Chapter J

Things I Learned Porting Quake to Win32

Chapter J

Learning to Coexist with a Seriously Game-Unfriendly API

I turned 40 yesterday, as I write this. Most of the time I feel like I'm the same hip, with-it person I was at 20, but occasionally I catch myself talking in intense, serious tones about things like 401(k) plans, gum disease, and fiber, and I worry less about how cool my friends think my car is than whether someday soon my daughter is going to want to pierce some part of her body that I can't see. Maybe my memory is going too, but I'm pretty sure I didn't have very many brain cells devoted to those topics 20 years ago. I think I can sum this whole aging business up pretty well by saying that when I stop to think about it, life is better at 40 than 20 in lots of ways, but overall it's definitely way more boring.

This is not entirely a bad thing. Life was more exciting when I was 20 largely in the sense that I did a lot of dumb things back then. For example, there's the time we tried to replace the gas tank in Danny Haselkorn's car at 2 a.m. with a streetlight for illumination, using a single crescent wrench, about which the best thing I can say is that not a single person required skin grafts afterward. Or the time we decided to conduct desk-hurdling races in the high-school Honors room, and learned—to the amazement of our brilliant, Honors-quality, soon-to-be-in-detention minds—that floors conduct sound, and that we were on the second floor over a class taking a test. But my point is that nowadays, I'm too busy flossing to get in trouble, which seems like a pretty good deal when you consider a guy I work with who I'll call "Art."

Art's a twentysomething guy who a few months back was at dinner with some friends, when his friend Zeke suddenly decided it would be fun to go for a ride on a partially completed highway nearby that wasn't yet open to the public. After all, how often do you get the chance to have an entire highway all to yourself? So they all piled into Zeke's car, and soon they were skirting a barrier and accelerating up an on-ramp.

After a while, as they headed down a ramp to get back on the normal, open highway, they noticed a light underneath an overpass, and someone said, "Is that a police car?" So they all strained to make out the shape behind the light, which is why no one noticed the cement barrier blocking the ramp until Art screamed, "Watch out!", giving Zeke plenty of time to slow from 50 mph all the way down to 40 mph before they slammed into the barrier and came to a dead stop.

It was indeed a police car. The Texas state trooper drove up the ramp, got out his flashlight, and checked them over. Miraculously, no one was seriously hurt. Then he said, "Yew kids been drinking?"

"No, officer," they assured him.

"Yew kids been doin' drugs?"

"Absolutely not, sir," they said.

"Well, then," he noted, "I guess you're just sheer stupid."

And with that, we come to this chapters's thesis, which is mainly that Art's story is good for a laugh, and a laugh is definitely useful both when turning 40 and when learning Win32 game programming. I wouldn't call either one exactly *fun*—but in the long run, they sure beat the alternatives!

Win32 and Games

During my tenure at id Software I spent a few months porting Quake to Win32. I can now unequivocally say that someday DirectX will make Win32 game programming a breeze—but today is not that day. DirectX, particularly DirectSound, does finally make DOS-quality games possible on Win32 (by which I mean Win95, and NT version 4.0 or later), but right now it's harder to write a bug-free, commercial-quality game for Win32 than for DOS—doable, but definitely hard. Most of the problems are the usual ones for new technology: buggy drivers, inadequate and sometimes incorrect documentation, and missing functionality. There are also some problems unique to DirectX, such as its tendency to grab the Win16 lock on Win95 (so that, for example, while you have a page-flipped back buffer locked through DirectDraw, neither DirectSound nor networking calls will work), and of course there are the normal complications that come with writing a Win32 app, such as handling loss of input focus and coexisting with screen savers and such.

Nonetheless, Win32 is surely the future of personal computers, and DirectX looks to be the future of personal computer games; we had best learn to write games that work well with both. The driver and bug issues will solve themselves over time, but the complexity of Win32 game programming won't, so our job is to learn as much as we can of the arcane lore that makes Win32 and DirectX programs work well, in pursuit of which I'll spend the rest of this chapter discussing hard-earned Win32 and DirectX information from the WinQuake port. I'm not going to try to provide tutorials on these topics, but rather point out specific information that isn't readily apparent from available documentation (particularly the DirectX SDK and the MSDN CD, both of which are absolutely essential for Win32 game programming; you can find out more about them at www.microsoft.com).

