

Chapter 66

Quake's Hidden-Surface Removal

Chapter

66

Struggling with Z-Order Solutions to the Hidden Surface Problem

Okay, I admit it: I'm sick and tired of classic rock. Admittedly, it's been a while, about 20 years, since I was last **excited** to hear anything by the Cars or Boston, and I was never particularly excited in the first place about Bob Seger or Queen, to say nothing of Elvis, so some things haven't changed. But I knew something was up when I found myself changing the station on the Allman Brothers and Steely Dan and Pink Floyd and, God help me, the Beatles (just stuff like "Hello Goodbye" and "I'll Cry Instead," *though*, not "Ticket to Ride" or "A Day in the Life"; I'm not *that* far gone). It didn't take long to figure out what the problem was; I'd been hearing the same songs for a quarter-century, and I was bored.

I tell you this by way of explaining why it was that when my daughter and I drove back from dinner the other night, the radio in my car was tuned, for the first time ever, to a station whose slogan is "There is no alternative."

Now, we're talking here about a 10-year-old who worships the Beatles and has been raised on a steady diet of oldies. She loves melodies, catchy songs, and good singers, none of which you're likely to find on an alternative rock station. So it's no surprise that when I turned on the radio, the first word out of her mouth was "Yuck!"

What did surprise me was that after listening for a while, she said, "You know, Dad, it's actually kind of interesting."

Apart from giving me a clue as to what sort of music I can expect to hear blasting through our house when she's a teenager, her quick uptake on alternative rock (versus my decades-long devotion to the music of my youth) reminded me of something that it's easy to forget as we become older and more set in our ways. It reminded me that it's essential to keep an open mind, and to be willing, better yet, eager, to try new things. Programmers tend to become attached to familiar approaches, and are inclined to stick with whatever is currently doing the job adequately well, but in programming there are always alternatives, and I've found that they're often worth considering.

Not that I should have needed any reminding, considering the ever-evolving nature of Quake.

Creative Flux and Hidden Surfaces

Back in Chapter 64, I described the creative flux that led to John Carmack's decision to use a precalculated potentially visible set (PVS) of polygons for each possible viewpoint in Quake, the game we're developing here at id Software. The precalculated PVS meant that instead of having to spend a lot of time searching through the world database to find out which polygons were visible from the current viewpoint, we could simply draw all the polygons in the PVS from back-to-front (getting the ordering courtesy of the world BSP tree) and get the correct scene drawn with no searching at all; letting the back-to-front drawing perform the final stage of hidden-surface removal (HSR). This was a terrific idea, but it was far from the end of the road for Quake's design.

Drawing Moving Objects

For one thing, there was still the question of how to sort and draw moving objects properly; in fact, this is the single technical question I've been asked most often in recent months, so I'll take a moment to address it here. The primary problem is that a moving model can span multiple BSP leaves, with the leaves that are touched varying as the model moves; that, together with the possibility of multiple models in one leaf, means there's no easy way to use BSP order to draw the models in correctly sorted order. When I wrote Chapter 64, we were drawing sprites (such as explosions), moveable BSP models (such as doors), and polygon models (such as monsters) by clipping each into all the leaves it touched, then drawing the appropriate parts as each BSP leaf was reached in back-to-front traversal. However, this didn't solve the issue of sorting multiple moving models in a single leaf against each other, and also left some ugly sorting problems with complex polygon models.

John solved the sorting issue for sprites and polygon models in a startlingly low-tech way: We now z-buffer them. (That is, before we draw each pixel, we compare its distance, or z, value with the z value of the pixel currently on the screen, drawing only if the new pixel is nearer than the current one.) First, we draw the basic world, walls, ceilings, and the like. No z-buffer *testing* is involved at this point (the world

visible surface determination is done in a different way, as we'll see soon); however, we do *fill* the z-buffer with the z values (actually, $1/z$ values, as discussed below) for all the world pixels. Z-filling is a much faster process than z-buffering the entire world would be, because no reads or compares are involved, just writes of z values. Once the drawing and z-filling of the world is done, we can simply draw the sprites and polygon models with z-buffering and get perfect sorting all around.

