gd

GAME DEVELOPER MAGAZINE

JUNE/JULY 1996

UBM Tech

Reboot

No, there's nothing wrong with your dials. You may have been expecting Larry O'Brien's sagacious words in this space, and here you are reading a column by the guy who formerly wrote Crossfire. "Just what the heck is going on?!" you might be asking.

Changes are afoot here at *Game Developer*, though nothing so radical as a Spindler reorg. Larry's been kicked upstairs to editorial director, which means that he gets to lean back in his chair and watch pandemonium unfold, rather than having to dive into the mire with the rest of us. Oh, and he's also doing a lot more programming (some people get to have all the fun). So, instead, I've taken over much of the day-to-day housekeeping here at the magazine.

Now that I've been entrusted with a smidge of power, my first goal is to work towards beefing up the number of articles every issue. More bang for your buck. I'm going to try to squeeze every inch of this magazine to fit as many articles as possible into upcoming issues. As my previous beat was industry news and analysis, this column will address these issues, and the Crossfire column will slowly fade into the sunset.

This issue, we examine Microsoft's DirectX II SDK. If you haven't looked at the SDK yet, check out Robert Hess's article on page 24 which provides an overview of the kit. If you've already started down the DirectX trail, however, this article may be slightly remedial. Don't fret, *Game Developer* isn't reneging on its commitment to high quality, technically deep content. We did feel, however, that since there is going to be substantial coverage in the magazine about the current and future versions of DirectX SDK, it would make sense to have a base to build upon. Besides the articles in this issue on the Windows 95 Game SDK, we'll feature three articles in the next issue on DirectDraw, DirectInput, and Direct3D. Though our demographics indicate that most of our readers work on games for Intel-based machines, we'll also add more diverse platform coverage. We'll be devoting coverage to the upcoming Macintosh Game SDK for those of you working on Mac games.

Beginning this fall, we'll review development tools. The first review was going to cover C++ compilers, but, wouldn't you know it, Chris Hecker stole my thunder with his latest series on compilers. But look for reviews of various 3D animation and C++ tools.

As with every byte of content that goes into the magazine, let us know what appeals to you and what doesn't. Want more space in the magazine devoted to articles, and have us post code on our FTP site instead of taking up space on the pages? Or perhaps you want more code in the magazine and fewer articles. Are there any subjects that we haven't covered that you'd like to read about? Let us know either through our web site, http://www.gdmag.com, or pick up a pen and scribble us a note.

I've just returned from the Computer Game Developers Conference in Santa Clara, Calif., which was more packed with people than ever before—close to 4,000. Check out the Bit Blasts column, as well as our web site for information about new products that were launched at the show. ■

Alex Dunne
Senior Editor

GAME DEVELOPER

Editorial Director Larry O'Brien *gdmag@mfi.com*
Senior Editor Alex Dunne *76702.1142@compuserve.com*
Managing Editor Diane Anderson *dianderson@mfi.com*
Editorial Assistant Jana Outlaw *joutlaw@mfi.com*
Contributing Editors Barbara Hanscome *bhanscome@mfi.com*, Chris Hecker *checker@bix.com*, David Sieks *103302.301@compuserve.com*
Web Site Manager Phil Keppeler *phil_keppeler@mfi.com*
Editor-at-Large Alexander Antoniades *sander@mfi.com*

Cover Photography Charles Ingram Photography

Publisher Veronica Costanza
Group Director Regina Starr Ridley
Special Projects Manager Nicole Freeman *76702.706@compuserve.com*

Advertising Sales Staff
Western Regional Sales Manager
Steve Nikkola (415) 905-2256 *snikkola@mfi.com*
Promotions Manager/Eastern Regional Sales Manager
Holly Meintzer (212) 615-2275 *hmeintzer@mfi.com*

Marketing Manager Susan McDonald
Marketing Graphic Designer Azriel Hayes
Advertising Production Coordinator Denise Temple
Director of Production Andrew A. Mickus
Vice President/Circulation Jerry M. Okabe
Circulation Director Gina Oh
Associate Circulation Director Kathy Henry
Group Circulation Manager Mike Poplardo
Assistant Circulation Manager Jamai Deuberry
Newsstand Manager Debra Caris
Reprints Stella Valdez (916) 729-3633

Chairman of the Board Graham J.S. Wilson
Chairman/CEO Marshall W. Freeman
President/COO Thomas L. Kemp
Senior Vice President/CFO Warren "Andy" Ambrose
Senior Vice Presidents David Nussbaum, Darrell Denny, Donald A. Pazour, Wini D. Ragus
Vice President/Production Andrew A. Mickus
Vice President/Circulation Jerry Okabe
Vice President/
Software Development Division Regina Starr Ridley

Miller Freeman
A United News & Media publication

Exploring the World of Texture Mapping

KEEPING CHRIS ON HIS TOES!

Dear Editor:

I've enjoyed reading Chris Hecker's series on perspective texture mapping. His articles were very informative, and they were very timely with respect to my current project. Thanks!

However, there are a few things in the February/March 1996 issue that confused me. First, on page 24, Figure 3, should the resulting exponent have the same binary exponent value as the higher magnitude value? In this case 10000111 (135)?

Also, in the first paragraph of page 24, he talks about subtracting "the integer representation of our large, floating-point shift number from the integer representation of the number we just converted...." What are these numbers? The values to be subtracted are now:

0 | 10010110 | 10000000000000000000000
(1.1 *2^{23})
-
1 | 10010110 | 00000000000000000001000
(Our 8.75 lined up)
=
1 | 10010110 | 11111111111111111111000
(This gets us our -8)

I think this is correct!

Kenneth Chao
Via e-mail

Chris Hecker replies:
Yes, this is a bug, thanks for spotting it! I must have cut and pasted when I did the diagram.

In regards to integer presentation, you have got one bit wrong in your result. The .1 bit gets borrowed from, so you end up with:

1 | 10010110 | 01111111111111111111000

*Now, we've got a result that has our -8 embedded in the bottom, but we want to get rid of the top garbage. To do this, we just subtract our "big number" (your 1.1 * 2^{23}) from this as an integer:*

11001011001111111111111111111000
-
01001011010000000000000000000000
=
11111111111111111111111111111000
(Which equals -8)

Thanks a lot! Keep up the good work!

X MODE MARKS THE SPOT

Dear Editor:

Your magazine is great! I use a lot of the articles constantly. Especially the ones about breaking into the game business. It actually helped me land a job!

Your X mode programming optimizations were great. Thanks for producing a much needed magazine.

John Bryant
Via e-mail

I HAVE A TEXTURE MAPPING QUESTION!

Dear Editor:

I've been getting *Game Developer* magazine since March 1995, and I've been especially following your series on texture mapping.

Do you have any suggestions on how I could make a quad texture mapper, assuming that the points are clockwise and coplanar? Theoretically, it should not be too much of a problem, since the gradients across the quad are the same as either of the two triangles which make it up, or at least, should be, as far as I have reasoned.

Also, how do games like Descent manage such fast texture mapping, while also doing game AI, Z buffering, and anything else that needs to be done? When it changes detail level, what is it doing?

Stuart Doyle
Via e-mail

Chris Hecker replies:
Well, general quads will work with a projective mapping (they won't with affine). But, if your quads (or even general polygons) are coplanar, you can use the plane equations to generate the gradients without using the vertex texture coordinates. The math is a bit much, but I might cover it in a later article. You can derive it yourself if you think about what you need to describe the plane of the texture (a vector from the origin to the texture origin, a U direction vector, and a V direction vector, all in view space). Solve for u and v in terms of screen space x and y (which are view point x/z and y/z) and you've got it. You'll end up with rational linear equations that look like:

$u = (ax + by + c)/(dx + ey + f)$
$u = (gx + hy + i)/(dx + ey + f)$

Also, Descent doesn't Z buffer. The major speedups in real games come from novel database traversal algorithms that cull out most of the world before they gets to the texture mapper. The fastest texture mapped triangles are the ones you didn't draw.

Say It!

Please send all feedback to: *Game Developer* magazine, Feedback, 600 Harrison St., S.F., Calif. 91407 or to joutlaw@mfi.com. Thanks!

Exploring the World of Texture Mapping

KEEPING CHRIS ON HIS TOES!

Dear Editor:

I've enjoyed reading Chris Hecker's series on perspective texture mapping. His articles were very informative, and they were very timely with respect to my current project. Thanks!

However, there are a few things in the February/March 1996 issue that confused me. First, on page 24, Figure 3, should the resulting exponent have the same binary exponent value as the higher magnitude value? In this case 10000111 (135)?

Also, in the first paragraph of page 24, he talks about subtracting "the integer representation of our large, floating-point shift number from the integer representation of the number we just converted...." What are these numbers? The values to be subtracted are now:

0 | 10010110 | 10000000000000000000000
(1.1 *2^{23})
-
1 | 10010110 | 00000000000000000001000
(Our 8.75 lined up)
=
1 | 10010110 | 11111111111111111111000
(This gets us our -8)

I think this is correct!

Kenneth Chao
Via e-mail

Chris Hecker replies:
Yes, this is a bug, thanks for spotting it! I must have cut and pasted when I did the diagram.

In regards to integer presentation, you have got one bit wrong in your result. The .1 bit gets borrowed from, so you end up with:

1 | 10010110 | 01111111111111111111000

*Now, we've got a result that has our -8 embedded in the bottom, but we want to get rid of the top garbage. To do this, we just subtract our "big number" (your 1.1 * 2^{23}) from this as an integer:*

11001011001111111111111111111000
-
01001011010000000000000000000000
=
11111111111111111111111111111000
(Which equals -8)

Thanks a lot! Keep up the good work!

X MODE MARKS THE SPOT

Dear Editor:

Your magazine is great! I use a lot of the articles constantly. Especially the ones about breaking into the game business. It actually helped me land a job!

Your X mode programming optimizations were great. Thanks for producing a much needed magazine.

John Bryant
Via e-mail

I HAVE A TEXTURE MAPPING QUESTION!

Dear Editor:

I've been getting *Game Developer* magazine since March 1995, and I've been especially following your series on texture mapping.

Do you have any suggestions on how I could make a quad texture mapper, assuming that the points are clockwise and coplanar? Theoretically, it should not be too much of a problem, since the gradients across the quad are the same as either of the two triangles which make it up, or at least, should be, as far as I have reasoned.

Also, how do games like Descent manage such fast texture mapping, while also doing game AI, Z buffering, and anything else that needs to be done? When it changes detail level, what is it doing?

