit that doesn't exist in the software-only version. Although Verite uses DMA to download surfaces, DMA does in fact steal performance from the CPU. This cost is increased by the requirement for 16-bpp surfaces, because twice as much data must be downloaded. Worse still, it takes about twice as long to build 16-bpp surfaces as 8-bpp surfaces, so the cost of missing the surface cache is well over twice as expensive in VQuake as in Quake. Fortunately, there's 4 MB of memory on Verite-based adapters, so the surface cache doesn't miss very often and VQuake runs fine (and looks very good, thanks to bilinear texture filtering, which by itself is pretty much worth the cost of 3-D hardware), but it's nonetheless true that a completely straightforward port of the surface-caching model is not as appealing for hardware as for software. This is especially true at high resolutions, where the needs of the surface cache increase due to more detailed surfaces but available memory decreases due to frame buffer size.

Does my recent experience indicate that as the PC market moves to hardware, there's no choice but to move to Gouraud shading, despite the quality issues? Not at all. First of all, surface caching does still work well, just not as relatively well compared to Gouraud shading as is the case in software. Second, there are at least two alternatives that preserve the advantages of surface caching without many of the disadvantages noted above.

Letting the Graphics Card Build the Textures

One obvious solution is to have the accelerator card build the textures, rather than having the CPU build and then download them. This eliminates downloading completely, and lets the accelerator, which should be faster at such things, do the texel manipulation. Whether this is actually faster depends on whether the CPU or the accelerator is doing more of the work overall, but it eliminates download time, which is a big help. This approach retains the ability to composite other effects, such as splatters and dents, onto surfaces, but by the same token retains the high memory requirements and dynamic lighting performance impact of the surface cache. It also requires that the 3-D API and accelerator being used allow drawing into a texture,

which is not universally true. Neither do all APIs or accelerators allow applications enough control over the texture heap so that an efficient surface cache can be implemented, a point that favors non-caching approaches. (A similar option that wasn't open to us due to time limitations is downloading 8-bpp surfaces and having the accelerator expand them to 16-bpp surfaces as it stores them in texture memory. Better yet, some accelerators support 8-bpp palettized hardware textures that are expanded to 16-bpp on the fly during texturing.)

The Light Map as Alpha Texture

Another appealing non-caching approach is doing unlit texture-mapping in one pass, then lighting from the light map as a second pass, using the light map as an alpha texture. In other words, the textured polygon is drawn first, with no lighting, then the light map is textured on top of the polygon, with the light map intensity used as an alpha value to determine how brightly to light each texel. The hardware's texture-mapping circuitry is used for both passes, so the lighting comes out perspective-correct and consistent under all viewing conditions, just as with the surface cache. The lighting polygons don't even have to match the texture polygons, so they can represent dynamically changing lighting.

Two-pass lighting not only looks good, but has no memory footprint other than texture and light map storage, and provides level performance, because it's not dependent on surface cache hit rate. The primary downside to two-pass lighting is that it requires at least twice as much performance from the accelerator as single-pass drawing. The current crop of 3-D accelerators is not particularly fast, and few of them are up to the task of doing two passes at high resolution, although that will change soon. Another potential problem is that some accelerators don't implement true alpha blending. Nonetheless, as accelerators get better, I expect two-pass drawing (or three-or-more-pass, for adding splatters and the like by overlaying sprite polygons) to be widely used. I also expect Gouraud shading to be widely used; it's easy to use and fast. Also, speedier CPUs and accelerators will enable much more detailed geometry to be used, and the smaller that polygons become, the better Gouraud shading looks compared to surface caching and two-pass lighting.

The next graphics engine you'll see from id Software will be oriented heavily toward hardware accelerators, and at this point it's a tossup whether the engine will use surface caching, Gouraud shading, or two-pass lighting.

Drawing Triangle Models

Most of the last group of chapters in this book discuss how Quake works. If you look closely, though, you'll see that almost all of the information is about drawing the world—the static walls, floors, ceilings, and such. There are several reasons for this, in particular that it's hard to get a world renderer working well, and that the world is the base on which everything else is drawn. However, moving entities, such as monsters,

are essential to a useful game engine. Traditionally, these have been done with sprites, but when we set out to build Quake, we knew that it was time to move on to polygon-based models. (In the case of Quake, the models are composed of triangles.) We didn't know exactly how we were going to make the drawing of these models fast enough, though, and went through quite a bit of experimentation and learning in the process of doing so. For the rest of this chapter I'll discuss some interesting aspects of our triangle-model architecture, and present code for one useful approach for the rapid drawing of triangle models.