Performance Impact

Whenever a z-buffer is involved, the questions inevitably are: What's the memory footprint and what's the performance impact? Well, the memory footprint at 320×200 is 128K, not trivial but not a big deal for a game that requires 8 MB to run. The performance impact is about 10 percent for z-filling the world, and roughly 20 percent (with lots of variation) for drawing sprites and polygon models. In return, we get a perfectly sorted world, and also the ability to do additional effects, such as particle explosions and smoke, because the z-buffer lets us flawlessly sort such effects into the world. All in all, the use of the z-buffer vastly improved the visual quality and flexibility of the Quake engine, and also simplified the code quite a bit, at an acceptable memory and performance cost.

Leveling and Improving Performance

As I said above, in the Quake architecture, the world itself is drawn first, without z-buffer reads or compares, but filling the z-buffer with the world polygons' z values, and then the moving objects are drawn atop the world, using full z-buffering. Thus far, I've discussed how to draw moving objects. For the rest of this chapter, I'm going to talk about the other part of the drawing equation; that is, how to draw the world itself, where the entire world is stored as a single BSP tree and never moves.

As you may recall from Chapter 64, we're concerned with both raw performance and level performance. That is, we want the drawing code to run as fast as possible, but we also want the difference in drawing speed between the average scene and the slowest-drawing scene to be as small as possible.



It does little good to average 30 frames per second if 10 percent of the scenes draw at 5 fps, because the jerkiness in those scenes will be extremely obvious by comparison with the average scene, and highly objectionable. It would be better to average 15 fps 100 percent of the time, even though the average drawing speed is only half as much.

The precalculated PVS was an important step toward both faster and more level performance, because it eliminated the need to identify visible polygons, a relatively slow step that tended to be at its worst in the most complex scenes. Nonetheless, in some spots in real game levels the precalculated PVS contains five times more polygons than are actually visible; together with the back-to-front HSR approach, this created

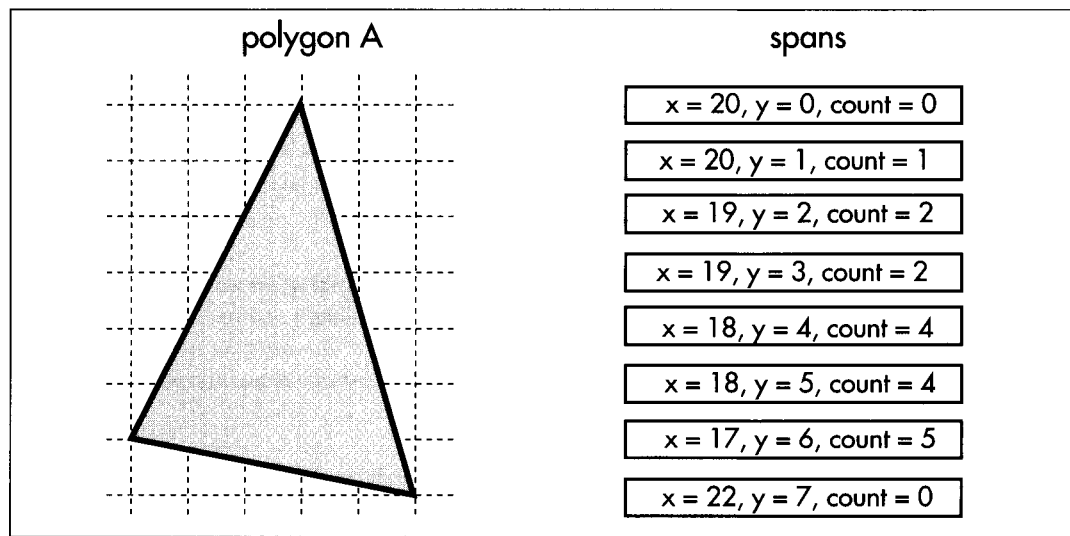
hot spots in which the frame rate bogged down visibly as hundreds of polygons are drawn back-to-front, most of those immediately getting overdrawn by nearer polygons. Raw performance in general was also reduced by the typical 50% overdraw resulting from drawing everything in the PVS. So, although drawing the PVS back-to-front as the final HSR stage worked and was an improvement over previous designs, it was not ideal. Surely, John thought, there's a better way to leverage the PVS than back-to-front drawing.

