Chapter H

Terra Incognita

Chapter H

Pondering Where Game Development Will Travel in the New World of 3-D Hardware

To begin, two anecdotes. Anecdote the first: Many years ago, I was a counselor's aide at a summer camp. For years, I had been a terrible student, with little to be proud of academically (unfortunately, reading SF by the bucketload didn't count for anything), but that school year I had gotten the hang of taking tests (though not, I admit, actually studying for them), and to my amazement had vaulted near the top of my class. Even more amazingly, I had done very well on the SATs, and I was so unused to this business of having something to boast about that it didn't take much to get me to mention my SAT score. In retrospect, I can see how obnoxious I must have been, but at the time I was in my own happy little world, and the other guys at camp were kind enough to make approving noises and keep their comments to themselves.

One afternoon, it was my job to man the headquarters office, playing the bugle calls and answering the phone. As the sun set, shadows crept across the office; I looked for the light switch, but couldn't find it. I could see the bulb on the ceiling plainly enough—heck, I could reach out and touch it—but though I traced the power cable along the unfinished ceiling and into the wall, I could find no sign of a switch. So the shadows grew deeper, until at last I was manning my post by the light of the moon. Finally, a counselor walked in and said, "Why the heck are you sitting in the dark?"

"I can't help it," I explained. "There's no light switch."

He walked to the bulb, reached up, gave it a quarter-turn, and it sprang to life. He looked at me, said, “1450, huh?” and walked out, grinning from ear to ear.

Anecdote the second: When John Carmack wrote DOOM, he discovered the wonders of using BSP trees for rendering. Most indoor renderers back then used approaches like tiling or sectors for spatial partitioning and for drawing order, but DOOM was far more flexible because it used a BSP tree. Another element John put into DOOM was line-of-sight testing, used to tell whether monsters could see the player. It took John a long time to get this code working well, and when he finally did get it working, it was quite messy and slow, doing many tests against a gridded set of world polygons.

Not long after DOOM was finished, John met Bruce Naylor, one of the leading BSP researchers in the world. In the course of their discussion, John mentioned his solution to the line-of-sight problem, and said he wished he could have thought of something cleaner. Bruce said, “But you have a BSP tree. Why not just clip the sight line into the tree?”

John thought about it, then said, “Oh my God, I’m an idiot!” It had simply never occurred to him. Alerted to the possibility, he put BSP trees to good use in Quake, reducing the line-of-sight calculations to a few dozen lines of recursive code, with O(log(n)) performance.

I’ve been asked by several people why I tell you tales of failure (relatively speaking), such as these two anecdotes, rather than cutting to the chase and telling you only the successful approaches learned in the wake of the failures. By way of an answer, I’ve had to think back over what I’ve tried to convey during the last 13 or 14 years that I’ve been writing about graphics, and I’ve concluded that in the end, if there is one insight I hope you can take away from these essays of mine and use in the future, it is this: The most important skill you can develop is not a set of techniques and solutions to problems, important as those are; rather, it is a sense of how to *develop* techniques and *solve* problems.

In your programming career, you are going to have to solve most of your problems on your own; each project is unique in many ways, and cookbook approaches will take you only so far. Certainly, if you plan to do leading-edge work, you’ll have to be inventive, and you’ll often have to fumble your way into unknown territory—*terra incognita*—as, for example, John did with BSP trees. When you do that, what you know or can look up will matter greatly, but your ability to identify and fill in the gaps in your current conception of the technology will matter far more. So be proud of what you get working, but always ask yourself what you’re missing, what simplifying assumption or tying together of seemingly unrelated areas might pay off.

Let’s start this chapter by looking at new territory that we’ll all be exploring soon: the potential for variation in hardware-based 3-D games.

3-D Hardware and Game Differentiation

As you might expect, I get quite a bit of e-mail related to my writings on graphics, and the concern most frequently expressed by readers over the past few months can be summed up as follows: Once 3-D hardware is widespread, 3-D games are all going to look pretty much the same. A common corollary is that with hardware taking over rasterization, the programming challenge—the thrill, if you will—will be largely gone from 3-D graphics. That's pretty serious stuff, especially since I expect 3-D acceleration to be standard in all new systems by the end of 1997—but are those concerns justified?