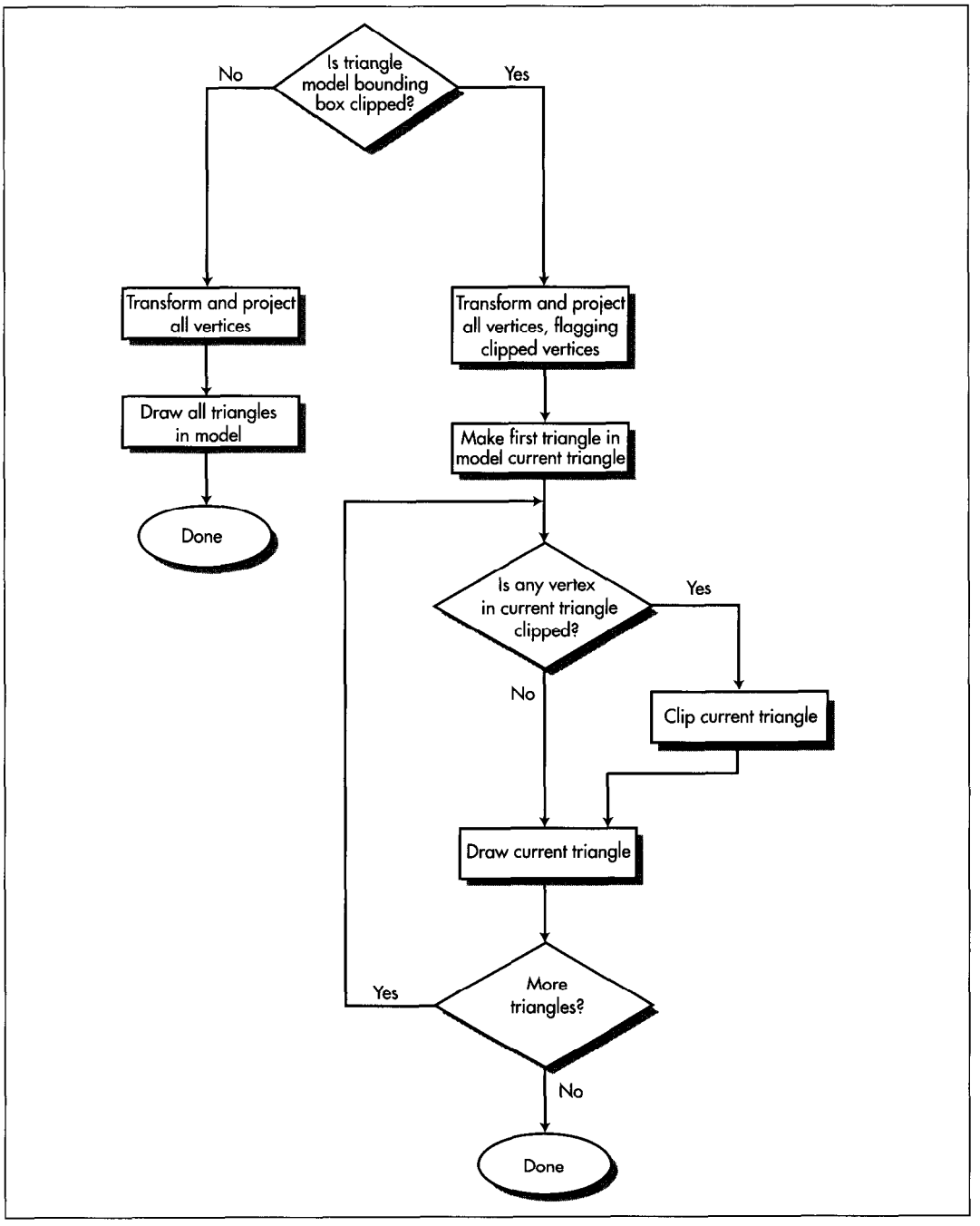
Drawing Triangle Models Fast

We would have liked one rendering model, and hence one graphics pipeline, for all drawing in Quake; this would have simplified the code and tools, and would have made it much easier to focus our optimization efforts. However, when we tried adding polygon models to Quake's global edge table, edge processing slowed down unacceptably. This isn't that surprising, because the edge table was designed to handle 200 to 300 large polygons, not the 2,000 to 3,000 tiny triangles that a dozen triangle models in a scene can add. Restructuring the edge list to use trees rather than linked lists would have helped with the larger data sets, but the basic problem is that the edge table requires a considerable amount of overhead per edge per scan line, and triangle models have too few pixels per edge to justify that overhead. Also, the much larger edge table generated by adding triangle models doesn't fit well in the CPU cache.