And indeed there is.

Sorted Spans

The ideal final HSR stage for Quake would reject all the polygons in the PVS that are actually invisible, and draw only the visible pixels of the remaining polygons, with no overdraw, that is, with every pixel drawn exactly once, all at no performance cost, of course. One way to do that (although certainly not at zero cost) would be to draw the polygons from front-to-back, maintaining a region describing the currently occluded portions of the screen and clipping each polygon to that region before drawing it. That sounds promising, but it is in fact nothing more or less than the beam tree approach I described in Chapter 64, an approach that we found to have considerable overhead and serious leveling problems.

We can do much better if we move the final HSR stage from the polygon level to the span level and use a sorted-spans approach. In essence, this approach consists of turning each polygon into a set of spans, as shown in Figure 66.1, and then sorting



Span generation.

Figure 66.1

and clipping the spans against each other until only the visible portions of visible spans are left to be drawn, as shown in Figure 66.2. This may sound a lot like z-buffering (which is simply too slow for use in drawing the world, although it's fine for smaller moving objects, as described earlier), but there are crucial differences.

By contrast with z-buffering, only visible portions of visible spans are scanned out pixel by pixel (although all polygon edges must still be rasterized). Better yet, the sorting that z-buffering does at each pixel becomes a per-span operation with sorted spans, and because of the coherence implicit in a span list, each edge is sorted only against some of the spans on the same line and is clipped only to the few spans that it overlaps horizontally. Although complex scenes still take longer to process than simple scenes, the worst case isn't as bad as with the beam tree or back-to-front approaches, because there's no overdraw or scanning of hidden pixels, because complexity is limited to pixel resolution and because span coherence tends to limit the worst-case sorting in any one area of the screen. As a bonus, the output of sorted spans is in precisely the form that a low-level rasterizer needs, a set of span descriptors, each consisting of a start coordinate and a length.

In short, the sorted spans approach meets our original criteria pretty well; although it isn't zero-cost, it's not horribly expensive, it completely eliminates both overdraw and pixel scanning of obscured portions of polygons and it tends to level worst-case performance. We wouldn't want to rely on sorted spans alone as our hidden-surface mechanism, but the precalculated PVS reduces the number of polygons to a level that sorted spans can handle quite nicely.

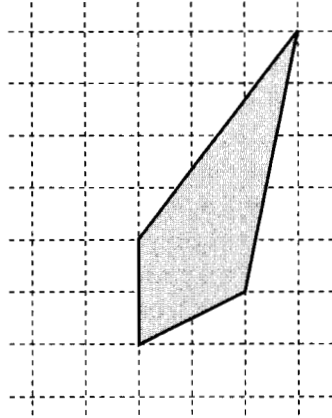
So we've found the approach we need; now it's just a matter of writing some code and we're on our way, right? Well, yes and no. Conceptually, the sorted-spans approach is simple, but it's surprisingly difficult to implement, with a couple of major design choices to be made, a subtle mathematical element, and some tricky gotchas that I'll have to defer until Chapter 67. Let's look at the design choices first.

Edges versus Spans

The first design choice is whether to sort spans or edges (both of which fall into the general category of "sorted spans"). Although the results are the same both ways, a list of spans to be drawn, with no overdraw, the implementations and performance implications are quite different, because the sorting and clipping are performed using very different data structures.