Stuart Doyle
Via e-mail

Chris Hecker replies:
Well, general quads will work with a projective mapping (they won't with affine). But, if your quads (or even general polygons) are coplanar, you can use the plane equations to generate the gradients without using the vertex texture coordinates. The math is a bit much, but I might cover it in a later article. You can derive it yourself if you think about what you need to describe the plane of the texture (a vector from the origin to the texture origin, a U direction vector, and a V direction vector, all in view space). Solve for u and v in terms of screen space x and y (which are view point x/z and y/z) and you've got it. You'll end up with rational linear equations that look like:

$u = (ax + by + c)/(dx + ey + f)$
$u = (gx + hy + i)/(dx + ey + f)$

Also, Descent doesn't Z buffer. The major speedups in real games come from novel database traversal algorithms that cull out most of the world before they gets to the texture mapper. The fastest texture mapped triangles are the ones you didn't draw.

Say It!

Please send all feedback to: *Game Developer* magazine, Feedback, 600 Harrison St., S.F., Calif. 91407 or to joutlaw@mfi.com. Thanks!

Schmooze news.....

Game developers unhappy about Microsoft's $125 Pax Romana toga party fee held the "First Annual DirectBeer 1A Beta" party at a San Jose microbrewery. T-shirt themes included: "Quick, How many polygons am I holding up?" and "Unrelocated Crash Address: 96DIRECT:BEER"

Not only did Microsoft unwittingly use Pax Romana (the forcible pacification of subject states by a larger power) as their theme, but their Game Evangelist offended even more developers by his "10 Reasons DirectX Doesn't Suck" list which was topped with the statement that it didn't matter that only .01% of toga attendees would be female...one of them may be a Playboy Bunny. Worse yet, he actually showed up with a well-endowed woman who spent the time dropping grapes in his mouth.

CUC, the shopping network, bought Sierra for a cool billion dollars. What did they buy? The inventory of old Sierra games is worth something, but they didn't get much in the way of game development talent other than designer Roberta Williams. Sierra is known for its high turnover and regularly loses such assets as Lori and Cory Cole. CUC promises hands off, so the takeover doesn't look like it's providing any management fix to stem defections.

Remember Choplifter? Developer Danny Gorlin dropped out for four years for a stint of African dancing and drumming but has come back to the fold. He's working on Gravity's forthcoming Banzi Bug game to be published by Grolier.

Wanna gossip?
E-mail The Gossip Lady at:
71501.3553@compuserve.com.

On With the Show

Diane Anderson

After a few days at CGDC, I must decompress. I remind myself other humans exist. Pog guns, dart guns, dog tags, mouse pads, Nerds, Lemonheads, silly string, neon necklaces, and temporary tattoos all seem strangely normal after a period of show saturation.

COMPUTER GAME DEVELOPERS CONFERENCE™

Unfortunately, we didn't have room to list all the products we wanted to devote space to, so check out our web site for a more complete scoop. Here's a *Reader's Digest* synopsis of some products that were announced at or around the time of the CGDC:

- Intel threw a big party at the CGDC to announce its MMX technology, a major multimedia enhancement to the Intel architecture. See http://www.intel.com/pc-supp/multimed/mmx/ to get MMX development information.
- Yamaha announced plans to design hardware and software solutions to support Intel's MMX. Call (408) 567-2300.
- Diamond Multimedia released Diamond Edge 3D, a multimedia accelerator with Microsoft Direct3D support. Call (408) 325-7000.
- Yamaha announced support for the Direct3D API and availability of Direct3D drivers for the RPA family of 3D graphics accelerators. Direct3D drivers are available to Yamaha's OEMs, content developers, and selected beta test sites. Call (408) 567-2300.
- 3D Labs Inc. announced that its Glint and Permedia 3D graphics accelerators will support the new Direct3D API. Call (408) 436-3455.
- Silicon Graphics announced SiliconStudio, an open-architecture for building digital studios. StudioCentral is its asset management companion, Firewalker is a content authoring system, and StudioLive is its Internet-based studio services network. Call (415) 960-1980.
- 3Dfx announced System3D, a scaleable system based on 3Dfx's Obsidian graphics board for LBE games; it is designed for texture-mapped 3D arcade games. 3Dfx also announced its Voodoo Graphics chipset. Call (415) 934-2400.
- Apple, Netscape, and Silicon Graphics recently agreed to cooperate on 3D graphics for the Internet. The three companies plan to develop a new binary file format for Moving Worlds (a leading proposal for VRML 2.0) based on Apple's 3D metafile format (3DMF) technology. Moving Worlds is an open, cross-platform specification for dynamic 3D environments on the Internet, which enables higher compression, file streaming, and faster parsing of 3D objects and virtual worlds across the Internet.
- Apple announced Game Sprockets, their new Game SDK. Check out http://www.dev.apple.com/games.

For full descriptions, pricing, and contact information on these and other products, please surf to the new and improved *Game Developer* web site located at http://www.gdmag.com.

PowerPC Compilers: Still Not So Hot

I believe it was Theodore Roosevelt who first called the presidency of the United States a "bully pulpit," which is a catchy way of saying that the president can rant on a subject, people will actually listen, and maybe those people will even do something about whatever the topic of the rant happens to be. Magazine columns can be bully pulpits as well, and, while a computer magazine column is clearly not a pulpit on the same level as the White House, I don't expect to hear Bill Clinton taking compiler vendors to task about lame optimization quality in the next State of the Union Address, so I might as well do it myself.

Review Problems

This article started out as a comparative review of compiler optimizations, but the more I learned about the various compilers and how they did or did not optimize, the more the article turned into an exploration of how we as programmers have to help the compilers do a good job with our code. So while I'm still going to talk about five compilers and give comparison charts like a normal review, I'm actually going to concentrate on how our source code changes affect the assembly the compiler generates.

Most other compiler reviews focus on the compiler's integrated development environment, on the fancy editor with color syntax highlighting that doesn't even let you write a simple macro, on the debugger's silly ToolTip windows (that pop up over variable names with their values if you hold your mouse there forever), and on the compiler supplied class library that violates just about every precept of good object-oriented design in C++ and is bloated and slow to boot. Wow! As you can see, I'm no fan of compiler reviews—I believe most are written by either nonprogrammers or nonproduction programmers writing toy programs. Compilers themselves are written for those reviewers, and so we end up with the current mess, where compiler vendors focus on silly new features to please silly reviewers instead of focusing on things that actually help production programmers do their jobs well.

When I evaluate a compiler I look for two things: C++ compliance and code optimization. The former is basically a lost cause at this point because the C++ draft standard is still a moving target and there's no solid conformance suite. I pray this will change soon. By contrast, compiler writers have had years to work on compiler optimizations, and not much has changed since the early days.

By focusing on optimizations, we'll not only learn which compilers optimize the best, we'll also learn what we can do to help a compiler do its best with our code. This time, we'll be covering compilers for the PowerPC chip on the Macintosh, and next time we'll cover the Intel x86. Even if you don't program for the PowerPC, reading this will help you learn a lot about compilers and how they optimize, and that knowledge will carry across to whatever CPU you care to program.

The compilers we'll cover this issue are: Metrowerks CodeWarrior 8, Symantec C++ 8, version 1.0f3e2 of Apple's MrCpp compiler (which is included with the Symantec compiler), Motorola's 2.1.1 PowerPC C++ compiler, and the Microsoft Visual C++ for Macintosh 4.0 cross compiler.

Table 1. Transform Cycle Counts

Compiler	Listing 1	Listing 2	KAPed 1 (not shown)	Listing 4	Listing 5	Listing 6
CodeWarrior	40.7	50.5	50.9	34.3	29.7	19.6
Symantec C++	76.6	94.9	82.8	50.9	31.9	25.7
Motorola C++	34.5	47.4	39.5	33.2	30.8	20.6
Apple's MrCpp	52.0	65.0	56.2	36.1	28.8	19.5
Microsoft VC++	41.6	49.3	42.8	31.9	21.9	22.7

The Test Code

We'll use a simple inner product of a three-by-three matrix and a three element column vector to evaluate each

compiler's optimization quality. Obviously, a single function is not going to tell the whole optimization story, but it should give us an idea of what sorts of optimizations we can expect from today's compilers.

Listing 1 shows the function `TransformVectors`. I made it transform an array of vectors so the compilers would have to work a bit harder, but, even so, the code is trivial. I used 1,000 calls to this function with 500 transforms on each call to gather timing information. The first column of data in Table 1 shows the approximate cycle counts for each product measured with the MacOS call `Microseconds` for the various compilers on my Power Computing 604. I turned on all the optimizations I could find on each compiler to gather this data. I made sure my test program was producing correct results on every compiler by making the source vectors eigenvectors of the transform matrix and checking to see if the transformed vector was the same as the source—it's always a good idea to make sure neither you nor the compiler has introduced any bugs while optimizing.

Anti-Alias

If you've looked ahead at the other results in Table 1 and the other listings, you're probably wondering what the second column of data means, and why Listing 2 is almost identical to Listing 1. Even though you and I know we wouldn't call `TransformVectors` from Listing 1 with the source or destination pointing to the same vector, or, worse yet, with the destination pointing into the middle of the matrix somewhere, the compiler doesn't know this, so it can't assume we didn't do something silly. When a variable points to another live variable in the function, it's called "pointer aliasing," and when the compiler sees a write through a pointer, it needs to assume that the data could have landed anywhere, including into variables it's already loaded into registers. This means

Listing 1. The Test Function

```
void TransformVectors( float *pDestVectors,
        float const (*pMatrix)[3], float const *pSourceVectors,
        int NumberOfVectors )
{
        int Counter, i, j;
        for(Counter = 0;Counter < NumberOfVectors;Counter++) {
                for(i = 0;i < 3;i++) {
                        float Value = 0;
                        for(j = 0;j < 3;j++) {
                                Value += pMatrix[i][j] * pSourceVectors[j];
                        }
                        *pDestVectors++ = Value;
                }
                pSourceVectors += 3;
        }
}
```

Chris Hecker

Compilers. What are they good for? Chris Hecker steps to the bully pulpit to rant about the state of current PowerPC compilers. Sadly, these days, most compilers need a lot of help optimizing code.

Listing 2. The Non-Aliasing Test Function

```
void TransformVectors2( float *pDestVectors,
          float const (*pMatrix)[3], float const *pSourceVectors,
          int NumberOfVectors )
{
          int Counter, i, j;
          for(Counter = 0;Counter < NumberOfVectors;Counter++) {
                    float aTemp[3];
                    for(i = 0;i < 3;i++) {
                              float Value = 0;
                              for(j = 0;j < 3;j++) {
                                        Value += pMatrix[i][j] * pSourceVectors[j];
                              }
                              aTemp[i] = Value;
                    }
                    pSourceVectors += 3;
                    for(i = 0;i < 3;i++) {
                              *pDestVectors++ = aTemp[i];
                    }
          }
}
```

the optimizer has to continually reload variables into registers in case we're aliasing parameters, so I wrote `TransformVectors2` in Listing 2 to give the compilers some help. Since `aTemp` is defined local to our function, the compiler knows it can't be aliased, so writes to `aTemp` shouldn't cause spurious register reloads.

Well, at least that's what I thought, anyway. As you can see from the timings, all the compilers got slower because not only did they still reload all the registers, they also naively implemented the copy loop at the end of Listing 2!

