# Chapter 64

# Quake's Visible-Surface Determination

# Chapter 64

## The Challenge of Separating All Things Seen from All Things Unseen

Years ago, I was working at Video Seven, a now-vanished video adapter manufacturer, helping to develop a VGA clone. The fellow who was designing Video Seven's VGA chip, Tom Wilson, had worked around the clock for months to make his VGA run as fast as possible, and was confident he had pretty much maxed out its performance. As Tom was putting the finishing touches on his chip design, however, news came fourth-hand that a competitor, Paradise, had juiced up the performance of the clone they were developing by putting in a FIFO.

That was all he knew; there was no information about what sort of FIFO, or how much it helped, or anything else. Nonetheless, Tom, normally an affable, laid-back sort, took on the wide-awake, haunted look of a man with too much caffeine in him and no answers to show for it, as he tried to figure out, from hopelessly thin information, what Paradise had done. Finally, he concluded that Paradise must have put a write FIFO between the system bus and the VGA, so that when the CPU wrote to video memory, the write immediately went into the FIFO, allowing the CPU to keep on processing instead of stalling each time it wrote to display memory.

Tom couldn't spare the gates or the time to do a full FIFO, but he could implement a one-deep FIFO, allowing the CPU to get one write ahead of the VGA. He wasn't sure how well it would work, but it was all he could do, so he put it in and taped out the chip.

The one-deep FIFO turned out to work astonishingly well; for a time, Video Seven's VGAs were the fastest around, a testament to Tom's ingenuity and creativity under pressure. However, the truly remarkable part of this story is that Paradise's FIFO design turned out to bear not the slightest resemblance to Tom's, and *didn't work as well.* Paradise had stuck a *read* FIFO between display memory and the video output stage of the VGA, allowing the video output to read ahead, so that when the CPU wanted to access display memory, pixels could come from the FIFO while the CPU was serviced immediately. That did indeed help performance—but not as much as Tom's write FIFO.

*What we have here is as neat a parable about the nature of creative design as one could hope to find. The scrap of news about Paradise's chip contained almost no actual information, but it forced Tom to push past the limits he had unconsciously set in coming up with his original design. And, in the end, I think that the single most important element of great design, whether it be hardware, software, or any creative endeavor, is precisely what the Paradise news triggered in Tom: the ability to detect the limits you have built into the way you think about your design, and then transcend those limits.*

The problem, of course, is how to go about transcending limits you don't even know you've imposed. There's no formula for success, but two principles can stand you in good stead: simplify and keep on trying new things.

Generally, if you find your code getting more complex, you're fine-tuning a frozen design, and it's likely you can get more of a speed-up, with less code, by rethinking the design. A really good design should bring with it a moment of immense satisfaction in which everything falls into place, and you're amazed at how little code is needed and how all the boundary cases just work properly.

As for how to rethink the design, do it by pursuing whatever ideas occur to you, no matter how off-the-wall they seem. Many of the truly brilliant design ideas I've heard of over the years sounded like nonsense at first, because they didn't fit my preconceived view of the world. Often, such ideas are in fact off-the-wall, but just as the news about Paradise's chip sparked Tom's imagination, aggressively pursuing seemingly outlandish ideas can open up new design possibilities for you.

Case in point: The evolution of Quake's 3-D graphics engine.

## VSD: The Toughest 3-D Challenge of All

I've spent most of my waking hours for the last several months working on Quake, id Software's successor to DOOM, and I suspect I have a few more months to go. The very best things don't happen easily, nor quickly—but when they happen, all the sweat becomes worthwhile.

In terms of graphics, Quake is to DOOM as DOOM was to its predecessor, Wolfenstein 3-D. Quake adds true, arbitrary 3-D (you can look up and down, lean, and even fall on your side), detailed lighting and shadows, and 3-D monsters and players in place of DOOM's sprites. Someday I hope to talk about how all that works, but for the here and now I want to talk about what is, in my opinion, the toughest 3-D problem of all: visible surface determination (drawing the proper surface at each pixel), and its close relative, culling (discarding non-visible polygons as quickly as possible, a way of accelerating visible surface determination). In the interests of brevity, I'll use the abbreviation VSD to mean both visible surface determination and culling from now on.