Consequently, we implemented a separate drawing pipeline for triangle models, as shown in Figure 69.1. Unlike the world pipeline, the triangle-model pipeline is in most respects a traditional one, with a few exceptions, noted below. The entire world is drawn first, and then the triangle models are drawn, using z-buffering for proper visibility. For each triangle model, all vertices are transformed and projected first, and then each triangle is drawn separately.

Triangle models are stored quite differently from the world itself. Each model consists of front and back skins stretched around a triangle mesh, and contains a full set of vertex coordinates for each animation frame, so animation is performed by simply using the correct set of coordinates for the desired frame. No interpolation, morphing, or other runtime vertex calculations are performed.

Early on, we decided to allow lower drawing quality for triangle models than for the world, in the interests of speed. For example, the triangles in the models are small, and usually distant—and generally part of a quickly moving monster that's trying its best to do you in—so the quality benefits of perspective texture mapping would add little value. Consequently, we chose to draw the triangles with affine texture mapping, avoiding the work required for perspective. Mind you, the models are perspective-correct at the vertices; it's just the pixels between the vertices that suffer slight warping.



Quake's triangle-model drawing pipeline.
Figure 69.1

Trading Subpixel Precision for Speed

Another sacrifice at the altar of performance was subpixel precision. Before each triangle is drawn, we snap its vertices to the nearest integer screen coordinates, rather than doing the extra calculations to handle fractional vertex coordinates. This causes some jumping of triangle edges, but again, is not a problem in normal gameplay, especially for the animation of figures in continuous motion.

One interesting benefit of integer coordinates is that they let us do backface culling and rejection of degenerate triangles in one operation, because the cross-product z component used for backface culling returns zero for degenerate triangles. Conveniently, that cross-product component is also the denominator for the lighting and texture gradient calculations used in drawing each triangle, so as soon as we check the cross-product z value and determine that the triangle is drawable, we immediately start the FDIV to calculate the reciprocal. By the time we get around to calculating the gradients, the FDIV has completed execution, effectively taking only the one cycle required to issue it, because the integer execution pipes can process independently while FDIV executes.

Finally, we decided to Gouraud-shade the triangle models, because this makes them look considerably more 3-D. However, we can't afford to calculate where all the relevant light sources for each model are in each frame, or even which is the primary light source. Instead, we select each model's lighting level based on how brightly the floor point it was standing on is lit, and use that lighting level for both ambient lighting (so all parts of the model have some illumination) and Gouraud shading—but the lighting vector for Gouraud shading is a fixed vector, so the model is always lit from the same direction. Somewhat surprisingly, in practice this looks considerably better than pure ambient lighting.

An Idea that Didn't Work

As we implemented triangle models, we tried several ideas that didn't work out. One that's notable because it seems so appealing is caching a model's image from one frame and reusing it in the next frame as a sprite. Our thinking was that clipping, transforming, projecting, and drawing a several-hundred-triangle model was going to be a lot more expensive than drawing a sprite, too expensive to allow very many models to be visible at once. We wanted to be able to display at least a dozen simultaneous models, so the idea was that for all but the closest models, we'd draw into a sprite, then reuse that sprite at the model's new locations for the next two or three frames, amortizing the 3-D drawing cost over several frames and boosting overall model-drawing performance. The rendering wouldn't be exactly right when the sprite was reused, because the view of the model would change from frame to frame as the viewer and model moved, but it didn't seem likely that that slight inaccuracy would be noticeable for any but the nearest and largest models.

As it turns out, though, we were wrong: The repeated frames were sometimes painfully visible, looking like jerky cardboard cutouts. In fact they looked a lot like the sprites used in DOOM—precisely the effect we were trying to avoid. This was especially true if we reused them more than once—and if we reused them only once, then we had to do one full 3-D rendering plus two sprite renderings every two frames, which wasn't much faster than simply doing two 3-D renderings.