Let's look at the code generated by the winner of this round, the Motorola C++ compiler. Listing 3 shows the PowerPC assembly language generated for `TransformVectors2`, our supposedly non-aliased function. Despite some odd ways of moving values into registers, this code is a pretty straightforward translation of our source into assembly language, which is disappointing. For example, the compiler doesn't bother to load the source vector into registers outside the loop, even though it's used three times and cannot be aliased because of our temporary results array. Also, instead of leaving the temporary results in registers, it actually copies them out to the stack and then copies the stack to the destination.

It even increments the destination pointer in the loop with three discrete instructions instead of using offsets and doing one addition at the end, or even using the PowerPC's autoincrement instructions. The Motorola compiler also produced the fastest code for Listing 1, and the difference between the timings for Listings 1 and 2 can be attributed to the naive compilation of the temporary copy loop at the end of Listing 2 (even though the temporary loop was supposed to help by eliminating the possibility of aliasing). Overall, not a great showing, even by our winner in this round. Clearly, the compilers need more help.

Bust a KAP

The Motorola compiler ships with an interesting tool, called the Kuck and Associates Preprocessor for C (KAP). Basically, KAP compiles your C code (it doesn't support C++), optimizes it, and then generates C code as its output

Listing 3. Motorola C++ Assembly for Listing 2

```
TransformVectors2__FPfPA3_CfPCfi.b:
        cmpi    0x7,0x0,r6,0        ; compare count to 0
        addi    r11,r0,0            ; Counter = r11 = 0
        bc      0x4,0x1d,L..11      ; bail out if count = 0
        addi    r8,sp,24            ; allocate some stack
L..8:   ori     r9,r4,0x0           ; r9 = pMatrix
        addi    r10,r0,0            ; r10 = 0
        ori     r7,r10,0x0          ; r7 = 0
        subfic  r10,r10,3           ; r10 = 3
        mtctr   r10                 ; ctr = 3
L..9:   lfs     f1,0(r9)            ; f1 = pMatrix[0]
        lfs     f2,0(r5)            ; f2 = pSource[0]
        lfs     f3,4(r9)            ; f3 = pMatrix[1]
        lfs     f4,4(r5)            ; f4 = pSource[1]
        fmuls   f1,f1,f2            ; f1 = f1 * f2
        lfs     f2,8(r9)            ; f2 = pMatrix[2]
        lfs     f5,8(r5)            ; f5 = pSource[2]
        fmadds  f3,f3,f4,f1         ; f3 = f3 * f4 + f1
        addi    r9,r9,12            ; pMatrix->next row
        fmadds  f2,f2,f5,f3         ; f2 = f2 * f5 + f3
        stfsx   f2,r8,r7            ; *(stack + r7) = f2
        addi    r7,r7,4             ; r7 next float
        bc      0x10,0x0,L..9       ; branch if (--ctr)
        lfs     f1,0(r8)            ; f1 = stack[0]
        lfs     f2,4(r8)            ; f2 = stack[1]
        lfs     f3,8(r8)            ; f3 = stack[2]
        stfs    f1,0(r3)            ; pDest[0] = f1
        addi    r3,r3,4             ; pDest++
        addi    r11,r11,1           ; Counter++
        stfs    f2,0(r3)            ; pDest[1] = f2
        addi    r3,r3,4             ; pDest++
        cmp     0x7,0x0,r11,r6      ;  flags  =  Counter  <
NumVecs
        addi    r5,r5,12            ; pSource += 3
        stfs    f3,0(r3)            ; pDest[2] = f3
        addi    r3,r3,4             ; pDest++
        bc      0xc,0x1c,L..8       ; branch if (Counter <
NumVecs)
L..11:  addi    sp,sp,48            ; clear stack
        bclr    0x14,0x0            ; return
```

Listing 4. The KAPed Listing 2

```
void TransformVectors2( float *pDestVectors,
        const float (*pMatrix)[3], const float *pSourceVectors,
        int NumberOfVectors )
{
    int Counter, i, j;
    float aTemp[3];
    float Value, _Krr1, _Krr2, _Krr4, _Krr5;
    long _Kii1, _Kii2;
    for ( Counter = 0; Counter<NumberOfVectors; Counter++ ) {
        _Krr1 = 0.0F; _Krr2 = 0.0F; Value = 0.0F;
        _Kii1 = Counter * 3;
        _Krr1 +=  pMatrix[0][0] * pSourceVectors[_Kii1];
        _Krr2 +=  pMatrix[1][0] * pSourceVectors[_Kii1];
        Value +=  pMatrix[2][0] * pSourceVectors[_Kii1];
        _Krr1 +=  pMatrix[0][1] * pSourceVectors[_Kii1+1];
        _Krr2 +=  pMatrix[1][1] * pSourceVectors[_Kii1+1];
        Value +=  pMatrix[2][1] * pSourceVectors[_Kii1+1];
        if (1) {
            _Krr1 +=  pMatrix[0][2] * pSourceVectors[_Kii1+2];
            _Krr2 +=  pMatrix[1][2] * pSourceVectors[_Kii1+2];
```

instead of assembly language. When I first got the Motorola compiler, I figured my test would be so simple that there was no way KAP could help out, but after looking at the results we just discussed, I figured anything was worth a try. Listing 4 shows the output of running Listing 2 through KAP (they call the processed code KAPed). If you've never seen machine-generated C code, don't be surprised by stuff like the `"if (1)"` block—compilers output weird stuff like that for bizarre reasons. However, you should be surprised at how poor the code is. It unrolls the loop, which is fine, but why does it go to the trouble of putting the temporaries in `aTemp` and then looping over `aTemp` to copy them into the destination? More absurd yet is that something as mundane as unrolling a loop in this simple function actually helped the compilers produce faster code.

You can see the timing results in Table 1. The KAPed Listing 1 is not even worthy of print, and as you can see the compilers all got slower on that version. The KAPed Listing 2 (shown in Listing 4) actually made a positive difference, even on the best compilers, and it made a huge difference on Symantec. Even so, if you thought it was bad that KAP generated the redundant loop at the end of Listing 4, it's even worse that every compiler generated actual assembly language code for that loop! The worst offender is clearly Symantec. Symantec ships a prerelease version of Apple's MrCpp compiler with their package, so it's unclear if I should even review Symantec on their optimization quality because I think they expect you to use MrCpp if you care about run-time speed. However, MrCpp and Symantec's main-

Listing 4. Continued from p. 16

```
            Value +=  pMatrix[2][2] * pSourceVectors[_Kii1+2];
        }
        aTemp[0] = _Krr1; aTemp[1] = _Krr2; aTemp[2] = Value;
        _Kii2 = Counter * 3;
        for ( i = 0; i<=2; i++ ) {
            _Krr5 = aTemp[i]; _Krr4 = _Krr5;
            pDestVectors[_Kii2+i] = _Krr4;
        }
    }
    pSourceVectors +=  NumberOfVectors * 3;
    pDestVectors +=  NumberOfVectors * 3;
}
```

Listing 5. Hand-optimized Listing 1

```
void TransformVectors( float *pDestVectors,
        const float (*pMatrix)[3], const float *pSourceVectors,
        int NumberOfVectors )
{
    int Counter;
    float Value0, Value1, Value2;
    for ( Counter = 0; Counter<NumberOfVectors; Counter++ ) {
            Value0 = pMatrix[0][0] * pSourceVectors[0];
            Value0 += pMatrix[0][1] * pSourceVectors[1];
            Value0 += pMatrix[0][2] * pSourceVectors[2];
            *pDestVectors++ = Value0;
            Value1 = pMatrix[1][0] * pSourceVectors[0];
            Value1 += pMatrix[1][1] * pSourceVectors[1];
            Value1 += pMatrix[1][2] * pSourceVectors[2];
            *pDestVectors++ = Value1;
            Value2 = pMatrix[2][0] * pSourceVectors[0];
            Value2 += pMatrix[2][1] * pSourceVectors[1];
            Value2 += pMatrix[2][2] * pSourceVectors[2];
            *pDestVectors++ = Value2;
            pSourceVectors += 3;
    }
}
```

line compiler are not C++ feature-equivalent, so I'm not sure how they can expect you to freely exchange them.

A Helping Hand

At this point, it was clear that the compilers by themselves—and even with the help of KAP, for what it's worth—were not going to be able to produce reasonable code for these functions, so I had to step in and give them a hand. I looked at what kind of improvement KAP got from using what I had assumed were brain-dead rewrites (I can't even bring myself to call them optimizations), and I decided to hand code the `Transform` functions to see what would come out. Listings 5 and 6 contain the hand-optimized versions of Listings 1 and 2, respectively. You can see from the

Listing 6. Hand-optimized Listing 2

```
void TransformVectors2( float *pDestVectors,
    const float (*pMatrix)[3], const float *pSourceVectors,
    int NumberOfVectors )
{
    int Counter;
    float Value, _Krr1, _Krr2;
    for ( Counter = 0; Counter<NumberOfVectors; Counter++ ) {
        _Krr1 =  pMatrix[0][0] * pSourceVectors[0];
        _Krr2 =  pMatrix[1][0] * pSourceVectors[0];
        Value =  pMatrix[2][0] * pSourceVectors[0];
        _Krr1 +=  pMatrix[0][1] * pSourceVectors[1];
        _Krr2 +=  pMatrix[1][1] * pSourceVectors[1];
        Value +=  pMatrix[2][1] * pSourceVectors[1];
        _Krr1 +=  pMatrix[0][2] * pSourceVectors[2];
        _Krr2 +=  pMatrix[1][2] * pSourceVectors[2];
        Value +=  pMatrix[2][2] * pSourceVectors[2];
        *pDestVectors++ = _Krr1;
        *pDestVectors++ = _Krr2;
        *pDestVectors++ = Value;
        pSourceVectors += 3;
    }
}
```

timing results in the last two columns of Table 1 that it made a big difference on all the compilers.

Why did it make such a big difference? I have no idea, and the only explanation I can come up with is that you need to hold your compiler's hand on any piece of code you care about. The changes I made for Listings 5 and 6 are very obvious (to a human, if not a compiler). I'm basically just stating explicitly where variables are accessed, where possible aliasing can occur, and which variables are constant throughout a loop iteration. These are all things the compiler is supposed to do for us, so we can work on more important stuff, like design and algorithms, or assembly language code for our most inner loops. We're supposed to trust the compiler will do a respectable job, without having to optimize every line of our code (an impossible task for all but the smallest programs).

