Blue's News Presents:

Ramblings in Realtime

by

Michael Abrash

Inside Quake: Visible-Surface Determination

by Michael Abrash

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 it; 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 or 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 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 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.

The toughest 3-D challenge of all

I've spent most of my waking hours for the last seven months working on Quake, id Software's successor to DOOM, and after spending the next three months in much the same way, I expect Quake will be out as shareware around the time you read this.

In terms of graphics, Quake is to DOOM as DOOM was to its predecessor, Wolfenstein 3D. 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. Sometime soon, I'll talk about how all that works, but this month I want to talk about what is, in my book, 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 doing realistic worlds. I expect VSD increasingly to be the 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 last time, 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 1.

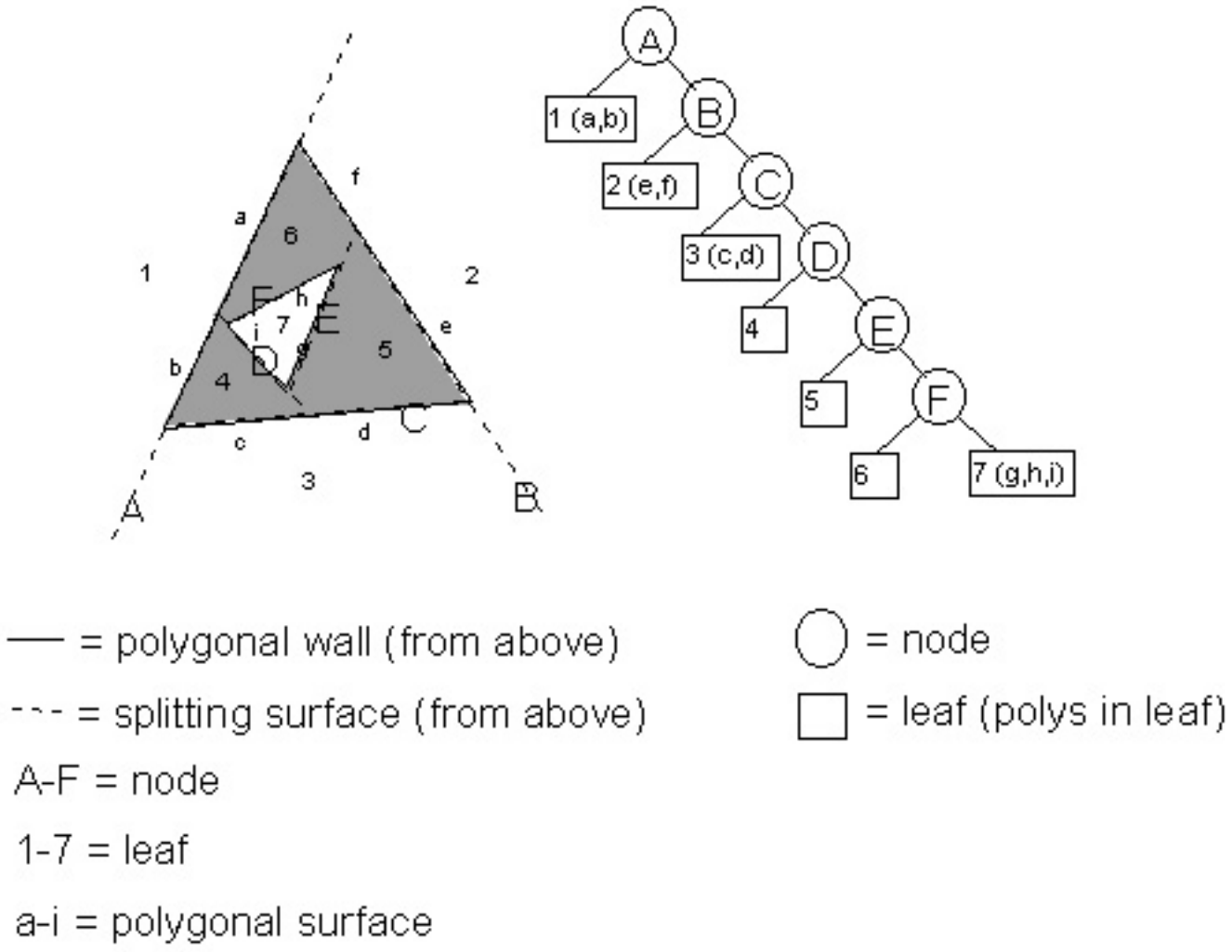


Figure 1: In Quake, polygons are stored in the empty leaves. Shaded areas are solid leaves (solid volumes, such as the insides of walls).

Correct drawing order can be obtained by drawing the leaves in front-to-back or back-to-front BSP order, again as discussed last time. 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 2, wasting no time overdrawing pixels multiple times; note how little of the polygon set in Figure 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, and the processing time would be the same for all possible viewpoints, giving the game a smooth visual flow.

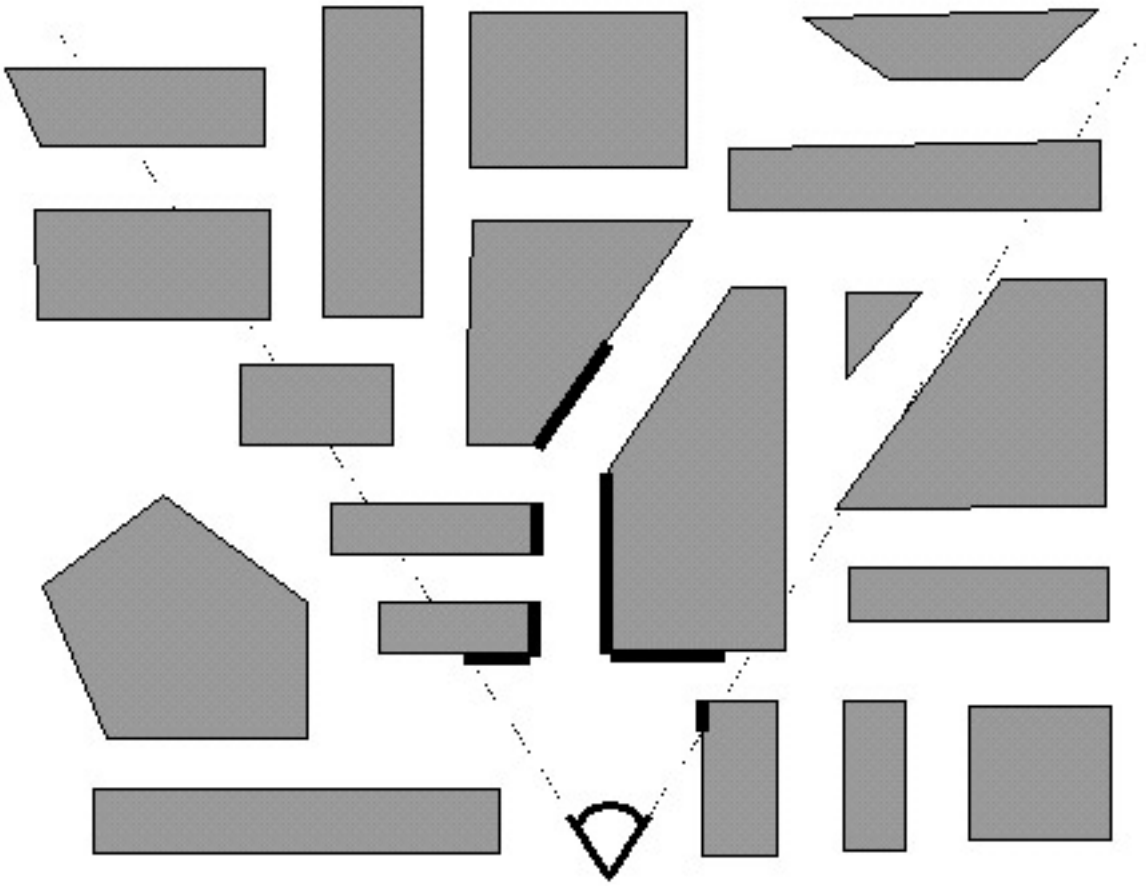


Figure 2: An ideal VSD architecture would draw only visible parts of visible polygons.

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 last time, 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 last time, 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 performance down considerably.

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 3. Another way of thinking of this is that each node splits into two pieces the subspace 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.

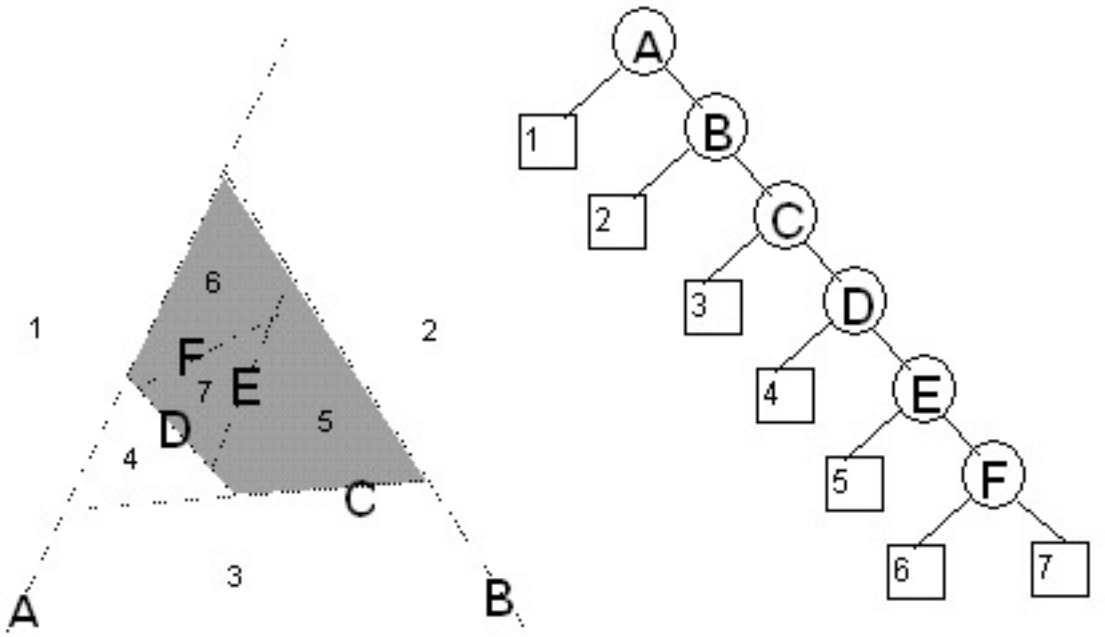


Figure 3: Node E describes the shaded subspace, which contains leaves 5, 6, and 7, and node F.

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's 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 above 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 magnitude 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, 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 Wolf 3D 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.

By three months after I arrived, only one element of the original VSD design was anywhere in sight, and John had taken "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 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 5.

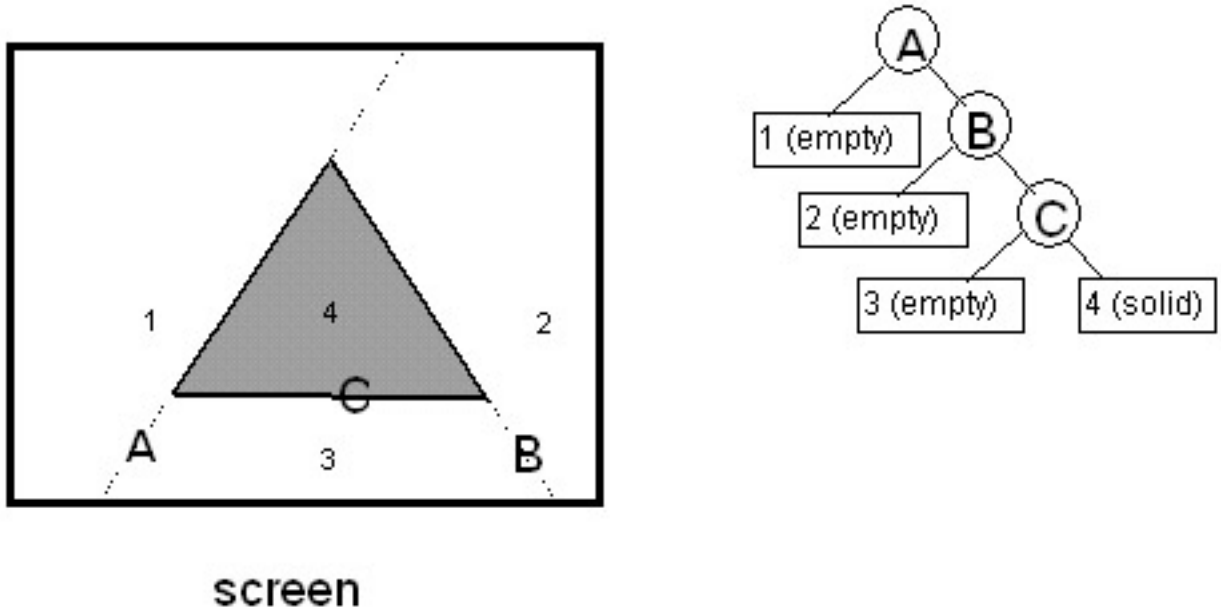


Figure 4: Quake's beam tree effectively partitioned the screen into 2-D regions.

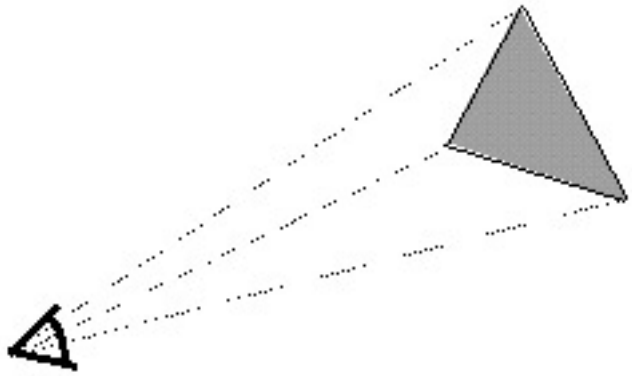


Figure 5: Quake's beam tree was composed of 3-D wedges, or beams, projecting out from the viewpoint to polygon edges.

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.

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 8x8 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.