The sprite architecture also introduced considerable code complexity, increased memory footprint because of the need to cache the sprites, and made it difficult to get hidden surfaces exactly right because sprites are unavoidably 2-D. The performance of drawing the sprites dropped sharply as models got closer, and that's also where the sprites looked worse when they were reused, limiting sprites to use at a considerable distance. All these problems could have been worked out reasonably well if necessary, but the sprite architecture just had the feeling of being fundamentally not the right approach, so we tried thinking along different lines.

An Idea that Did Work

John Carmack had the notion that it was just way too much effort per pixel to do all the work of scanning out the tiny triangles in distant models. After all, distant models are just indistinct blobs of pixels, suffering heavily from effects such as texture aliasing and pixel quantization, he reasoned, so it should work just as well if we could come up with another way of drawing blobs of approximately equal quality. The trick was to come up with such an alternative approach. We tossed around half-formed ideas like flood-filling the model's image within its silhouette, or encoding the model as a set of deltas, picking a visible seed point, and working around the visible side of the model according to the deltas. The first approach that seemed practical enough to try was drawing the pixel at each vertex replicated to form a 2×2 box, with all the vertices together forming the approximate shape of the model. Sometimes this worked quite well, but there were gaps where the triangles were large, and the quality was very erratic. However, it did point the way to something that in the end did the trick.

One morning I came in to the office to find that overnight (and well into the morning), John had designed and implemented a technique I'll call *subdivision rasterization.* This technique scans out approximately the right pixels for each triangle, with almost no overhead, as follows. First, all vertices in the model are drawn. Ideally, only the vertices on the visible side of the model would be drawn, but determining which vertices those are would take time, and the occasional error from a visible back vertex is lost in the noise.

Once the vertices are drawn, the triangles are processed one at a time. Each triangle that makes it through backface culling is then drawn with recursive subdivision. If any of the triangle's sides is more than one pixel long in either x or y—that is, if the triangle contains any pixels that aren't at vertices—then that side is split in half as nearly as possible at given integer coordinates, and a new vertex is created at the

split, with texture and screen coordinates that are halfway between those of the vertices at the endpoints. (The same splitting could be done for lighting, but we found that for small triangles—the sort that subdivision works well on—it was adequate to flat-shade each triangle at the light level of the first vertex, so we didn't bother with Gouraud shading.) The halfway values can be calculated very quickly with shifts. This vertex is drawn, and then each of the two resulting triangles is then processed recursively in the same way, as shown in Figure 69.2. There are some additional details, such as the fill rule that ensures that each pixel is drawn only once (except for backside vertices, as noted above), but basically subdivision rasterization boils down to taking a triangle, splitting a side that has at least one undrawn pixel and drawing the vertex at the split, and repeating the process for each of the two new triangles. The code to do this, shown in Listing 69.1, is very simple and easily optimized, especially by comparison with a generalized triangle rasterizer.