Why do I think VSD is the toughest 3-D challenge? Although rasterization issues such as texture mapping are fascinating and important, they are tasks of relatively finite scope, and are being moved into hardware as 3-D accelerators appear; also, they only scale with increases in screen resolution, which are relatively modest.

In contrast, VSD is an open-ended problem, and there are dozens of approaches currently in use. Even more significantly, the performance of VSD, done in an unsophisticated fashion, scales directly with scene complexity, which tends to increase as a square or cube function, so this very rapidly becomes the limiting factor in rendering realistic worlds. I expect VSD to be the increasingly dominant issue in realtime PC 3-D over the next few years, as 3-D worlds become increasingly detailed. Already, a good-sized Quake level contains on the order of 10,000 polygons, about three times as many polygons as a comparable DOOM level.

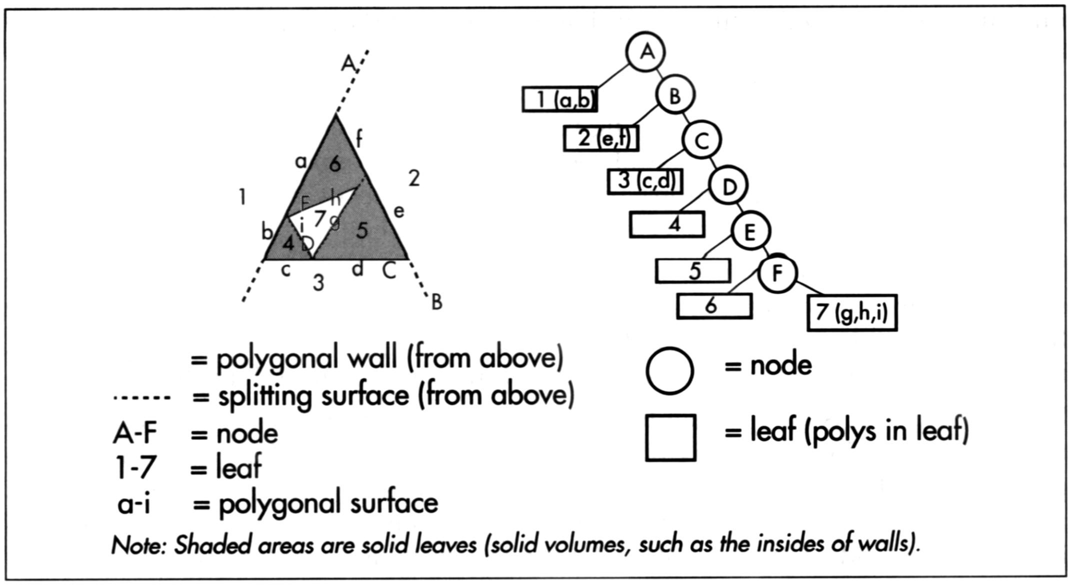
## The Structure of Quake Levels

Before diving into VSD, let me note that each Quake level is stored as a single huge 3-D BSP tree. This BSP tree, like any BSP, subdivides space, in this case along the planes of the polygons. However, unlike the BSP tree I presented in Chapter 62, Quake's BSP tree does not store polygons in the tree nodes, as part of the splitting planes, but rather in the empty (non-solid) leaves, as shown in overhead view in Figure 64.1.

