Chapter 1

Quake's 3-D Engine: The Big Picture

Chapter

Understanding Quake's Client/Server Architecture from a Height

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 development 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 a 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 it: Put your head down and grind it out until it's done. Do that, and I promise you the programming world will be yours.

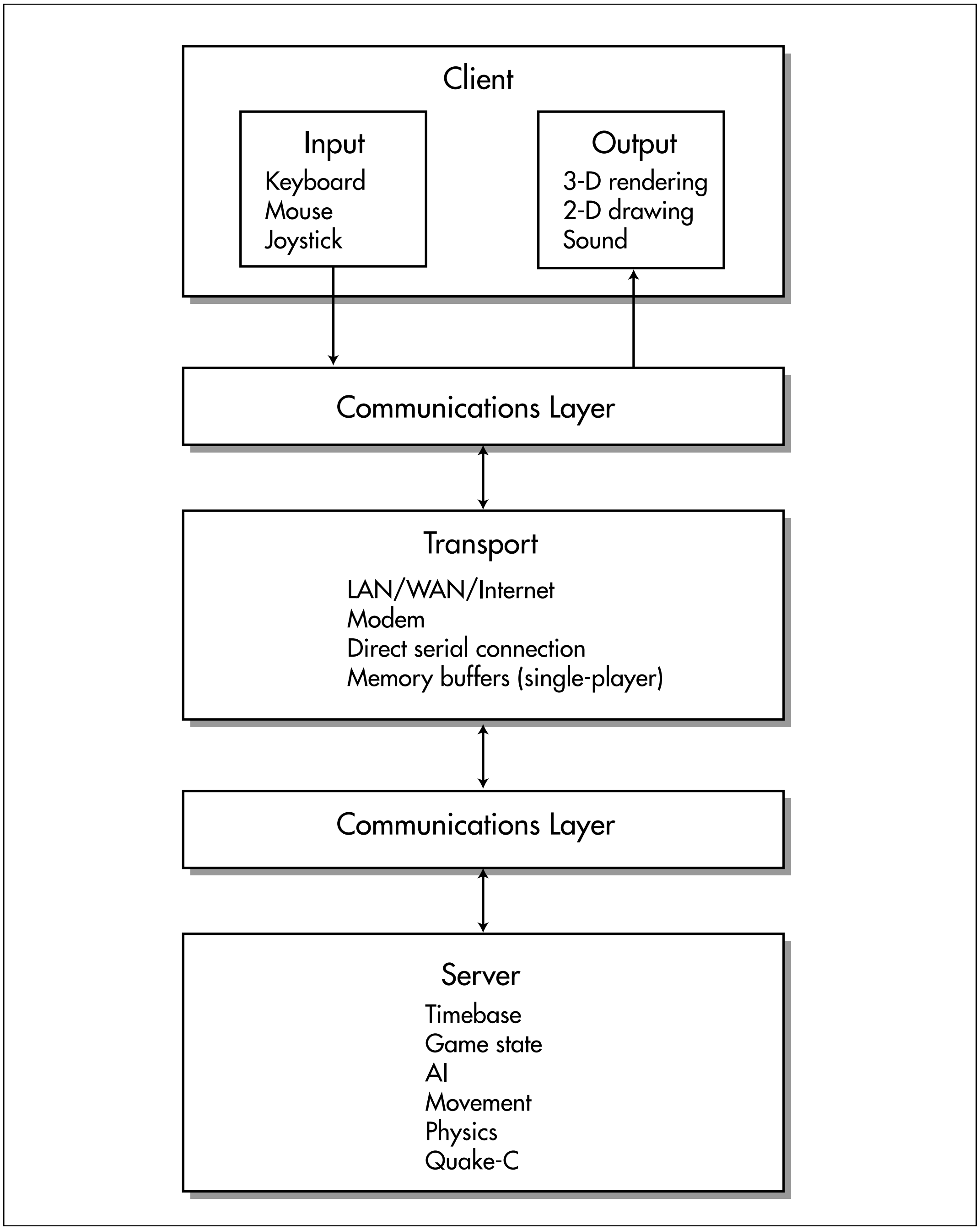
It worked for Dave; Cinematronics became a successful company and was acquired by Maxis. It worked for us at id, as well. DOOM was one of the most successful games in history, and we wanted to top it. For the programmers, the goal was to set new standards for 3-D and multiplayer—especially Internet—technology 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 I.1. 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).

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.



Quake's architecture.
Figure I.1

Communications

An interesting issue with the 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 acknowledgment is sent back, and if acknowledgment 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 round trip from server to client), and then it takes a long time to resend it (at least another round trip). 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 acknowledgment, and neither the server nor client knows nor cares whether those packets arrive or not. Each update contains all state information 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 round trip 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, which I've explained in some detail in the printed portion of this book. 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, by a process known as the *painter's 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 (front-most) 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.

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.

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.

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 texture mapping 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 I've described in the printed portion of this book.

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%!