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 are 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 AM 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 (contrary to what I said a few months back, choosing as the next splitter the polygon that splits the fewest other polygons is clearly the best heuristic, based on the latest data) 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% but ranging up to 150%, 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 up shop for this month. 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.) 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 how Quake works, even before Quake has shipped. That's also why id has placed the full source code for Wolfenstein 3D on ftp.idsoftware.com/idstuff/source; you can't just recompile the code and sell it, but 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.

Consider the Alternatives: Quake's Hidden-Surface Removal

by Michael Abrash

Okay, I admit it: I'm sick and tired of classic rock. Admittedly, it's been a while--about 20 years--since last I was 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 ten-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

Back in January, 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 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; check out the May, July, and November 1995 columns for a discussion of BSP trees), 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 question I've been asked most often since the January column came out, so I'll take a moment to address it. 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 the January column, 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 drawing and z-filling the world is done, we can simply draw the sprites and polygon models with z-buffering and get perfect sorting all around.

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 320x200 is 128K, not trivial but not a big deal for a game that

equires 8 Mb. The performance impact is about 10% for z-filling the world, and roughly 20% (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 column, I'm going to talk about the other part of the drawing equation, 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 the January column, 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% 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% 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 the January column, 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 1, and then sorting and clipping the spans against each other until only the visible portions of visible spans are left to be drawn, as shown in Figure 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 against only against some of the spans on the same line, and 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.

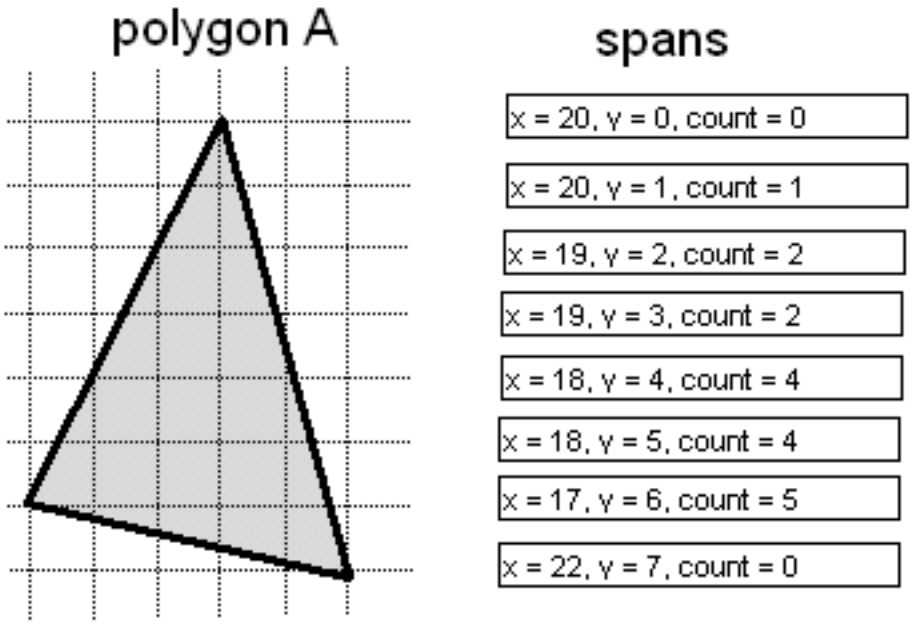


Figure 1: Span generation.

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 we'll see in the next column. 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 1, and each span is sorted and clipped into the bucket for the scan line the span is on, as shown in Figure 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.

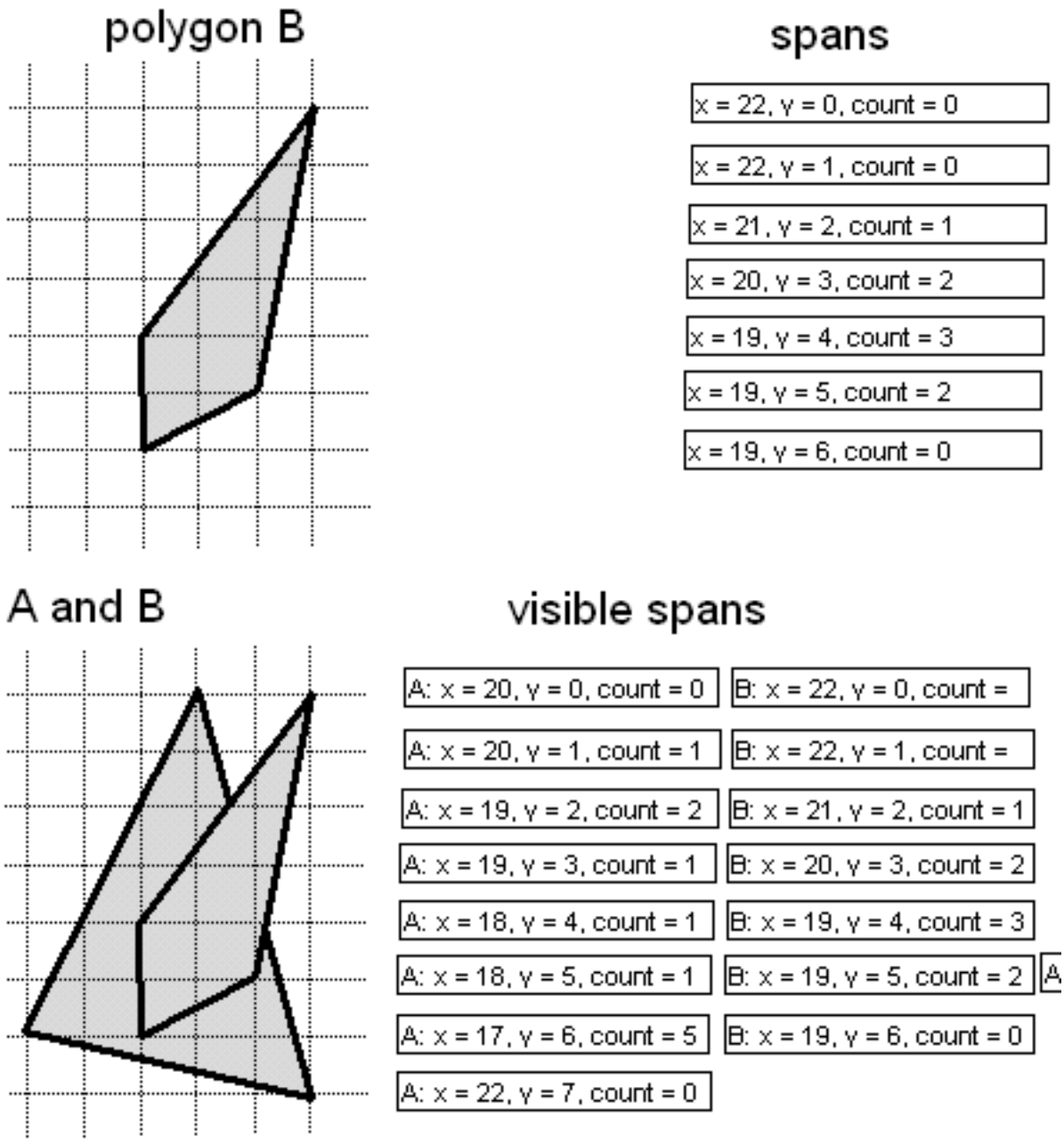


Figure 2: The spans from polygon A from Figure 1 sorted and clipped with the spans from polygon B, where polygon A is at a constant z distance of 100 and polygon B is at a constant z distance of 50 (polygon B is closer).

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 3, or deactivated and removed from the APL, as shown in Figure 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 because 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.

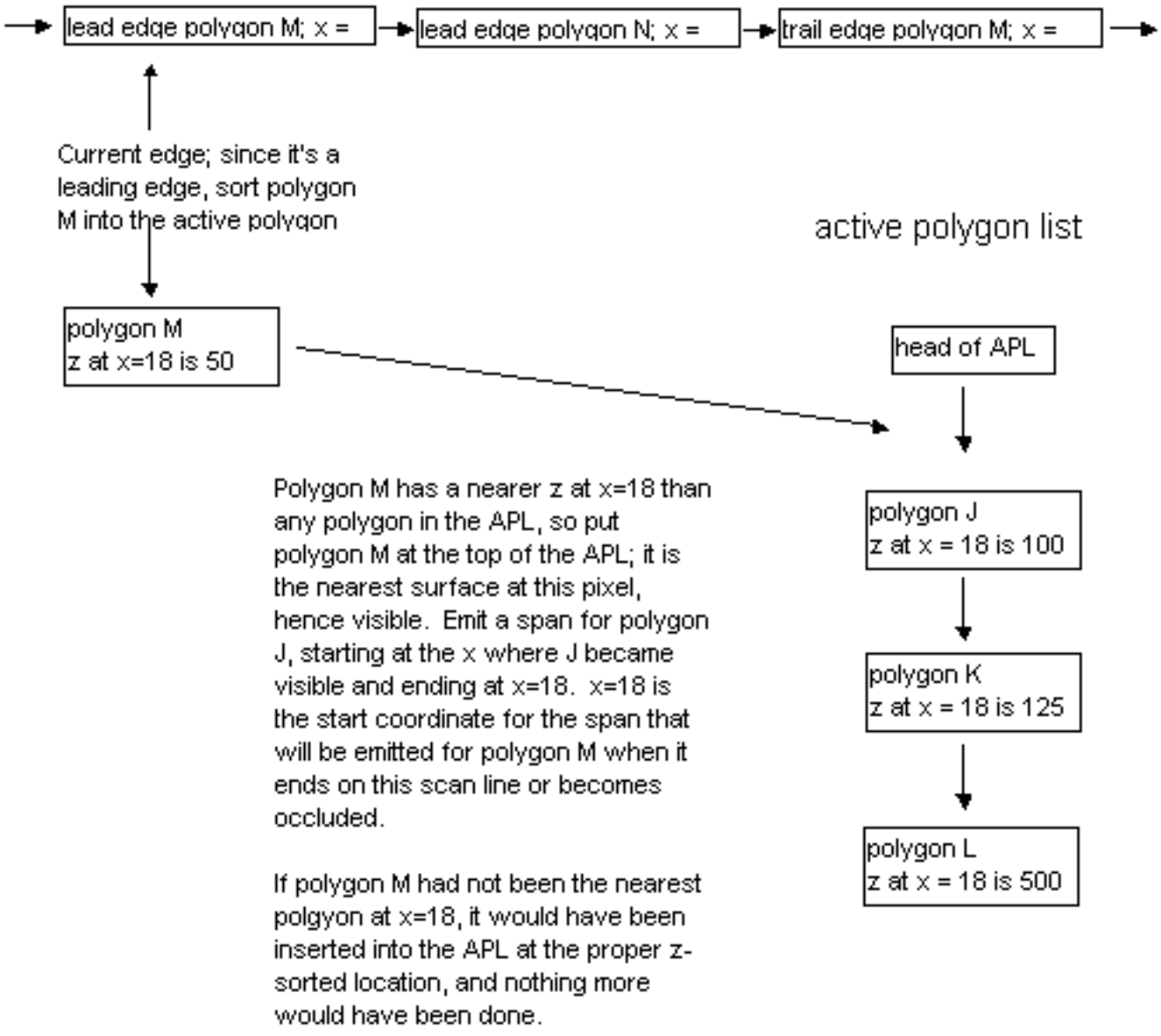


Figure 3: Activating a polygon when a leading edge is encountered in the AEL.

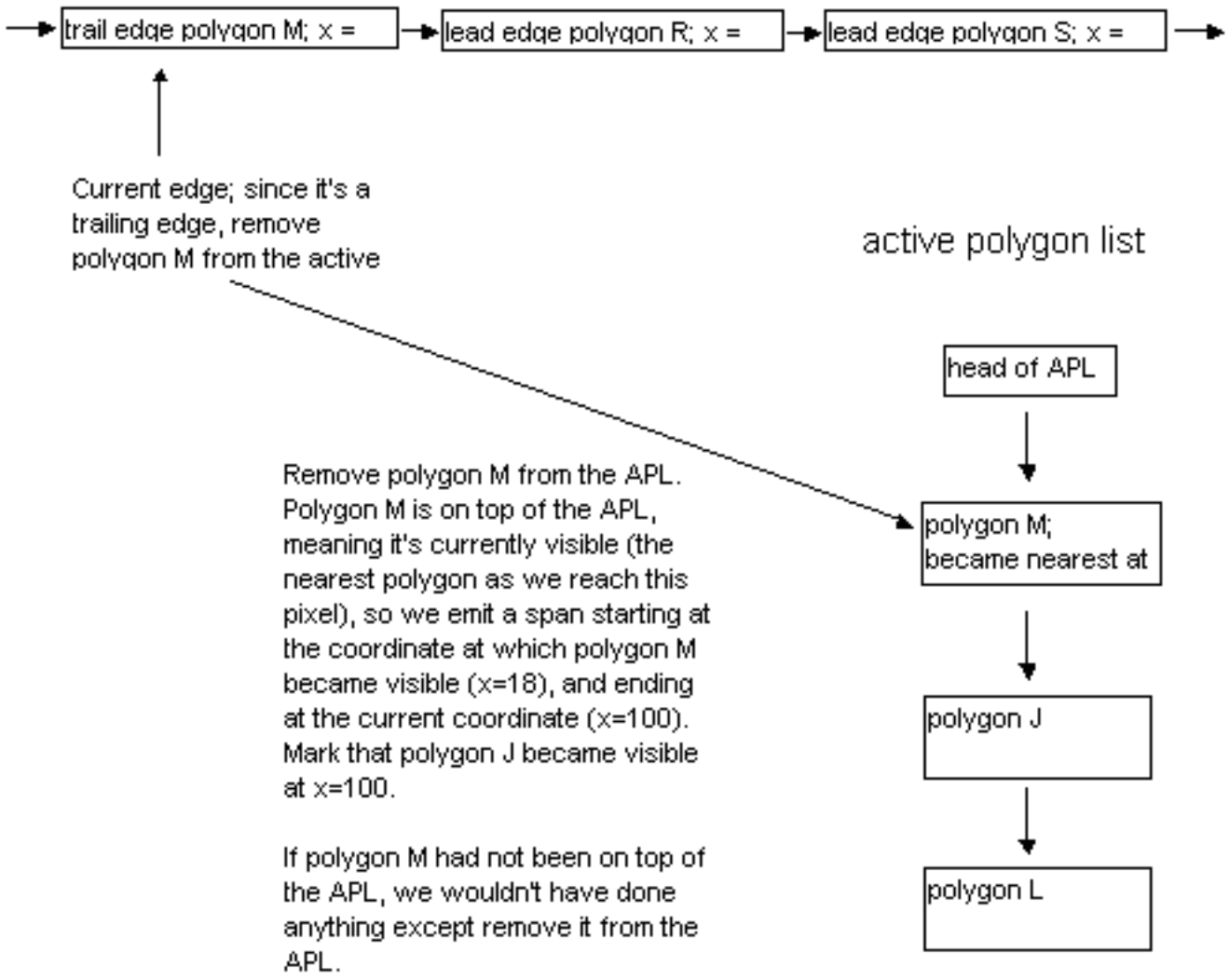


Figure 4: Deactivating a polygon when a trailing edge is encountered in the AEL.

