Chapter 70

Quake: A Post-Mortem and a Glimpse into the Future

Chapter 70

Why did not any of the children in the first group think of this faster method of going across the room? It is simple. They looked at what they were given to use for materials and, they are like all of us, they wanted to use everything. But they did not need everything. They could do better with less, in a different way.

—*Frederik Pohl,* The Gold at the Starbow's End

Eleven years ago, I started the first serious graphics article I ever wrote with the above quote. The point I was making at the time was that programming assumptions based on high-level languages or other processors had to be discarded in the quest for maximum x86 performance. While that's certainly still true, time and the microcomputer world have moved on, and today there's a more important lesson 3-D game programmers can draw from Frederik Pohl's words. Nowadays, CPUs, 3-D hardware, 3-D algorithms, and 3-D data structures are evolving so rapidly that the enemy is now often the assumptions and techniques from the last product—and sometimes the assumptions and techniques in the *current* product. We all feel most comfortable with techniques we've already mastered, but leading-edge 3-D game technology is such a delicate balancing act between performance, features (particularly with game designers always wanting to add more), and workflow (as we'll see, preprocessing that improves performance often hurts designer productivity) that it's never safe to stop looking for a better approach until the game has actually shipped. Change is the rule, and we must always be looking to "do better with less, in a different way."

I've talked about Quake's technology elsewhere in this book, However, those chapters focused on specific areas, not overall structure. Moreover, Quake changed in significant ways between the writing of those chapters and the final shipping. Then, after shipping, Quake was ported to 3-D hardware. And the post-Quake engine, code-named Trinity, is already in development at this writing (Spring 1997), with some promising results. So in wrapping up this book, I'll recap Quake's overall structure relatively quickly, then bring you up to date on the latest developments. And in the spirit of Frederik Pohl's quote, I'll point out that we implemented and discarded at least half a dozen 3-D engines in the course of developing Quake (and all of Quake's code was written from scratch, rather than using Doom code), and almost switched to another one in the final month, as I'll describe later. And even at this early stage, Trinity uses almost no Quake technology.

In fact, I'll take this opportunity to coin Carmack's Law, as follows: *Fight code entropy.* If you have a new fundamental assumption, throw away your old code and rewrite it from scratch. Incremental patching and modifying seems easier at first, and is the normal course of things in software development, but ends up being much harder and producing bulkier, markedly inferior code in the long run, as we'll see when we discuss the net code for QuakeWorld. It may seem safer to modify working code, but the nastiest bugs arise from unexpected side effects and incorrect assumptions, which almost always arise in patched-over code, not in code designed from the ground up. Do the hard work up front to make your code simple, elegant, great—and just plain *right*—and it'll pay off many times over in the long run.

Before I begin, I'd like to remind you that all of the Doom and Quake material I'm presenting in this book is presented in the spirit of sharing information to make our corner of the world a better place for everyone. I'd like to thank John Carmack, Quake's architect and lead programmer, and id Software for allowing me to share this technology with you, and I encourage you to share your own insights by posting on the Internet and writing books and articles whenever you have the opportunity and the right to do so. (Of course, check with your employer first!) We've all benefited greatly from the shared wisdom of people like Knuth, Foley and van Dam, Jim Blinn, Jim Kajiya, and hundreds of others—are you ready to take a shot at making your own contribution to the future?

Preprocessing the World

For the most part, I'll discuss Quake's 3-D engine in this chapter, although I'll touch on other areas of interest. For 3-D rendering purposes, Quake consists of two basic sorts of objects: the world, which is stored as a single BSP model and never changes shape or position; and potentially moving objects, called *entities*, which are drawn in several different ways. I'll discuss each separately.

The world is constructed from a set of brushes, which are n-sided convex polyhedra placed in a level by a designer using a map editor, with a selectable texture on each

face. When a level is completed, a preprocessing program combines all brushes to form a skin around the solid areas of the world, so there is no interpenetration of polygons, just a continuous mesh delineating solid and empty areas. Once this is done, the next step is generating a BSP tree for the level.

The BSP consists of splitting planes aligned with polygons, called nodes, and of leaves, which are the convex subspaces into which all the nodes carve space. The top node carves the world into two subspaces, and divides the remaining polygons into two sets, splitting any polygon that spans the node into two pieces. Each subspace is then similarly split by one node each, and so on until all polygons have been used to create nodes. A node's subspace is the total space occupied by all its children: the subspace that the node splits into two parts, and that its children continue to subdivide. When the only polygon in a node's subspace is the polygon that splits the subspace—the polygon whose plane defines the node—then the two child subspaces are called leaves, and are not divided any further.

The BSP tree is built using the polygon that splits the fewest of the polygons in the current node's subspace as the heuristic for choosing splitters, which is not an optimal solution—but an optimal solution is NP-complete, and our heuristic adds only 10% to 15% more polygons to the level as a result of BSP splits. Polygons are not split all the way into leaves; rather, they are placed on the nodes with which they are coplanar (one set on the front and one on the back, which has the advantage of letting us reuse the BSP-walking dot product for backface culling as well), thereby reducing splitting considerably, because polygons are split only by parent nodes, not by child nodes (as would be necessary if polygons were split into leaves). Eliminating polygon splits, thus reducing the total number of polygons per level, not only shrinks Quake's memory footprint, but also reduces the number of polygons that need to be processed by the 3-D pipeline, producing a speedup of about 10% in Quake's overall performance.

Getting proper front-to-back drawing order is a little more complicated with polygons on nodes. As we walk the BSP tree front-to-back, in each leaf we mark the polygons that are at least partially in that leaf, and then after we've recursed and processed everything in front of a node, we then process all the marked polygons on that node, after which we recurse to process the polygons behind the node. So putting the polygons on the nodes saves memory and improves performance significantly, but loses the simple approach of simply recursing the tree and processing the polygons in each leaf as we come to it, in favor of recursing and marking in front of a node, processing marked polygons on the node, then recursing behind the node.

After the BSP is built, the outer surfaces of the level, which no one can ever see (because levels are sealed spaces), are removed, so the interior of the level, containing all the empty space through which a player can move, is completely surrounded by a solid region. This eliminates a great many irrelevant polygons, and reduces the complexity of the next step, calculating the potentially visible set.