In a word, no. Sure, things will be different—*everything* changes in this industry every couple of years—but the thrill, and the programming challenge, will remain.

First off, I think we all can agree that the programming challenge would remain even if there were no variation in graphics. In my opinion, graphics currently gets an outsized share of game programming mindshare and development time, while areas such as physics, AI, biped motion, and networking are relatively shortchanged. That's understandable, because graphics is better eye-candy and is just plain more fun to develop, but there are huge steps to be taken in other areas, steps that would go farther toward improving gameplay and sales than yet another flashy water effect or a few hundred more polygons in a scene. For example, good physics modeling software—capable of handling not idealized rigid bodies but much more realistic objects with features like deformation—can currently simulate motion quite well. Unfortunately, it can't do it anywhere near fast enough; a general micro-impulse-based simulation of a pool table runs at less than one-tenth the speed of real time—for only 16 moving objects. Getting that level of realism into a game is certainly a programming challenge!

I think we can also agree that games wouldn't really look alike even if they all used exactly the same 3-D engine. Art, animation, models, and world design are all highly individualized efforts; after all, books aren't all the same because they all use the same output engine, and neither are Web pages. So what I think these concerns really come down to is that 1) games will all have the look of SGI demos, and 2) there won't be any challenging graphics programming left to do, because rendering will just be a matter of tossing a pile of polygons at the hardware.

Pick a Pile of Polygons

Ah, but *which* pile of polygons? Good hardware acceleration lets you draw a lot of polygons, but still nowhere near enough to allow brute force to do the job; there's no way the hardware is going to be fast enough to let you just draw all the polygons in your world database, with no large-scale culling. The baseline system for id Software's next-generation 3-D technology—the Trinity engine—is a 233 MHz Pentium Pro with MMX, 32 MB of memory, and a 3-D accelerator capable of 300 K texture-mapped, alpha-blended, z-buffered triangles a second and 50 megapixels/second fill rate. That's more powerful than the best PC you can buy as I write this, yet even that system is not capable of brute-force-drawing all the polygons in a large Quake level at 15 frames/second, let alone the considerably more detailed levels Trinity is targeted at. So the first part of the new programming challenge is *scene management*—efficiently extracting the limited set of polygons that need to be drawn from increasingly large world databases.

Hardware acceleration makes it possible to extend scene management into new territory as well. The ability to draw more polygons makes it possible to display highly detailed surfaces; for example, a stone wall could actually have thousands of bumps and cracks, rather than being a flat surface with a *picture* of bumps and cracks on it, as is currently the case in 3-D games. However, there is no way that an entire visible scene can be drawn with that level of detail; the transformation, projection, and set-up load would instantly bring any accelerator to its knees. Moreover, there's no reason to draw the entire scene in so much detail, because the subtleties would be lost for all but the nearest surfaces. Consequently, variable level of detail (LOD) is a huge challenge for accelerated 3-D games. For Trinity, John Carmack has experimented with using fractal surfaces to generate smoothly emergent LOD with decreasing distance, with the new vertices morphing toward their final positions, to avoid popping as new triangles are added; the early results are very encouraging. (To understand why LOD morphing is necessary, check out the current crop of flight simulators and racing games, many of which have mountains and buildings that abruptly pop into and out of existence, or snap between detail levels, reducing the sense of immersion.)

Still other programming challenges arise as you try to improve the quality of high-LOD surfaces by lighting and shadowing them properly. In fact, high-quality dynamic lighting and shadowing typify the new generation of knotty graphics problems, having more to do with scene and database management than with rasterization.