Now, if you're like me, you've been waiting to say something about all this loop unrolling for a while now. You're waiting to say that you don't actually want the compiler to unroll loops all over the place, because that makes your code bigger and probably slower. Hah! I was waiting for you to say that, because the most amazing thing of all about this test is that a couple of the compilers produced code for Listing 6 that is smaller than the code for the original non-unrolled Listing 2. Listing 7 shows the

CodeWarrior version of Listing 6; it's at least 6 instructions smaller than any of the compiled versions of Listing 2, and about two to five times as fast. Motorola produced similar code. (MrCpp decided now was the time to unroll the entire function, which tripled the size of the code for absolutely no performance increase.) Don't for a minute think this is a compliment for CodeWarrior or Motorola, it's really a damning insult to all the compilers: on maximum optimizations they didn't find the smaller and faster version of a basic function like a matrix transform. Heck, just by inspection I can see how to save a couple more instructions in Listing 7. And people actually say that writing assembly language is a dying art.

You Lose Some

If this was a normal compiler review, it would be time to pick a winner, but, instead, it's time to point out that you and I are the losers in this situation. Pundits have been saying that assembly language is dead—especially on RISC chips like the PowerPC—and it should be eminently clear from the listings in this article that those people have no clue what they're talking about. Even a beginning assembly language programmer could produce better code than any of the compilers for Listings 1 and 2, and this is simple code. While you might not choose to write your code in assembly language, you end up with C code that looks like assembly language if you want respectable performance, like Listings 5 and 6.

If I had to choose a winner, I'd pick the Motorola C++ compiler, because it seems like the least incompetent optimizer of the bunch. The Microsoft compiler showed some promising aggressiveness by loading the entire matrix into registers once at the top of the loop for its version of Listings 5 and 6. Microsoft has an option I turned on that tells the compiler there's no pointer aliasing that allowed them to perform this optimization, but they didn't take advantage of the assumption anywhere else that I could see. Given this optimization, I'm not sure why their version of Listing 6 wasn't faster than the others…it may have been a pipelining issue, or the loop might have been cache bound. As soon as I learn a bit more about the subtleties of the PowerPC 604, I'll get back to you on this one.

In the next issue, I'll quickly cover a bunch of Intel x86 compilers, but we will still have room to talk about some optimization programming techniques of our own, because the compilers clearly aren't going to do it for us.

This Just In: I was keeping Mike Phillip at Motorola's compiler group posted on my results, and he just got back to me with a command line switch for KAP that will assume there's no aliasing. When you turn it on, KAP produces something resembling Listing 6,

Listing 7. CodeWarrior version of Listing 6

```
TransformVectors2__FPfPA3_CfPCfi
     mr       r0,r6       ; r0 = NumVecs
     cmpwi    r6,0        ; flags = NumVecs == 0
     mtctr    r0          ; ctr = NumVecs
     blelr                ; bail if(NumVects == 0)
L1:  lfs      fp1,0(r4)   ; fp1 = pMatrix[0][0]
     lfs      fp3,0(r5)   ; fp3 = pSource[0]
     lfs      fp0,12(r4)  ; fp0 = pMatrix[1][0]
     lfs      fp2,24(r4)  ; fp2 = pMatrix[2][0]
     fmuls    fp7,fp1,fp3; fp7 = fp1 * fp3
     lfs      fp1,4(r4)   ; fp1 = pMatrix[0][1]
     lfs      fp5,4(r5)   ; fp5 = pSource[1]
     fmuls    fp8,fp0,fp3; fp8 = fp0 * fp3
     fmuls    fp6,fp2,fp3; fp6 = fp2 * fp3
     lfs      fp0,16(r4)  ; fp0 = pMatrix[1][1]
     lfs      fp4,28(r4)  ; fp4 = pMatrix[2][1]
     lfs      fp2,8(r5)   ; fp2 = pSource[2]
     fmadds   fp7,fp1,fp5,fp7
                          ; fp7 = fp1 * fp5 + fp7
     addi     r5,r5,12    ; pSource += 3
     lfs      fp3,8(r4)   ; fp3 = pMatrix[0][2]
     fmadds   fp8,fp0,fp5,fp8
                          ; fp8 = fp0 * fp5 + fp8
     lfs      fp1,20(r4); fp1 = pMatrix[1][2]
     fmadds   fp6,fp4,fp5,fp6
                          ; fp6 = fp4 * fp5 + fp6
     lfs      fp0,32(r4); fp0 = pMatrix[2][2]
     fmadds   fp7,fp3,fp2,fp7
                          ; fp7 = fp3 * fp2 + fp7
     fmadds   fp8,fp1,fp2,fp8
                          ; fp8 = fp1 * fp2 + fp8
     fmadds   fp6,fp0,fp2,fp6
                          ; fp6 = fp0 * fp2 + fp6
     stfs     fp7,0(r3) ; pDest[0] = fp7
     stfsu    fp8,4(r3)
                         ; pDest[1] = fp8, pDest++
     stfsu    fp6,4(r3)
                          ; pDest[1] = fp6, pDest++
     addi     r3,r3,4     ; pDest++
     bdnz     L1          ; branch if(--ctr)
     blr                  ; return
```

so all the compilers do well. However, not to be out-done, I decided to take this no-aliasing assumption to the limit and explicitly load the matrix into temporaries (much like the Microsoft compiler tried to do). The result: another 25% speedup, with times around 15 cycles, for something the compiler could have done itself. The compilers lose again. ■

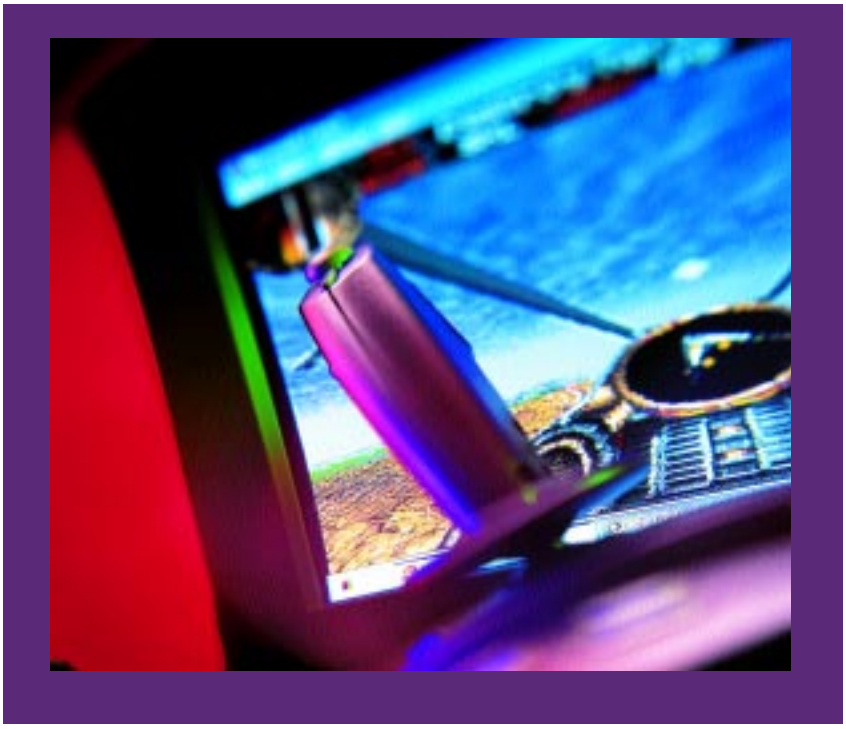
Chris Hecker tries to live an optimized life, but he does about as good of a job on his own life as the current crop of compilers does on his code. Contact him at gdmag@mfi.com.

Introduction to the DirectX II APIs

Once upon a time, developing games for the PC wasn't easy. DOS allowed game developers access to low-level systems functions, enabling good performance, but few standards were in place to support the wide variety of hardware found in PCs. Developing under Windows 3.x wasn't any better, due to performance bottlenecks. However, with last year's release of Windows 95, a door has been opened to the development of high-performance, Windows-based games. And the key to that door is Microsoft's Game Software Development Kit (SDK).

The primary goal of the Game SDK is to make performance on Windows rival or exceed performance on DOS-based platforms and console systems and to provide a robust, standardized, and well-documented platform for game developers. A high-performance Windows-based game installs successfully and takes advantage of hardware accelerator cards, Windows hardware and software standards (such as Plug-and-Play), and the Windows communications services.

The Game SDK, which you can find in the Microsoft Developer Network (MSDN) Level II, provides a consistent interface for hardware manufacturers and game developers. It reduces the complexity of installing and configuring games and uses a computer's hardware to the best advantage. DOS-based games can still take advantage of the hardware cards available to the Game SDK developer; however, DOS-based game developers must conform to varying implementations of cards—which complicates the installation.



New Standards for Hardware Vendors

The Game SDK provides portable access to the features of DOS today, keeps a high level of performance, and removes obstacles to hardware innovation. The Game SDK tries to provide guidelines for hardware companies based on feedback from game developers and independent hardware vendors (IHVs). Therefore, the Game SDK components often provide specifications for hardware accelerator features that do not yet exist. In many cases, these specifications are emulated in software. In other cases, the capabilities of the hardware are polled first, and the feature is bypassed if it is not supported by the hardware.

The Game SDK supports a number of video hardware features that have already come out (or will be released in the near future). These include:

- Overlays
- Page flipping
- Sprite engines
- Stretching with interpolation
- Alpha blending
- Z buffer-aware bit-block transfers.

Overlays will be supported so that page flipping will also be enabled within a window using graphic device interface (GDI). Page flipping is the double-buffer scheme used to display frames on the entire screen. Sprite engines are used to make overlaying sprites easier.

Stretching with interpolation is stretching a smaller frame to fit the entire screen; it can be an efficient way to conserve video RAM. Alpha blending is used to mix colors at the hardware pixel level. Three-dimensional (3D) accelerators with perspective-correct textures will allow textures to be displayed on a 3D surface; for example, hallways in a castle generated by 3D software can be textured with a brick wall bitmap that maintains the correct perspective. As 3D games generally need at least 2MB of video memory, the Game SDK supports this. A compression standard to put more data into display memory will include transparency compression, will be usable for textures, and will be very fast when implemented in software as well as hardware.

The audio hardware features that the Game SDK supports or will soon include are the following:

- 3D audio enhancers that provide a spatial placement for different sounds (particularly effective with headphones)
- Onboard memory for audio boards
- Audio-video combination boards that share onboard memory.

In addition, video playback will see the benefit from Game SDK-compatible hardware. Future releases of the Game SDK will support hardware-accelerated decompression of YUV video.

Taking It Apart

The Game SDK is made of several interfaces that target performance issues of game programming under Windows 95. The following are APIs:

- The DirectDraw API accelerates hardware and software animation techniques by providing direct access to bitmaps in offscreen video memory as well as fast access to the bit-block transferring and buffer-flipping capabilities of the hardware.
- The DirectSound API enables hardware and software sound mixing and playback.
- The DirectPlay API lets you develop games for play over a modem line or a network.
- The Direct3D API provides direct low-level access to 3D hardware, allowing DirectDraw surfaces to be used both as 3D rendering targets and as source texture maps.
- The DirectInput API provides joystick input capabilities that are scaleable to future Windows hardware input API and drivers.
- The AutoPlay feature lets your CD-ROM run an installation program or the game itself immediately upon insertion of the disc.