The Potentially Visible Set (PVS)

After the BSP tree is built, the potentially visible set (PVS) for each leaf is calculated. The PVS for a leaf consists of all the leaves that can be seen from anywhere in that leaf, and is used to reduce to a near-minimum the polygons that have to be considered for drawing from a given viewpoint, as well as the entities that have to be updated over the network (for multiplayer games) and drawn. Calculating the PVS is expensive; Quake levels take 10 to 30 minutes to process on a four-processor Alpha, and even with speedup tweaks to the BSPer (the most effective of which was replacing many calls to **malloc()** with stack-based structures—beware of **malloc()** in performance-sensitive code), Quake 2 levels are taking up to an hour to process. (Note, however, that that includes BSPing, PVS calculations, and radiosity lighting, which I'll discuss later.)

Some good news, though, is that in the nearly two years since we got the Alpha, Pentium Pros have become as fast as that generation of Alphas, so it is now possible to calculate the PVS on an affordable machine. On the other hand, even 10 minutes of BSPing does hurt designer productivity. John has always been a big advocate of moving code out of the runtime program into utilities, and of preprocessing for performance and runtime simplicity, but even he thinks that in Quake, we may have pushed that to the point where it interfered too much with workflow. The real problem, of course, is that even a huge amount of money can't buy orders of magnitude more performance than commodity computers; we are getting an eight-R10000 SGI compute server, but that's only about twice as fast as an off-the-shelf four-processor Pentium Pro.

The size of the PVS for each leaf is manageable because it is stored as a bit vector, with a 1-bit for the position in the overall leaf array of each leaf that's visible from the current leaf. Most leaves are invisible from any one leaf, so the PVS for each leaf consists mostly of zeros, and compacts nicely with run-length encoding.

There are two further interesting points about the PVS. First, the Quake PVS does not exclude quite as many leaves from potential visibility as it could, because the surfaces that precisely describe leaf-to-leaf visibility are quadratic surfaces; in the interests of speed and simplicity, planar surfaces with some slope are used instead. Second, the PVS describes visibility from anywhere in a leaf, rather than from a specific viewpoint; this can cause two or three times as many polygons as are actually visible to be considered. John has been researching the possibility of an EVS—an *exactly visible set*—and has concluded that a 6-D BSP with hyperbolic separating planes could do the job; the problem now is that he doesn't know how to get the math to work, at least at any reasonable speed.

An interesting extension of the PVS is what John calls the *potentially hearable set* (PHS)—all the leaves visible from a given leaf, plus all the leaves visible from *those* leaves—in other words, both the directly visible leaves and the one-bounce visible leaves. Of course, this is not exactly the hearable space, because sounds could echo or carry further than that, but it does serve quite nicely as a potentially *relevant* space—the set

of leaves that have any interest to the player. In Quake, all sounds that happen anywhere in the world are sent to the client, and are heard, even through walls, if they're close enough; an explosion around the corner could be well within hearing and very important to hear, so the PVS can't be used to reject that sound, but unfortunately an explosion on the other side of a solid wall will sound exactly the same. Not only is it confusing hearing sounds through walls, but in a modem game, the bandwidth required to send all the sounds in a level can slow things down considerably. In a recent version of QuakeWorld, a specifically multiplayer variant of Quake I'll discuss later, John uses the PHS to determine which sounds to bother sending, and the resulting bandwidth improvement has made it possible to bump the maximum number of players from 16 to 32. Better yet, a sound on the other side of a solid wall won't be heard unless there's an opening that permits the sound to come through. (In the future, John will use the PVS to determine fully audible sounds, and the PHS to determine muted sounds.) Also, the PHS can be used for events like explosions that might not have their center in the PVS, but have portions that reach into the PVS. In general, the PHS is useful as an approximation of the space in which the client might need to be notified of events.

The final preprocessing step is light map generation. Each light is traced out into the world to see what polygons it strikes, and the cumulative effect of all lights on each surface is stored as a light map, a sampling of light values on a 16-texel grid. In Quake 2, radiosity lighting—a considerably more expensive process, but one that produces highly realistic lighting—is performed, but I'll save that for later.

Passages: The Last-Minute Change that Didn't Happen

Earlier, I mentioned that we almost changed 3-D engines again in the last month of Quake's development. Here's what happened: One of the alternatives to the PVS is the use of *portals*, where the focus is on the places where polygons don't exist along leaf faces, rather than the more usual focus on the polygons themselves. These "empty" places are themselves polygons, called portals, that describe all the places that visibility can pass from one leaf to another. Portals are used by the PVS generator to determine visibility, and are used in other 3-D engines as the primary mechanism for determining leaf or sector visibility. For example, portals can be projected to screenspace, then used as a 2-D clipping region to restrict drawing of more distant polygons to only those that are visible through the portal. Or, as in Quake's preprocessor, visibility boundary planes can be constructed from one portal to the next, and 3-D clipping to those planes can be used to determine visible polygons or leaves. Used either way, portals can support more changeable worlds than the PVS, because, unlike the PVS, the portals themselves can easily be changed on the fly.

The problem with portal-based visibility is that it tends to perform at its worst in complex scenes, which can have many, many portals. Since those are the most expensive scenes to draw, as well, portals tend to worsen the worst case. However, late

in Quake's development, John realized that the approach of storing portals themselves in the world database could readily be improved upon. (To be clear, Quake wasn't using portals at that point, and didn't end up using them.) Since the aforementioned sets of 3-D visibility clipping planes *between* portals—which he named *passages*—were what actually got used for visibility, if he stored those, instead of generating them dynamically from the portals, he would be able to do visibility much faster than with standard portals. This would give a significantly tighter polygon set than the PVS, because it would be based on visibility through the passages from the viewpoint, rather than the PVS's approach of visibility from anywhere in the leaf, and that would be a considerable help, because the level designers were running right up against performance limits, partly because of the PVS's relatively loose polygon set. John immediately decided that passages-based visibility was a sufficiently superior approach that if it worked out, he would switch Quake to it, even at that late stage, and within a weekend, he had implemented it and had it working—only to find that, like portals, it improved best cases but worsened worst cases, and overall wasn't a win for Quake. In truth, given how close we were to shipping, John was as much thankful as disappointed that passages didn't work out, but the possibilities were too great for us not to have taken a shot at it.