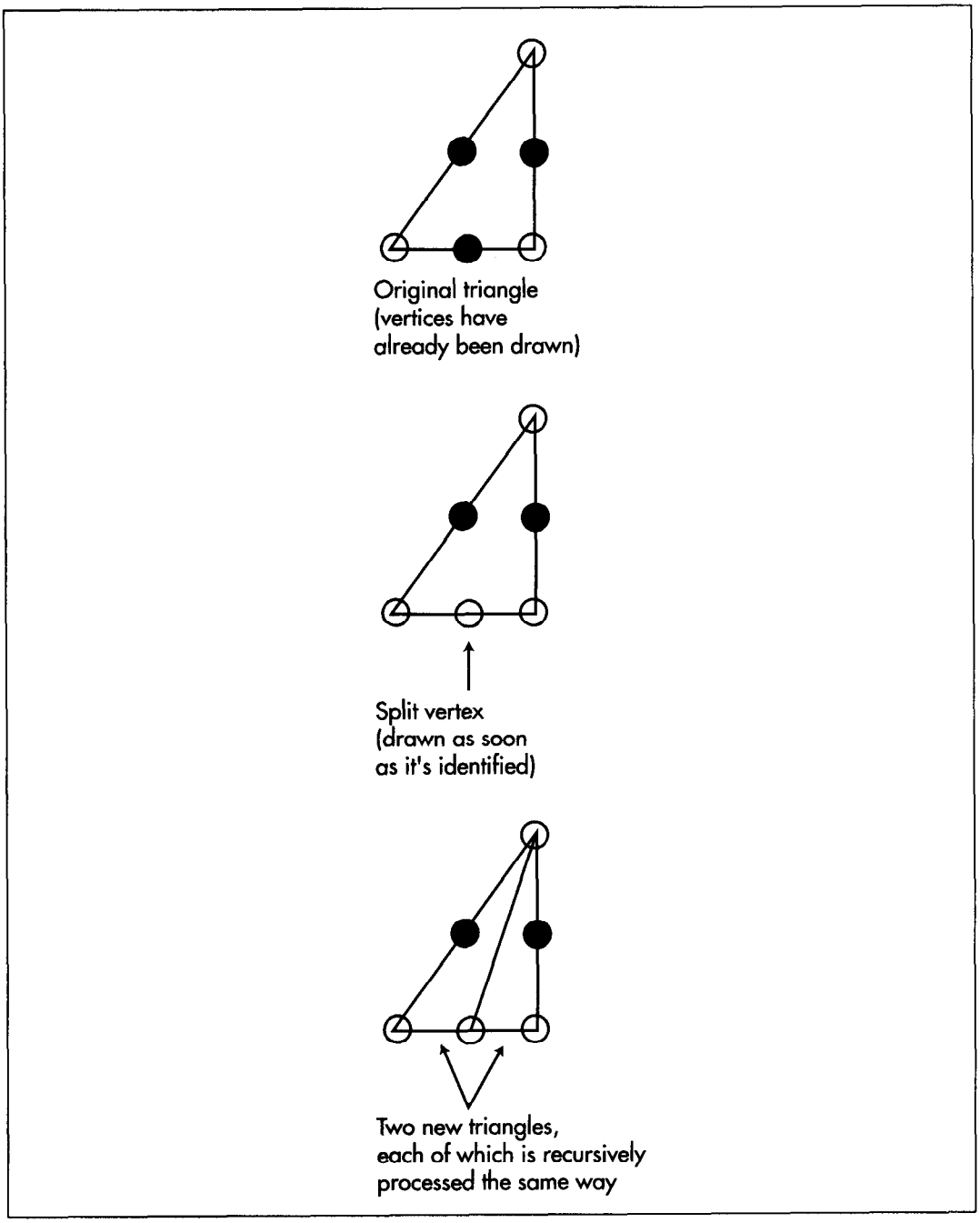
Subdivision rasterization introduces considerably more error than affine texture mapping, and doesn't draw exactly the right triangle shape, but the difference is very hard to detect for triangles that contain only a few pixels. We found that the point at which the difference between the two rasterizers becomes noticeable was surprisingly close: 30 or 40 feet for the Ogres, and about 12 feet for the Zombies. This means that most of the triangle models that are visible in a typical Quake scene are drawn with subdivision rasterization, not affine texture mapping.

How much does subdivision rasterization help performance? When John originally implemented it, it more than doubled triangle-model drawing speed, because the affine texture mapper was not yet optimized. However, I took it upon myself to see how fast I could make the mapper, so now affine texture mapping is only about 20 percent slower than subdivision rasterization. While 20 percent may not sound impressive, it includes clipping, transform, projection, and backface-culling time, so the rasterization difference alone is more than 50 percent. Besides, 20 percent overall means that we can have 12 monsters now where we could only have had 10 before, so we count subdivision rasterization as a clear success.

LISTING 69.1 L69-1.C

```
// Quake's recursive subdivision triangle rasterizer; draws all
// pixels in a triangle other than the vertices by splitting an
// edge to form a new vertex, drawing the vertex, and recursively
// processing each of the two new triangles formed by using the
// new vertex. Results are less accurate than from a precise
// affine or perspective texture mapper, and drawing boundaries
// are not identical to those of a precise polygon drawer, although
// they are consistent between adjacent polygons drawn with this
// technique.
//
// Invented and implemented by John Carmack of id Software.

void D_PolysetRecursiveTriangle (int *lp1, int *lp2, int *lp3)
{
      int    *temp;
      int    d;
```