Note: DirectInput and AutoPlay exist in the Microsoft Win32 API and aren't unique to the Game SDK.

DirectDraw

The biggest gain in performance in the Game SDK comes from the DirectDraw services, which are a combination of four COM interfaces: `IDirectDraw`, `IDirectDrawSurface`, `IDirectDrawPalette`, and `IDirectDrawClipper`. A `DirectDraw` object, created using the function `DirectDrawCreate`, represents the video display card. One of the object's member functions, `IDirectDraw::CreateSurface`, creates the primary `DirectDrawSurface` object, which repre-

Robert Hess

Through the use of DirectX APIs, you can develop with a consistent interface; your Windows games can outperform their DOS-based ancestors and automatically take advantage of hardware acceleration where it's available.

sents the video display memory being viewed on the monitor. From the primary surface, offscreen surfaces can be created in a linked-list fashion.

In the most common case, a back-buffer surface is created (in addition to the primary surface) and is used to exchange images with the primary surface. While the screen is busy displaying the lines of the image in the primary surface, the back-buffer surface frame is composed by transferring a series of offscreen bitmaps stored on other `DirectDrawSurface` objects in video RAM. Call the `DirectDrawSurface::Flip` member function to display the recently composed frame, which sets a register so that the exchange occurs when the screen performs a vertical retrace. This is asynchronous, so the game can continue processing after calling `DirectDrawSurface::Flip`. (The back buffer is automatically write-blocked after calling `DirectDrawSurface::Flip` until the exchange occurs.) After the exchange occurs, the game continues to compose the next frame in the back buffer, calls `DirectDrawSurface::Flip`, and so on.

DirectDraw improves performance over the Windows 3.1 GDI model, which does not have direct access to bitmaps in video memory. Therefore, bit-block transfers always occur in host RAM and are then transferred to video display memory. Using DirectDraw, all processing is done on the card whenever possible. DirectDraw improves performance over the Windows 95 and Windows NT GDI model that uses the `CreateDIBSection` function to enable hardware processing.

The third type of DirectDraw object is `DirectDrawPalette`. Because the physical display palette is usually maintained in video hardware, an object represents and manipulates it. The `IDirectDrawPalette` interface implements palettes in hardware. These bypass Windows palettes and are therefore only available when a game has exclusive access to the video hardware. `DirectDrawPalette` objects are also created from `DirectDraw` objects.

The fourth type of DirectDraw object is the `DirectDrawClipper`. DirectDraw manages clipped regions of display memory by using these objects. A bitmap is transferred to a surface using a transparent bit-block transfer, and a certain color (or range of colors) in the bitmap is defined as transparent. Color keying achieves a transparent bit-block transfer. Source color keying defines which color or color range on the bitmap is transparent (and therefore not copied during a transfer operation). Destination color keying defines which color or color range on the surface will be covered by pixels of that color or color range in the source bitmap.

Finally, DirectDraw supports overlays in hardware and by software emulation. Overlays are an easy means of implementing sprites and managing multiple layers of animation. Any `DirectDrawSurface` object can be created as an overlay, and any overlay has all of the capabilities of any other surface—plus extra capabilities associated only with overlays. These capabilities require extra display memory, and, if there are no overlays in display memory, the overlay surfaces can exist in host memory.

Color keying works the same for overlays as for transparent bit-block transfers. The Z order of the overlay automatically handles the occlusion and transparency manipulations between overlays.

You can find source code for a DirectDraw application on the *Game Developer* web site. The bulk of this code comes from the generic sample sources I wrote for the Win32 SDK. I made minor additions and modifications to enable this application to utilize the DirectDraw features of the new DirectX APIs for Windows. This code is not meant to illustrate a great DirectDraw application, but simply to show you how easily DirectDraw can be added to a Windows application model. There are several sample applications that come with the Game SDK that better illustrate some of the more advanced features and capabilities of DirectX.

DirectSound

Game programming requires efficient and dynamic sound production. Microsoft provides two methods for achieving this: MIDI streams and DirectSound. MIDI streams are actually part of the Windows 95 multimedia API. They provide the ability to time-stamp MIDI messages and send a buffer of these messages to the system, which can then efficiently integrate them with its processes. More information about MIDI streams can be found in the Win32 SDK documentation.

DirectSound is built on the COM-based interfaces `IDirectSound` and `IDirectSoundBuffer`, and it's extensible to other interfaces. DirectSound implements a new model for playing back digitally recorded sound samples and mixing different sample sources together. As with other object classes in the Game SDK, you should use the hardware to its greatest advantage whenever possible and emulate a hardware feature in the software when the feature is not present in hardware. You can query hardware capabilities at run-time to determine the best solution for any given personal computer configuration.

The `DirectSound` object represents the sound card and its various attributes. Using a `DirectSound` object, you create the `DirectSoundBuffer` object, which represents a buffer containing sound data. Several `DirectSoundBuffer` objects can exist and be mixed together into the primary `DirectSoundBuffer` object. DirectSound buffers are used to start, stop, and pause sound playback and to set attributes such as frequency, format, and so on. Depending on the card type, DirectSound buffers can exist in hardware as onboard RAM, wave table memory, a DMA channel, or a virtual buffer (for an I/O, port-based audio card). Where there is no hardware implementation of a DirectSound buffer, it is emulated in host system memory.

The primary buffer is generally used to mix sound from secondary buffers but can be accessed directly for custom mixing or other specialized activities. (Use caution in locking the primary buffer, because this blocks all access to the sound hardware from other sources.) Secondary buffers can store

common sounds that are played throughout a game. A sound stored in a secondary buffer can be played as a single event or as a looping sound that plays repeatedly. You can use secondary buffers to play sounds that are larger than available sound buffer memory. When used to play a sound larger than the buffer, the secondary buffer is a queue that stores the portions of the sound about to be played.

DirectPlay

One of the best features of PCs as a game platform is their easy access to communication services. DirectPlay capitalizes on this and allows multiple players to interact during game play through standard modems, network connections, or online services.

The `IDirectPlay` interface contains methods providing capabilities such as creating and destroying players, adding players to and deleting players from groups, sending messages to players, inviting players to participate in a game, and so on.

DirectPlay is composed of the interface to the game, as defined by the `IDirectPlay` interface and the DirectPlay server. DirectPlay servers are provided by Microsoft for modems and networks, as well as by third parties. When using a supported server, DirectPlay-enabled games can bypass connectivity and communication overhead details.

Direct3D

Direct3D is the newest addition to Microsoft's DirectX family of APIs and provides developers with an API and system services for real-time 3D graphics. Direct3D is based on the Reality Lab technology acquired by Microsoft in 1995 when they purchased RenderMorphics and has been significantly enhanced to include tight integration with the DirectDraw API. Direct3D consists of the following components: integrated retained mode and immediate mode APIs, extensible file format, and a device-independent driver model for transparent access to 3D hardware acceleration.

Direct3D is exposed to the developer as COM-based interfaces for the retained (`IDirect3DRM`) and immediate mode APIs (`IDirect3D`). These interfaces are responsible for operations including DirectDraw rendering devices, textures, materials, lights, viewports, animations, and picking.

The Direct3D high-level retained-mode API is designed for manipulating 3D objects and managing 3D scenes, while insulating the developer from the mesh structures and transformation calculations. It is targeted at developers who don't want to create their own geometry engines or object database routines but want to easily add 3D capabilities to new or existing Windows-based applications. For example, the retained mode API supports the loading of a predefined, textured 3D object with a single API command; the application can use additional simple API commands to rotate, move, or scale the object to manipulate it in the scene in real time. The retained mode API also supports key frame animations.

The Direct3D low-level immediate-mode API, on the other hand, is a thin polygon- and vertex-based API layer that gives you direct access to features of 3D hardware in a device-independent manner. Because the immediate-mode API does not provide its own geometry engine (unlike the retained-mode API), the application handles the object and scene management. The immediate-mode API lets you port existing high-performance multimedia applications, such as games, to the Windows operating system. It also gives you the flexibility to make use of your own rendering and scene-management technologies while transparently taking advantage of the new generation of 3D hardware accelerators.

Direct3D provides a rich file format for storing meshes, textures, animation sets, and user-definable objects. This format facilitates the exchange of 3D information between applications. Support for animation sets allows predefined paths to be stored for playback in real time. Instancing and hierarchies are also supported and allow multiple references to a single data object, such as a mesh, but store the data for the object only once per file. The Direct3D file format is used natively by the Direct3D retained-mode API, providing support for reading predefined objects into an application or writing mesh information constructed by the application in real time. The file format will be supported by content creators for modeling 3D objects and scenes and defining complex animation paths, and it will be used by title developers for incorporation into their titles.

The Direct3D hardware abstraction layer (HAL) provides a driver interface for giving developers a transparent, device-independent means to access the features of 3D hardware acceleration. The Direct3D hardware emulation layer (HEL) provides software-based emulation of 3D rendering services not supported by the hardware device. For example, Direct3D supports the acceleration of any or part of the 3D rendering pipeline including transformations, lighting, and rasterization—many 3D hardware accelerators on the market today only offload the rasterization module of the pipeline, so the transformations and lighting are handled by the software emulation routines. This architecture ensures that services exposed by the Direct3D APIs are always available to the application, whether the underlying hardware supports it or not. Developers can query the underlying characteristics of the hardware to identify the capabilities supported and determine whether the hardware is providing the rendering services, to support tuning and scaling of the application in real time as appropriate for the given configuration.

Direct3D integrates with DirectDraw to provide 2D drawing and texture services for 3D rendering. Applications use Direct3D and DirectDraw for 3D rendering in a relatively straightforward manner. For example, the steps to set up a scene and to render a triangle using the Direct3D immediate mode API are as follows: First, you use DirectDraw to create the rendering surfaces, which consist of the front buffer, back buffer, and (optionally) the Z-buffer, as DirectDraw surfaces. Next, you use the `IDirect3D` COM interface to

set up the world, view, and projection matrices and to create a viewport to control the 3D clipping information. You create a material for the background of the viewport. You then create a DirectDraw surface to serve as the texture for the triangle; then you create a material to define the surface reflectivity and color. Then you create a light source to add lighting to the scene. Next, create an execute buffer to represent the display list for holding the vertices and to define how the vertices are tied together into primitives for rendering the 3D object. You add any transformations, lighting, and rendering state information to the execute buffer, followed by the vertex and primitive operation information for representing the 3D object. Finally, you clear the viewport and render the execute buffer, followed by a page flip operation to display the rendered scene on the front buffer—repeat the process as the execute buffer is modified to animate the object.

DirectInput

The joystick represents a class of devices that report tactile movements and actions that players make within a game. DirectInput provides the functionality to process the data representing these movements and actions from joysticks, as well as other related devices, such as trackballs and flight harnesses.

DirectInput is currently another name for an existing Win32 function, `joyGetPosEx`. This function provides extended capabilities to its predecessor, `joyGetPos`, and should be used for any joystick services. In future support for input devices, including virtual reality hardware, games that use `joyGetPosEx` will be automatically supported for joystick input services. This is not the case for `joyGetPos`.