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 column clearer. There will a thorough discussion in the next column.

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.

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 has 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 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 the next column, I'll go into considerable detail about edge-sorting, complete with a full implementation. I'm going the spend the rest of this column laying the foundation for next time by discussing sorting keys and 1/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 covered in detail; my apologies, but it's unavoidable, and all should become clear by the end of the next column.

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 we can 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 our 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 series of columns on texture mapping over the past year 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.

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 next time.

Quake and z-sorting

I mentioned above 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.

Onward to next time

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 next time should help; if you want to peek ahead, the code should be available from ftp.idsoftware.com/mikeab/ddjsort.zip by the time you read this column. You may also want to take a look at Foley & 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.

Sorted Spans in Action

by Michael Abrash

Last time, we dove headlong into the intricacies of hidden surface removal by way of z-sorted (actually, 1/z-sorted) spans. At the end, I noted that we were currently using 1/z-sorted spans in Quake, but it was unclear whether we'd switch back to BSP order. Well, it's clear now: We're back to sorting spans by BSP order.

In Robert A. Heinlein's wonderful story "The Man Who Sold the Moon," the chief engineer of the Moon rocket project tries to figure out how to get a payload of three astronauts to the Moon and back. He starts out with a four-stage rocket design, but finds that it won't do the job, so he adds a fifth stage. The fifth stage helps, but not quite enough, "Because," he explains, "I've had to add in too much dead weight, that's why." (The dead weight is the control and safety equipment that goes with the fifth stage.) He then tries adding yet another stage, only to find that the sixth stage actually results in a net slowdown. In the end, he has to give up on the three-person design and build a one-person spacecraft instead.

1/z-sorted spans in Quake turned out pretty much the same way, as we'll see in a moment. First, though, I'd like to note up front that this column is very technical and builds heavily on previously-covered material; reading the last column is strongly recommended, and reading the six columns before that, which cover BSP trees, 3-D clipping, and 3-D math, might be a good idea as well. I regret that I can't make this column stand completely on its own, but the truth is that commercial-quality 3-D graphics programming requires vastly more knowledge and code than did the 2-D graphics I've written about in years past. And make no mistake about it, this is commercial quality stuff; in fact, the code in this column uses the same sorting technique as the test version of Quake, qtest1.zip, that we just last week placed on the Internet. These columns are the Real McCoy, reports from the leading edge, and I trust that you'll be patient if careful rereading and some catch-up reading of prior columns are required to absorb everything contained herein. Besides, the ultimate reference for any design is working code, which you'll find in part in Listing 1 and in its entirety in ftp.idsoftware.com/mikeab/ddjzsort.zip.

Quake and sorted spans

As you'll recall from last time, Quake uses sorted spans to get zero overdraw while rendering the world, thereby both improving overall performance and leveling frame rates by speeding up scenes that would otherwise experience heavy overdraw. Our original design used spans sorted by BSP order; because we traverse the world BSP tree from front to back relative to the viewpoint, the order in which BSP nodes are visited is a guaranteed front to back sorting order. We simply gave each node a increasing BSP sequence number as it was visited, set each polygon's sort key to the BSP sequence number of the node (BSP splitting plane) it lay on, and used those sort keys when generating spans.

(In a change from earlier designs, polygons now are stored on nodes, rather than leaves, which are the convex subspaces carved out by the BSP tree. Visits to potentially-visible leaves are used only to mark that the polygons that touch those leaves are visible and need to be drawn, and each marked-visible polygon is then drawn after everything in front of its node has been

drawn. This results in less BSP splitting of polygons, which is A Good Thing, as explained below.)

This worked flawlessly for the world, but had a couple of downsides. First, it didn't address the issue of sorting small, moving BSP models such as doors; those models could be clipped into the world BSP tree's leaves and assigned sort keys corresponding to the leaves into which they fell, but there was still the question of how to sort multiple BSP models in the same world leaf against each other. Second, strict BSP order requires that polygons be split so that every polygon falls entirely within a single leaf. This can be stretched by putting polygons on nodes, allowing for larger polygons on average, but even then, polygons still need to be split so that every polygon falls within the bounding volume for the node on which it lies. The end result, in either case, is more and smaller polygons than if BSP order weren't used--and that, in turn, means lower performance, because more polygons must be clipped, transformed, and projected, more sorting must be done, and more spans must be drawn.

We figured that if only we could avoid those BSP splits, Quake would get a lot faster. Accordingly, we switched from sorting on BSP order to sorting on 1/z, and left our polygons unsplit. Things did get faster at first, but not as much as we had expected, for two reasons.

First, as the world BSP tree is descended, we clip each node's bounding box in turn to see if it's inside or outside each plane of the view frustum. The clipping results can be remembered, and often allow the avoidance of some or all clipping for the node's polygons. For example, all polygons in a node that has a trivially accepted bounding box are likewise guaranteed to be unclipped and in the frustum, since they all lie within the node's volume, and need no further clipping. This efficient clipping mechanism vanished as soon as we stepped out of BSP order, because a polygon was no longer necessarily confined to its node's volume.

Second, sorting on 1/z isn't as cheap as sorting on BSP order, because floating-point calculations and comparisons are involved, rather than integer compares. So Quake got faster, but, like Heinlein's fifth rocket stage, there was clear evidence of diminishing returns.

That wasn't the bad part; after all, even a small speed increase is a good thing. The real problem was that our initial 1/z sorting proved to be unreliable. We first ran into problems when two forward-facing polygons started at a common edge, because it was hard to tell which one was really in front (as discussed below), and we had to do additional floating-point calculations to resolve these cases. This fixed the problems for a while, but then odd cases started popping up where just the right combination of polygon alignments caused new sorting errors. We tinkered with those too, adding more code and incurring additional slowdowns in the process. Finally, we had everything working smoothly again, although by this point Quake was back to pretty much the same speed it had been with BSP sorting.

And then yet another crop of sorting errors popped up.

We could have fixed those errors too; we'll take a quick look at how to deal with such cases shortly. However, like the sixth rocket stage, the fixes would have made Quake slower than it had been with BSP sorting. So we gave up and went back to BSP order, and now the code is simpler and sorting works reliably. It's too bad our experiment didn't work out, but it wasn't wasted time, because we learned quite a bit. In particular, we learned that the information

provided by a simple, reliable world ordering mechanism such as a BSP tree can do more good than is immediately apparent, in terms of both performance and solid code.

Nonetheless, sorting on 1/z can be a valuable tool, used in the right context; drawing a Quake world just doesn't happen to be such a case. In fact, sorting on 1/z is how we're now handling the sorting of multiple BSP models that lie within the same world leaf in Quake; here we don't have the option of using BSP order (because we're drawing multiple independent trees), so we've set restrictions on the BSP models to avoid running into the types of 1/z sorting errors we encountered drawing the Quake world. Below, we'll look at another application in which sorting on 1/z is quite useful, one where objects move freely through space. As is so often the case in 3-D, there is no one "right" technique, but rather a great many different techniques, each one handy in the right situations. Often, a combination of techniques is beneficial, as for example the combination in Quake of BSP sorting for the world and 1/z sorting for BSP models in the same world leaf.

For the remainder of this column, I'm going to look at the three main types of 1/z span sorting, then discuss a sample 3-D app built around 1/z span sorting.

Types of 1/z span sorting

As a quick refresher, with 1/z span sorting, all the polygons in a scene are treated as sets of screenspace pixel spans, and 1/z (where z is distance from the viewpoint in viewspace, as measured along the viewplane normal) is used to sort the spans so that the nearest span overlapping each pixel is drawn. As discussed last time, in the sample program we're actually going to do all our sorting with polygon edges, which represent spans in an implicit form.

There are three types of 1/z span sorting, each requiring a different implementation. In order of increasing speed and decreasing complexity, they are: intersecting, abutting, and independent. (These are names of my own devising; I haven't come across any standard nomenclature.)

Intersecting span sorting

Intersecting span sorting occurs when polygons can interpenetrate. Thus, two spans may cross such that part of each span is visible, in which case the spans have to be split and drawn appropriately, as shown in Figure 1.

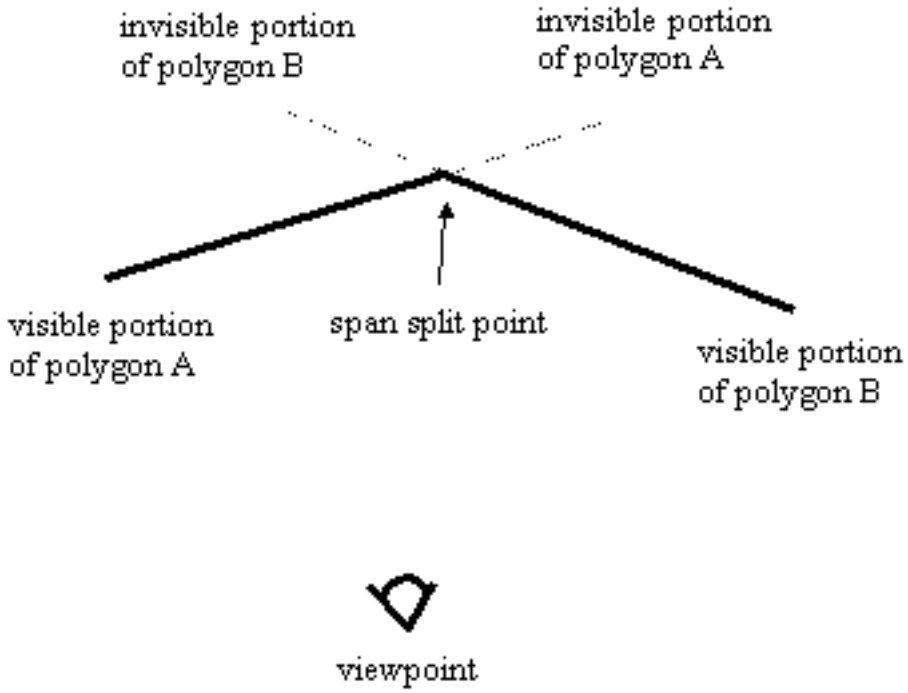


Figure 1: Intersecting span sorting. Polygons A and B are viewed from above.

Intersecting is the slowest and most complicated type of span sorting, because it is necessary to compare 1/z values at two points in order to detect interpenetration, and additional work must be done to split the spans as necessary. Thus, although intersecting span sorting certainly works, it's not the first choice for performance.

Abutting span sorting

Abutting span sorting occurs when polygons that are not part of a continuous surface can butt up against each other, but don't interpenetrate, as shown in Figure 2. This is the sorting used in Quake, where objects like doors often abut walls and floors, and turns out to be more complicated than you might think. The problem is that when an abutting polygon starts on a given scan line, as with polygon B in Figure 2, it starts at exactly the same 1/z value as the polygon it abuts, in this case, polygon A, so additional sorting is needed when these ties happen. Of course, the two-point sorting used for intersecting polygons would work, but we'd like to find something faster.

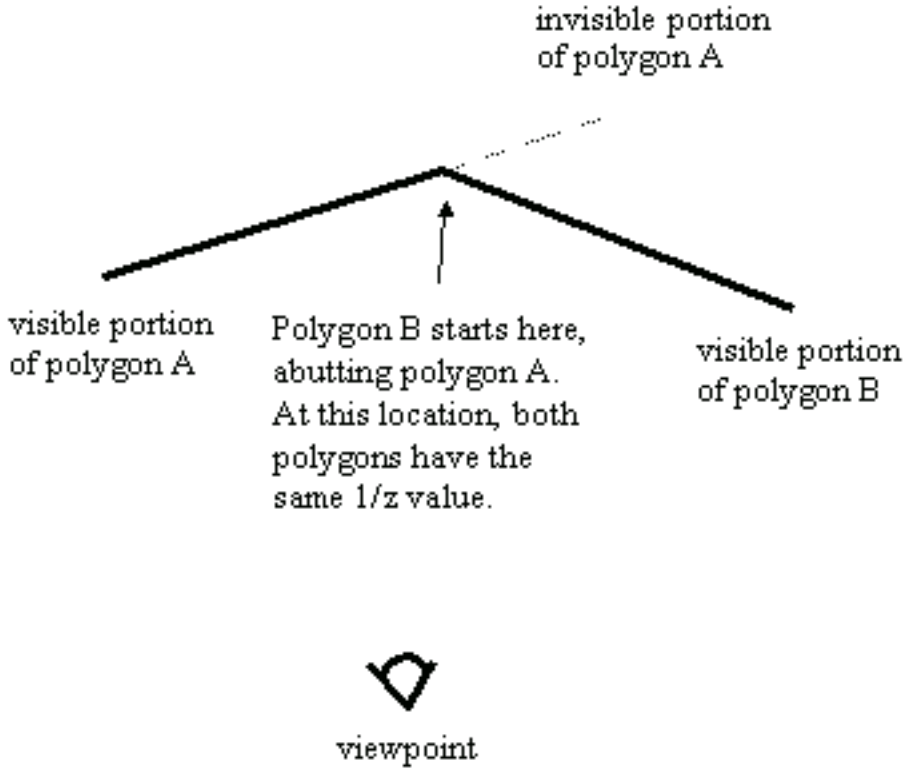


Figure 2: Abutting span sorting. Polygons A and B are viewed from above.