One recursive subdivision triangle-drawing step.
Figure 69.2

```
      int    new[6];
      int    z;
      short  *zbuf;

// try to find an edge that's more than one pixel long in x or y
      d = lp2[0] - lp1[0];
      if (d < -1 || d > 1)
            goto split;
      d = lp2[1] - lp1[1];
      if (d < -1 || d > 1)
            goto split;
      d = lp3[0] - lp2[0];
      if (d < -1 || d > 1)
            goto split2;
      d = lp3[1] - lp2[1];
      if (d < -1 || d > 1)
            goto split2;
      d = lp1[0] - lp3[0];
      if (d < -1 || d > 1)
            goto split3;
      d = lp1[1] - lp3[1];
      if (d < -1 || d > 1)
      {
split3:
// shuffle points so first edge is edge to split
      temp = lp1;
      lp1 = lp3;
      lp3 = lp2;
      lp2 = temp;
      goto split;
}

return;         // no pixels left to fill in triangle

split2:
// shuffle points so first edge is edge to split
      temp = lp1;
      lp1 = lp2;
      lp2 = lp3;
      lp3 = temp;

split:
// split first edge screen x, screen y, texture s, texture t, and z
// to form a new vertex.  Lighting (index 4) is ignored; the
// difference between interpolating lighting and using the same
// shading for the entire triangle is unnoticeable for small
// triangles, so we just use the lighting for the first vertex of
// the original triangle (which was used during set-up to set
// d_colormap, used below to look up lit texels)
      new[0] = (lp1[0] + lp2[0]) >> 1;        // split screen x
      new[1] = (lp1[1] + lp2[1]) >> 1;        // split screen y
      new[2] = (lp1[2] + lp2[2]) >> 1;        // split texture s
      new[3] = (lp1[3] + lp2[3]) >> 1;        // split texture t
      new[5] = (lp1[5] + lp2[5]) >> 1;        // split z

// draw the point if splitting a leading edge
      if (lp2[1] > lp1[1])
            goto nodraw;
      if ((lp2[1] == lp1[1]) && (lp2[0] < lp1[0]))
            goto nodraw;
```

```
      z = new[5]>>16;

// point to the pixel's z-buffer entry, looking up the scanline start
// address based on screen y and adding in the screen x coordinate
      zbuf = zspantable[new[1]] + new[0];

// draw the split vertex if it's not obscured by something nearer, as
// indicated by the z-buffer
      if (z >= *zbuf)
      {
            int     pix;

      // set the z-buffer to the new pixel's distance
            *zbuf = z;

      // get the texel from the model's skin bitmap, according to
      // the s and t texture coordinates, and translate it through
      // the lighting look-up table set according to the first
      // vertex for the original (top-level) triangle.  Both s and
      // t are in 16.16 format
            pix = d_pcolormap[skintable[new[3]>>16][new[2]>>16]];

      // draw the pixel, looking up the scanline start address
      // based on screen y and adding in the screen x coordinate
            d_viewbuffer[d_scantable[new[1]] + new[0]] = pix;
      }

nodraw:
// recursively draw the two new triangles we created by adding the
// split vertex
      D_PolysetRecursiveTriangle (lp3, lp1, new);
      D_PolysetRecursiveTriangle (lp3, new, lp2);
}
```

More Ideas that Might Work

Useful as subdivision rasterization proved to be, we by no means think that we've maxed out triangle-model drawing, if only because we spent far less design and development time on subdivision than on the affine rasterizer, so it's likely that there's quite a bit more performance to be found for drawing small triangles. For example, it could be faster to precalculate drawing masks or even precompile drawing code for all possible small triangles (say, up to 4×4 or 5×5), and the memory footprint looks reasonable. (It's worth noting that both precalculated drawing and subdivision rasterization are only possible because we snap to integer coordinates; none of this stuff works with fixed-point vertices.)

More interesting still is the stack-based rendering described in the article "Time/Space Tradeoffs for Polygon Mesh Rendering," by Bar-Yehuda and Gotsman, in the April, 1996 *ACM Transactions on Graphics*. Unfortunately, the article is highly abstract and slow going, but the bottom line is that it's possible to represent a triangle mesh as a stream of commands that place vertices in a stack, remove them from the stack, and draw triangles using the vertices in the stack. This results in excellent CPU cache coherency, because rather than indirecting all over a vertex pool to retrieve vertex data, all vertices reside in a tiny stack that's guaranteed to be in the cache. Local

variables used while drawing can be stored in a small block next to the stack, and the stream of commands representing the model is accessed sequentially from start to finish, so cache utilization should be very high. As processors speed up at a much faster rate than main memory access, cache optimizations of this sort will become steadily more important in improving drawing performance.

As with so many aspects of 3-D, there is no one best approach to drawing triangle models, and no such thing as the fastest code. In a way, that's frustrating, but the truth is, it's these nearly infinite possibilities that make 3-D so interesting; not only is it an endless, varied challenge, but there's almost always a better solution waiting to be found.