AutoPlay

AutoPlay is the feature of Windows 95 that automatically plays a CD or audio CD when inserted into a CD-ROM drive. Any CD-ROM product that bears the Windows 95 logo must be enabled with the AutoPlay feature.

Game SDK COM Interfaces

The interfaces in the Game SDK have been created at a very base level of the COM programming hierarchy. Each main device object interface, such as `IDirectDraw`, `IDirectSound`, or `IDirectDraw` derives directly from `IUnknown`. The creation of these base objects is handled by specialized functions in the library rather than by the Win32 `CoCreateInstance` function normally used to create COM objects. The Game SDK object model provides one main object for each device, from which other support service objects are derived. For example, the `DirectDraw` object represents the display adapter. It is used to create `DirectDrawSurface` objects that represent the video RAM and `DirectDrawPalette` objects that represent hardware palettes. Similarly, the `DirectSound` object represents the audio card and creates `DirectSoundBuffer` objects that represent the sound sources on that card.

Besides the ability to generate subordinate objects, the main device object determines the capabilities of the hardware device it represents, such as the screen size and number of colors, or whether the audio card has wave table synthesis.

By utilizing DirectX, it is finally possible to write state-of-the-art, fast-action, "rip the nerves from the tips of your fingers" games for Windows. And not only can these games far exceed the wildest dreams of Windows programmers of the past, but they can leave DOS games twitching in the dust. Windows, it isn't just for spreadsheets anymore. ■

When not underwater basketweaving, Robert Hess spends his time as a software design engineer in the Developer Relations Group at Microsoft. You can contact him at gdmag@mfi.com.

An extended version of this article is available on the Internet at the Game Developer *web site.*

Networking Your Game Using DirectPlay

With the advent of the Windows 95 Game SDK, Windows 95 is now positioned as a powerful and interesting platform for network gaming. More specifically, the DirectPlay component of the Game SDK provides a network communication protocol that stands to make life much easier for network game developers and players alike. It provides a device- and network-independent communications model for multiplayer games and a consistent user interface for establishing and maintaining network connections.

DirectPlay provides all the overhead, which enables players to connect to each other in a consistent manner across a wide range of network types. At the code level, you simply call the correct DirectPlay API functions. The one missing element in DirectPlay, however, is synchronization support. Because of the many different approaches to solving the game synchronization problem, DirectPlay forces you to implement your own game-specific solution. Although it might seem like Microsoft took the easy way out, in reality they just didn't want to force a specific synchronization solution on game developers.

DirectPlay Architecture

DirectPlay provides a network-independent programmatic interface to network game development. This network independence means that you write game-communication code to the DirectPlay API, and it sends the information over the network connection established for the game. This saves you from needing to learn the details of all the different network protocols. At this point, if you haven't breathed a huge sigh of relief, please feel free to. The ability to write network games without having to learn the details of network interfaces is truly a giant step in game programming. DirectPlay lets you focus on the network aspects directly related to your game.

DirectPlay is composed of two parts: the DirectPlay COM (Component Object Model) object and the DirectPlay service provider. The COM object provides the programmatic interface with which you establish network connections, maintain available sessions and players, and handle the details of sending and receiving game data. The DirectPlay service provider is a lower level DirectPlay component that handles the dirty work of implementing network-specific communications. The service provider is implemented as a network server for each type of supported network. Microsoft provides DirectPlay servers for IPX, TCP/IP, and modem networks. Third-party vendors must develop their own DirectPlay servers for supporting specialized network hardware and online services.

DirectPlay servers are the network game equivalents of drivers in other parts of the Windows system. Servers take on the difficulties of implementing the DirectPlay API for a specific network. This approach works well because it maintains a consistent interface at the application level, while allowing extensibility at the network level. When a DirectPlay COM object is created, a DirectPlay server is specified. DirectPlay then dynamically binds to this server,

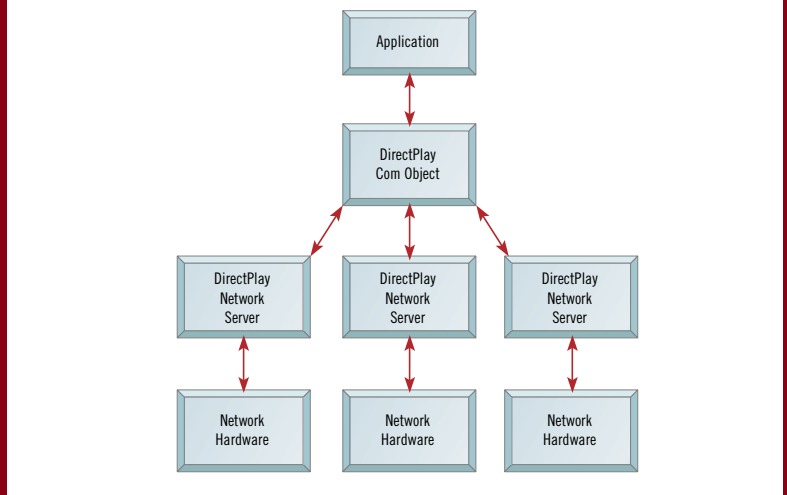


Figure 1. The DirectPlay communications model.

through which all DirectPlay communications are carried out. Figure 1 shows the DirectPlay communications model, which illustrates how an application communicates through DirectPlay on a particular type of network.

DirectPlay Fundamentals

DirectPlay provides a means of establishing a connection and communicating over a network in a consistent manner. This is no small feat and puts a lot of responsibility on DirectPlay network servers. DirectPlay itself keeps up with information regarding the network connections and all parties involved. The key components of a DirectPlay network connection are sessions and players.

Sessions

Every DirectPlay game must establish a session, which is a communication channel. Multiple sessions on a given network correspond to different multiplayer games running on the network. The exception is a modem network, where only one session can exist. Players in a particular network game are in the same session. Suppose you want to join one of two sessions of a network Poker game. You must choose one poker game or the other to connect to. Players choose from a list of sessions that DirectPlay supports.

DirectPlay can save information about a session in the registry for future use. With a modem network, for example, the remote player's name, phone number, and optional password are saved. Speaking of modem connections, modem code is another huge responsibility taken on by DirectPlay. Remember that DirectPlay servers handle the details of actually making the network connections. The DirectPlay modem server uses the Windows 95 Telephony Application Programming Interface (TAPI) to manage the intricacies of modem connections.

To join a DirectPlay game, you connect to an existing session on the network. Because this connection usually takes place from within a game, you select from the list of sessions that typically shows only one type of game. In other words, if you run a Chess game and try to connect to a session, it will only show you other Chess sessions on the network. This limiting of sessions is implemented at the application

Listing 1. The CGame::DPInit Member Function for TicTacToe

```
BOOL
CGame::DPInit()
{
  // Clear the players
  m_dpidPlayer[0] = 0;
  m_dpidPlayer[1] = 0;

   // Prompt user to select a DP server, then create the DP
object
  CServerSelDlg dlgServerSel;
  if (dlgServerSel.DoModal() == IDOK)
    return (::DirectPlayCreate(dlgServerSel.GetSelServer(),
      &m_pDirectPlay, NULL) == DP_OK);

  return FALSE;
```

Michael Morrison

DirectPlay takes care of developing a network-based game while shielding you from all those messy network protocol and modem details.

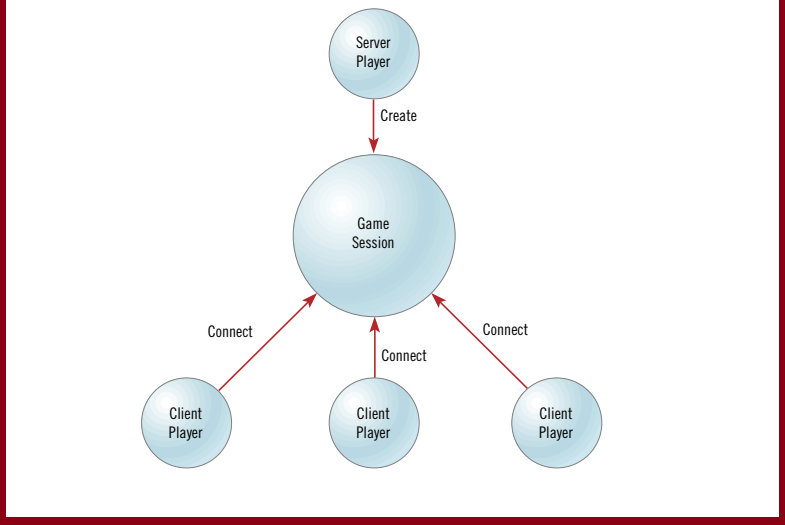


Figure 2. DirectPlay client/server session connections.

level, so it is technically possible to show all available sessions of all game types, which might be useful in a game finder application that shows all game sessions and then launches the appropriate one based on the user's choice.

How are sessions created to begin with? The original player is responsible for initially creating the game session to which other players will connect. When creating a new session, you assign a name to it so other players can find it, such as "Bill's No-Holds-Barred Cage Match." Because all available sessions are likely to be for the same type of game, it is important for you to give your session an identifiable name. Then just sit back and wait for someone to connect to your session so you can get down to business.

Each type of session must be assigned a global identifier, which is guaranteed to be unique for all sessions. DirectPlay uses this identifier when referring to the session internally. This is how DirectPlay keeps up with games created independently. You can generate a global identifier for your game by running UUIDGEN, which is an application that comes with the Win32 SDK. It requires a network card to generate unique identifiers, since all network cards have a unique identifier associated with them.

You might have noticed that DirectPlay imposes a client/server model for initially connecting to game sessions. One of the players must perform the initial session creation. This is the server game. All other players connect to this game as clients. After connections are made, it doesn't matter who made the initial connection. In this way, the client/server model is in effect only during the initial session creation and connection. Figure 2 shows multiple client players connecting to a game created by the server player.

Players

DirectPlay maintains a list of current players in a session and provides an interface to manage them. Each player generally corresponds to other game instances on the network. Each player has a friendly name and a formal name that are set when the player is created, as well as a player identifier (ID). DirectPlay does not use the player names internally; they are solely for player communication during the game or for a high score list. DirectPlay always uses a player's identifier when working with players internally.

DirectPlay also supports player groups, which can be thought of as teams. A player group appears like a player in the session. Information then can be sent to the group, in which case DirectPlay routes the message to each individual player in the group.

Messages

DirectPlay manages communication between players. DirectPlay messages are different from Windows messages and are sent and received through a different protocol. A few DirectPlay system messages let you determine when a connection has been established and when players and groups have been added or deleted. Other messages are custom, game-specific messages that you define. To send a message to another player, you simply call the appropriate DirectPlay function and provide the ID of the player with the message to be sent. The target game then receives the message and processes it accordingly.