So why even bother mentioning this? Partly to show that not every interesting idea pans out; I tend to discuss those that *did* pan out, and it's instructive to point out that many ideas don't. That doesn't mean you shouldn't try promising ideas, though. First, some do pan out, and you'll never know which unless you try. Second, an idea that doesn't work out in one case can still be filed away for another case. It's quite likely that passages will be useful in a different context in a future engine.

The more approaches you try, the larger your toolkit and the broader your understanding will be when you tackle your next project.

Drawing the World

Everything described so far is a preprocessing step. When Quake is actually running, the world is drawn as follows: First, the PVS for the view leaf is decompressed, and each leaf flagged as visible is marked as being in the current frame's PVS. (The marking is done by storing the current frame's number in the leaf; this avoids having to clear the PVS marking each frame.) All the parent nodes of each leaf in the PVS are also marked; this information could have been stored as additional PVS flags, but to save space is bubbled up the BSP from each visible leaf.

After the PVS is marked, the BSP is walked front-to-back. At each node, the bounding box of the node's subspace is clipped against the view frustum; if the bounding box is fully clipped, then that node and all its children are ignored. Likewise, if the node is not in the PVS for the current viewpoint leaf, the node and all its children are ignored. If the bounding box is partially clipped or not clipped at all, that information is passed to the children so that any unnecessary clip tests can be avoided.

The children in front of the node are then processed recursively. When a leaf is reached, polygons that touch that leaf are marked as potentially drawable. When recursion in front of a node is finished, all polygons on the front side of the node that are marked as potentially drawable are added to the edge list, and then the children on the back side of that node are similarly processed recursively.

The edge list is a special, intermediate step between polygons and drawing. Each polygon is clipped, transformed, and projected, and its non-horizontal edges are added to a global list of potentially drawable edges. After all the potentially drawable edges in the world have been added, the global edge list is scanned out all at once, and all the visible spans (the nearest spans, as determined by sorting on BSP-walk order) in the world are emitted into span lists linked off the respective surface descriptors (for now, you can think of a surface as being the same as a polygon). Taken together, these spans cover every pixel on the screen once and only once, resulting in zero overdraw; surfaces that are completely hidden by nearer surfaces generate no spans at all. The spans are then drawn; all the spans for one surface are drawn, and then all the spans for the next, so that there's texture coherency between spans, which is very helpful for processor cache coherency, and also to reduce setup overhead.

The primary purpose of the edge list is to make Quake's performance as level—that is, as consistent—as possible. Compared to simply drawing all potentially drawable polygons front-to-back, the edge list certainly slows down the best case, that is, when there's no overdraw. However, by eliminating overdraw, the worst case is helped considerably; in Quake, there's a ratio of perhaps 4:1 between worst and best case drawing time, versus the 10:1 or more that can happen with straight polygon drawing. Leveling is very important, because cases where a game slows down to the point of being unplayable dictate game and level design, and the fewer constraints placed on design, the better.

A corollary is that best case performance can be seductively misleading; it's a great feeling to see a scene running at 30 or even 60 frames per second, but if the bulk of the game runs at 15 fps, those best cases are just going to make the rest of the game look worse.

The edge list is an atypical technology for John; it's an extra stage in the engine, it's complex, and it doesn't scale well. A Quake level might have a maximum of 500 potentially drawable polygons that get placed into the edge list, and that runs fine, but if you were to try to put 5,000 polygons into the edge list, it would quickly bog down due to edge sorting, link following, and dataset size. Different data structures (like using a tree to store the edges rather than a linear linked list) would help to some degree, but basically the edge list has a relatively small window of applicability; it was appropriate technology for the degree of complexity possible in a Pentium-based game (and even then, only with the reduction in polygons made possible by the PVS), but will probably be poorly suited to more complex scenes. It served well in the Quake engine, but remains an inelegant solution, and, in the end, it feels like

there's something better we didn't hit on. However, as John says, "I'm pragmatic above all else"—and the edge list did the job.

Rasterization

Once the visible spans are scanned out of the edge list, they must still be drawn, with perspective-correct texture mapping and lighting. This involves hundreds of lines of heavily optimized assembly language, but is fundamentally pretty simple. In order to draw the spans for a given surface, the screenspace equations for 1/z, s/z, and t/z (where s and t are the texture coordinates and z is distance) are calculated for the surface. Then for each span, these values are calculated for the points at each end of the span, the reciprocal of 1/z is calculated with a divide, and s and t are then calculated as (s/z)*z and (t/z)*z. If the span is longer than 16 pixels, s and t are likewise calculated every 16 pixels along the span. Then each stretch of up to 16 pixels is drawn by linearly interpolating between these correctly calculated points. This introduces some slight error, but this is almost never visible, and even then is only a small ripple, well worth the performance improvement gained by doing the perspective-correct math only once every 16 pixels. To speed things up a little more, the FDIV to calculate the reciprocal of 1/z is overlapped with drawing 16 pixels, taking advantage of the Pentium's ability to perform floating-point in parallel with integer instructions, so the FDIV effectively takes only one cycle.

Lighting

Lighting is less simple to explain. The traditional way of doing polygon lighting is to calculate the correct light at the vertices and linearly interpolate between those points (Gouraud shading), but this has several disadvantages; in particular, it makes it hard to get detailed lighting without creating a lot of extra polygons, the lighting isn't perspective correct, and the lighting varies with viewing angle for polygons other than triangles. To address these problems, Quake uses surface-based lighting instead. In this approach, when it's time to draw a surface (a world polygon), that polygon's texture is tiled into a memory buffer. At the same time, the texture is lit according to the surface's light map, as calculated during preprocessing. Lighting values are linearly interpolated between the light map's 16-texel grid points, so the lighting effects are smooth, but slightly blurry. Then, the polygon is drawn to the screen using the perspective-correct texture mapping described above, with the prelit surface buffer being the source texture, rather than the original texture tile. No additional lighting is performed during texture mapping; all lighting is done when the surface buffer is created.