As it turns out, the additional sorting for abutting polygons is actually quite simple; whichever polygon has a greater 1/z gradient with respect to screen x (that is, whichever polygon is heading fastest toward the viewer along the scan line) is the front one. The hard part is identifying when ties--that is, abutting polygons--occur; due to floating-point imprecision, as well as fixed-point edge-stepping imprecision that can move an edge slightly on the screen, calculations of 1/z from the combination of screen coordinates and 1/z gradients (as discussed last time) can be slightly off, so most tie cases will show up as near matches, not exact matches. This imprecision makes it necessary to perform two comparisons, one with an adjust-up by a small epsilon and one with an adjust-down, creating a range in which near-matches are considered matches. Fine-tuning this epsilon to catch all ties without falsely reporting close-but-not-abutting edges as ties proved to be troublesome in Quake, and the epsilon calculations and extra comparisons slowed things down.

I do think that abutting 1/z span sorting could have been made reliable enough for production use in Quake, were it not that we share edges between adjacent polygons in Quake, so that the world is a large polygon mesh. When a polygon ends and is followed by an adjacent polygon that shares the edge that just ended, we simply assume that the adjacent polygon sorts relative to other active polygons in the same place as the one that ended (because the mesh is continuous and there's no interpenetration), rather than doing a 1/z sort from scratch. This speeds things up by saving a lot of sorting, but it means that if there is a sorting error, a whole string of adjacent polygons can be sorting incorrectly, pulled in by the one missorted polygon. Missorting is a very real hazard when a polygon is very nearly perpendicular to the screen, so that the 1/z calculations push the limits of numeric precision, especially in single-precision

floating point.

Many caching schemes are possible with abutting span sorting, because any given pair of polygons, being noninterpenetrating, will sort in the same order throughout a scene. However, in Quake at least, the benefits of caching sort results were outweighed by the additional overhead of maintaining the caching information, and every caching variant we tried actually slowed Quake down.

Independent span sorting

Finally, we come to independent span sorting, the simplest and fastest of the three, and the type the sample code in Listing 1 uses. Here, polygons never intersect or touch any other polygons except adjacent polygons with which they form a continuous mesh. This means that when a polygon starts on a scan line, a single 1/z comparison between that polygon and the polygons it overlaps on the screen is guaranteed to produce correct sorting, with no extra calculations or tricky cases to worry about.

Independent span sorting is ideal for scenes with lots of moving objects that never actually touch each other, such as a space battle. Next, we'll look at an implementation of independent 1/z span sorting.

1/z span sorting in action

Listing 1 is a portion of a program that demonstrates independent 1/z span sorting. This program is based on the sample 3-D clipping program from the March column; however, the earlier program did hidden surface removal (HSR) by simply z-sorting whole objects and drawing them back to front, while Listing 1 draws all polygons by way of a 1/z-sorted edge list. Consequently, where the earlier program worked only so long as object centers correctly described sorting order, Listing 1 works properly for all combinations of non-intersecting and non-abutting polygons. In particular, Listing 1 correctly handles concave polyhedra; a new L-shaped object (the data for which is not included in Listing 1) has been added to the sample program to illustrate this capability. The ability to handle complex shapes makes Listing 1 vastly more useful for real-world applications than the earlier 3-D clipping demo.

By the same token, Listing 1 is quite a bit more complicated than the earlier code. The earlier code's HSR consisted of a z-sort of objects, followed by the drawing of the objects in back-to-front order, one polygon at a time. Apart from the simple object sorter, all that was needed was backface culling and a polygon rasterizer.

Listing 1 replaces this simple pipeline with a three-stage HSR process. After backface culling, the edges of each of the polygons in the scene are added to the global edge list, by way of AddPolygonEdges(). After all edges have been added, the edges are turned into spans by ScanEdges(), with each pixel on the screen being covered by one and only one span (that is, there's no overdraw). Once all the spans have been generated, they're drawn by DrawSpans(), and rasterization is complete.

There's nothing tricky about AddPolygonEdges(), and DrawSpans(), as implemented in Listing 1, is very straightforward as well. In an implementation that supported texture mapping, however, all the spans wouldn't be put on one global span list and drawn at once, as is done in

Listing 1, because that would result in drawing spans from all the surfaces in no particular order. (A surface is a drawing object that's originally described by a polygon, but in ScanEdges() there is no polygon in the classic sense of a set of vertices bounding an area, but rather just a set of edges and a surface that describes how to draw the spans outlined by those edges.) That would mean constantly skipping from one texture to another, which in turn would hurt processor cache coherency a great deal, and would also incur considerable overhead in setting up gradient and perspective calculations each time a surface was drawn. In Quake, we have a linked list of spans hanging off each surface, and draw all the spans for one surface before moving on to the next surface.

The core of Listing 1, and the most complex aspect of 1/z-sorted spans, is ScanEdges(), where the global edge list is converted into a set of spans describing the nearest surface at each pixel. This process is actually pretty simple, though, if you think of it as follows.

For each scan line, there is a set of active edges, those edges that intersect the scan line. A good part of ScanEdges() is dedicated to adding any edges that first appear on the current scan line (scan lines are processed from the top scan line on the screen to the bottom), removing edges that reach their bottom on the current scan line, and x-sorting the active edges so that the active edges for the next scan can be processed from left to right. All this is per-scan-line maintenance, and is basically just linked list insertion, deletion, and sorting.

The heart of the action is the loop in ScanEdges() that processes the edges on the current scan line from left to right, generating spans as needed. The best way to think of this loop is as a surface event processor, where each edge is an event with an associated surface. Each leading edge is an event marking the start of its surface on that scan line; if the surface is nearer than the current nearest surface, then a span ends for the nearest surface, and a span starts for the new surface. Each trailing edge is an event marking the end of its surface; if its surface is currently nearest, then a span ends for that surface, and a span starts for the next-nearest surface (the surface with the next-largest 1/z at the coordinate where the edge intersects the scan line). One handy aspect of this event-oriented processing is that leading and trailing edges do not need to be explicitly paired, because they are implicitly paired by pointing to the same surface. This saves the memory and time that would otherwise be needed to track edge pairs.

One more element is required in order for ScanEdges() to work efficiently. Each time a leading or trailing edge occurs, it must be determined whether its surface is nearest (at a larger 1/z value than any currently active surface); in addition, for leading edges, the currently topmost surface must be known, and for trailing edges, it may be necessary to know the currently next-to-topmost surface. The easiest way to accomplish this is with a surface stack; that is, a linked list of all currently active surfaces, starting with the nearest surface and progressing toward the farthest surface, which, as described below, is always the background surface. (The operation of this sort of edge event-based stack was described and illustrated in the May column.) Each leading edge causes its surface to be 1/z-sorted into the surface stack, with a span emitted if necessary. Each trailing edge causes its surface to be removed from the surface stack, again with a span emitted if necessary. As you can see from Listing 1, it takes a fair bit of code to implement this, but all that's really going on is a surface stack driven by edge events.

Implementation notes

Finally, a few notes on Listing 1. First, you'll notice that although we clip all polygons to the view frustum in worldspace, we nonetheless later clamp them to valid screen coordinates before adding them to the edge list. This catches any cases where arithmetic imprecision results in clipped polygon vertices that are a bit outside the frustum. I've only found such imprecision to be significant at very small z distances, so clamping would probably be unnecessary if there were a near clip plane, and might not even be needed in Listing 1, because of the slight nudge inward that we give the frustum planes, as described in the March column. However, my experience has consistently been that relying on worldspace or viewspace clipping to produce valid screen coordinates 100 percent of the time leads to sporadic and hard-to-debug errors.

There is no separate clear of the background in Listing 1. Instead, a special background surface at an effectively infinite distance is added, so whenever no polygons are active the background color is drawn. If desired, it's a simple matter to flag the background surface and draw the background specially. For example, the background could be drawn as a starfield or a cloudy sky.

The edge-processing code in Listing 1 is fully capable of handling concave polygons as easily as convex polygons, and can handle an arbitrary number of vertices per polygon, as well. One change is need needed for the latter case: Storage for the maximum number of vertices per polygon must be allocated in the polygon structures. In a fully polished implementation, vertices would be linked together or pointed to, and would be allocated dynamically from a vertex pool, so each polygon wouldn't have to contain enough space for the maximum possible number of vertices.

Each surface has a field named **state**, which is incremented when a leading edge for that surface is encountered, and decremented when a trailing edge is reached. A surface is activated by a leading edge only if **state** increments to 1, and is deactivated by a trailing edge only if **state** decrements to 0. This is another guard against arithmetic problems, in this case quantization during the conversion of vertex coordinates from floating point to fixed point. Due to this conversion, it is possible, although rare, for a polygon that is viewed nearly edge-on to have a trailing edge that occurs slightly before the corresponding leading edge, and the span-generation code will behave badly if it tries to emit a span for a surface that hasn't started yet. It would help performance if this sort of fix-up could be eliminated by careful arithmetic, but I haven't yet found a way to do so for 1/z-sorted spans.

Lastly, as discussed last time, Listing 1 uses the gradients for 1/z with respect to changes in screen x and y to calculate 1/z for active surfaces each time a leading edge needs to be sorted into the surface stack. The natural origin for gradient calculations is the center of the screen, which is (x,y) coordinate (0,0) in viewspace. However, when the gradients are calculated in AddPolygonEdges(), the origin value is calculated at the upper left corner of the screen. This is done so that screen x and y coordinates can be used directly to calculate 1/z, with no need to adjust the coordinates to be relative to the center of the screen. Also, the screen gradients grow more extreme as a polygon is viewed closer to edge-on. In order to keep the gradient calculations from becoming meaningless or generating errors, a small epsilon is applied to backface culling, so that polygons that are very nearly edge-on are culled. This calculation

would be more accurate if it were based directly on the viewing angle, rather than on the dot product of a viewing ray to the polygon with the polygon normal, but that would require a square root, and in my experience the epsilon used in Listing 1 works fine.

Bretton Wade's BSP Web page has moved

A while back, I mentioned that Bretton Wade was constructing a promising Web site on BSPs. He has moved that site, which has grown to contain a lot of useful information, to http://www.qualia.com/bspfaq/; alternatively, mail bspfaq@qualia.com with a subject line of "help".

Quake's Lighting Model: Surface Caching

by Michael Abrash

It was during my senior year in college that I discovered computer games. Not Wizardry, or Choplifter, or Ultima, because none of those existed yet--the game that hooked me was the original Star Trek game, in which you navigated from one 8x8 quadrant to another in search of starbases, occasionally firing phasers or photon torpedoes. This was less exciting than it sounds; after each move, the current quadrant had to be reprinted from scratch, along with the current stats--and the output device was a 10 cps printball console. A typical game took over an hour, during which nothing particularly stimulating ever happened (Klingons appeared periodically, but they politely waited for your next move before attacking, and your photon torpedoes never missed, so the outcome was never in doubt), but none of that mattered; nothing could detract from the sheer thrill of being in a computer-simulated universe.

Then the college got a PDP-11 with four CRT terminals, and suddenly Star Trek could redraw in a second instead of a minute. Better yet, I found the source code for the Star Trek program in the recesses of the new system, the first time I'd ever seen any real-world code other than my own, and excitedly dove into it. One evening, as I was looking through the code, a really cute girl at the next terminal asked me for help getting a program to run. After I had helped her, eager to get to know her better, I said, "Want to see something? This is the actual source for the Star Trek game!" and proceeded to page through the code, describing each subroutine. We got to talking, and eventually I worked up the nerve to ask her out. She said sure, and we ended up having a good time, although things soon fell apart because of her two or three other boyfriends (I never did get an exact count). The interesting thing, though, was her response when I finally got around to asking her out. She said, "It's about time!" When I asked what she meant, she said, "I've been trying to get you to ask me out all evening--but it took you forever! You didn't actually think I was interested in that Star Trek program, did you?"

Actually, yes, I had thought that, because I was interested in it. One thing I learned from that experience, and have had reinforced countless times since, is that we--you, me, anyone who programs because they love it, who would do it for free if necessary--are a breed apart. We're different, and luckily so; while everyone else is worrying about downsizing, we're in one of the hottest industries in the world. And, so far as I can see, the biggest reason we're in such a good situation isn't intelligence, or hard work, or education, although those help; it's that we actually like this stuff.

It's important to keep it that way. I've seen far too many people start to treat programming like a job, forgetting the joy of doing it, and burn out. So keep an eye on how you feel about the programming you're doing, and if it's getting stale, it's time to learn something new; there's plenty of interesting programming of all sorts to be done. Follow your interests--and don't forget to have fun!

Lighting

As I've mentioned in previous columns, I've spent the last year and a half working with John Carmack on Quake's 3-D graphics engine. John faced several fundamental design issues while architecting Quake. I've written in past columns about some of those issues, including eliminating non-visible polygons quickly via a precalculated potentially visible set (PVS), and

improving performance by inserting potentially visible polygons into a global edge list and scanning out only the nearest polygon at each pixel.

For the rest of this column, I'm going to talk about another, equally crucial design issue: how we developed our lighting approach for the part of the Quake engine that draws the world itself, the static walls and floors and ceilings. Monsters and players are drawn using completely different rendering code, with speed the overriding factor. A primary goal for the world, on the other hand, was to be as precise as possible, getting everything right so that polygons, textures, and sophisticated lighting would be pegged in place, with no visible shifting or distortion under all viewing conditions, for maximum player immersion--all with good performance, of course. As I'll discuss, the twin goals of performance and rock-solid, complex lighting proved to be difficult to achieve with traditional lighting approaches; ultimately, a dramatically different approach was required.

Gouraud shading

The traditional way to do realistic lighting in polygon pipelines is Gouraud shading (also known as smooth shading). Gouraud shading involves generating a lighting value at each polygon vertex by applying all relevant world lighting, linearly interpolating between lighting values down the edges of the polygon, and then linearly interpolating between the edges of polygon across each span. If texture mapping is desired (all polygons are texture mapped in Quake), then at each pixel in each span, the pixel's corresponding texture map location (texel) is determined, and the interpolated lighting is applied to the texel to generate a final, lit pixel. Texels are generally taken from a 32x32 or 64x64 texture that's tiled repeatedly across the polygon, for several reasons: Performance (a 64x64 texture sits nicely in the 486 or Pentium cache), database size, and less artwork.