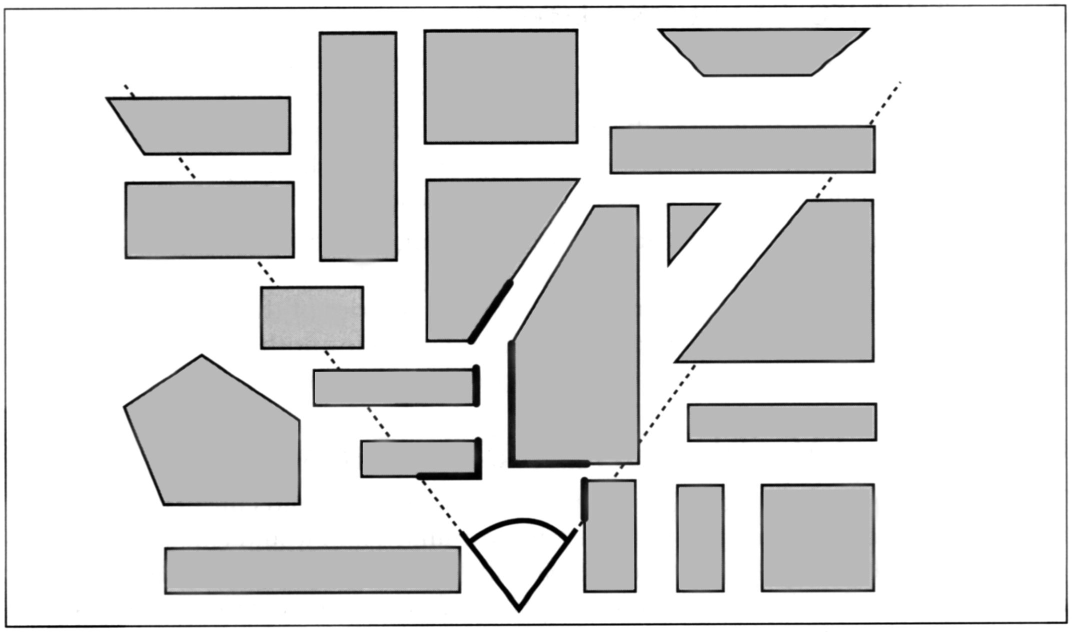
Correct drawing order can be obtained by drawing the leaves in front-to-back or back-to-front BSP order, again as discussed in Chapter 62. Also, because BSP leaves are always convex and the polygons are on the boundaries of the BSP leaves, facing inward, the polygons in a given leaf can never obscure one another and can be drawn in any order. (This is a general property of convex polyhedra.)

## Culling and Visible Surface Determination

The process of VSD would ideally work as follows: First, you would cull all polygons that are completely outside the view frustum (view pyramid), and would clip away the irrelevant portions of any polygons that are partially outside. Then, you would draw only those pixels of each polygon that are actually visible from the current viewpoint, as shown in overhead view in Figure 64.2, wasting no time overdrawing pixels multiple times; note how little of the polygon sets in Figure 64.2 actually need to be drawn. Finally, in a perfect world, the tests to figure out what parts of which polygons are visible would be free,


A
B
C
D
E
F
1
2
3
4
5
6
7
a
b
c
d
e
f
g
h
i
1 (a,b)
2 (e,f)
3 (c,d)
4
5
6
7 (g,h,i)
= polygonal wall (from above)
...... = splitting surface (from above)
A-F = node
1-7 = leaf
a-i = polygonal surface
= node
= leaf (polys in leaf)
Note: Shaded areas are solid leaves (solid volumes, such as the insides of walls).

*Quake's polygons are stored as empty leaves.*
**Figure 64.1**



*Pixels visible from the current viewpoint.*
**Figure 64.2**

and the processing time would be the same for all possible viewpoints, giving the game a smooth visual flow.