With span-sorting, spans are stored in x-sorted, linked list buckets, typically with one bucket per scan line. Each polygon in turn is rasterized into spans, as shown in Figure 66.1, and each span is sorted and clipped into the bucket for the scan line the span is on, as shown in Figure 66.2, so that at any time each bucket contains the nearest spans encountered thus far, always with no overlap. This approach involves generating all spans for each polygon in turn, with each span immediately being sorted, clipped, and added to the appropriate bucket.

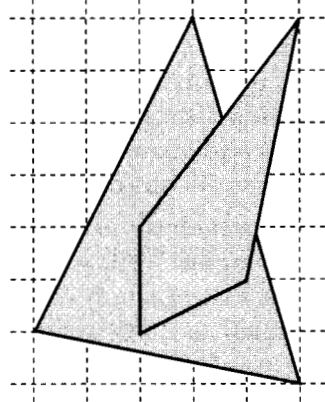
polygon A



spans

x = 22, y = 0, count = 0
x = 22, y = 1, count = 0
x = 21, y = 2, count = 1
x = 20, y = 3, count = 2
x = 19, y = 4, count = 3
x = 19, y = 5, count = 2
x = 19, y = 6, count = 0

A and B composited



visible spans

A: x = 20, y = 0, count = 0	B: x = 22, y = 0, count = 0	
A: x = 20, y = 1, count = 1	B: x = 22, y = 1, count = 0	
A: x = 19, y = 2, count = 2	B: x = 21, y = 2, count = 1	
A: x = 19, y = 3, count = 1	B: x = 20, y = 3, count = 2	
A: x = 18, y = 4, count = 1	B: x = 19, y = 4, count = 3	
A: x = 18, y = 5, count = 1	B: x = 19, y = 5, count = 2	A: x = 20, y = 5, count = 1
A: x = 17, y = 6, count = 5	B: x = 19, y = 6, count = 0	
A: x = 22, y = 7, count = 0		