So challenges aplenty remain, but there's still the nagging concern that accelerated 3-D games will all look basically the same, once voxels and raytracing and other unique CPU-based approaches are displaced by accelerated texture-mapped, alpha-blended, z-buffered polygons. There's no question that there will be a greater tendency to use polygons, given hardware acceleration, but while this will make some approaches impractical, it also enables a wider variety of capabilities than you might imagine, especially given alpha blending, z-buffering, and the richness of the blending operations supported by 3-D APIs such as OpenGL.

Real-Time Fog Banks

For example, everyone knows that transparency and translucency can be rendered using alpha blending. (Consider the drawing-order constraints involved in translucency, though, and you'll see another reason why programming for an accelerator involves more than just sending polygons to the hardware.) However, it's far less obvious that a second, alpha-blend pass with a light map for a texture can produce precisely the same detailed, perspective-correct, vertex-independent lighting as

Quake's surface caching. Better yet, lighting with a second alpha pass uses little extra texture memory (the light maps have a density of one light point every 256 texels, and intensities are 8-bit alpha values), and makes for level performance, because, unlike surface caching, there are no cache misses to deal with. Given the richness of OpenGL's 3-D pipeline, there are surely many similar techniques yet to be devised.

Also, surface caching is quite feasible with a hardware accelerator; the only downside is that surfaces are larger than texture tiles, so they take longer to download and require more texture memory. Still, MMX and faster processors make it possible to do quite a bit of procedural texturing while building a surface, all the more so since an accelerator can rasterize in parallel while the CPU builds surfaces. It might be too expensive to use surface caching for all surfaces in a game, but it's certainly a possibility for special surfaces when effects like pseudo-bump-mapping, swirling plasma, and burning paper are needed. Which brings us to another class of 3-D programming challenges yet to be solved in the new world of 3-D acceleration: fire, water, rain, snow, swirling fog, and the like. Consider just the problem of a spotlight beam illuminating floating motes of dust, or a streetlight shining into thick, shifting fog, and I think you'll agree: It's going to be a lot of fun to be a 3-D programmer over the next few years.

More on Win32 Game Programming

Earlier in this book, I discussed some information about Win32 game programming that I had found useful as I ported Quake to Win32. Now, with some additional experience, I have some corrections and clarifications that need to be made, as well as some new information.

Last time, I described how **ChangeDisplaySettings()** can be used to change the video mode, even if DirectDraw isn't installed (although on Win95, only the resolution can be changed this way; changing the bit depth requires DirectDraw). I described the **CDS_FULLSCREEN** flag to **ChangeDisplaySettings()**, which tells GDI not to move and/or resize the windows on the desktop to match the new resolution; if this flag is not used, then when you return from, say, 640×480 to a 1024×768 desktop, all the windows will be scrunched into the upper left 640×480 of the desktop. However, I neglected to mention that when you call **ChangeDisplaySettings()** with a **NULL** pointer for the **DEVMODE** parameter (as you'd typically do when your program exits, to reset the mode to the desktop default), you must again use **CDS_FULLSCREEN**, as in: **ChangeDisplaySettings(NULL, CDS_FULLSCREEN)**, or else window resizing may occur. Also note that due to a peculiarity of Windows, DOS boxes may get scrunched on some systems even if you do use this flag.

Letting Sleeping Screen Savers Sleep

I also reported that calling **SetCursorPos()** each frame, to recenter the mouse so that it would have maximum range of motion before clipping against the edge of the screen, seemed to keep the screen saver from starting, a potentially useful side effect. That still appears to be true of Win95, but there is no guarantee that this applies to all systems, and it is definitely not true of NT. An alternative, documented way to keep the screen saver from starting is **SystemParametersInfo(SPI_SETSCREENSAVEACTIVE, FALSE, NULL, 0)**; however, this has the drawback of leaving the system in an incorrect state if your app should happen to crash or otherwise terminate improperly. A better approach, sent to me by Todd Laney, is to block the screen saver messages in your window proc, as shown in Listing H.1.