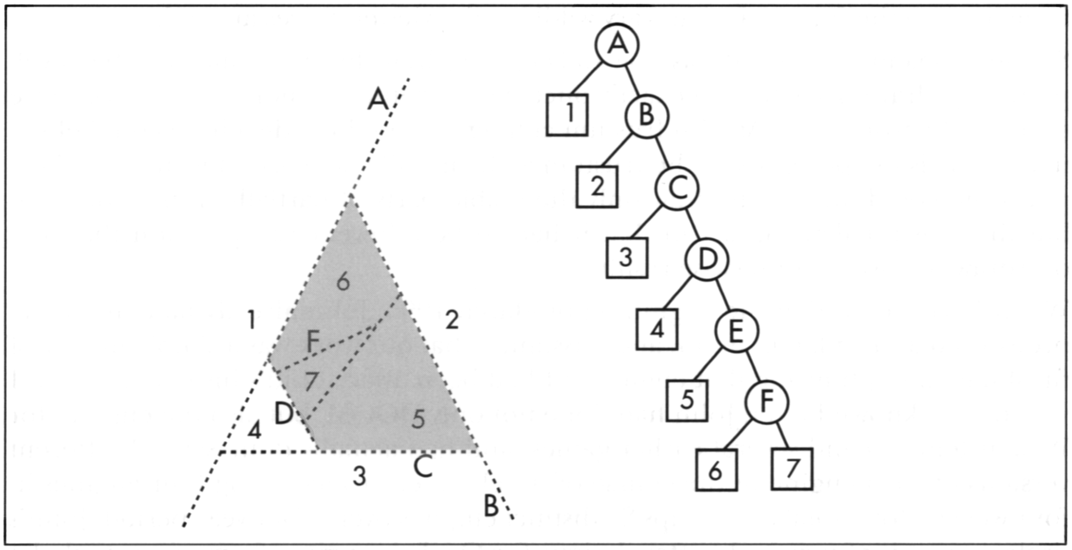
As it happens, it is easy to determine which polygons are outside the frustum or partially clipped, and it's quite possible to figure out precisely which pixels need to be drawn. Alas, the world is far from perfect, and those tests are far from free, so the real trick is how to accelerate or skip various tests and still produce the desired result.

As I discussed at length in Chapter 62, given a BSP, it's easy and inexpensive to walk the world in front-to-back or back-to-front order. The simplest VSD solution, which I in fact demonstrated earlier, is to simply walk the tree back-to-front, clip each polygon to the frustum, and draw it if it's facing forward and not entirely clipped (the painter's algorithm). Is that an adequate solution?

For relatively simple worlds, it is perfectly acceptable. It doesn't scale very well, though. One problem is that as you add more polygons in the world, more transformations and tests have to be performed to cull polygons that aren't visible; at some point, that will bog considerably performance down.

## Nodes Inside and Outside the View Frustum

Happily, there's a good workaround for this particular problem. As discussed earlier, each leaf of a BSP tree represents a convex subspace, with the nodes that bound the leaf delimiting the space. Perhaps less obvious is that each node in a BSP tree also describes a subspace—the subspace composed of all the node's children, as shown in Figure 64.3. Another way of thinking of this is that each node splits the subspace



*The substance described by node E.*
Figure 64.3

into two pieces created by the nodes above it in the tree, and the node's children then further carve that subspace into all the leaves that descend from the node.

Since a node's subspace is bounded and convex, it is possible to test whether it is entirely outside the frustum. If it is, *all* of the node's children are certain to be fully clipped and can be rejected without any additional processing. Since most of the world is typically outside the frustum, many of the polygons in the world can be culled almost for free, in huge, node-subspace chunks. It's relatively expensive to perform a perfect test for subspace clipping, so instead bounding spheres or boxes are often maintained for each node, specifically for culling tests.

So culling to the frustum isn't a problem, and the BSP can be used to draw back-to-front. What, then, *is* the problem?

## Overdraw

The problem John Carmack, the driving technical force behind DOOM and Quake, faced when he designed Quake was that in a complex world, many scenes have an awful lot of polygons in the frustum. Most of those polygons are partially or entirely obscured by other polygons, but the painter's algorithm described earlier requires that every pixel of every polygon in the frustum be drawn, often only to be overdrawn. In a 10,000-polygon Quake level, it would be easy to get a worst-case overdraw level of 10 times or more; that is, in some frames each pixel could be drawn 10 times or more, on average. No rasterizer is fast enough to compensate for an order of such magnitude and more work than is actually necessary to show a scene; worse still, the painter's algorithm will cause a vast difference between best-case and worst-case performance, so the frame rate can vary wildly as the viewer moves around.

So the problem John faced was how to keep overdraw down to a manageable level, preferably drawing each pixel exactly once, but certainly no more than two or three times in the worst case. As with frustum culling, it would be ideal if he could eliminate all invisible polygons in the frustum with virtually no work. It would also be a plus if he could manage to draw only the visible parts of partially-visible polygons, but that was a balancing act in that it had to be a lower-cost operation than the overdraw that would otherwise result.