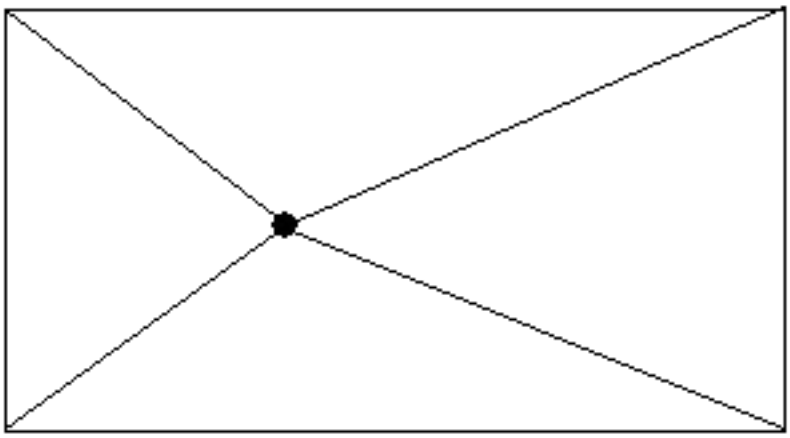
The interpolated lighting can consist of either a color intensity value or three separate red, green, and blue values. RGB lighting produces more sophisticated results, such as colored lights, but is slower and best suited to RGB modes. Games like Quake that are targeted at palettized 256-color modes generally use intensity lighting; each pixel is lit by looking up the pixel color in a table, using the texel color and the lighting intensity as the look-up indices.

Gouraud shading allows for decent lighting effects with a relatively small amount of calculation and a compact data set that's a simple extension of the basic polygon model. However, there are several important drawbacks to Gouraud shading, as well.

Problems with Gouraud shading

The quality of Gouraud shading depends heavily on the average size of the polygons being drawn. Linear interpolation is used, so highlights can only occur at vertices, and color gradients are monotonic across the face of each polygon. This can make for bland lighting effects if polygons are large, and makes it difficult to do spotlights and other detailed or dramatic lighting effects. After John brought the initial, primitive Quake engine up using Gouraud shading for lighting, the first thing he tried to improve lighting quality was adding a single vertex and creating new polygons wherever a spotlight was directly overhead a polygon, with the new vertex added directly underneath the light, as shown in Figure One. This produced fairly attractive highlights, but simultaneously made evident several problems.

Wall is a single polygon before adding a light vertex



Wall becomes four polygons after adding a light vertex directly beneath a light

Figure One: Adding an extra vertex directly beneath a light.

A primary problem with Gouraud shading is that it requires the vertices used for world geometry to serve as lighting sample points as well, even though there isn't necessarily a close relationship between lighting and geometry. This artificial coupling often forces the subdivision of a single polygon into several polygons purely for lighting reasons, as with the spotlights mentioned above; these extra polygons increase the world database size, and the extra transformations and projections that they induce can harm performance considerably.

Similar problems occur with overlapping lights, and with shadows, where additional polygons are required in order to approximate lighting detail well. In particular, good shadow edges need small polygons, because otherwise the gradient between light and dark gets spread across too wide an area. Worse still, the rate of lighting change across a shadow edge can vary considerably as a function of the geometry the edge crosses; wider polygons stretch and diffuse the transition between light and shadow. A related problem is that lighting discontinuities can be very visible at t-junctions (although ultimately we had to add edges to eliminate t-junctions anyway, because otherwise dropouts can occur along polygon edges).

These problems can be eased by adding extra edges, but that increases the rasterization load.

Another problem is that Gouraud shading isn't perspective correct. With Gouraud shading, lighting varies linearly across the face of a polygon, in equal increments per pixel--but unless the polygon is parallel to the screen, the same sort of perspective correction is needed to step lighting across the polygon properly as is required for texture mapping. Lack of perspective correction is not as visibly wrong for lighting as it is for texture mapping, because smooth lighting gradients can tolerate considerably more warping than can the detailed bitmapped images used in texture mapping, but it nonetheless shows up in several ways.

First, the extent of the mismatch between Gouraud shading and perspective lighting varies with the angle and orientation of the polygon being lit. As a polygon turns to become more on-edge, for example, the lighting warps more and therefore shifts relative to the perspective-texture mapped texels it's shading, an effect I'll call viewing variance. Lighting can similarly shift as a result of clipping, for example if one or more polygon edges are completely clipped; I'll refer to this as clipping variance.

These are fairly subtle effects; more pronounced is the rotational variance that occurs when Gouraud shading any polygon with more than three vertices. Consistent lighting for a polygon is fully defined by three lighting values; taking four or more vertices and interpolating between them, as Gouraud shading does, is basically a hack, and does not reflect any consistent underlying model. If you view a Gouraud-shaded quad head-on, then rotate it like a pinwheel, the lighting will shift as the quad turns, as shown in Figure Two. The extent of the lighting shift can be quite drastic, depending on how different the colors at the vertices are.

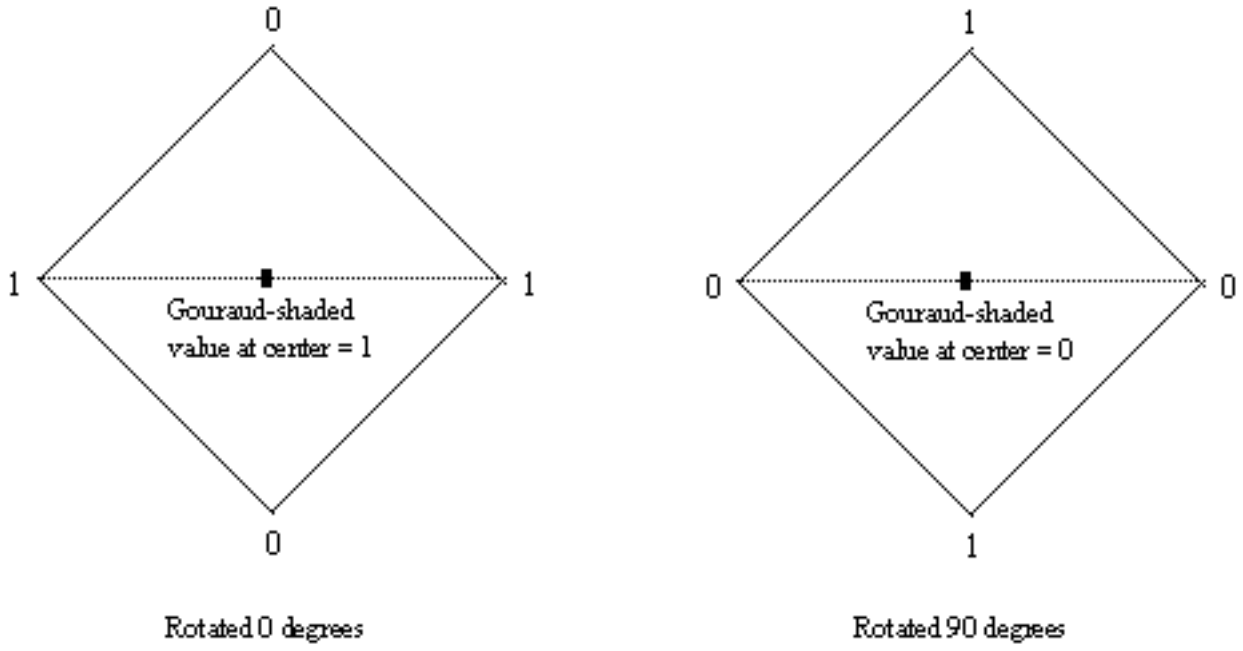


Figure Two: Gouraud shading varies with polygon screen orientation.

It was rotational variance that finally brought the lighting issue to a head for Quake. We'd look at the floors, which were Gouraud-shaded quads; then we'd pivot, and the lighting would

shimmy and shift, especially where there were spotlights and shadows. Given the goal of rendering the world as accurately and convincingly as possible, this was unacceptable.

The obvious solution to rotational variance is to use only triangles, but that brings with it a new set of problems. It takes twice as many triangles as quads to describe the same scene, increasing the size of the world database and requiring extra rasterization, at a performance cost. Triangles still don't provide perspective lighting; their lighting is rotationally invariant, but it's still wrong--just more consistently wrong. Gouraud-shaded triangles still result in odd lighting patterns, and require lots of triangles to support shadowing and other lighting detail. Finally, triangles don't solve clipping or viewing variance.

Yet another problem is that while it may work well to add extra geometry so that spotlights and shadows show up well, that's feasible only for static lighting. Dynamic lighting--light cast by sources that move--has to work with whatever geometry the world has to offer, because its needs are constantly changing.

These issues led us to conclude that if we were going to use Gouraud shading, we would have to build Quake levels from many small triangles, with sufficiently finely-detailed geometry so that complex lighting could be supported and the inaccuracies of Gouraud shading wouldn't be too noticeable. Unfortunately, that line of thinking brought us back to the problem of a much larger world database and a much heavier rasterization load (all the worse because Gouraud shading requires an additional interpolant, slowing the inner rasterization loop), so that not only would the world still be less than totally solid, because of the limitations of Gouraud shading, but the engine would also be too slow to support the complex worlds we had hoped for in Quake.

The quest for alternative lighting

None of which is to say that Gouraud shading isn't useful in general. Descent uses it to excellent effect, and in fact Quake uses Gouraud shading for moving entities, because these consist of small triangles and are always in motion, which helps hide the relatively small lighting errors. However, Gouraud shading didn't seem capable of meeting our design goals for rendering quality and speed for drawing the world as a whole, so it was time to look for alternatives.

There are many alternative lighting approaches, most of them higher-quality than Gouraud, starting with Phong shading, in which the surface normal is interpolated across the polygon's surface, and going all the way up to ray-tracing lighting techniques in which full illumination calculations are performed for all direct and reflected paths from each light sources for each pixel. What all these approaches have in common is that they're slower than Gouraud shading, too slow for our purposes in Quake. For weeks, we kicked around and rejected various possibilities and continued working with Gouraud shading for lack of a better alternative--until the day John came into work and said, "You know, I have an idea..."

Decoupling lighting from rasterization

John's idea came to him while was looking at a wall that had been carved into several pieces because of a spotlight, with an ugly lighting glitch due to a t-junction. He thought to himself that if only there were some way to treat it as one surface, it would look better and draw faster--and

then he realized that there was a way to do that.

The insight was to split lighting and rasterization into two separate steps. In a normal Gouraud-based rasterizer, there's first an off-line preprocessing step when the world database is built, during which polygons are added to support additional lighting detail as needed, and lighting values are calculated at the vertices of all polygons. At runtime, the lighting values are modified if dynamic lighting is required, and then the polygons are drawn with Gouraud shading.

Quake's approach, which I'll call surface-based lighting, preprocesses differently, and adds an extra rendering step. During off-line preprocessing, a grid, called a light map, is calculated for each polygon in the world, with a lighting value every 16 texels horizontally and vertically. This lighting is done by casting light from all the nearby lights in the world to each of the grid points on the polygon, and summing the results for each grid point. The Quake preprocessor filters the values, so shadow edges don't have a stairstep appearance (a technique suggested by Billy Zelsnack); additional preprocessing could be done, for example Phong shading to make surfaces appear smoothly curved. Then, at runtime, the polygon's texture is tiled into a buffer, with each texel lit according to the weighted average intensities of the four nearest light map points, as shown in Figure Three. If dynamic lighting is needed, the light map is modified accordingly before the buffer, which I'll call a surface, is built. Then the polygon is drawn with perspective texture mapping, with the surface serving as the input texture, and with no lighting performed during the texture mapping.

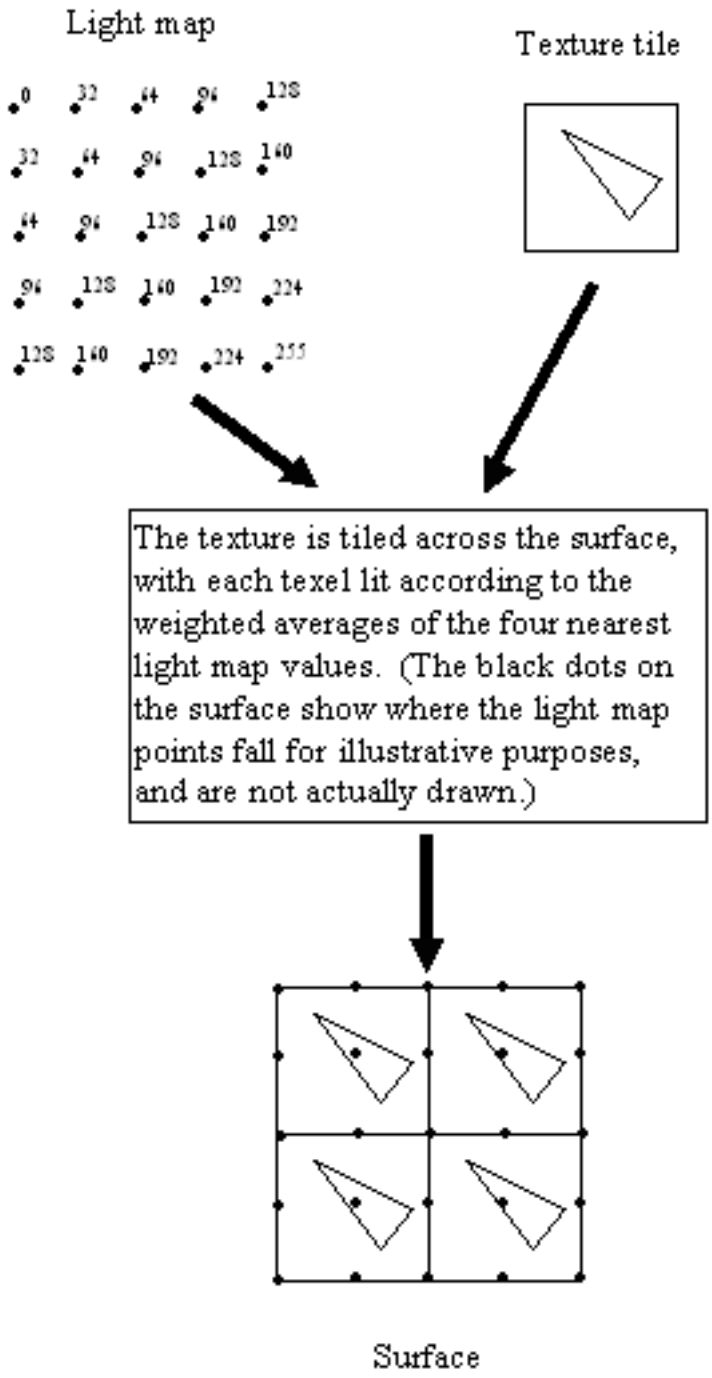


Figure Three: A surface is built by tiling the texture and lighting the texels from the light map.