Certainly it takes longer to build a surface buffer and then texture map from it than it does to do lighting and texture mapping in a single pass. However, surface buffers are cached for reuse, so only the texture mapping stage is usually needed. Quake surfaces tend to be big, so texture mapping is slowed by cache misses; however, the Quake approach doesn't need to interpolate lighting on a pixel-by-pixel basis, which

helps speed things up, and it doesn't require additional polygons to provide sophisticated lighting. On balance, the performance of surface-based drawing is roughly comparable to tiled, Gouraud-shaded texture mapping—and it looks much better, being perspective correct, rotationally invariant, and highly detailed. Surface-based drawing also has the potential to support some interesting effects, because anything that can be drawn into the surface buffer can be cached as well, and is automatically drawn in correct perspective. For instance, paint splattered on a wall could be handled by drawing the splatter image as a sprite into the appropriate surface buffer, so that drawing the surface would draw the splatter as well.

Dynamic Lighting

Here we come to a feature added to Quake after last year's Computer Game Developer's Conference (CGDC). At that time, Quake did not support dynamic lighting; that is, explosions and such didn't produce temporary lighting effects. We hadn't thought dynamic lighting would add enough to the game to be worth the trouble; however, at CGDC Billy Zelsnack showed us a demo of his latest 3-D engine, which was far from finished at the time, but did have impressive dynamic lighting effects. This caused us to move dynamic lighting up the priority list, and when I got back to id, I spent several days making the surface-building code as fast as possible (winding up at 2.25 cycles per texel in the inner loop) in anticipation of adding dynamic lighting, which would of course cause dynamically lit surfaces to constantly be rebuilt as the lighting changed. (A significant drawback of dynamic lighting is that it makes surface caching worthless for dynamically lit surfaces, but if most of the surfaces in a scene are not dynamically lit at any one time, it works out fine.) There things stayed for several weeks, while more critical work was done, and it was uncertain whether dynamic lighting would, in fact, make it into Quake.

Then, one Saturday, John suggested that I take a shot at adding the high-level dynamic lighting code, the code that would take the dynamic light sources and project their sphere of illumination into the world, and which would then add the dynamic contributions into the appropriate light maps and rebuild the affected surfaces. I said I would as soon as I finished up the stuff I was working on, but it might be a day or two. A little while later, he said, "I bet I can get dynamic lighting working in less than an hour," and dove into the code. One hour and nine minutes later, we had dynamic lighting, and it's now hard to imagine Quake without it. (It sure is easier to imagine the impact of features and implement them once you've seen them done by someone else!)

One interesting point about Quake's dynamic lighting is how inaccurate it is. It is basically a linear projection, accounting properly for neither surface angle nor lighting falloff with distance—and yet that's almost impossible to notice unless you specifically look for it, and has no negative impact on gameplay whatsoever. Motion and fast action can surely cover for a multitude of graphics sins.

It's well worth pointing out that because Quake's lighting is perspective correct and independent of vertices, and because the rasterizer is both subpixel and subtexel correct, Quake worlds are visually very solid and stable. This was an important design goal from the start, both as a point of technical pride and because it greatly improves the player's sense of immersion.

Entities

So far, all we've drawn is the static, unchanging (apart from dynamic lighting) world. That's an important foundation, but it's certainly not a game; now we need to add moving objects. These objects fall into four very different categories: BSP models, polygon models, sprites, and particles.

BSP Models

BSP models are just like the world, except that they can move. Examples include doors, moving bridges, and health and ammo boxes. The way these are rendered is by clipping their polygons into the world BSP tree, so each polygon fragment is in only one leaf. Then these fragments are added to the edge list, just like world polygons, and scanned out, along with the rest of the world, when the edge list is processed. The only trick here is front-to-back ordering. Each BSP model polygon fragment is given the BSP sorting order of the leaf in which it resides, allowing it to sort properly versus the world polygons. If two or more polygons from different BSP models are in the same leaf, however, BSP ordering is no longer useful, so we then sort those polygons by $1/z$, calculated from the polygons' plane equations.

Interesting note: We originally tried to sort all world polygons on $1/z$ as well, the reason being that we could then avoid splitting polygons except when they actually intersected, rather than having to split them along the lines of parent nodes. This would result in fewer edges, and faster edge list processing and rasterization. Unfortunately, we found that precision errors and special cases such as seamlessly abutting objects made it difficult to get global $1/z$ sorting to work completely reliably, and the code that we had to add to work around these problems slowed things up to the point where we were getting no extra performance for all the extra code complexity. This is not to say that $1/z$ sorting can't work (especially in something like a flight sim, where objects never abut), but BSP sorting order can be a wonderful thing, partly because it always works perfectly, and partly because it's simpler and faster to sort on integer node and leaf orders than on floating-point $1/z$ values.

BSP models take some extra time because of the cost of clipping them into the world BSP tree, but render just as fast as the rest of the world, again with no overdraw, so closed doors, for example, block drawing of whatever's on the other side (although it's still necessary to transform, project, and add to the edge list the polygons the door occludes, because they're still in the PVS—they're potentially visible if the door opens). This makes BSP models most suitable for fairly simple structures, such as

boxes, which have relatively few polygons to clip, and cause relatively few edges to be added to the edge list.

Polygon Models and Z-Buffering

Polygon models, such as monsters, weapons, and projectiles, consist of a triangle mesh with front and back skins stretched over the model. For speed, the triangles are drawn with affine texture mapping; the triangles are small enough, and the models are generally distant enough, that affine distortion isn't visible. (However, it is visible on the player's weapon; this caused a lot of extra work for the artists, and we will probably implement a perspective-correct polygon-model rasterizer in Quake *2* for this specific purpose.) The triangles are also Gouraud shaded; interestingly, the light vector used to shade the models is always from the same direction, and has no relation to any actual lights in the world (although it does vary in intensity, along with the model's ambient lighting, to match the brightness of the spot the player is standing above in the world). Even this highly inaccurate lighting works well, though; the Gouraud shading makes models look much more three-dimensional, and varying the lighting in even so crude a way allows hiding in shadows and illumination by explosions and muzzle flashes.