Two sets of spans sorted and clipped against one another.
Figure 66.2

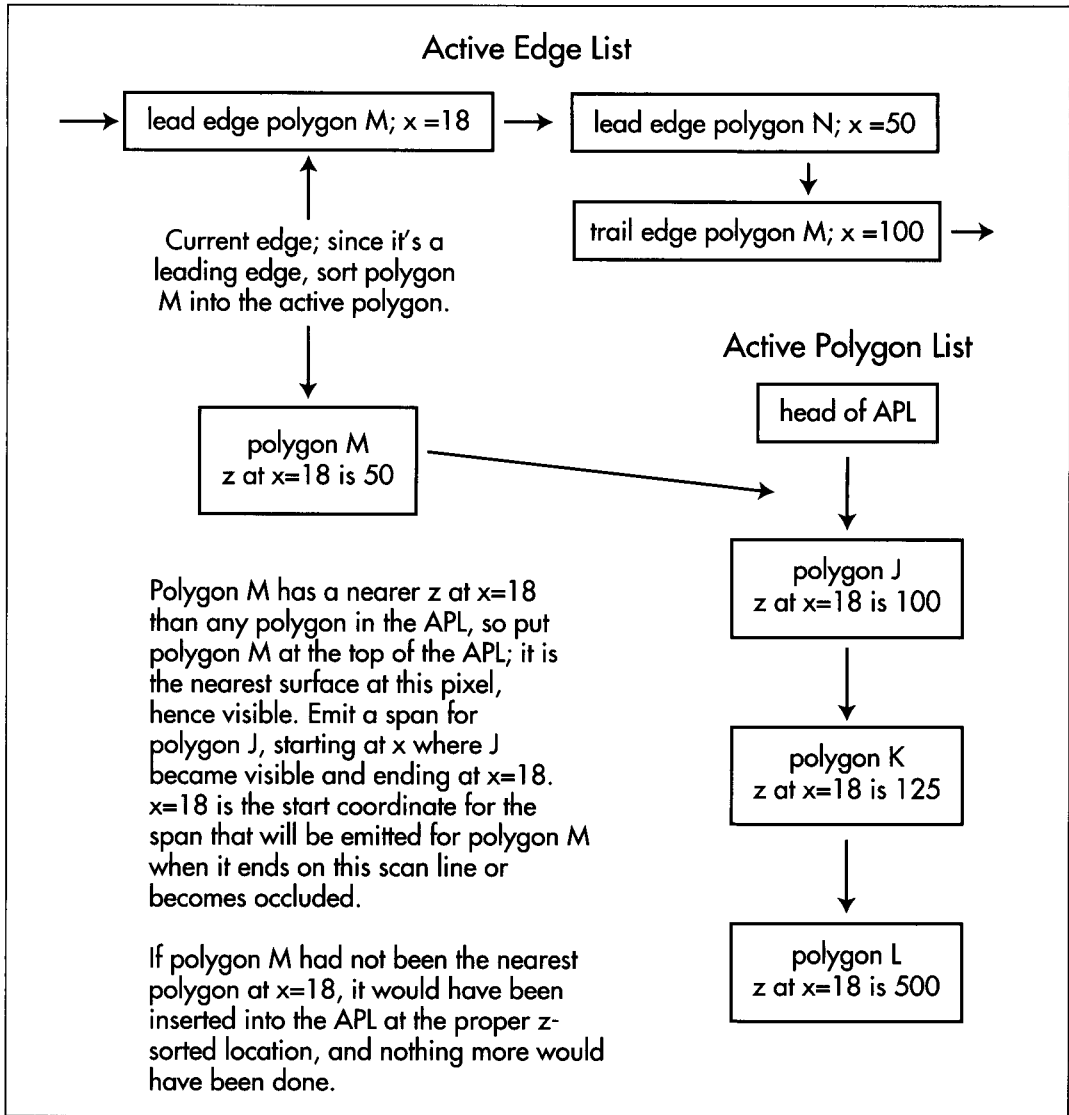
With edge-sorting, edges are stored in x-sorted, linked list buckets according to their start scan line. Each polygon in turn is decomposed into edges, cumulatively building a list of all the edges in the scene. Once all edges for all polygons in the view frustum have been added to the edge list, the whole list is scanned out in a single top-to-bottom, left-to-right pass. An active edge list (AEL) is maintained. With each step to a new scan line, edges that end on that scan line are removed from the AEL, active edges are stepped to their new x coordinates, edges starting on the new scan line are added to the AEL, and the edges are sorted by current x coordinate.

For each scan line, a z-sorted active polygon list (APL) is maintained. The x-sorted AEL is stepped through in order. As each new edge is encountered (that is, as each polygon starts or ends as we move left to right), the associated polygon is activated and sorted into the APL, as shown in Figure 66.3, or deactivated and removed from the APL, as shown in Figure 66.4, for a leading or trailing edge, respectively. If the nearest polygon has changed (that is, if the new polygon is nearest, or if the nearest polygon just ended), a span is emitted for the polygon that just stopped being the nearest, starting at the point where the polygon first became nearest and ending at the x coordinate of the current edge, and the current x coordinate is recorded in the polygon that is now the nearest. This saved coordinate later serves as the start of the span emitted when the new nearest polygon ceases to be in front.

Don't worry if you didn't follow all of that; the above is just a quick overview of edge-sorting to help make the rest of this chapter a little clearer. My thorough discussion of the topic will be in Chapter 67.

The spans that are generated with edge-sorting are exactly the same spans that ultimately emerge from span-sorting; the difference lies in the intermediate data structures that are used to sort the spans in the scene. With edge-sorting, the spans are kept implicit in the edges until the final set of visible spans is generated, so the sorting, clipping, and span emission is done as each edge adds or removes a polygon, based on the span state implied by the edge and the set of active polygons. With span-sorting, spans are immediately made explicit when each polygon is rasterized, and those intermediate spans are then sorted and clipped against other the spans on the scan line to generate the final spans, so the states of the spans are explicit at all times, and all work is done directly with spans.