So what does surface-based lighting buy us? First and foremost, it provides consistent, perspective-correct lighting, eliminating all rotational, viewing, and clipping variance, because lighting is done in surface space rather than in screen space. By lighting in surface space, we bind the lighting to the texels in an invariant way, and then the lighting gets a free ride through the perspective texture mapper and ends up perfectly matched to the texels. Surface-based lighting also supports good, although not perfect, detail for overlapping lights and shadows. The 16-texel grid has a resolution of two feet in the Quake frame of reference, and this relatively fine resolution, together with the filtering performed when the light map is built, is

sufficient to support complex shadows with smoothly fading edges. Additionally, surface-based lighting eliminates lighting glitches at t-junctions, because lighting is unrelated to vertices. In short, surface-based lighting meets all of Quake's visual quality goals, which leaves only one question: How does it perform?

Size and speed

As it turns out, the raw speed of surface-based lighting is pretty good. Although an extra step is required to build the surface, moving lighting and tiling into a separate loop from texture mapping allows each of the two loops to be optimized very effectively, with almost all variables kept in registers. The surface-building inner loop is particular efficient, because it consists of nothing more than interpolating intensity, combining it with a texel and using the result to look up a lit texel color, and storing the results with a dword write every four texels. In assembly language, we've gotten this code down to 2.25 cycles per lit texel in Quake. Similarly, the texture-mapping inner loop, which overlaps an FDIV for floating-point perspective correction with integer pixel drawing in 16-pixel bursts, has been squeezed down to 7.5 cycles per pixel on a Pentium, so the combined inner loop times for building and drawing a surface is roughly in the neighborhood of 10 cycles per pixel. It's certainly possible to write a Gouraud-shaded perspective-correct texture mapper that's somewhat faster than 10 cycles, but 10 cycles/pixel is fast enough to do 40 frames/second at 640x400 on a Pentium/100, so the cycle counts of surface-based lighting are acceptable. It's worth noting that it's possible to write a one-pass texture mapper that does approximately perspective-correct lighting. However, I have yet to hear of or devise such an inner loop that isn't complicated and full of special cases, which makes it hard to optimize; worse, this approach doesn't work well with the procedural and post-processing techniques I'll discuss shortly.

Moreover, surface-based lighting tends to spend more of its time in inner loops, because polygons can have any number of sides and don't need to be split into multiple smaller polygons for lighting purposes; this reduces the amount of transformation and projection that are required, and makes polygon spans longer. So the performance of surface-based lighting stacks up very well indeed--except for caching.

I mentioned earlier that a 64x64 texture tile fits nicely in the processor cache. A typical surface doesn't. Every texel in every surface is unique, so even at 320x200 resolution, something on the rough order of 64,000 texels must be read in order to draw a single scene. (The number actually varies quite a bit, as discussed below, but 64,000 is in the ballpark.) This means that on a Pentium, we're guaranteed to miss the cache once every 32 texels, and the number can be considerably worse than that if the texture access patterns are such that we don't use every texel in a given cache line before that data gets thrown out of the cache. Then, too, when a surface is built, the surface buffer won't be in the cache, so the writes will be uncached writes that have to go to main memory, then get read back from main memory at texture mapping time, potentially slowing things further still. All this together makes the combination of surface building and unlit texture mapping a potential performance problem, but that never posed a problem during the development of Quake, thanks to surface caching.

Surface caching

When he thought of surface-based lighting, John immediately realized that surface building

would be relatively expensive. (In fact, he assumed it would be considerably more expensive than it actually turned out to be with full assembly-language optimization.) Consequently, his design included the concept of caching surfaces, so that if the same surface was visible in the next frame, it could be reused without having to be rebuilt.

With surface rebuilding needed only rarely, thanks to surface caching, Quake's rasterization speed is generally the speed of the unlit, perspective-correct texture-mapping inner loop, which suffers from more cache misses than Gouraud-shaded, tiled texture mapping, but doesn't have the overhead of Gouraud shading, and allows the use of larger polygons. In the worst case, where everything in a frame is a new surface, the speed of the surface-caching approach is somewhat slower than Gouraud shading, but generally surface caching provides equal or better performance, so once surface caching was implemented in Quake, performance was no longer a problem--but size became a concern.

The amount of memory required for surface caching looked forbidding at first. Surfaces are large relative to texture tiles, because every texel of every surface is unique. Also, a surface can contain many texels relative to the number of pixels actually drawn on the screen, because due to perspective foreshortening, distant polygons have only a few pixels relative to the surface size in texels. Surfaces associated with partly hidden polygons must be fully built, even though only part of the polygon is visible, and if polygons are drawn back to front with overdraw, some polygons won't even be visible, but will still require surface building and caching. What all this meant was that the surface cache initially looked to be very large, on the order of several megabytes, even at 320x200--too much for a game intended to run on an 8 Mb machine.

Mipmapping to the rescue

Two factors combined to solve this problem. First, polygons are drawn through an edge list with no overdraw, as discussed a few columns back, so no surface is ever built unless at least part of it is visible. Second, surfaces are built at four mipmap levels, depending on distance, with each mipmap level having one-quarter as many texels as the preceding level, as shown in Figure Four. The mipmap level for a given surface is selected to result in a texel:pixel ratio approximately between 1:1 and 1:2, so texels map roughly to pixels, and more distant surfaces are correspondingly smaller. As a result, the number of surface texels required to draw a scene at 320x200 is on the rough order of 64,000; the number is actually somewhat higher, because of portions of surfaces that are obscured and viewspace-tilted polygons, which have high texel-to-pixel ratios along one axis, but not a whole lot higher. Thanks to mipmapping and the edge list, 600K has proven to be plenty for the surface cache at 320x200, even in the most complex scenes, and at 640x480, a little more than 1 Mb suffices.

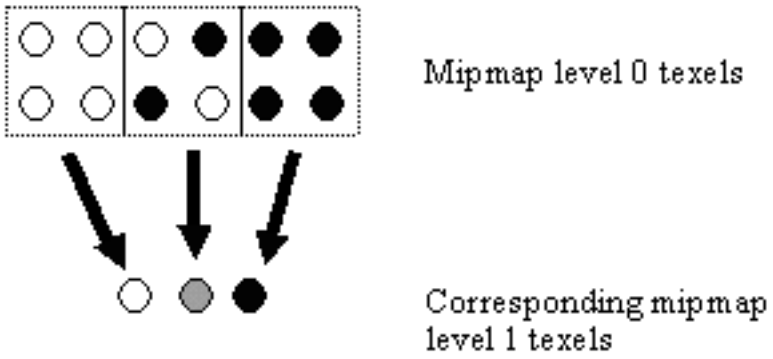


Figure Four: Each texel at a given mipmap level corresponds to four texels at the preceding mipmap level.

All mipmapped texture tiles are generated as a preprocessing step, and loaded from disk at runtime. One interesting point is that a key to making mipmapping look good turned out to be box-filtering down from one level to the next by averaging four adjacent pixels, then using error diffusion dithering to generate the mipmapped texels.

Also, mipmapping is done on a per-surface basis; the mipmap level for a whole surface is selected based on the distance from the viewer of the nearest vertex. This led us to limit surface size to a maximum of 256x256. Otherwise, surfaces such as floors would extend for thousands of texels, all at the mipmap level of the nearest vertex, and would require huge amounts of surface cache space while displaying a great deal of aliasing in distant regions due to a high texel:pixel ratio.

One final issue with surface caching involves 3-D hardware accelerators. Surfaces are effectively large textures (and larger at the mipmap levels typically used at the high resolutions of accelerators than they are at 320x200), and texture memory tends to be a limited resource on accelerators. Worse, accelerators are built for 16- or 32-bpp graphics, and surfaces are twice as large at 16-bpp as they are at 8-bpp, and correspondingly slower to build. Although the edge list can still be used to cull invisible polygons, it's nonetheless true that a surface cache around 2 Mb is best on a hardware accelerator.

The first generation of accelerators was originally designed for 2 Mb of RAM, which would have been a squeeze, but plummeting memory prices seem to have solved the problem; 4 Mb is fast becoming the standard. And given sufficient memory, surface caching runs at about the same speed on accelerators as Gouraud shading (slower because of building and downloading surfaces, but faster because of fewer, larger polygons), and still offers the same advantage as in software: detailed and consistently correct lighting.

Two final notes on surface caching

Dynamic lighting has a significant impact on the performance of surface caching, because whenever the lighting on a surface changes, the surface has to be rebuilt. In the worst case, where the lighting changes on every visible surface, the surface cache provides no benefit, and rendering runs at the combined speed of surface building and texture mapping. This

worst-case slowdown is tolerable but certainly noticeable, so it's best to design games that uses surface caching so only some of the surfaces change lighting at any one time. If necessary, you could alternate surface relighting so that half of the surfaces change on even frames, and half on odd frames, but large-scale, constant relighting is not surface caching's strongest suit.

Finally, Quake barely begins to tap surface caching's potential. All sorts of procedural texturing and post-processing effects are possible. If a wall is shot, a sprite of pockmarks could be attached to the wall's data structure, and the sprite could be drawn into the surface each time the surface is rebuilt. The same could be done for splatters, or graffiti, with translucency easily supported. These effects would then be cached and drawn as part of the surface, so the performance cost would be much less than effects done by on-screen overdraw every frame. Basically, the surface is a handy repository for all sorts of effects, because multiple techniques can be composited, because it caches the results for reuse without rebuilding, and because the texels constructed in a surface are automatically drawn in perspective.

Surface Caching Revisited, Quake's Triangle Models, and More

by Michael Abrash

In the late 70's, 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, a few months 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.

More on surface caching

Last time, I discussed in detail the surface caching technique that Quake uses to do detailed, high-quality lighting without lots of polygons. Since then, I've spent a considerable amount of time working on the port of Quake to Rendition's Verite 3-D accelerator chip, so I'll start off this month 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 combined 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 last time, 3-D hardware accelerators are designed to optimize Gouraud shading, but surface caching can also work on accelerators, with some significant quality advantages.

The surface-caching architecture of the Verite version of Quake (VQuake) is essentially the same as in software 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 format.

Downloading surfaces to the accelerator is a performance hit that doesn't exist in the software version. Although Verite uses DMA to download surfaces, DMA does 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 too 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, 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.

The 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 possible 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.)

One 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 (or three-or-more-pass, for adding splatters and the like by overlaying sprite polygons) drawing 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 polygons become, the better Gouraud shading looks compared to surface caching and two-pass lighting.

Our next engine at id Software will be oriented heavily toward hardware accelerators, and at this point it's a toss-up whether we'll use surface caching, Gouraud shading, or two-pass lighting. I'll keep you posted.

Drawing triangle models

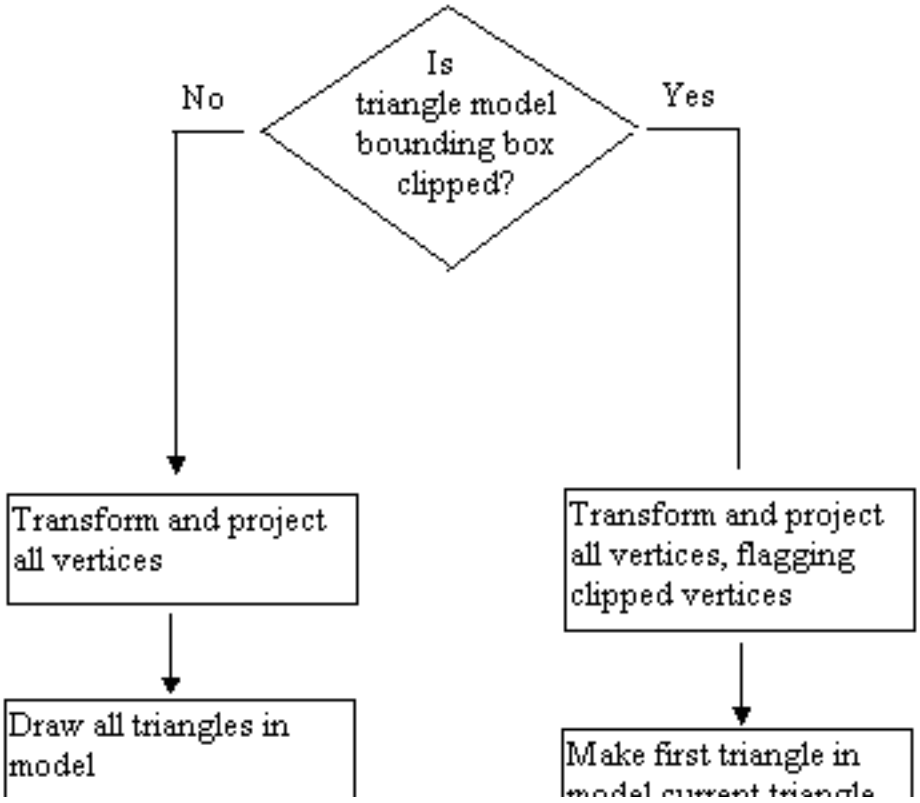
I've spent a number of columns over the past year discussing how Quake works. If you look closely, though, you'll see that almost all of the information was about drawing the world--the static walls, floors, ceilings, and such. There are several reasons for this, in particular that it's hard to get a 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 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 column I'll discuss some interesting aspects of our triangle-model architecture, and present code for one useful approach for rapid drawing of triangle models.

Drawing triangle models fast

We would have liked to have had 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 200-300 large polygons, not the 2000-3000 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 One. 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.