DirectSound

DirectSound filled the most glaring game-support hole in the Win32 API: low-latency sound. Win32's built-in wave output is fine for normal applications, but isn't quite good enough for real-time games because playing of sound is audibly delayed from the time the wave buffer is played, a problem that DirectSound completely fixes when a proper DirectSound driver is installed. (More on driver issues shortly.) Generally, DirectSound works well, but I've run into a few significant gotchas, especially concerning primary sound buffers.

DirectSound supports two types of buffers, primary and secondary. Secondary buffers are basically individual sound sources for DirectSound to mix together and play. Secondary buffers could possibly be in hardware, but basically they're just places to store sound so you can tell DirectSound where to find them to mix them for you. Primary buffers are quite different. There can only be one primary buffer per sound device, and that buffer is the real hardware sound buffer, generally the DMA buffer from which the sound card pulls its data in real time.

Bugs in DirectSound Primary Buffers

The DirectSound documentation recommends using only secondary buffers, because secondary buffers are simpler (DirectSound does all the mixing and format translation), and because hardware acceleration for sound mixing can't be used with primary buffers. To that list I'll add another, more compelling reason: *Primary buffers don't work reliably.*

Quake wrote directly to the DMA buffer under DOS, so I figured I'd just port the sound code to use the primary buffer in the same way under DirectSound, without having to change much, but as it turned out I could never get primary buffer sound to work reliably. The killer problem that finally caused me to give up and switch to a secondary buffer was that on some systems, when I'd minimize the WinQuake window and then restore it, the sound would play back twice as fast as it should have, making the fearsome Ogres sound like Munchkins on helium. DirectSound automatically takes away sound output when an app loses the focus and restores it when the app regains the focus, and apparently there's a bug when DirectSound tries to do that with a primary buffer. The bottom line is, secondary buffers work fine, and you should avoid playing sound through primary buffers like the plague.

Matching Sound Data Format to the Hardware Playback Speed

One of the simplifications of secondary buffers is that you can store the sound in whatever format (11/22/44 KHz, 8- or 16-bit) that you'd like, and DirectSound will handle the conversion. However, we've found it works best if your secondary buffers match the speed the hardware is operating at. In particular, if you play a secondary buffer with 11 KHz data on hardware configured for 22 KHz playback, you'll get soft, scratchy sound artifacts. The problem is that playing each 11 KHz sample twice at 22 KHz is not the same as playing each sample once at 22 KHz, because the sound hardware is effectively an integrating, smoothing device, and responds differently in the two situations, producing high-frequency artifacts.

One possible solution is to filter your sound data into whatever format the hardware is using; you can determine this format by creating a primary buffer and using the **GetFormat** method to query the configuration. Another solution, now used in WinQuake, is to create a primary buffer and use the **SetFormat** method to set the hardware playback speed, as shown in Listing J.1; all WinQuake sound data is at 11 KHz, 8-bits, so we set the primary buffer to 11 Khz (11025, to be exact), 8-bits, two channels. However, we've found that some sound cards don't allow this, instead failing the **SetFormat** call; in those cases, we just leave the hardware in the default configuration, and live with the resulting sound artifacts.