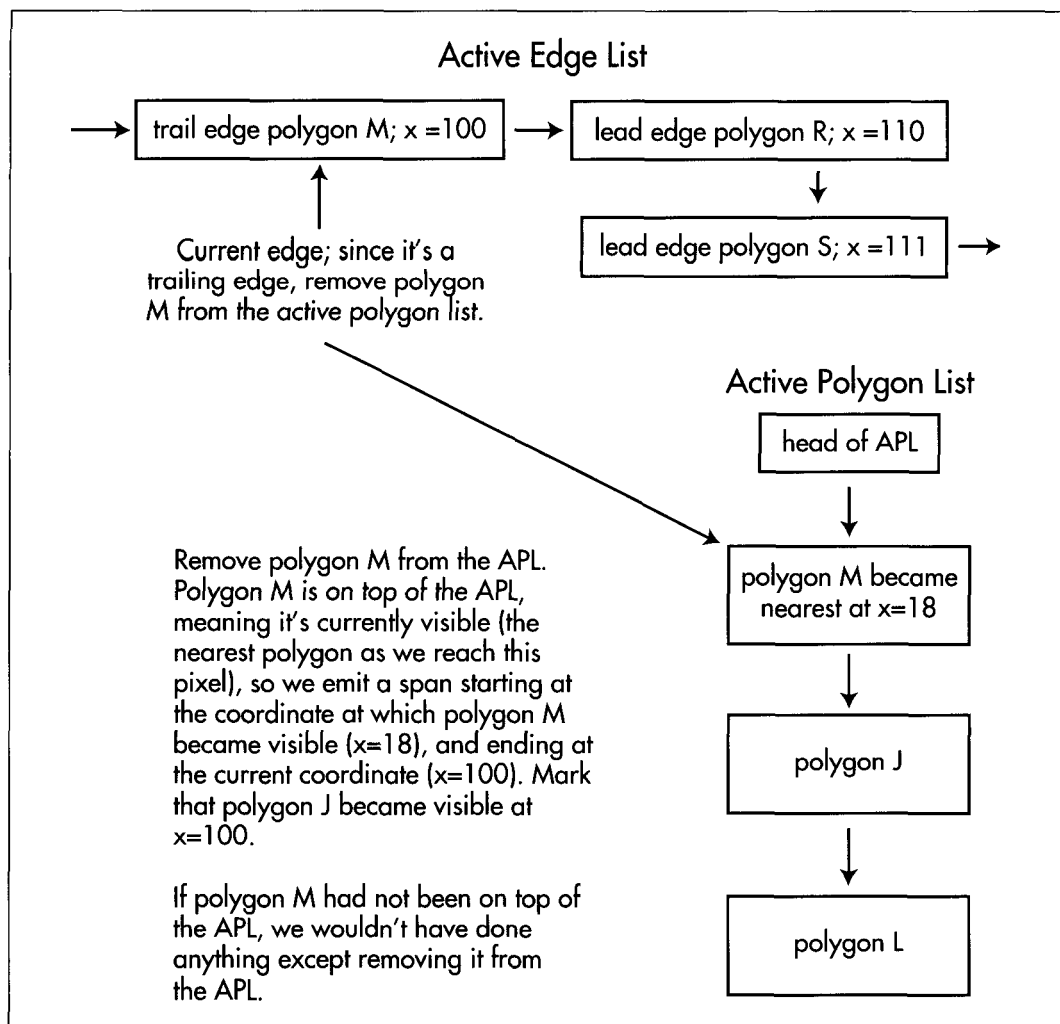
Both span-sorting and edge-sorting work well, and both have been employed successfully in commercial projects. We've chosen to use edge-sorting in Quake partly because it seems inherently more efficient, with excellent horizontal coherence that makes for minimal time spent sorting, in contrast with the potentially costly sorting into linked lists that span-sorting can involve. A more important reason, though, is that with edge-sorting we're able to share edges between adjacent polygons, and that cuts the work involved in sorting, clipping, and rasterizing edges nearly in half, while also shrinking the world database quite a bit due to the sharing.