One issue with polygon models was how to handle occlusion issues; that is, what parts of models were visible, and what surfaces they were in front of. We couldn't add models to the edge list, because the hundreds of polygons per model would overwhelm the edge list. Our initial occlusion solution was to sort polygon-model polygons into the world BSP, drawing the portions in each leaf at the right points as we drew the world in BSP order. That worked reasonably well with respect to the world (not perfectly, though, because it would have been too expensive to clip all the polygon-model polygons into the world, so there was some occlusion error), but didn't handle the case of sorting polygon models in the same leaf against each other, and also didn't help the polygons in a given polygon model sort properly against each other.

The solution to this turned out to be z-buffering. After all the spans in the world are drawn, the z-buffer is filled in for those spans. This is a write-only operation, and involves no comparisons or overdraw (remember, the spans cover every pixel on the screen exactly once), so it's not that expensive—the performance cost is about 10%. Then polygon models are drawn with z-buffering; this involves a z-compare at each polygon-model pixel, but no complicated clipping or sorting—and occlusion is exactly right in all respects. Polygon models tend to occupy a small portion of the screen, so the cost of z-buffering is not that high, anyway.

Opinions vary as to the desirability of z-buffers; some people who favor more analytical approaches to hidden surface removal claim that John has been seduced by the z-buffer. Maybe so, but there's a lot there to be seduced by, and that will be all the more true as hardware rendering becomes the norm. The addition of particles—thousands of tiny colored rectangles—to Quake illustrated just how seductive the

z-buffer can be; it would have been very difficult to get all those rectangles to draw properly using any other occlusion technique. Certainly z-buffering by itself can't perform well enough to serve for all hidden surface removal; that's why we have the PVS and the edge list (although for hardware rendering the PVS would suffice), but z-buffering pretty much means that if you can figure out how to draw an effect, you can readily insert it into the world with proper occlusion, and that's a powerful capability indeed.

Supporting scenes with a dozen or more models of 300 to 500 polygons each was a major performance challenge in Quake, and the polygon-model drawing code was being optimized right up until the last week before it shipped. One help in allowing more models per scene was the PVS; we only drew those models that were in the PVS, meaning that levels could have a hundred or more models without requiring a lot of work to eliminate most of those that were occluded. (Note that this is not unique to the PVS; whatever high-level culling scheme we had ended up using for world polygons would have provided the same benefit for polygon models.) Also, model bounding boxes were used to trivially clip those that weren't in the view pyramid, and to identify those that were unclipped, so they could be sent through a special fast path. The biggest breakthrough, though, was a very different sort of rasterizer that John came up with for relatively distant models.

The Subdivision Rasterizer

This rasterizer, which we call the *subdivision rasterizer*, first draws all the vertices in the model. Then it takes each front-facing triangle, and determines if it has a side that's at least two pixels long. If it does, we split that side into two pieces at the pixel nearest to the middle (using adds and shifts to average the endpoints of that side), draw the vertex at the split point, and process each of the two split triangles recursively, until we get down to triangles that have only one-pixel sides and hence have nothing left to draw. This approach is hideously slow and quite ugly (due to inaccuracies from integer quantization) for 100-pixel triangles—but it's very fast for, say, five-pixel triangles, and is indistinguishable from more accurate rasterization when a model is 25 or 50 feet away. Better yet, the subdivider is ridiculously simple—a few dozen lines of code, far simpler than the affine rasterizer—and was implemented in an evening, immediately making the drawing of distant models about three times as fast, a very good return for a bit of conceptual work. The affine rasterizer got fairly close to the same performance with further optimization—in the range of 10% to 50% slower—but that took weeks of difficult programming.

We switch between the two rasterizers based on the model's distance and average triangle size, and in almost any scene, most models are far enough away so subdivision rasterization is used. There are undoubtedly faster ways yet to rasterize distant models adequately well, but the subdivider was clearly a win, and is a good example of how thinking in a radically different direction can pay off handsomely.

Sprites

We had hoped to be able to eliminate sprites completely, making Quake 100% 3-D, but sprites—although sometimes very visibly 2-D—were used for a few purposes, most noticeably the cores of explosions. As of CGDC last year, explosions consisted of an exploding spray of particles (discussed below), but there just wasn't enough visual punch with that representation; adding a series of sprites animating an explosion did the trick. (In hindsight, we probably should have made the explosions polygon models rather than sprites; it would have looked about as good, and the few sprites we used didn't justify the considerable amount of code and programming time required to support them.) Drawing a sprite is similar to drawing a normal polygon, complete with perspective correction, although of course the inner loop must detect and skip over transparent pixels, and must also perform z-buffering.

Particles

The last drawing entity type is particles. Each particle is a solid-colored rectangle, scaled by distance from the viewer and drawn with z-buffering. There can be up to 2,000 particles in a scene, and they are used for rocket trails, explosions, and the like. In one sense, particles are very primitive technology, but they allow effects that would be extremely difficult to do well with the other types of entities, and they work well in tandem with other entities, as, for example, providing a trail of fire behind a polygon-model lava ball that flies into the air, or generating an expanding cloud around a sprite explosion core.

How We Spent Our Summer Vacation: After Shipping Quake

Since shipping Quake in the summer of 1996, we've extended it in several ways: We've worked with Rendition to port it to the Verite accelerator chip, we've ported it to OpenGL, we've ported it to Win32, we've done QuakeWorld, and we've added features for Quake 2. I'll discuss each of these briefly.

Verite Quake

Verite Quake (VQuake) was the first hardware-accelerated version of Quake. It looks extremely good, due to bilinear texture filtering, which eliminates most pixel aliasing, and because it provides good performance at higher resolutions such as 512×384 and 640×480. Implementing VQuake proved to be an interesting task, for two reasons: The Verite chip's fill rate was marginal for Quake's needs, and Verite contains a programmable RISC chip, enabling more sophisticated processing than most 3-D accelerators. The need to squeeze as much performance as possible out of Verite ruled out the use of a standard API such as Direct 3D or OpenGL; instead, VQuake uses Rendition's proprietary API, Speedy3D, with the addition of some special calls and custom Verite code.