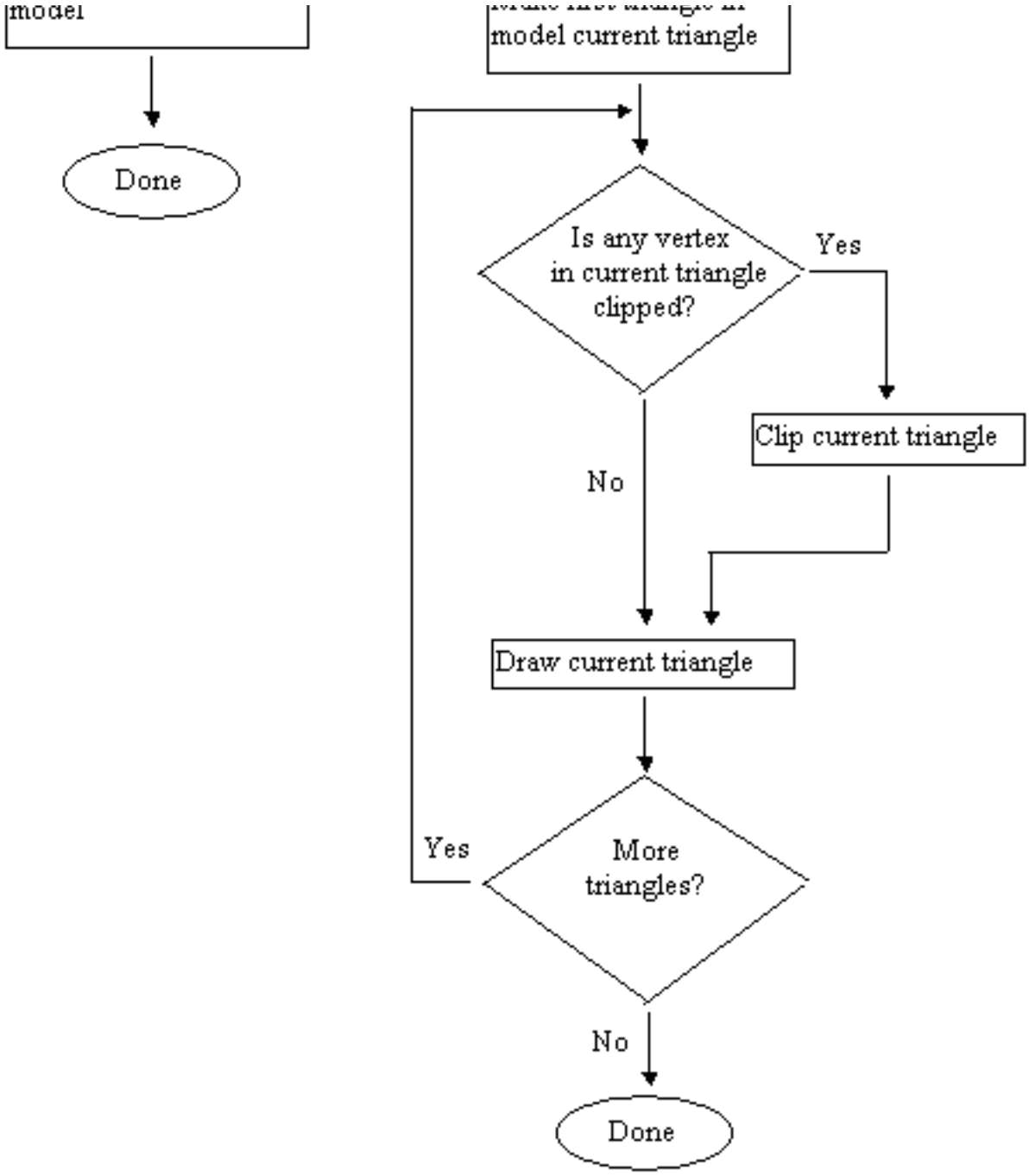


Figure One: Quake's triangle-model drawing pipeline.

Triangle models are stored quite differently from the world. 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 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.

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. One interesting benefit of integer coordinates is that it lets 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 gradients 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, 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 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 the 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 have 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 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, 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 just 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 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 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 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 2x2 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 did the trick.

One morning I came in, to find that overnight (and well into the morning), John had designed and implemented a technique I'll call subdivision rasterization, which 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 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 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 Two. There are some 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 One, is very simple and easily optimized, especially by comparison with a general triangle rasterizer.

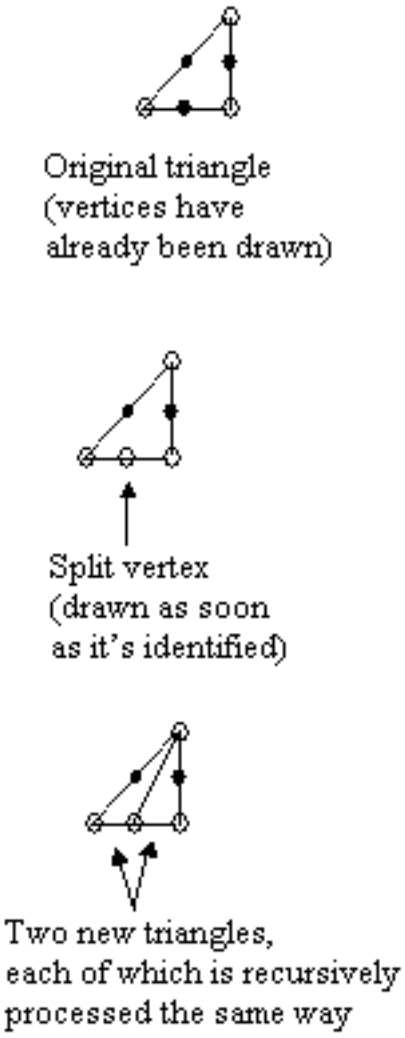


Figure Two: One recursive subdivision triangle-drawing step.

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% slower than subdivision rasterization. While 20% may not sound impressive, it includes clipping, transform, projection, and backface-culling time, so the rasterization difference alone is more than 50%. Besides, 20% overall means that we can have 12 monsters where we could only have had 10 before, so we count subdivision rasterization as a clear success.

Some 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 4x4 or 5x5), 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. The result is similar to a tristrip, but with 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 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.

Quake's 3-D Engine: The Big Picture

by Michael Abrash

If you want to be a game programmer, or for that matter any sort of programmer at all, here's the secret to success in just two words: Ship it. Finish the product and get it out the door, and you'll be a hero. It sounds simple, but it's a surprisingly rare skill, and one that's highly prized by software companies. Here's why.

My friend David Stafford, co-founder of the game company Cinematronics, says that shipping software is an unnatural act, and he's right. Most of the fun stuff in a software project happens early on, when anything's possible and there's a ton of new code to write. By the end of a project, the design is carved in stone, and most of the work involves fixing bugs, or trying to figure out how to shoehorn in yet another feature that was never planned for in the original design. All that is a lot less fun than starting a project, and often very hard work--but it has to be done before the project can ship. As a former manager of mine liked to say, "After you finish the first 90% of a project, you have to finish the other 90%." It's that second 90% that's the key to success.

This is true for even the most interesting projects. I spent the last year and a half as one of three programmers writing the game Quake at id Software, doing our best to push the state of the art of multiplayer and 3-D game technology ahead of anything else on the market, working on what was probably the most-anticipated game of all time. Exciting as it was, we hit the same rough patches toward the end as any other software project. I am quite serious when I say that a month before shipping, we were sick to death of working on Quake.

A lot of programmers get to that second 90%, get tired and bored and frustrated, and change jobs, or lose focus, or find excuses to procrastinate. There are a million ways not to finish a project, but there's only one way to finish: Put your head down and grind it out until it's done. Do that, and I promise you the programming world will be yours.

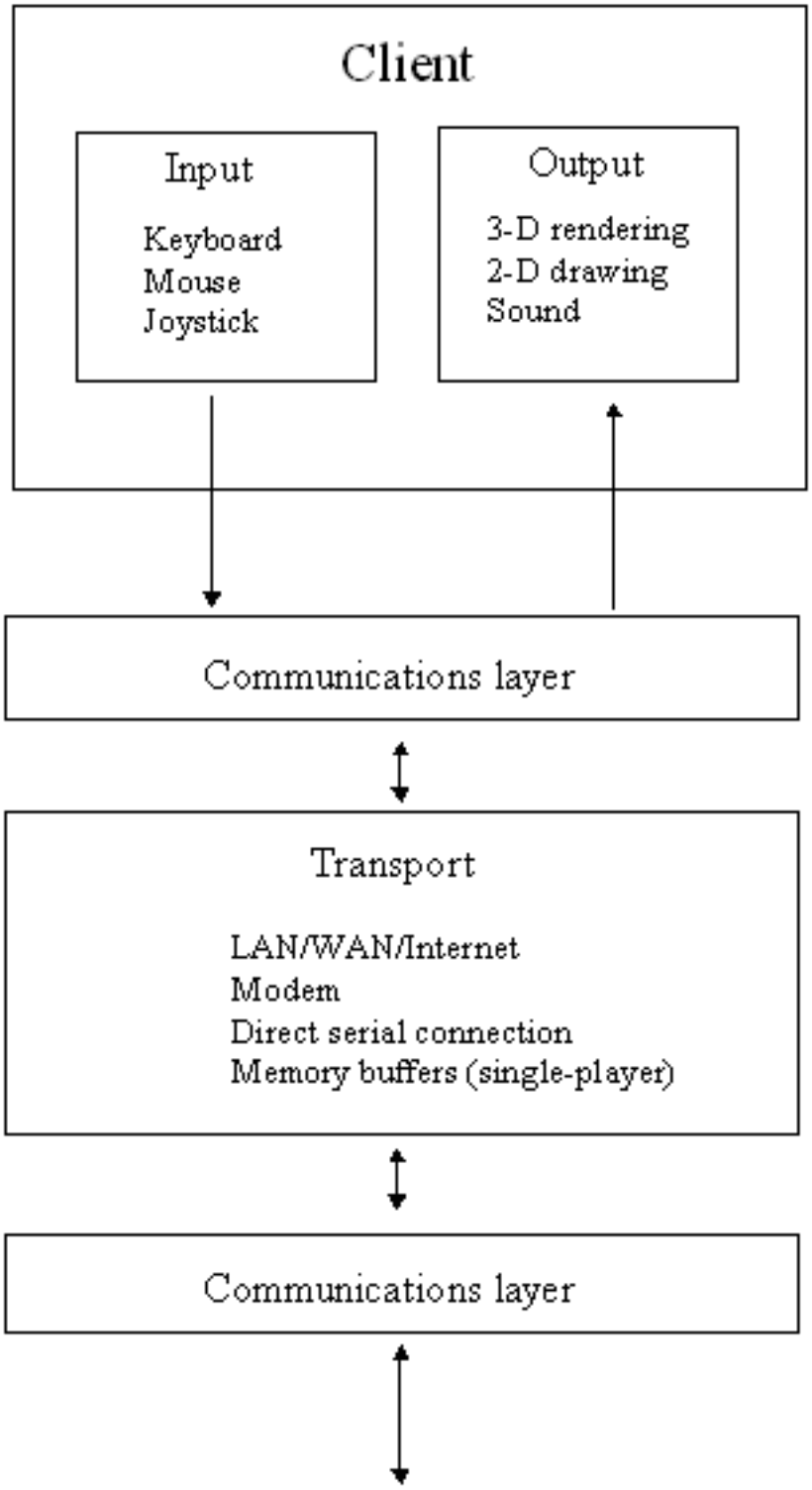
It worked for Dave; Cinematronics became a successful company and was acquired by Maxis. It worked for us at id, as well. DOOM was one of the most successful games in history, and we wanted to top it. For the programmers, the goal was to set new standards for 3-D and multiplayer--especially Internet--technology for the DOOM genre, and Quake did just that. Let's take a look at how it works.

Client-server

DOOM had a synchronous peer-to-peer networking architecture, where each player's machine runs a parallel game engine, and the machines proceed in lockstep. This works reasonably well for two-player modem games, but makes it hard to support lots of players coming and going at will, and is less well suited to the Internet, so we went with a different approach for Quake.

Quake is a client-server application, as shown in Figure One. All gameplay and simulation are performed on the server, and all input and output take place on the client, which is basically nothing more than a specialized terminal. Each client gathers up keyboard, mouse, and joystick input for each frame and sends it off to the server; the server receives the input from all

clients, runs the game for a fixed timeslice, and sends the results off to the clients; and the clients display the results during the next frame after they're received. This is true even in single-player mode, but here the client and the server can't actually be separate processes, because Quake has to run on non-multitasking DOS; instead, during each frame the input portion of the client is run, then the server executes, and finally the output portion of the client displays the current frame, with all communications between the client and the server flowing through the communications layer using memory buffers as the transport. In multiplayer games, the client and server are separate processes, running on different machines (except for the special case of listen servers, where both the multiplayer server and one of the clients run in the same process on one machine).



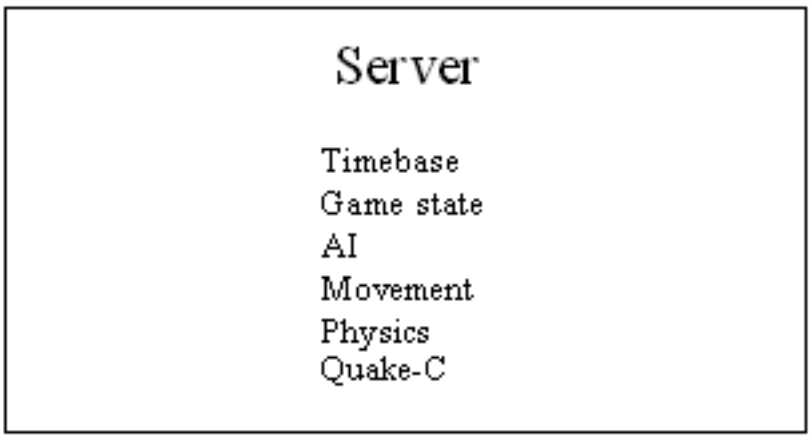


Figure 1: Quake's architecture.

Client-server has obvious benefits for multiplayer games, because it vastly simplifies issues of synchronization between various players. Perhaps less obvious is that client-server is useful even in single-player mode, because it enforces a modular design, and has a single communications channel between client and server that simplifies debugging. It's also a big help to have identical code for single-player and multiplayer modes.

Communications

An interesting issue with client-server architecture is Internet play. Quake was designed from the start for multiplayer gaming, but Internet play, which hadn't been a major issue for earlier games, raised some interesting and unique issues, because communications latencies are longer over the Internet than they are on a LAN, and often even longer than directly-connected modems, and because packet delivery is less reliable.

In the early stages of development, Quake used reliable packet delivery for everything. With this approach, packets are sent out and acknowledgement is sent back, and if acknowledgement isn't received, everything is brought to a halt until a resend succeeds. This was necessary because the clients were sent only changes to the current state, rather than the current state, in order to reduce the total amount of data that needed to be sent, and when sending nothing but changes, it's essential that every change be received, or else the cumulative state will be incorrect.