Activating a polygon when a leading edge is encountered in the AEL.

Figure 66.3

One final advantage of edge-sorting is that it makes no distinction between convex and concave polygons. That's not an important consideration for most graphics engines, but in *Quake*, edge clipping, transformation, projection, and sorting have become a major bottleneck, so we're doing everything we can to get the polygon and edge counts down, and concave polygons help a lot in that regard. While it's possible



Deactivating a polygon when a trailing edge is encountered in the AEL.

Figure 66.4

to handle concave polygons with span-sorting, that can involve significant performance penalties.

Nonetheless, there's no cut-and-dried answer as to which approach is better. In the end, span-sorting and edge-sorting amount to the same functionality, and the choice between them is a matter of whatever you feel most comfortable with. In Chapter 67, I'll go into considerable detail about edge-sorting, complete with a full implementation. I'm going to spend the rest of this chapter laying the foundation for Chapter 67 by discussing sorting keys and I/z calculation. In the process, I'm going to have to

make a few forward references to aspects of edge-sorting that I haven't yet covered in detail; my apologies, but it's unavoidable, and all should become clear by the end of Chapter 67.

Edge-Sorting Keys

Now that we know we're going to sort edges, using them to emit spans for the polygons nearest the viewer, the question becomes: How can we tell which polygons are nearest? Ideally, we'd just store a sorting key in each polygon, and whenever a new edge came along, we'd compare its surface's key to the keys of other currently active polygons, and could easily tell which polygon was nearest.

That sounds too good to be true, but it is possible. If, for example, your world database is stored as a BSP tree, with all polygons clipped into the BSP leaves, then BSP walk order is a valid drawing order. So, for example, if you walk the BSP back-to-front, assigning each polygon an incrementally higher key as you reach it, polygons with higher keys are guaranteed to be in front of polygons with lower keys. This is the approach Quake used for a while, although a different approach is now being used, for reasons I'll explain shortly.

If you don't happen to have a BSP or similar data structure handy, or if you have lots of moving polygons (BSPs don't handle moving polygons very efficiently), another way to accomplish your objectives would be to sort all the polygons against one another before drawing the scene, assigning appropriate keys based on their spatial relationships in viewspace. Unfortunately, this is generally an extremely slow task, because every polygon must be compared to every other polygon. There are techniques to improve the performance of polygon sorts, but I don't know of anyone who's doing general polygon sorts of complex scenes in realtime on a PC.

An alternative is to sort by z distance from the viewer in screenspace, an approach that dovetails nicely with the excellent spatial coherence of edge-sorting. As each new edge is encountered on a scan line, the corresponding polygon's z distance can be calculated and compared to the other polygons' distances, and the polygon can be sorted into the APL accordingly.

Getting z distances can be tricky, however. Remember that we need to be able to calculate z at any arbitrary point on a polygon, because an edge may occur and cause its polygon to be sorted into the APL at any point on the screen. We could calculate z directly from the screen x and y coordinates and the polygon's plane equation, but unfortunately this can't be done very quickly, because the z for a plane doesn't vary linearly in screenspace; however, $1/z$ *does* vary linearly, so we'll use that instead. (See Chris Hecker's 1995 series of columns on texture mapping in *Game Developer* magazine for a discussion of screenspace linearity and gradients for $1/z$.) Another advantage of using $1/z$ is that its resolution increases with decreasing distance, meaning that by using $1/z$, we'll have better depth resolution for nearby features, where it matters most.