Interestingly, VQuake is very similar to software Quake; in order to allow Verite to handle the high pixel processing loads of high-res, VQuake uses an edge list and builds span lists on the CPU, just as in software Quake, then Verite DMAs the span descriptors to onboard memory and draws them. (This was only possible because Verite is fully programmable; most accelerators wouldn't be able to support this architecture.) Similarly, the CPU builds lit, tiled surfaces in system RAM, then Verite DMAs them to an onboard surface cache, from which they are texture-mapped. In short, VQuake is very much like normal Quake, except that the drawing of the spans is done by a specialized processor.

This approach works well, but some of the drawbacks of a surface cache become more noticeable when hardware is involved. First, the DMAing is an extra step that's not necessary in software, slowing things down. Second, onboard memory is a relatively limited resource (4 MB total), and textures must be 16-bpp (because hardware can only do filtering in RGB modes), thus eating up twice as much memory as the software version's 8-bpp textures—and memory becomes progressively scarcer at higher resolutions, especially given the need for a z-buffer and two 16-bpp pages. (Note that using the edge list helps here, because it filters out spans from polygons that are in the PVS but fully occluded, reducing the number of surfaces that have to be downloaded.) Surface caching in VQuake usually works just fine, but response when coming around corners into complex scenes or when spinning can be more sluggish than in software Quake.

An alternative to surface caching would have been to do two passes across each span, one tiling the texture, and the other doing an alpha blend using the light map as a texture, to light the texture (two-pass alpha lighting). This approach produces exactly the same results as the surface cache, without requiring downloading and caching of large surfaces, and has the advantage of very level performance. However, this approach requires at least twice the fill rate of the surface cache approach, and Verite didn't have enough fill rate for that at higher resolutions. It's also worth noting that two-pass alpha lighting doesn't have the same potential for procedural texturing that surface caching does. In fact, given MMX and ever-faster CPUs, and the ability of the CPU and the accelerator to process in parallel, it will become increasingly tempting to use the CPU to build surfaces with procedural texturing such as bump mapping, shimmers, and warps; this sort of procedural texturing has the potential to give accelerated games highly distinctive visuals. So the choice between surface caching and two-pass alpha lighting for hardware accelerators depends on a game's needs, and it seems most likely that the two approaches will be mixed together, with surface caching used for special surfaces, and two-pass alpha lighting used for most drawing.

GLQuake

The second (and, according to current plans, last) port of Quake to a hardware accelerator was an OpenGL version, GLQuake, a native Win32 application. I have

no intention of getting into the 3-D API wars currently raging; the observation I want to make here is that GLQuake uses two-pass alpha lighting, and runs very well on fast chips such as the 3Dfx, but rather slowly on most of the current group of accelerators. The accelerators coming out this year should all run GLQuake fine, however. It's also worth noting that we'll be using two-pass alpha lighting in the N64 port of Quake; in fact, it looks like the N64's hardware is capable of performing both texture-tiling and alpha-lighting in a single pass, which is pretty much an ideal hardware-acceleration architecture: It's as good looking and generally faster than surface caching, without the need to build, download, and cache surfaces, and much better looking and about as fast as Gouraud shading. We hope to see similar capabilities implemented in PC accelerators and exposed by 3-D APIs in the near future.

Dynamic lighting is done differently in GLQuake than in software Quake. It could have been implemented by changing the light maps, as usual, but current OpenGL drivers are not very fast at downloading textures (when the light maps are used as in GLQuake); also, it takes time to identify and change the affected light maps. Instead, GLQuake simply alpha-blends an approximate sphere around the light source. This requires very little calculation and no texture downloading, and as a bonus allows dynamic lights to be colored, so a rocket, for example, can cast a yellowish light.

Unlike Quake or VQuake, GLQuake does not use the edge list and draws all polygons in the potentially visible set. Because OpenGL drivers are not currently very fast at selecting new textures, GLQuake sorts polygons by texture, so that all polygons that use a given texture are drawn together. Once texture selection is faster, it might be worthwhile to draw back-to-front with z-fill, because some hardware can do z-fill faster than z-compare, or to draw front-to-back, so that z-buffering can reject as many pixels as possible, saving display-memory writes. GLQuake also avoids having to do z-buffer clearing by splitting the z range into two parts, and alternating between the two parts from frame to frame; at the same time, the z-compare polarity is switched (from greater-than-or-equal to less-than-or-equal), so that the previous frame's z values are always considered more distant than the current frame's.

GLQuake was very easy to develop, taking only a weekend to get up and running, and that leads to another important point: OpenGL is also an excellent API on which to build tools. QuakeEd, the tool we use to build levels, is written for OpenGL running on Win32, and when John needed a 3-D texture editing tool for modifying model skins, he was able to write it in one night by building it on OpenGL. After we finished Quake, we realized that about half our code and half our time was spent on tools, rather than on the game engine itself, and the artists' and level designers' productivity is heavily dependent on the tools they have to use; considering all that, we'd be foolish not to use OpenGL, which is very well suited to such tasks.

One good illustration of how much easier a good 3-D API can make development is how quickly John was able to add two eye-candy features to GLQuake: dynamic shadows and reflections. Dynamic shadows were implemented by projecting a model's

silhouette onto the ground plane, then alpha-blending that silhouette into the world. This doesn't always work properly—for example, if the player is standing at the edge of a cliff, the shadow sticks out in the air—but it was added in a few hours, and most of the time looks terrific. Implementing it properly will take only a day or two more and should run adequately fast; it's a simple matter of projecting the silhouette into the world, and onto the surfaces it encounters.

Reflections are a bit more complex, but again were implemented in a day. A special texture is designated as a mirror surface; when this is encountered while drawing, a hole is left. Then the z-range is changed so that everything drawn next is considered more distant than the scene just drawn, and a second scene is drawn, this time from the reflected viewpoint behind the mirror; this causes the mirror to be behind any nearer objects in the true scene. The only drawback to this approach (apart from the extra processing time to draw two scenes) is that because of the z-range change, the mirror must be against a sealed wall, with nothing in the PVS behind it, to ensure that a hole is left into which the reflection can be drawn. (Note that an OpenGL stencil buffer would be ideal here, but while OpenGL accelerators can be relied upon to support z-buffering and alpha-blending in hardware, the same is not yet true of stencil buffers.) As a final step, a marbled texture is blended into the mirror surface, to make the surface itself less than perfectly reflective and visible enough to seem real.