LISTING J.1 LJ-1.C

```
// try to create a primary buffer and set its format, to configure
// the hardware
#include <windows.h>
#include <dsound.h>
LPDIRECTSOUNDBUFFER SetSoundHWFormat (LPDIRECTSOUND pDS,   int channels, int bits_per_sample,
int samples_per_sec)
{
   DSBUFFERDESC         dsbuf;
   WAVEFORMATEX         format;
   LPDIRECTSOUNDBUFFER  pDSPBuf;

   memset (&format, 0, sizeof(format));
   format.wFormatTag = WAVE_FORMAT_PCM;
   format.nChannels = channels;
   format.wBitsPerSample = bits_per_sample;
   format.nSamplesPerSec = samples_per_sec;
   format.nBlockAlign = format.nChannels*format.wBitsPerSample / 8;
   format.cbSize = 0;
   format.nAvgBytesPerSec = format.nSamplesPerSec*format.nBlockAlign;

   memset (&dsbuf, 0, sizeof(dsbuf));
   dsbuf.dwSize = sizeof(DSBUFFERDESC);
   dsbuf.dwFlags = DSBCAPS_PRIMARYBUFFER;
   dsbuf.dwBufferBytes = 0;
   dsbuf.lpwfxFormat = NULL;
```

```
    if (pDS->lpVtbl->CreateSoundBuffer(pDS, &dsbuf, &pDSPBuf, NULL) ==
         DS_OK)
    {
       if (pDSPBuf->lpVtbl->SetFormat (pDSPBuf, &format) == DS_OK)
          return pDSPBuf; // succeeded in configuring hardware
       else
          return NULL;    // didn't succeed in configuring hardware
    }
    else
    {
       return NULL;   // couldn't create a primary buffer
    }
}
```

When DirectSound Drivers Are Missing

Although someday soon coverage will be universal, not all sound hardware has a DirectSound driver right now, and in fact there are no DirectSound drivers for NT at all. (Before you say, "Who cares about games on NT," let me say that I consider a PentiumPro running WinQuake to be a great platform for Internet play, thanks to NT's networking; the only flaw is high sound latency.) When there is no DirectSound driver installed, DirectSound transparently switches to using wave APIs, which sounds like a good thing, but didn't work well in our case. The problem is that the submission chunk size DirectSound uses is apparently quite large, and sometimes DirectSound using wave output exhibited extremely long latencies (hundreds of milliseconds), and very coarse granularity for advancing streaming sound buffers' write pointers. It's possible that this problem is unique to our sound architecture, and it's only an issue if you're generating sound into a streaming buffer on the fly, but if you encounter sound problems on systems without true DirectSound drivers, you might try switching to wave output with small submission chunks (we used 1 K blocks) and see if that fixes it; we observed that a whole host of sound problems, especially on NT, went away when we did this. You can tell if a given system has a true DirectSound driver with the code in Listing J.2.

LISTING J.2 LJ-2.C

```
// determines whether there's real DirectSound support in the sound
// driver, or just wave-based emulation
#include <windows.h>
#include <dsound.h>

int RealDirectSoundDriver (LPDIRECTSOUND pDS)
{
    DSCAPS   dscaps;

    dscaps.dwSize = sizeof(dscaps);

    if (pDS->lpVtbl->GetCaps (pDS, &dscaps) != DS_OK)
       return 0;   // couldn't get capabilities

    if (dscaps.dwFlags & DSCAPS_EMULDRIVER)
       return 0;   // no real DirectSound driver support, just waves
```

```
   else
      return 1;   // real DirectSound driver support
}
```

A final note: We've fixed some nasty sound (and graphics) problems simply by installing the latest Microsoft-supplied version of DirectX. This file is available at this writing as the 6.3 MB file http://www.microsoft.com/mediadev/downloads/directx.zip. Depending on when you read this book, a newer and better version may be available; always check the Microsoft Web site, where this and a great many other useful things can be found. There have been many DirectX releases, with many bug fixes, and there are also problems with some manufacturers' driver support for DirectX. (For example, as I write this, the latest SB16 drivers from Creative Labs crash WinQuake on some machines, but the MS-supplied DirectSound drivers don't.) Now, when WinQuake tech support problems crop up on only certain machines, the first course of action is to have the owners of those machines upgrade to the latest Microsoft-supplied version of DirectX.