The obvious way to get a $1/z$ value at any arbitrary point on a polygon is to calculate $1/z$ at the vertices, interpolate it down both edges of the polygon, and interpolate between the edges to get the value at the point of interest. Unfortunately, that requires doing a lot of work along each edge, and worse, requires division to calculate the $1/z$ step per pixel across each span.

A better solution is to calculate $1/z$ directly from the plane equation and the screen x and y of the pixel of interest. The equation is

$$1/z = (a/d)x' - (b/d)y' + c/d$$

where z is the viewspace z coordinate of the point on the plane that projects to screen coordinate (x',y') (the origin for this calculation is the center of projection, the point on the screen straight ahead of the viewpoint), $[a\ b\ c]$ is the plane normal in viewspace, and d is the distance from the viewspace origin to the plane along the normal. Division is done only once per plane, because a , b , c , and d are per-plane constants.

The full $1/z$ calculation requires two multiplies and two adds, all of which should be floating-point to avoid range errors. That much floating-point math sounds expensive but really isn't, especially on a Pentium, where a plane's $1/z$ value at any point can be calculated in as little as six cycles in assembly language.

Where That $1/Z$ Equation Comes From

For those who are interested, here's a quick derivation of the $1/z$ equation. The plane equation for a plane is

$$ax + by + cz - d = 0$$

where x and y are viewspace coordinates, and a , b , c , d , and z are defined above. If we substitute $x=x'z$ and $y=-y'z$ (from the definition of the perspective projection, with y inverted because y increases upward in viewspace but downward in screenspace), and do some rearrangement, we get:

$$z = d / (ax' - by' + c)$$

Inverting and distributing yields:

$$1/z = ax'/d - by'/d + c/d$$

We'll see $1/z$ sorting in action in Chapter 67.

Quake and Z-Sorting

I mentioned earlier that Quake no longer uses BSP order as the sorting key; in fact, it uses $1/z$ as the key now. Elegant as the gradients are, calculating $1/z$ from them is clearly slower than just doing a compare on a BSP-ordered key, so why have we switched Quake to $1/z$?

The primary reason is to reduce the number of polygons. Drawing in BSP order means following certain rules, including the rule that polygons must be split if they cross BSP planes. This splitting increases the numbers of polygons and edges considerably. By

sorting on $1/z$, we're able to leave polygons unsplit but still get correct drawing order, so we have far fewer edges to process and faster drawing overall, despite the added cost of $1/z$ sorting.

Another advantage of $1/z$ sorting is that it solves the sorting issues I mentioned at the start involving moving models that are themselves small BSP trees. Sorting in world BSP order wouldn't work here, because these models are separate BSPs, and there's no easy way to work them into the world BSP's sequence order. We don't want to use z-buffering for these models because they're often large objects such as doors, and we don't want to lose the overdraw-reduction benefits that closed doors provide when drawn through the edge list. With sorted spans, the edges of moving BSP models are simply placed in the edge list (first clipping polygons so they don't cross any solid world surfaces, to avoid complications associated with interpenetration), along with all the world edges, and $1/z$ sorting takes care of the rest.

Decisions Deferred

There is, without a doubt, an awful lot of information in the preceding pages, and it may not all connect together yet in your mind. The code and accompanying explanation in the next chapter should help; if you want to peek ahead, the code is available on the CD-ROM as DDJZSORT.ZIP in the directory for Chapter 67. You may also want to take a look at Foley and van Dam's *Computer Graphics* or Rogers' *Procedural Elements for Computer Graphics*.

As I write this, it's unclear whether Quake will end up sorting edges by BSP order or $1/z$. Actually, there's no guarantee that sorted spans in any form will be the final design. Sometimes it seems like we change graphics engines as often as they play Elvis on the '50s oldies stations (but, one would hope, with more aesthetically pleasing results!) and no doubt we'll be considering the alternatives right up until the day we ship.