LISTING H.1 LH-1.C

```
//
// Disables screen saver
//
LRESULT CALLBACK ToplevelWindowProc(HWND hWnd, UINT uMsg,
     WPARAM wParam, LPARAM lParam)
{

    switch (uMsg)
    {
        case WM_SYSCOMMAND:
            switch (wParam & ~0x0F)
            {
                case SC_SCREENSAVE:
                case SC_MONITORPOWER:
                    //
                    // Don't call DefWindowProc() or
                    // it will start the screen saver
                    //
                    return 0;
            }
        //  :
        // other cases
        //  :
    }
}
```

The reason that disabling the screen saver was of interest to us was that the sound hardware is not currently a fully shareable resource. If any app has a wave device open when you try to create a DirectSound object, the creation will fail. Likewise, if a DirectSound object exists or a wave device is already open when you try to call **waveOutOpen()**, your wave open will fail. In the interests of being a good Win32 citizen, we used to shut down Quake's sound (either DirectSound or waveOut, depending on whether the system had a DirectSound driver) every time we lost the focus, including if a screen saver started. The problem was that in systems that had sounds that played on maximization, the system would play that sound using **waveOut()** when the screen saver was turned off and Quake took over the screen again—and because the sound was playing when Quake tried to restart its sound code, Quake's sound failed to reinitialize and sound was lost, much to the player's irritation.

We originally thought to work around this by preventing the screen saver from starting, but similar problems can happen if Quake is minimized or running in the background. For example, if Quake is running in a window, and another window is activated and starts playing sound (such as Media Player being used to start playing a wave file), then Quake will be unable to produce sound when it's made the active app. We decided to resolve this by having Quake never relinquish the sound hardware once it has initialized it; the DirectSound object or wave handle is kept around for the entire session, even if Quake isn't in the foreground. (At Quake startup, if the sound hardware is already allocated to another app, the user is prompted and given the option of trying again to start sound, presumably after shutting down the offending app, or running Quake with no sound.) The drawback of this approach is that when Quake is running, even if it's minimized, no other app can play wave sound, although other DirectSound-based apps will be able to make sound when Quake isn't the active app, because DirectSound has built-in resource-sharing capabilities.

Should DirectSound Do Its Own Mixing?

I have one more tale to tell about DirectSound. As I described earlier, we're using DirectSound in an unusual way, by using our own mixing code to mix all sound channels into a single DirectSound streaming secondary buffer, taking care to stay ahead of the location at which DirectSound allows us to write to the buffer. This necessitates mixing about a tenth of a second ahead, resulting in significant latency. Consequently, we decided to try using DirectSound as it's intended to be used, by having each of our sounds in a separate secondary buffer, and letting DirectSound mix the buffers for us. The result was startling: Latency did indeed drop, but performance dropped even more. It appears that DirectSound is slow at starting lots of sounds (and when you're firing the nail gun in Quake, a *lot* of sounds are starting), so in scenes where dozens of sounds were constantly starting, Quake's performance dropped by as much as half compared to our old, mix-it-ourselves architecture. At this point, I would have to conclude that the performance of DirectSound is not good enough for a real-time game with many sounds constantly starting, unless you do your own mixing.

Nearly as surprising was the discovery that we could get a 5% to 10% speedup by mixing our sounds into a wave buffer rather than a secondary DirectSound buffer, and playing through **waveOut()** rather than DirectSound. Unfortunately, **waveOut()** imposes still more latency, enough to be quite noticeable on some systems, so neither approach is ideal. We've made DirectSound the default in Quake, but let the player select wave sound if they'd rather have a higher frame rate at the cost of greater sound latency.

Overall, DirectSound is a good sound API, and I expect that as upcoming releases of DirectSound solve these performance problems and clean up a few rough spots, both wave sound and mix-it-yourself approaches will become distant memories for game programmers. Right now, though, you may not be able to get the performance you want using DirectSound in a straightforward manner.

Keep Looking for the Switch!

There is limitless great software yet to be written, so keep on coding. And no matter how talented and experienced you are, remember—somewhere in your code there's almost certainly a light bulb waiting to be screwed in just a quarter-turn more!