Painless Mode Changes

DirectDraw makes it easy to change both the resolution and bits per pixel of the screen. Unfortunately, before you can do that, there must be a DirectDraw driver installed, and therein lies the rub. As with DirectSound, someday soon DirectDraw support will be universal, at least in factory-configured systems, but that's not the case in mid-1997. One workaround is to have your app's installation program install DirectX (including drivers) as well as your program, but there are two problems there. First, there may simply not be a driver for whatever adapter you're installing on; there are tens of millions of adapters out there that are no longer in production, and manufacturers have little incentive to write DirectDraw drivers for those cards, even assuming the manufacturer is still in business. Even if a driver does exist, installing it can cause the end user problems. For example, when I first installed DirectX on my home machine, the 1024×768 mode I was using suddenly became interlaced, and thoroughly unusable. This is the sort of thing that causes support calls and general unhappiness, and a number of developers I know are understandably reluctant to do it.

APIs for Full-Screen 640X480 Windows at Any Resolution

What choice do they really have, though? Sure, they could run in a window, but on most desktops these days, even a 640×480 window is pretty small, and doesn't deliver a great game experience. A full-screen window is another possibility, but that means 800×600, 1024×768, or even 1280×1024 (heck, my desktop at work is 1600×1200), and even with **StretchBlt()**, there's just no way that that's going to run fast enough; decent performance even at 640×480 is often a stretch for a sophisticated game. It turns out, though, that this is largely fixable with the help of two little-known APIs,

EnumDisplaySettings() and **ChangeDisplaySettings()**. You can find these APIs if you search on them by name in MSDN, but I've never found any links that lead to them, so it's easy to miss them.

EnumDisplaySettings() describes all the modes that your system is capable of, and **ChangeDisplaySettings()** (CDS, for short) lets you change the current video mode to any of those modes. This means that no matter what Win32 system you're running on, you can always switch to a 640×480 resolution, create a fullscreen popup window, create a DIB section to draw into and to copy or **StretchBlt()** to the screen, and have the screen to yourself, just like a DOS app—all without installing anything, or relying on anything other than built-in Win32 APIs.

Listing J.3 shows how to set a video mode; a larger sample program that exercises enumerating and setting display modes is available on the CD-ROM in the file dispset.zip. One important point illustrated by dispset.zip is that when you use DIB sections for real-time graphics, you should make sure you have an identity palette, or else copying the DIB to the screen will be slower than it needs to be.

LISTING J.3 LJ-3.C

```
// functions to manipulate the desktop video mode
#include <windows.h>
// attempts to set the desktop video mode to the specified
// resolution and bits per pixel
int SetVideoMode (int bpp, int width, int height)
{
   DEVMODE  gdevmode;

   gdevmode.dmFields = DM_BITSPERPEL | DM_PELSWIDTH | DM_PELSHEIGHT;
   gdevmode.dmBitsPerPel = bpp;
   gdevmode.dmPelsWidth = width;
   gdevmode.dmPelsHeight = height;

   // CDS_FULLSCREEN keeps icons and windows from being moved to
   // fit within the new dimensions of the desktop
   if (ChangeDisplaySettings (&gdevmode, CDS_FULLSCREEN) !=
         DISP_CHANGE_SUCCESSFUL)
   {
      return 0;
   }

   return 1;
}

// puts back the default desktop video mode
void RestoreDefaultVideoMode (void)
{
   ChangeDisplaySettings (NULL, 0);
}
```

This sort of full-screen window has several disadvantages relative to DirectDraw full-screen sessions: no page flipping, and hence possible frame shearing (although under DOS, Quake doesn't page flip by default, and we've had no complaints); no ability to change palette colors 0 and 255 (note, though, that currently these colors are never changeable, even under DirectDraw, on NT); no support for DirectDraw hardware acceleration such as transparent blts; and usually no resolutions below 640×480 (although currently few DirectDraw drivers support resolutions lower than 640×480, except for the limited Mode X support).