Both alpha-blending and z-buffering are relatively new to PC games, but are standard equipment on accelerators, and it's a lot of fun seeing what sorts of previously very difficult effects can now be up and working in a matter of hours.

WinQuake

I'm not going to spend much time on the Win32 port of Quake; most of what I learned doing this consists of tedious details that are doubtless well covered elsewhere, and frankly it wasn't a particularly interesting task and was harder than I expected, and I'm pretty much tired of the whole thing. However, I will say that Win32 is clearly the future, especially now that NT is coming on strong, and like it or not, you had best learn to write games for Win32. Also, Internet gaming is becoming ever more important, and Win32's built-in TCP/IP support is a big advantage over DOS; that alone was enough to convince us we had to port Quake. As a last comment, I'd say that it is nice to have Windows take care of device configuration and interfacing—now if only we could get manufacturers to write drivers for those devices that actually worked reliably! This will come as no surprise to veteran Windows programmers, who have suffered through years of buggy 2-D Windows drivers, but if you're new to Windows programming, be prepared to run into and learn to work around—or at least document in your readme files—driver bugs on a regular basis.

Still, when you get down to it, the future of gaming is a networked Win32 world, and that's that, so if you haven't already moved to Win32, I'd say it's time.

QuakeWorld

QuakeWorld is a native Win32 multiplayer-only version of Quake, and was done as a learning experience; it is not a commercial product, but is freely distributed on the Internet. The idea behind it was to try to improve the multiplayer experience, especially for people linked by modem, by reducing actual and perceived latency. Before I discuss QuakeWorld, however, I should discuss the evolution of Quake's multiplayer code.

From the beginning, Quake was conceived as a client-server app, specifically so that it would be possible to have persistent servers always running on the Internet, independent of whether anyone was playing on them at any particular time, as a step toward the long-term goal of persistent worlds. Also, client-server architectures tend to be more flexible and robust than peer-to-peer, and it is much easier to have players come and go at will with client-server. Quake is client-server from the ground up, and even in single-player mode, messages are passed through buffers between the client code and the server code; it's quite likely that the client and server would have been two processes, in fact, were it not for the need to support DOS. Client-server turned out to be the right decision, because Quake's ability to support persistent, come-and-go-as-you-please Internet servers with up to 16 people has been instrumental in the game's high visibility in the press, and its lasting popularity.

However, client-server is not without a cost, because, in its pure form, latency for clients consists of the round trip from the client to the server and back. (In Quake, orientation changes instantly on the client, short-circuiting the trip to the server, but all other events, such as motion and firing, must make the round trip before they happen on the client.) In peer-to-peer games, maximum latency can be just the cost of the one-way trip, because each client is running a simulation of the game, and each peer sees its own actions instantly. What all this means is that latency is the downside of client-server, but in many other respects client-server is very attractive. So the big task with client-server is to reduce latency.

As of the release of QTest1, the first and last prerelease of Quake, John had smoothed net play considerably by actually keeping the client's virtual time a bit earlier than the time of the last server packet, and interpolating events between the last two packets to the client's virtual time. This meant that events didn't snap to whatever packet had arrived last, and got rid of considerable jerking and stuttering. Unfortunately, it actually increased latency, because of the retarding of time needed to make the interpolation possible. This illustrates a common tradeoff, which is that reduced latency often makes for rougher play.

Reduced latency also often makes for more frustrating play. It's actually not hard to reduce the latency perceived by the player, but many of the approaches that reduce latency introduce the potential for paradoxes that can be quite distracting and annoying. For example, a player may see a rocket go by, and think they've dodged it, only to find themselves exploding a second later as the difference of opinion between his simulation and the other simulation is resolved to his detriment.

Worse, QTest1 was prone to frequent hitching over all but the best connections, because it was built around reliable packet delivery (TCP) provided by the operating system. Whenever a packet didn't arrive, there was a long pause waiting for the retransmission. After QTest1, John realized that this was a fundamentally wrong assumption, and changed the code to use unreliable packet delivery (UDP), sending the relevant portion of the full state every time (possible only because the PVS can be used to cull most events in a level), and letting the game logic itself deal with packets that didn't arrive. A reliable sideband was used as well, but only for events like scores, not for gameplay state. However, this was a good example of Carmack's Law: John did not rewrite the net code to reflect this new fundamental assumption, and wound up with 8,000 lines of messy code that took right up until Quake shipped to debug. For QuakeWorld, John did rewrite the net code from scratch around the assumption of unreliable packet delivery, and it wound up as just 1,500 lines of clean, bug-free code.

In the long run, it's cheaper to rewrite than to patch and modify!

So as of shipping Quake, multiplayer performance was quite smooth, but latency was still a major issue, often in the 250 to 400 ms range for modem players. QuakeWorld attacked this in two ways. First, it reduced latency by around 50 to 100 ms with a server change. The Quake server runs 10 or 20 times a second, batching up inputs in between ticks, and sending out results after the tick. By contrast, QuakeWorld servers run immediately whenever a client sends input, knocking up to 50 or 100 ms off response time, although at the cost of a greater server processing load. (A similar anti-latency idea that wasn't implemented in QuakeWorld is having a separate thread that can send input off to the server as soon as it happens, instead of incurring up to a frame of latency.)

The second way in which QuakeWorld attacks latency is by not interpolating. The player is actually predicted well ahead of the latest server packet (after all, the client has all the information needed to move the player, unless an outside force intervenes), giving very responsive control. The rest of the world is drawn as of the latest server packet; this is jerkier than Quake, again showing that smoothness is often a tradeoff for latency. The player's prediction may, of course, result in a minor paradox; for example, if an explosion turns out to have knocked the player sideways, the player's location may suddenly jump without warning as the server packet arrives with the correct location. In the latest version of QuakeWorld, the other players are predicted as well, with consequently more frequent paradoxes, but smoother, more convincing motion. Platforms and doors are still not predicted, and consequently are still pretty jerky. It is, of course, possible to predict more and more objects into the future; it's a tradeoff of smoothness and perceived low latency for the frustration of paradoxes—and that's the way it's going to stay until most people are connected to the Internet by something better than modems.