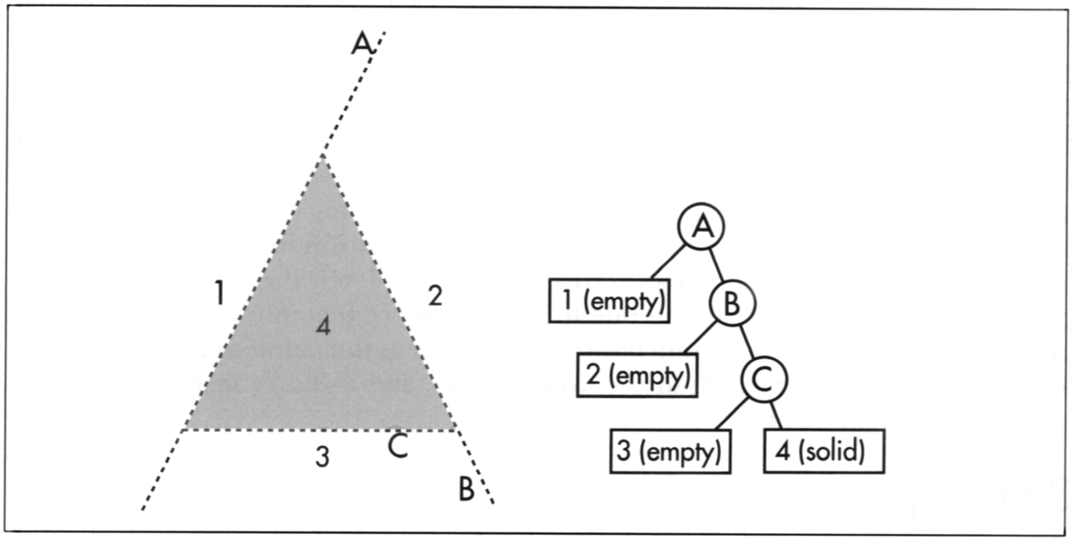
When I arrived at id at the beginning of March 1995, John already had an engine prototyped and a plan in mind, and I assumed that our work was a simple matter of finishing and optimizing that engine. If I had been aware of id's history, however, I would have known better. John had done not only DOOM, but also the engines for Wolfenstein 3-D and several earlier games, and had actually done several different versions of each engine in the course of development (once doing four engines in four weeks), for a total of perhaps 20 distinct engines over a four-year period. John's tireless pursuit of new and better designs for Quake's engine, from every angle he could think of, would end only when we shipped the product.

By three months after I arrived, only one element of the original VSD design was anywhere in sight, and John had taken the dictum of "try new things" farther than I'd ever seen it taken.