DirectPlay Implementation

DirectPlay is implemented as a COM object that represents the entire communications environment for an application. The DirectPlay COM object, `DirectPlay`, provides access to DirectPlay's functionality. DirectPlay contains two API functions used to enumerate DirectPlay servers and create `DirectPlay` objects. You always use one of these functions to create an initial `DirectPlay` object. In fact, you will usually use both functions; you will enumerate and display the available DirectPlay servers and then create a `DirectPlay` object based on the server selected by the user. The DirectPlay API functions are `DirectPlayCreate` and `DirectPlayEnumerate`.

`DirectPlayCreate` creates and initializes a `DirectPlay` object:

```
HRESULT DirectPlayCreate(LPGUID lpGUID,
LPDIRECTPLAY FAR *lplpDP,
  IUnknown FAR * pUnkOuter)
```

`DirectPlayEnumerate` is the other half of the DirectPlay API function pair, which is used to query the system for the available network service providers:

```
HRESULT DirectPlayEnumerate(LPDPENUMDP-
CALLBACK lpEnumDPCallback,
  LPVOID lpContext)
```

Each installed network service provider contains an entry in the registry. DirectPlayEnumerate searches for these entries and notifies you of each supported network server. Practically speaking, you will always want to enumerate and display the available network servers so the user can select from them. After the user selects a server, you pass its global identifier into DirectPlayCreate to create the DirectPlay object and bind it to the selected network server.

The DirectPlay object itself represents the physical network connection and associated information about the connection. To create the DirectPlay object, you specify which DirectPlay server the object will bind to for actual communication. Once the DirectPlay object is created, you can establish a network connection. When you get a pointer to a DirectPlay object via a call to DirectPlayCreate, you don't have a pointer to the DirectPlay object itself; you have a pointer to the IDirectPlay interface of the DirectPlay object. The IDirectPlay interface defines the functions implemented by the DirectPlay object. The most useful functions supported by the IDirectPlay interface are: Close, EnumSessions, Open, CreatePlayer, GetCaps, Receive, DestroyPlayer, GetMessageCount, SaveSession, EnableNewPlayers, GetPlayerCaps, Send, EnumPlayers, GetPlayerName, and SetPlayerName.

The Close member function, HRESULT IDirectPlay::Close(), closes the communications channel (session) for the DirectPlay object.

This means the session will be closed, and all communications will be stopped. Because Close ultimately destroys the session connection, you always must destroy any local players before calling it. Some service providers will not allow a session to close until all players have been destroyed. This is especially important when the player who created the session tries to close it.

The CreatePlayer member function creates a player for a particular session:

```
HRESULT IDirectPlay::CreatePlayer(LPDPID
lpDPId, LPSTR
  lpPlayerFriendlyName, LPSTR lpPlayer-
FormalName,
  LPHANDLE lpReceiveEvent)
```

After you create or connect to a session, you call CreatePlayer to create a local player. When you successfully create a new player using CreatePlayer, DirectPlay sends a DPSYS_ADDPLAYER system message to all other players in the session notifying them of the new player. You are allowed to create multiple local players, in which you use a single machine for multiple player interaction. An example of this scenario is having two joysticks connected to one machine. DirectPlay imposes no limitations on the number of local and remote players, although you can limit the number of players that can be added to your game.

The DestroyPlayer member function destroys a player from a game session:

```
HRESULT IDirect-
Play::DestroyPlay-
er(DPID DPId)
```

You must call DestroyPlayer to destroy any local players you have created before closing the game session. After you successfully destroy a player using DestroyPlayer, DirectPlay sends a DPSYS_DELETEPLAYER system message to all the other players in the session notifying them of the player exiting the session.

The EnableNewPlayers member function toggles the capability to add new players and groups to a session and can be used to keep other players from joining a session:

```
HRESULT IDirect-
Play::EnableNewPlay-
ers (BOOL bEnable)
```

The EnumPlayers member function enumerates the current players in a session:

```
HRESULT    IDirectPlay::EnumPlayers
(LPDPENUMPLAYERSCALLBACK
  lpEnumPlayersCallback, LPVOID lpCon-
text, DWORD dwFlags)
```

The EnumSessions member function enumerates the current game sessions:

```
HRESULT   IDirectPlay::EnumSessions
(LPDPSESSIONDESC lpDPSessionDesc,
  LPDPENUMSESSIONSCALLBACK lpEnumSes-
sionCallback, LPVOID
  lpContext, DWORD dwFlags)
```

EnumSessions is used to build a list of the available sessions, in which you can

Listing 2. The CGame::DPCreateSession Member Function for TicTacToe

```
BOOL
CGame::DPCreateSession()
{
  if (m_pDirectPlay)
  {
    // Get session information
    CSessionInfoDlg dlgSessionInfo;
    if (dlgSessionInfo.DoModal() == IDOK)
    {
      // Create a new DP session
      DPSESSIONDESC dpsdDesc;
      ::ZeroMemory(&dpsdDesc, sizeof(DPSESSIONDESC));
      dpsdDesc.dwSize = sizeof(DPSESSIONDESC);
      dpsdDesc.dwMaxPlayers = 2;
      dpsdDesc.dwFlags = DPOPEN_CREATESESSION;
      dpsdDesc.guidSession = TICTACTOE_10;
        ::strcpy(dpsdDesc.szSessionName, dlgSessionInfo.Get-
Name());
      if (m_pDirectPlay->Open(&dpsdDesc) == DP_OK)
      {
        // Create local player and set game info
        m_pDirectPlay->EnableNewPlayers(TRUE);
        if (DPCreateLocalPlayer())
        {
          DPCreateEventThread();
          m_bMyTurn = TRUE;
          return TRUE;
        }
      }
    }
  }
  return FALSE;
}
```

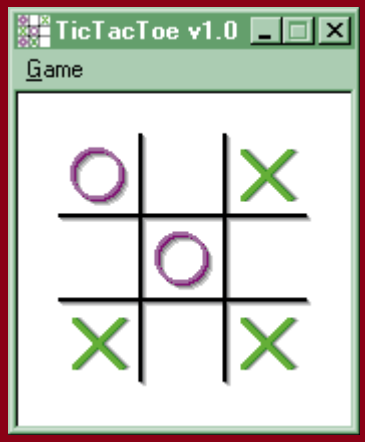


Figure 3. The TicTacToe sample game.

provide an interface for the user to select a session to join. This technique is useful on the client end of a game connection, because it looks for preexisting game sessions to select from.

The GetCaps member function gets the capabilities of the DirectPlay object, which is dependent on the network server to which the object is bound:

```
HRESULT IDirectPlay::GetCaps (LPDPCAPS
lpDPCaps)
```

The GetMessageCount member function determines the number of DirectPlay messages waiting for a particular player and is used to determine when to receive messages for a player:

```
HRESULT IDirectPlay::GetMessageCount
(DPID DPId, LPDWORD lpdwCount)
```

The GetPlayerCaps member function retrieves the capabilities of a particular player:

```
HRESULT  IDirectPlay::GetPlayerCaps
(DPID DPId, LPDPCAPS lpDPPlayerCaps)
```

The GetPlayerName member function queries DirectPlay for a player's friendly and formal names:

```
HRESULT  IDirectPlay::GetPlayerName
(DPID  DPId,  LPSTR  lpFriendlyName,
LPDWORD lpdwFriendlyNameLength, LPSTR
lpFormalName, LPDWORD lpdwFormalName-
Length)
```

The GetPlayerName function is very useful if you want to notify others about a player's actions—for example, a player leaving the game.

The Open member function opens the DirectPlay object and establishes a network connection, which means either creating a new session or connecting to an existing session:

```
HRESULT IDirectPlay::Open(LPDPSESSIONDE-
SC lpDPSessionDesc)
```

The user interface required to actually establish the connection is handled by DirectPlay, such as the dialing interface for a modem connection.

The Receive member function receives pending messages for a player:

```
HRESULT IDirectPlay::Receive(LPDPID
lpDPIdFrom, LPDPID lpDPIdTo,
  DWORD dwReceiveFlags, LPSTR lpMessage,
LPDWORD lpdwLength)
```

You use this function to receive information from other players and from DirectPlay regarding the status of the game. Receive always processes messages with respect to a particular player. DirectPlay has a set of system messages with corresponding structures containing information specific to the system message. You can access the information in each system message first by casting the message data to the generic message structure, DPMSG_GENERIC, and looking at the dwType message type member. The message type will correspond to one of the DirectPlay system messages. Once you know the type, you then can cast the data to the message structure of the appropriate type to access the message-specific data.

The SaveSession member function saves information regarding the current session to the registry:

```
HRESULT IDirectPlay::SaveSession(LPSTR
lpName)
```

This includes information, such as the player's friendly and formal names and phone number, in the case of a modem connection.

The Send member function is the companion to Receive and is used to send information to other players in the session:

```
HRESULT IDirectPlay::Send(DPID DPIdFrom,
DPID DPIdTo, DWORD
   dwFlags, LPSTR lpMessage, DWORD
dwLength)
```

The SetPlayerName member function

Listing 3. The CGameDPCreateLocalPlayer Member Function for TicTacToe

```
BOOL
CGame::DPCreateLocalPlayer()
{
  // Create local DP player
  CPlayerInfoDlg dlgPlayerInfo;
  if (dlgPlayerInfo.DoModal() == IDOK)
    return (m_pDirectPlay->CreatePlayer(&m_dpidPlayer[0],
      dlgPlayerInfo.GetFriendlyName(),
      dlgPlayerInfo.GetFormalName(), &m_hDPEvent) == DP_OK);

  return FALSE;
}
```

Figure 4. The TicTacToe Server Selection dialog box.

sets the friendly and formal names of a player:

```
HRESULT IDirectPlay::SetPlayerName(DPID
DPId, LPSTR
  lpFriendlyName, LPSTR lpFormalName)
```

Using DirectPlay in Games

The first function of a DirectPlay game is to determine whether the user intends to create a new session or connect to an existing session. This function is accomplished through some type of user interface, as determined by the DirectPlay service provider. After the user decides whether to create a new session or connect to an existing session, the `DirectPlay` object must be created and opened with the proper settings.

The next step is to create local players for the session. After the session is open and the players are created, the game is ready to begin. Remember, the same application will be running on both ends, so all players will be visible to each other as soon as they are created. The game then begins, and the play carries on in a way determined by your game-specific messaging protocol.

In addition to handling your own messages, you need to handle DirectPlay system messages. This is very important because it is possible for players to drop out of the game, in which case you will get a system message indicating that the player has left the session.

TicTacToe is a sample game that uses DirectPlay to manage network communications for two players. It is a very simple turn-based game that shows the basics of DirectPlay communication. Figure 3 shows what TicTacToe looks like during a game.

Running TicTacToe

The TicTacToe main menu contains a Game pull-down menu with the following menu choices: Create, Connect, End, and Exit. Create makes a new network session, Connect joins to an existing network session, End stops a network session, and Exit terminates the application. To establish a two-player network connection, one player creates a new session and the other player connects to it. So, the server player first must choose Create from the menu, which causes the Server Selection dialog box (shown in Figure 4) to appear.