The most important limitation, however, is that on Win95 the screen resolution can be changed on the fly via CDS, but the bits per pixel value cannot be changed without rebooting. This means that if someone is running their system at 1024×768, 16-bpp under Win95, you can use CDS to easily switch to 640×480, 16-bpp, but there is no way short of rebooting to get 640×480, 8-bpp. (There is no such limitation on NT; any mode may be selected at any time.) Your 8-bpp app will still work at 16-bpp, because GDI will translate the 8-bpp DIB to 16-bpp while copying or stretch bltting to the screen, but performance will suffer somewhat from the translation. (You could alternatively choose to work directly with 16-bpp DIBs in 16-bpp modes, but most games are currently written only for 8-bpp operation, both for performance reasons and because supporting multiple color depths is a lot of work.)

On the plus side, not only does CDS work on all systems, it also doesn't conflict with normal Windows debuggers, because your app is still running on the normal desktop, just in a different mode. So, for example, you can set a breakpoint in MSVC++, run a CDS-based app, and actually use the debugger when it breaks in the middle of rendering; try the same thing in a DirectDraw mode, and you'll hang the system. (I use SoftIce to get around this, and SoftIce is certainly a handy thing to have around, but MSVC++ is both easier to use and cheaper.) Also, unlike DirectDraw, full-screen windows based on CDS don't need to lock surfaces. In fact, full-screen windows can use much of the same code as regular windows, although there are some details to attend to, such as catching deactivation via **WM_ACTIVATE** and putting back the default video mode, which is done by calling **ChangeDisplaySettings(NULL, 0)**.

One very important—and poorly documented—aspect of CDS is **CDS_FULLSCREEN**. Setting this flag in the second parameter to CDS keeps all the icons and windows on the desktop from being shifted around to fit the new desktop size. Without this, changing the mode to 640×480 and then returning to, say, 1024×768 will result in everything on the desktop being scrunched into the upper-left corner of the display.

All other things being equal, DirectDraw has the potential to produce better results than CDS, but CDS is good enough for all but the most demanding apps, and beside being a lot easier to work with, it is (at this writing) more reliable and more widely supported. Even if you decide that DirectDraw is the best choice for your apps, it's still handy to implement CDS support so that if a user is unable to run with DirectDraw due to bugs or lack of drivers, your apps can still offer good full-screen performance; this fallback is now implemented in WinQuake.

Win32 Mouse Matters

The final Win32 topic I'll address in this chapter is the mouse. Win32 has mouse APIs, of course, but they're problematic for twitch games, for reasons I'll discuss in a moment. DirectX versions 1 and 2 didn't have any special mouse support; DirectX 3 does add low-level mouse support to DirectInput, but I've heard reports of problems with DirectInput 3 running on a number of systems, and besides, DirectInput 3 doesn't yet support NT mice, so for now we're back to square one: the Win32 mouse APIs.

The basic problem with Win32's mouse support is that it's screen-oriented, not mouse-oriented. DOS mouse drivers can tell you how far the physical mouse moves, in units called "mickeys." In contrast, Win32 tells you where the mouse pointer is on the screen, in pixels; to measure mouse movement, you have to remember the last position (which for twitch games is usually reset to the center of the screen after each reading), and calculate the difference between that and the current position. The problem is that Windows doesn't let the mouse move off the screen, so mouse movement—which for a twitch game has nothing to do with screen pixels—gets clamped to the limits of the screen. Even at 640×480, it's easy to move the mouse far enough in a single graphics frame to go from the center past the edge of the screen, in which case Windows just stops it dead when it reaches the edge, causing sharp mouse movements to turn into leaden game response.

Worse, Windows allows the user to select mouse responsiveness through Control Panel, and has a concept of mouse acceleration, under which if the mouse is moved more than a few pixels in a single measurement, Windows will transparently double or quadruple the movement (the idea being that if the user isn't performing a fine movement, they probably want to get the mouse across the screen to another window or another control in a hurry). All that makes good sense for a GUI, but makes for erratic, often-unresponsive gameplay.