Quake 2

I can't talk in detail about Quake 2 as a game, but I can describe some interesting technology features. The Quake 2 rendering engine isn't going to change that much from Quake; the improvements are largely in areas such as physics, gameplay, artwork, and overall design. The most interesting graphics change is in the preprocessing, where John has added support for radiosity lighting; that is, the ability to put a light source into the world and have the light bounced around the world realistically. This is sometimes terrific—it makes for great glowing light around lava and hanging light panels—but in other cases it's less spectacular than the effects that designers can get by placing lots of direct-illumination light sources in a room, so the two methods can be used as needed. Also, radiosity is *very* computationally expensive, approximately as expensive as BSPing. Most of the radiosity demos I've seen have been in one or two rooms, and the order of the problem goes up tremendously on whole Quake levels. Here's another case where the PVS is essential; without it, radiosity processing time would be O(polygons^2), but with the PVS it's O(polygons*average_potentially_visible_polygons), which is over an order of magnitude less (and increases approximately linearly, rather than as a squared function, with greater-level complexity).

Also, the moving sky texture will probably be gone or will change. One likely replacement is an enclosing texture-mapped box around the world, at a virtually infinite distance; this will allow open vistas, much like Doom, a welcome change from the claustrophobic feel of Quake.

Another likely change in Quake 2 is a shift from interpreted Quake-C code for game logic to compiled DLLs. Part of the incentive here is performance—interpretation isn't cheap—and part is debugging, because the standard debugger can be used with DLLs. The drawback, of course, is portability; Quake-C program files are completely portable to any platform Quake runs on, with no modification or recompilation, but DLLs compiled for Win32 require a real porting effort to run anywhere else. Our thinking here is that there are almost no non-console platforms other than the PC that matter that much anymore, and for those few that do (notably the Mac and Linux), the DLLs can be ported along with the core engine code. It just doesn't make sense for easy portability to tiny markets to impose a significant development and performance cost on the one huge market. Consoles will always require serious porting effort anyway, so going to Win32-specific DLLs for the PC version won't make much difference in the ease of doing console ports.

Finally, Internet support will improve in Quake 2. Some of the QuakeWorld latency improvements will doubtless be added, but more important, there will be a new interface, especially for monitoring and joining net games, in the form of an HTML page. John has always been interested in moving as much code as possible out of the game core, and letting the browser take care of most of the UI makes it possible to eliminate menuing and such from the Quake 2 engine. Think of being able to browse hundreds of Quake servers from a single Web page (much as you can today with

QSpy, but with the advantage of a standard, familiar interface and easy extensibility), and I think you'll see why John considers this the game interface of the future.

By the way, Quake 2 is currently being developed as a native Win32 app only; no DOS version is planned.

Looking Forward

In my address to the Computer Game Developer's Conference in 1996, I said that it wasn't a bad time to start up a game company aimed at hardware-only rasterization, and trying to make a game that leapfrogged the competition. It looks like I was probably a year early, because hardware took longer to ship than I expected, although there was a good living to be made writing games that hardware vendors could bundle with their boards. Now, though, it clearly is time. By Christmas 1997, there will be several million fast accelerators out there, and by Christmas 1998, there will be tens of millions. At the same time, vastly more people are getting access to the Internet, and it's from the convergence of these two trends that I think the technology for the next generation of breakthrough real-time games will emerge.

John is already working on id's next graphics engine, code-named Trinity and targeted around Christmas of 1998. Trinity is not only a hardware-only engine, its baseline system is a Pentium Pro 200-plus with MMX, 32 MB, and an accelerator capable of at least 50 megapixels and 300 K triangles per second with alpha blending and z-buffering. The goals of Trinity are quite different from those of Quake. Quake's primary technical goals were to do high-quality, well-lit, complex indoor scenes with 6 degrees of freedom, and to support client-server Internet play. That was a good start, but only that. Trinity's goals are to have much less-constrained, better-connected worlds than Quake. Imagine seeing through open landscape from one server to the next, and seeing the action on adjacent servers in detail, in real time, and you'll have an idea of where things are heading in the near future.

A huge graphics challenge for the next generation of games is level of detail (LOD) management. If we're to have larger, more open worlds, there will inevitably be more geometry visible at one time. At the same time, the push for greater detail that's been in progress for the past four years or so will continue; people will start expecting to see real cracks and bumps when they get close to a wall, not just a picture of cracks and bumps painted on a flat wall. Without LOD, these two trends are in direct opposition; there's no way you can make the world larger and make all its surfaces more detailed at the same time, without bringing the renderer to its knees.

The solution is to draw nearer surfaces with more detail than farther surfaces. In itself, that's not so hard, but doing it without popping and snapping being visible as you move about is quite a challenge. John has implemented fractal landscapes with constantly adjustable level of detail, and has made it so new vertices appear as needed and gradually morph to their final positions, so there is no popping. Trinity is already

capable of displaying oval pillars that have four sides when viewed from a distance, and add vertices and polygons smoothly as you get closer, such that the change is never visible, and the pillars look oval at all times.

Similarly, polygon models, which maxed out at about 5,000 polygon-model polygons total—for all models—per scene in Quake, will probably reach 6,000 or 7,000 per scene in Quake 2 in the absence of LOD. Trinity will surely have many more moving objects, and those objects will look far more detailed when viewed up close, so LOD for moving polygon models will definitely be needed.

One interesting side effect of morphing vertices as part of LOD is that Gouraud shading doesn't work very well with this approach. The problem is that adding a new vertex causes a major shift in Gouraud shading, which is, after all, based on lighting at vertices. Consequently, two-pass alpha lighting and surface caching seem to be much better matches for smoothly changing LOD.

Some people worry that the widespread use of hardware acceleration will mean that 3-D programs will all look the same, and that there will no longer be much challenge in 3-D programming. I hope that this brief discussion of the tightly interconnected, highly detailed worlds toward which we're rapidly heading will help you realize that both the challenge and the potential of 3-D programming are in fact greater than they've ever been. The trick is that rather than getting stuck in the rut of established techniques, you must constantly strive to "do better with less, in a different way"; keep learning and changing and trying new approaches—and working your rear end off—and odds are you'll be part of the wave of the future.