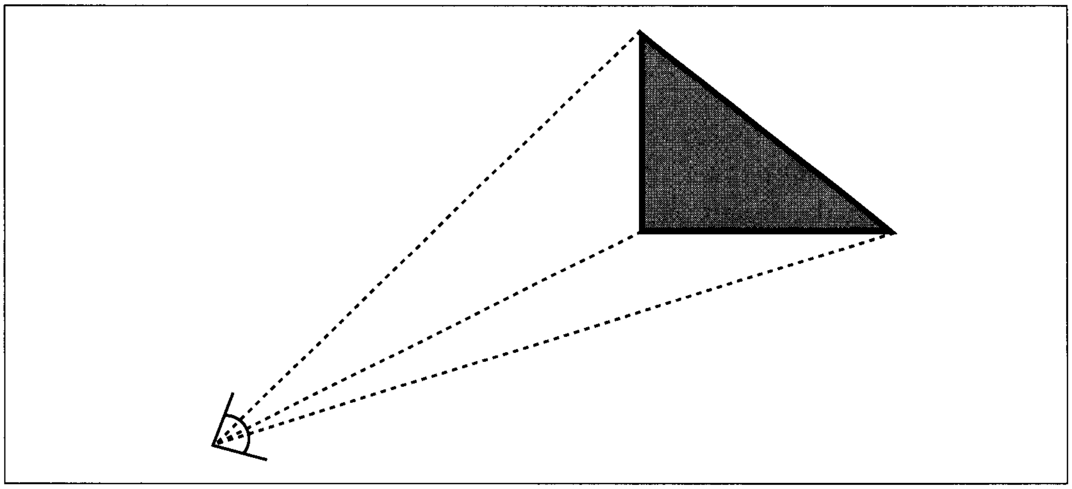
## The Beam Tree

John's original Quake design was to draw front-to-back, using a second BSP tree to keep track of what parts of the screen were already drawn and which were still empty and therefore drawable by the remaining polygons. Logically, you can think of this BSP tree as being a 2-D region describing solid and empty areas of the screen, as shown in Figure 64.4, but in fact it is a 3-D tree, of the sort known as a *beam tree.* A beam tree is a collection of 3-D wedges (beams), bounded by planes, projecting out from some center point, in this case the viewpoint, as shown in Figure 64.5.

In John's design, the beam tree started out consisting of a single beam describing the frustum; everything outside that beam was marked solid (so nothing would draw there), and the inside of the beam was marked empty. As each new polygon was reached while walking the world BSP tree front-to-back, that polygon was converted to a beam by running planes from its edges through the viewpoint, and any part of the beam that intersected empty beams in the beam tree was considered drawable and added to the beam tree as a solid beam. This continued until either there were no more polygons or the beam tree became entirely solid. Once the beam tree was completed, the visible portions of the polygons that had contributed to the beam tree were drawn.



*Partitioning the screen into 2-D regions.*
Figure 64.4



*Beams as wedges projecting from the viewpoint to polygon edges.*
**Figure 64.5**

The advantage to working with a 3-D beam tree, rather than a 2-D region, is that determining which side of a beam plane a polygon vertex is on involves only checking the sign of the dot product of the ray to the vertex and the plane normal, because all beam planes run through the origin (the viewpoint). Also, because a beam plane is completely described by a single normal, generating a beam from a polygon edge requires only a cross-product of the edge and a ray from the edge to the viewpoint. Finally, bounding spheres of BSP nodes can be used to do the aforementioned bulk culling to the frustum.

The early-out feature of the beam tree—stopping when the beam tree becomes solid—seems appealing, because it appears to cap worst-case performance. Unfortunately, there are still scenes where it's possible to see all the way to the sky or the back wall of the world, so in the worst case, all polygons in the frustum will still have to be tested against the beam tree. Similar problems can arise from tiny cracks due to numeric precision limitations. Beam-tree clipping is fairly time-consuming, and in scenes with long view distances, such as views across the top of a level, the total cost of beam processing slowed Quake's frame rate to a crawl. So, in the end, the beam-tree approach proved to suffer from much the same malady as the painter's algorithm: The worst case was much worse than the average case, and it didn't scale well with increasing level complexity.

## 3-D Engine *du Jour*

Once the beam tree was working, John relentlessly worked at speeding up the 3-D engine, always trying to improve the design, rather than tweaking the implementation. At least once a week, and often every day, he would walk into my office and say

"Last night I couldn't get to sleep, so I was thinking..." and I'd know that I was about to get my mind stretched yet again. John tried many ways to improve the beam tree, with some success, but more interesting was the profusion of wildly different approaches that he generated, some of which were merely discussed, others of which were implemented in overnight or weekend-long bursts of coding, in both cases ultimately discarded or further evolved when they turned out not to meet the design criteria well enough. Here are some of those approaches, presented in minimal detail in the hopes that, like Tom Wilson with the Paradise FIFO, your imagination will be sparked.

## Subdividing Raycast

Rays are cast in an 8×8 screen-pixel grid; this is a highly efficient operation because the first intersection with a surface can be found by simply clipping the ray into the BSP tree, starting at the viewpoint, until a solid leaf is reached. If adjacent rays don't hit the same surface, then a ray is cast halfway between, and so on until all adjacent rays either hit the same surface or are on adjacent pixels; then the block around each ray is drawn from the polygon that was hit. This scales very well, being limited by the number of pixels, with no overdraw. The problem is dropouts; it's quite possible for small polygons to fall between rays and vanish.