The problem with reliable packet delivery is that if a packet gets dropped, it takes a long time to find that out (at least one roundtrip from server to client), and then it takes a long time to resend it (at least another roundtrip). If the ping time to the client is 200 ms (about the best possible with a PPP connection), then a dropped packet will result in a glitch of several hundred milliseconds--long enough to be very noticeable and annoying.

Instead, Quake now uses reliable packet delivery only for information such as scores and level changes. Current game state, such as the locations of players and objects, is sent each timeslice not as changes, but in its entirety, compressed so it doesn't take up too much bandwidth. However, this information is not sent with reliable delivery; there is no

acknowledgement, and neither the server nor client knows nor cares whether those packets arrive or not. Each update contains all state relevant to each client for that frame, so all a dropped packet means is a freezing of the world for one server timeslice; server timeslices come in a constant stream at a rate of 10 or 20 a second, so a dropped packet results in a glitch of no more than 100 ms, which is quite acceptable.

Latency

Client-server imposes a potentially large latency between a player's action, such as pressing the jump key, and the player seeing the resulting action, such as his viewpoint jumping into the air. The action has to make a roundtrip to the server and back, so the latency can vary from close to no time at all, on a LAN, up to hundreds of milliseconds on the Internet. Longer latencies can make the game difficult to play; by the time the player actually jumps, he might have moved many feet forward and fallen into a pit. This problem raises the possibility of running some or all of the game logic on the client in parallel with the server, or in parallel with other clients in a peer-to-peer architecture, so the client can have faster response.

Faster response is all to the good, but there are some serious problems with simulating on the client. For one thing, it makes communications and game logic much more difficult, because instead of one central master simulation on the server, there are now potentially a dozen or more simulations that need to be synchronized. Only one outcome from any event can be allowed, so with client simulation there must be a mechanism for determining whose decision wins in case of conflict, and undoing actions that are overruled by another simulation. Worse, there are inevitably paradoxes, as, for example, a player firing a rocket and seeing it hit an opponent--but then seeing the opponent magically resurrect as the local client gets overruled. While Quake has lag, it doesn't have paradox, and that helps a lot in making the experience feel real.

Quake does use one shortcut to help with lag; if a player turns to look in another direction, that happens immediately, without waiting for the server to process the input and return the new state. There are no paradoxes or synchronization issues associated with turning in Quake, and instant turning makes the game feel much more responsive. (In QuakeWorld, a multiplayer-only follow-up currently in development, we've gone a step further and simulated the movement of the player, but nothing else, on the client, and we've found that this does improve the feel of Internet play quite a bit, albeit at the cost of an occasional minor paradox.)

The server

The Quake server maintains the game's timebase and state, performs object movement and physics, and runs monster AI. The most interesting aspect of the server is the extent to which it's data-driven. Each level (the current "world") is completely described by object locations and types, wall locations, and so on stored in a database loaded from disk. The behavior of objects and monsters is likewise externally programmable, controlled by functions written in a built-in interpreted language, Quake-C. Quake is controlled by its external database to the extent that not only have people been able to make new levels, but they've also been able to add new game elements, such as smart rockets that track people, planes that can be climbed into and flown, and alerters that stick to players and screech "Here I am!"--all without writing a single line of C or assembly code. This flexibility not only makes Quake a great platform for creativity, but

also helped a great deal as we developed the game, because it allowed us to try out changes without having to recompile the program. Indeed, levels and Quake-C programs can be reloaded and tried out without even exiting Quake.

If you're curious, there's lots of Quake-C code available on the Internet. One excellent site is Quake Developer's Pages (http://www.gamers.org/dEngine/quake/). You can find information about making custom monsters and levels there, as well.

The client

The server and communications layer are crucial elements of Quake, but it's the client with which the player actually interacts, and it's the client that has the glamour component--the 3-D engine. The client also handles keyboard, mouse, and joystick input, sound mixing, and 2-D drawing such as menus, the status bar, and text messages, but those are straightforward; 3-D is where the action is. The challenges with Quake's 3-D engine were twofold: allow true 3-D viewing in all directions (unlike DOOM's 2.5-D), and improve visual quality with lighting, more precise pixel placement, or whatever else it took--all with good performance, of course.

As with the server, the 3-D engine is data-driven, with the drawing data falling into two categories, the world and entities. Each level contains information about the geometry of walls, floors, and so on, and also about the textures (bitmaps) painted onto those faces. The Quake database also contains triangle meshes and textures describing players, monsters, and other moving objects, called entities. Originally, we planned to draw everything in Quake through a single rendering pipeline, but it turned out that there was no way to get good performance for both huge walls and monsters made of hundreds of tiny polygons out of a single pipeline, so the world and entities are drawn by completely different code paths.

The world is stored as a data structure known as a Binary Space Partitioning (BSP) tree. BSP trees are quite complicated to explain, so I won't go into detail here, but if you're interested, I've written about BSP trees in Dr. Dobb's Sourcebook; see the May, July, and November, 1995, issues. For Quake's purposes, BSP trees do two very useful things: they make it easy to traverse a set of polygons in front-to-back or back-to-front order, and they partition space into convex volumes.

Back-to-front order is handy if you're drawing complete polygons, because you can draw all your polygons back-to-front and get correct occlusion, a process known as the painters algorithm. In Quake, however, we draw polygons front-to-back. To be precise, we take all our polygons and put their edges into a global list; then we rasterize this list and draw only the visible (frontmost) portions. The big advantage of this approach is that we draw each pixel in the world once and only once, saving precious drawing time in complex scenes because we don't overdraw polygons one atop another. (See the May and July, 1996, issues of DDS for further discussion of Quake's edge list.)

However, the edge list isn't fast enough to handle the thousands of polygons that can be in the view pyramid; if you put that many edges into an edge list, what you get is a very slow frame rate, because there's just too much data to process and sort. So we limited the number of polygons that have to be considered by taking advantage of the convex-partitioning property of BSP trees to calculate a potentially visible set (PVS). When a level is processed into the

Quake format (a separate preprocessing step done once when a map is built, by a utility program), a BSP tree is built from the level, and then, for each convex subspace (called a "leaf") of the BSP tree, a visibility calculation is performed. For a given leaf , the utility calculates which other subspaces are visible from anywhere in that leaf, and that information is stored with the leaf in the BSP tree. In other words, if no matter where you're standing in the kitchen downstairs, you can't possibly see up the stairs into the bedroom, the bedroom polygons are omitted from the kitchen leaf's PVS; if you can see into the living room from the corner of the kitchen, the kitchen leaf's PVS remembers that living room polgyons are potentially visible. We can be sure that the PVS for a leaf contains all the polygons we ever need to consider if the player is standing anywhere in that leaf, so at rendering time, rather than processing the thousands of polygons in a level, we only have to handle--clip, transform, project, and insert in the edge list--the few hundred polygons in the current leaf's PVS. This reduces the polygon load to a level that the edge list can handle, and the PVS and the edge list together make for fast, consistent performance in a wide variety of scenes. The January, 1996, DDS covers the PVS in more detail.

One point about the PVS: it can be quite expensive to calculate. PVS determination can take 15 or 20 minutes to finish--on a four-processor Alpha system! Fortunately, Pentium Pro systems are getting fast enough to handle the job well, and no doubt the code can be made faster, but be aware that the power of the PVS comes at a price.

Once the edge list has finished processing all the edges, we're left with a set of spans that cover the screen exactly once. We pass this list to a rasterizer, which texture maps the appropriate bitmap onto those spans, accounting for perspective (which requires an expensive divide to get exactly right) by doing a divide every 16 pixels, and interpolating linearly between those points. (This is the key step in allowing true 3-D viewing in any direction.)

Lighting involves true light sources and shadowing, unlike DOOM's crude sector lighting. This is performed by having a separate lighting map (basically a texture map, but with light values instead of colors) for each polygon, with light samples on a 16-pixel grid, and prelighting the texture for each polygon according to the grid as the texture is drawn into a memory buffer; the actual texture mapping works from these pre-lit textures, with no lighting occurring during the texture mapping itself. I don't have space to explain how this differs from normal lighting, or why it's so desirable, except to say that it results in detailed, high-quality lighting and good performance, but you can find a thorough explanation in the November, 1996, DDS.

Entities

DOOM used flat posters--sprites--for monsters and other moving objects, and one of the big advances in Quake was switching to polygonal entities, which are true 3-D objects. However, this raised a new concern; entities can contain hundreds of polygons, and there can be a dozen of them visible at once, so drawing them fast was one of our major challenges. Each entity consists of a set of vertices, and a mesh of triangles across those vertices. All the vertices in an entity are transformed and projected as a set, and then all the triangles are drawn, using affine (linear) rather than perspective-correct texture mapping; affine is faster, and entity polygons are typically so small and far away that the imperfections of affine aren't noticeable. Also, the entity drawer is optimized for small triangles, rather than the long span drawing that

the world drawer is optimized for; in fact, there's a special ultra-fast drawer for distant entities, which you can read about in the January, 1997, DDS.

The big difference, however, is that entities are drawn with z-buffering; that is, the distance of each pixel to be drawn is compared to the distance of the pixel being drawn over (stored in a memory area called a z-buffer), and the new pixel is drawn only if it's closer. This lets entities sort seamlessly with the world and each other, no matter where they move or what angle they're viewed at. There's a cost, to be sure; z-buffering is slower than non-z-buffered drawing, and the z-buffer has to be initialized to match the visible world pixels, at a cost of about 10% of Quake's performance. That cost is, however, more than repaid by the simplicity and accuracy of z-buffering, which saves us from having to perform complex clipping and sorting operations in order to draw entities properly, and gives us flawless drawing under all circumstances.

The PVS helps improve entity performance, because we only need to draw entities that are in the PVS for the current leaf. Other entities don't exist, so far as the client is concerned; the server doesn't even bother sending information about anything outside the PVS, which not only helps reduce the drawing load, but also minimizes the amount of information that has to be sent over modems or the Internet.

As a final effect, we wanted to have effects like smoke trails and huge explosions in Quake, but couldn't figure out how to do them fast and well with standard sprites (although the cores of explosions are sprites) or with polygon models. The solution was to use clouds of hundreds of square, colored, z-buffered rectangles that scale with distance, called particles. A few hundred particles strewn behind a rocket looks amazingly like a trail of flame and smoke, especially if they start out yellow and fade to red and then to gray, and as a group they do an excellent job of convincing the eye that they represent a true 3-D object.

Particles and unreliable packet delivery, along with dynamic lighting, which allows explosions and muzzle flashes to light up the world, were among the last additions to the Quake engine; these features made a well-rendered but somewhat sterile world come alive. These, together with details such as menus, a ton of optimization, and a healthy dose of bug fixing, were the "second 90%" that propelled Quake from being a functional 3-D and multiplayer engine to a technological leap ahead.

Ah, if only we'd had time for a third 90%!

Permissions and Author's Note

These are the articles about how Quake works that I wrote during the period 1995-1997, when John Carmack and I were developing Quake. They certainly don't cover everything, but they cover a number of design and implementation aspects of Quake, and offer some history of the development process as well. John recently made the Quake source code freely available, and in that spirit I am making these articles, which together are a useful annotation to that code, available as well. I'd like to thank the Coriolis Group, publisher of "Michael Abrash's Graphics Programming Black Book," in which these articles first appeared as a group, for granting me permission to do this.

The architecture of Quake, and most of the technology discussed in these articles, is the creation of John Carmack. John, I figure that making this freely available is the form of "thank-you" that would mean the most to you.

I could not possibly have written about the many topics covered in the "Black Book" without help and encouragement from many people. Due to a mistake by an editor, the acknowledgements were omitted from that book. I would like to correct that omission here.

Thanks to my good friend Jeff Duntemann, editor of PC Techniques, without whom not only this material but pretty much my entire writing career wouldn't exist. Thanks to Jon Erickson, editor of Dr. Dobb's (in which most of this material originally appeared), both for encouragement and general good cheer and for giving me a place to write whatever I wanted about realtime 3-D. I'd also like to thank Chris Hecker and Jennifer Pahlka of the Game Developers Conference, without whose encouragement, nudging, and occasional well-deserved nagging there is no chance I would ever have written a paper for the GDC—a paper that's included in this material and is the most comprehensive overview of the Quake technology that's ever likely to be written.

Thanks to Dan Illowsky for getting me started writing articles long ago, when I lacked the confidence to do it on my own—and for teaching me how to handle the business end of things. Thanks to Will Fastie for giving me my first crack at writing for a large audience in the long-gone but still-missed PC Tech Journal, and for showing me how much fun it could be in his even longer-vanished but genuinely terrific column in Creative Computing (the most enjoyable single column I have ever read in a computer magazine; I used to haunt the mailbox around the beginning of the month just to see what Will had to say). Thanks to Robert Keller,

Erin O’Conner, Liz Oakley, Steve Baker, and the rest of the cast of thousands that made Programmer’s Journal a uniquely fun magazine—especially Erin, who did more than anyone else to teach me the proper use of the English language. (To this day, Erin will still patiently explain to me when one should use “that” and when one should use “which,” even though eight years of instruction on this and related topics have left no discernible imprint on my brain.) Thanks to Tami Zemel, Monica Berg, and the rest of the Dr. Dobb’s crew for excellent, professional editing, and for just being great people. Thanks to the Coriolis gang for their tireless hard work: Jeff Duntemann and Keith Weiskamp on the editorial and publishing side; Brad Grannis, Rob Mauhar, and Michelle Stroup, who handled art, design, and layout; and Jim Mischel for helping in a variety of ways. Thanks to Jack Tseng, for teaching me a lot about graphics hardware, and even more about how much difference hard work can make. Thanks to John T. Cockerham, David Stafford, Terje Mathisen, the BitMan, Jim Mackraz, Melvin Lafitte, John Navas, Phil Coleman, Anton Truenfels, John Miles, John Bridges, Jim Kent, Hal Hardenberg, Dave Miller, Steve Levy, Jack Davis, Duane Strong, Daev Rohr, Bill Weber, Dan Gochnauer, Patrick Milligan, Tom Wilson, the people in the ibm.pc/fast.code topic on Bix, and all the rest of you who have been so generous with your ideas and suggestions. I’ve done my best to acknowledge contributors by name, but if your name is omitted, my apologies, and please consider yourself thanked.

Special thanks to loonyboi and Blue for converting and formatting these files.

And, of course, thanks to Shay and Emily for their generous patience with my passion for writing and computers.

Michael Abrash
Bellevue, WA 2000