The solution we came up with for WinQuake consists of two parts. First, we call the **SystemParametersInfo()** interface to get the current settings for mouse speed and acceleration, using the **SPI_GETMOUSE** function, then use the **SPI_SETMOUSE** function to force the mouse speed to medium and to turn mouse acceleration off, thereby removing the variability associated with the normal GUI mouse configuration. We also use **ShowCursor(FALSE)** to make the cursor invisible. Second, we insert calls to get the current mouse position (via **GetCursorPos()**) at several approximately evenly spaced points in WinQuake's game loop, so the mouse position is checked and re-centered (via **SetCursorPos()**) a half-dozen times per frame, rather than the usual once per frame. With the mouse speed at medium and no mouse acceleration, the extra re-centerings are enough to keep the mouse from ever reaching the edge of the screen. The multiple mouse movements per frame are summed, and the result is used as the mouse movement for the next frame. Listing J.4 presents code to alter and restore the mouse settings, and to accumulate the mouse position multiple times per frame. The file dispset.zip, mentioned earlier, demonstrates this method of mouse input.

LISTING J.4 LJ-4.C

```
// functions to start up and shutdown the game configuration for
// the mouse, and to accumulate mouse moves over multiple calls
// during a frame and to calculate the total move for the frame
#include <windows.h>

extern int window_center_x, window_center_y;

// the two 0 values disable the two acceleration thresholds, and the
// 1 value specifies medium mouse speed
static int     originalmouseparms[3], newmouseparms[3] = {0, 0, 1};
static int     mouse_x_accum, mouse_y_accum;
static POINT   current_pos;

int StartGameMouse (void)
{
   if (!SystemParametersInfo (SPI_GETMOUSE, 0, originalmouseparms, 0))
      return 0;

   if (!SystemParametersInfo (SPI_SETMOUSE, 0, newmouseparms, 0))
      return 0;

   ShowCursor (FALSE);
   SetCursorPos (window_center_x, window_center_y);
   return 1;
}

void StopGameMouse (void)
{
   SystemParametersInfo (SPI_SETMOUSE, 0, originalmouseparms, 0);
   ShowCursor (TRUE);
}

void AccumulateGameMouseMove (void)
{
   GetCursorPos (&current_pos);

   mouse_x_accum += current_pos.x - window_center_x;
   mouse_y_accum += current_pos.y - window_center_y;

   // force the mouse to the center, so there's room to move
   SetCursorPos (window_center_x, window_center_y);
}

void GetGameMouseMoveForFrame (int * mouse_x_move, int * mouse_y_move)
{
   GetCursorPos (&current_pos);
   *mouse_x_move = current_pos.x - window_center_x + mouse_x_accum;
   *mouse_y_move = current_pos.y - window_center_y + mouse_y_accum;
   mouse_x_accum = 0;
   mouse_y_accum = 0;

   // force the mouse to the center, so there's room to move
   SetCursorPos (window_center_x, window_center_y);
}
```

One thing to look out for if you use **SystemParametersInfo()** to change the mouse settings is that you should restore those settings not only on exit, but also if you lose the focus, for example due to the user pressing Ctrl+Esc or Alt+Tab. I can tell you from experience that it's a weird sensation to return from 320×200 to a 1600×1200 desktop and try to use a medium-speed mouse with no acceleration.

Another interesting point is that each time you call **SetCursorPos()** to re-center the mouse, a mouse-movement message is sent to your message loop. This wouldn't be of much consequence, except that apparently Windows sees this message and concludes the mouse is active, and as a result doesn't turn on the screen saver. This can actually be a benefit if you don't want to deal with the complications of having a screen saver kick in and take over the screen from your app, but in any case, be aware of this unexpected behavior.

Throwing the Mouse Out the Window

You can, if you wish, use the mouse to control a game in a window exactly as you would full-screen. However, in this case you should use **SetCapture()** to limit mouse movement to your window. Otherwise, it's possible that a sharp mouse movement will snap the cursor out of the window; once it's out of the window, it becomes visible and, worse, will cause a focus switch if the user clicks a mouse button. And, of course, don't forget to call **ReleaseCapture()** when you lose focus or exit your app. (If you do forget, don't worry—it's just nature's way of telling you you're not as young as you used to be. I highly recommend adding more fiber to your diet. Or possibly getting a nose ring.)