## Vertex-Free Surfaces

The world is represented by a set of surface planes. The polygons are implicit in the plane intersections, and are extracted from the planes as a final step before drawing. This makes for fast clipping and a very small data set (planes are far more compact than polygons), but it's time-consuming to extract polygons from planes.

## The Draw-Buffer

Like a z-buffer, but with 1 bit per pixel, indicating whether the pixel has been drawn yet. This eliminates overdraw, but at the cost of an inner-loop buffer test, extra writes and cache misses, and, worst of all, considerable complexity. Variations include testing the draw-buffer a byte at a time and completely skipping fully-occluded bytes, or branching off each draw-buffer byte to one of 256 unrolled inner loops for drawing 0–8 pixels, in the process possibly taking advantage of the ability of the x86 to do the perspective floating-point divide in parallel while 8 pixels are processed.

## Span-Based Drawing

Polygons are rasterized into spans, which are added to a global span list and clipped against that list so that only the nearest span at each pixel remains. Little sorting is needed with front-to-back walking, because if there's any overlap, the span already in the list is nearer. This eliminates overdraw, but at the cost of a lot of span arithmetic; also, every polygon still has to be turned into spans.

### Portals

The holes where polygons are missing on surfaces are tracked, because it's only through such portals that line-of-sight can extend. Drawing goes front-to-back, and when a portal is encountered, polygons and portals behind it are clipped to its limits, until no polygons or portals remain visible. Applied recursively, this allows drawing only the visible portions of visible polygons, but at the cost of a considerable amount of portal clipping.

## Breakthrough!

In the end, John decided that the beam tree was a sort of second-order structure, reflecting information already implicitly contained in the world BSP tree, so he tackled the problem of extracting visibility information directly from the world BSP tree. He spent a week on this, as a byproduct devising a perfect DOOM (2-D) visibility architecture, whereby a single, linear walk of a DOOM BSP tree produces zero-overdraw 2-D visibility. Doing the same in 3-D turned out to be a much more complex problem, though, and by the end of the week John was frustrated by the increasing complexity and persistent glitches in the visibility code. Although the direct-BSP approach was getting closer to working, it was taking more and more tweaking, and a simple, clean design didn't seem to be falling out. When I left work one Friday, John was preparing to try to get the direct-BSP approach working properly over the weekend.

When I came in on Monday, John had the look of a man who had broken through to the other side—and also the look of a man who hadn't had much sleep. He had worked all weekend on the direct-BSP approach, and had gotten it working reasonably well, with insights into how to finish it off. At 3:30 Monday morning, as he lay in bed, thinking about portals, he thought of precalculating and storing in each leaf a list of all leaves visible from that leaf, and then at runtime just drawing the visible leaves back-to-front for whatever leaf the viewpoint happens to be in, ignoring all other leaves entirely.

Size was a concern; initially, a raw, uncompressed potentially visible set (PVS) was several megabytes in size. However, the PVS could be stored as a bit vector, with 1 bit per leaf, a structure that shrunk a great deal with simple zero-byte compression. Those steps, along with changing the BSP heuristic to generate fewer leaves (choosing as the next splitter the polygon that splits the fewest other polygons appears to be the best heuristic) and sealing the outside of the levels so the BSPer can remove the outside surfaces, which can never be seen, eventually brought the PVS down to about 20 Kb for a good-size level.

In exchange for that 20 Kb, culling leaves outside the frustum is speeded up (because only leaves in the PVS are considered), and culling inside the frustum costs nothing more than a little overdraw (the PVS for a leaf includes all leaves visible from anywhere in the leaf, so some overdraw, typically on the order of 50 percent but ranging up to 150 percent, generally occurs). Better yet, precalculating the PVS results in a leveling of performance; worst case is no longer much worse than best case, because there's no longer extra VSD processing—just more polygons and perhaps some extra overdraw—associated with complex scenes. The first time John showed me his working prototype, I went to the most complex scene I knew of, a place where the frame rate used to grind down into the single digits, and spun around smoothly, with no perceptible slowdown.

John says precalculating the PVS was a logical evolution of the approaches he had been considering, that there was no moment when he said "Eureka!" Nonetheless, it was clearly a breakthrough to a brand-new, superior design, a design that, together with a still-in-development sorted-edge rasterizer that completely eliminates overdraw, comes remarkably close to meeting the "perfect-world" specifications we laid out at the start.

## Simplify, and Keep on Trying New Things

What does it all mean? Exactly what I said up front: Simplify, and keep trying new things. The precalculated PVS is simpler than any of the other schemes that had been considered (although precalculating the PVS is an interesting task that I'll discuss another time). In fact, at runtime the precalculated PVS is just a constrained version of the painter's algorithm. Does that mean it's not particularly profound?

Not at all. All really great designs seem simple and even obvious—once they've been designed. But the process of getting there requires incredible persistence and a willingness to try lots of different ideas until the right one falls into place, as happened here.

*My friend Chris Hecker has a theory that all approaches work out to the same thing in the end, since they all reflect the same underlying state and functionality. In terms of underlying theory, I've found that to be true; whether you do perspective texture mapping with a divide or with incremental hyperbolic calculations, the numbers do exactly the same thing. When it comes to implementation, however, my experience is that simply time-shifting an approach, or matching hardware capabilities better, or caching can make an astonishing difference.*

My friend Terje Mathisen likes to say that "almost all programming can be viewed as an exercise in caching," and that's exactly what John did. No matter how fast he made his VSD calculations, they could never be as fast as precalculating and looking up the visibility, and his most inspired move was to yank himself out of the "faster code" mindset and realize that it was in fact possible to precalculate (in effect, cache) and look up the PVS.

The hardest thing in the world is to step outside a familiar, pretty good solution to a difficult problem and look for a different, better solution. The best ways I know to do that are to keep trying new, wacky things, and always, always, always try to simplify. One of John's goals is to have fewer lines of code in each 3-D game than in the previous game, on the assumption that as he learns more, he should be able to do things better with less code.

So far, it seems to have worked out pretty well for him.

## Learn Now, Pay Forward

There's one other thing I'd like to mention before I close this chapter. Much of what I've learned, and a great deal of what I've written, has been in the pages of *Dr. Dobb's Journal.* As far back as I can remember, *DDJ* has epitomized the attitude that sharing programming information is A Good Thing. I know a lot of programmers who were able to leap ahead in their development because of Hendrix's Tiny C, or Stevens' D-Flat, or simply by browsing through *DDJ*'s annual collections. (Me, for one.) Understandably, most companies understandably view sharing information in a very different way, as potential profit lost—but that's what makes *DDJ* so valuable to the programming community.

It is in that spirit that id Software is allowing me to describe in these pages (which also appeared in one of the *DDJ* special issues) how Quake works, even before Quake has shipped. That's also why id has placed the full source code for Wolfenstein 3-D on ftp.idsoftware.com/idstuff/source; and although you can't just recompile the code and sell it, you can learn how a full-blown, successful game works. Check wolfsrc.txt in the above-mentioned directory for details on how the code may be used.

So remember, when it's legally possible, sharing information benefits us all in the long run. You can pay forward the debt for the information you gain here and elsewhere by sharing what you know whenever you can, by writing an article or book or posting on the Net. None of us learns in a vacuum; we all stand on the shoulders of giants such as Wirth and Knuth and thousands of others. Lend your shoulders to building the future!

## References

Foley, James D., *et al., Computer Graphics: Principles and Practice,* Addison Wesley, 1990, ISBN 0-201-12110-7 (beams, BSP trees, VSD).

Teller, Seth, *Visibility Computations in Densely Occluded Polyhedral Environments* (dissertation), available on http://theory.lcs.mit.edu/~seth/ along with several other papers relevant to visibility determination.

Teller, Seth, *Visibility Preprocessing for Interactive Walkthroughs,* SIGGRAPH 91 proceedings, pp. 61-69.