In the example in Figure 4, a modem connection has been selected. The Enter Session Information dialog box appears and prompts for the name of the new session. After you enter the session information, the session is opened and the Enter Player Information dialog box appears. Here, you enter information regarding the local player, yourself.

TicTacToe then creates a player using the friendly and formal names you entered in the Enter Player Information dialog box. At this point, a new session has been created with a player representing you, the local player. Now you just sit back and wait for a remote player to join in.

On the remote end, the player chooses Connect from the Game menu. He or she must choose a modem connection from the Server Selection dialog box, just as you did. After selecting the network server, things change. Instead of specifying a new session name, the remote player is prompted with a list of available sessions from which to choose. In this case, there is only a single entry for dialing up a modem session.

After selecting the modem dial session, the remote player sees the Dial dialog box. The DirectPlay model server handles this interface.

After the remote player specifies the phone number of the server session, DirectPlay dials the number and establishes the modem connection. Once connected, the remote player must enter his or her own player information so that DirectPlay can create a local player. After entering player information, the remote player then sees a list of players currently in the game and must select one to play with. Of course, in TicTacToe, there is always just one other player. The main reason for including this feature in TicTacToe is to show how to enumerate other players when joining a session. The remote player must select the server player from a Player Selection dialog box.

Listing 4. The CGame::DPConnectSession Member Function for TicTacToe

```
BOOL
CGame::DPConnectSession()
{
  if (m_pDirectPlay)
  {
    // Select a DP session
    CSessionSelDlg dlgSessionSel(m_pDirectPlay);
    if (dlgSessionSel.DoModal() == IDOK)
    {
      // Open remote DP session
      DPSESSIONDESC dpsdDesc;
      ::ZeroMemory(&dpsdDesc, sizeof(DPSESSIONDESC));
      dpsdDesc.dwSize = sizeof(DPSESSIONDESC);
      dpsdDesc.dwFlags = DPOPEN_OPENSESSION;
      dpsdDesc.guidSession = TICTACTOE_10;
      dpsdDesc.dwSession = dlgSessionSel.GetSelSession();
      if (m_pDirectPlay->Open(&dpsdDesc) == DP_OK)
      {
        // Prompt user to select the remote player
        CPlayerSelDlg dlgPlayerSel(m_pDirectPlay);
        if (dlgPlayerSel.DoModal() == IDOK)
        {
          // Set remote player
          m_dpidPlayer[1] = dlgPlayerSel.GetSelPlayer();
          // Create local player and set game info
          m_pDirectPlay->EnableNewPlayers(TRUE);
          if (DPCreateLocalPlayer())
          {
            DPCreateEventThread();
            m_bMyTurn = FALSE;
            NewGame();
            return TRUE;
          }
        }
      }
    }
  }
  return FALSE;
}
```

Listing 5. The CGame::DPEventMsg Member Function for TicTacToe

```
UINT
CGame::DPEventMsgStart(LPVOID pData)
{
  // Call the DP event handler
  ASSERT((CGame*)pData);
  ((CGame*)pData)->DPEventMsg();
  return 0;
}
void
CGame::DPEventMsg()
{
  while(TRUE)
  {
    // Wait for event
    if (::WaitForSingleObject(m_hDPEvent, INFINITE) !=
      WAIT_TIMEOUT)
    {
      // Process event message
      if (m_pDirectPlay)
      {
        DPID  dpidFrom, dpidTo;
        BYTE  Msg[256];
        DWORD dwLen = 128;
        if (m_pDirectPlay->Receive(&dpidFrom, &dpidTo,
          DPRECEIVE_ALL, Msg, &dwLen) == DP_OK)
        {
          if (dpidFrom == 0)
          {
            // Got a system message
            DPMSG_GENERIC*  pmsgGeneric = (DPMSG_GENERIC*)Msg;
            CString         sText;
            switch(pmsgGeneric->dwType)
            {
              case DPSYS_CONNECT:
                AfxGetMainWnd()->MessageBox("Connected!",
                  AfxGetAppName());
                break;
              case DPSYS_SESSIONLOST:
                AfxGetMainWnd()->MessageBox("Session lost!",
                  AfxGetAppName());
                DPCleanup();
                break;
              case DPSYS_ADDPLAYER:
                // Notify of new player
                sText.Format("New Player : %s", ((DPMSG_ADDPLAYER*)
                  pmsgGeneric)->szShortName);
                AfxGetMainWnd()->MessageBox(sText,
                  AfxGetAppName());
                // Set new player and start new game
                if (((DPMSG_ADDPLAYER*)pmsgGeneric)->dpId !=
                  m_dpidPlayer[0])
                {
                  m_dpidPlayer[1] = ((DPMSG_ADDPLAYER*)
                    pmsgGeneric)->dpId;
                  NewGame();
                }
                break;

              case DPSYS_DELETEPLAYER:
                AfxGetMainWnd()->MessageBox("Player Deleted!",
                  AfxGetAppName());
                if (((DPMSG_DELETEPLAYER*)pmsgGeneric)->dpId ==
                  m_dpidPlayer[1])
                {
                  m_dpidPlayer[1] = 0;
                  DPEndSession();
                }
                break;
            }
          }
          else
            if (dpidTo == m_dpidPlayer[0])
            {
              // Got a remote player turn message
              if (dwLen == sizeof(POINT))
              {
                CPoint ptTile(*((POINT*)Msg));
                DPReceiveTurnMsg(ptTile);
              }
              else
                AfxGetMainWnd()->MessageBox(
                  "Unknown player message.", AfxGetAppName());
            }
        }
      }
    }
  }
}
```

TicTacToe is set up so that the server player always gets to go first. Even so, it is important for the remote player to know that the game has begun. That is the reason for notifying the remote player of the server player's turn. At this point, the game has begun, and the remote player is waiting for the server player to make the first move.

Let's jump back to the server side of things for a moment. When the remote player first connected to the server session, the server player received a connection system message. After being notified of the remote player's connection, the server player is sent an `AddPlayer` message containing information about the remote player. At this point, the server side now knows about the remote connection and the remote player, so the game begins.

Regardless of the outcome, the player who was to go next starts the next game playing Xs. And the game goes on until one of the players ends the session by choosing End from the Game menu or by closing the application.

Under The Hood

Now that you've got a feel for how TicTacToe runs, let's take a look at how it

works. The code that supports DirectPlay is mostly located in the CGame class. Incidentally, all the source code files for the TicTacToe game can be found on the *Game Developer* web site.

The CGame class models the TicTacToe game itself and maintains the DirectPlay connection, along with the players and the game-synchronization logic. CGame keeps a pointer to the DirectPlay object in m_pDirectPlay. This pointer is set by the DPInit member function, which is called by the application to initialize DirectPlay services for the game. The source code for DPInit is shown in Listing 1.

DPInit initializes the player members, m_dpidPlayer[2], and prompts the user to select a network game server by using the dialog object CServerSelDlg. The server identifier retrieved from the Server Selection dialog box then creates the DirectPlay object by calling DirectPlayCreate.

DPCleanup is the corresponding member function for cleaning up the DirectPlay support. It first calls DPEndSession, which destroys the local player by calling DPDestroyLocalPlayer. It closes the DirectPlay object and deletes the DirectPlay event thread used to process messages. Then DPCleanup releases the DirectPlay object and NULLs the member pointer.

CGame creates a DirectPlay session through the DPCreateSession member function, as shown in Listing 2.

DPCreateSession first prompts for the name of the new session by using the CSessionInfoDlg dialog object. It then uses this name to help fill out a DPSESSIONDESC structure that passes into the Open member function of the DirectPlay object. The maximum number of players is set to 2 and the open flag is set to DPOPEN_CREATESESSION. The session identifier is set to TICTACTOE_10, which specifies that this is version 1.0 of TicTacToe. TICTACTOE_10 is a global identifier that uniquely identifies the TicTacToe game. It was obtained by running the UUIDGEN application, and is defined in the file GUID.H.

After opening the new session, DPCreateSession enables the addition of new players and calls DPCreateLocalPlayer. The source code for DPCreateLocalPlayer is shown in Listing 3.

DPCreateLocalPlayer displays a dialog box using the CPlayerInfoDlg dialog object to obtain the friendly and formal names of the new player. It then uses these names in a call to the DirectPlay object's CreatePlayer member function to create the local player. You'll notice that CreatePlayer is passed a pointer to an event handle, m_hDPEvent, as its last parameter. This event handle specifies a Win32 Manual Reset event that is signaled when the player has waiting messages. After creating the local player, DPCreateSession creates an event thread by calling DPCreateEventThread. Finally, DPCreateSession sets the turn member variable, m_bMyTurn, to TRUE, which indicates that the server side of the game goes first. At this point, the session has been created and the local player is eagerly awaiting a connection by another player.

So how does the remote player connect to an existing session, like the one created by the server player with DPCreateSession? DPConnectSession connects to existing sessions and is very similar to DPCreateSession. The primary difference is that DPConnectSession displays the Session Selection dialog box using the CSessionSelDlg dialog object, instead of prompting for information regarding a new session. The DPOPEN_OPENSESSION flag is used in the DPSESSIONDESC structure to specify that you are attempting to open an existing session. The source code for DPConnectSession is shown in Listing 4.

After opening the session, the local player is prompted to select the server player from the dialog box displayed by the CPlayerSelDlg dialog object. The identifier of this player is stored away for later communication, and the local player is created by calling DPCreateLocalPlayer. The player event thread is then created, and the m_bMyTurn member variable is set to FALSE to indicate that the client player goes second. Finally, a new game is started.

During the course of the game, all DirectPlay messages are processed by the DPEventMsg member function (Listing 5). This function is called automatically when a DirectPlay event occurs.

The first call is to WaitForSingleObject, which is a Win32 API function that waits for an event to be signaled before returning. The significance of WaitForSingleObject is that it remains in a sleep state while waiting for the event to occur. You specify an infinite time-out period so that it will never time out.

The first step in processing DirectPlay messages is to receive the message and check the identifier of the source player to see whether it is a system message. System messages always are sent from player 0. If the message is a system message, you cast the data to a generic message structure to get the type of message. If a player has been added, you notify the local player, set the remote player identifier member variable, and start a new game. This scenario occurs when a remote player connects to a session created by the local player. If a player is deleted, which would correspond to the remote player quitting, the local player is notified and the session is terminated.

If the message is not a system message, the message is cast to a POINT struc-

Listing 6. The CGame::DPReceiveTurnMsg Member Function for TicTacToe

```
BOOL
CGame::DPReceiveTurnMsg(CPoint& ptTile)
{
  // Check remote turn message for valid tile bounds
  if ((ptTile.x >= 0) && (ptTile.x <= 2) && (ptTile.y >= 0) &&
    (ptTile.y <= 2))
  {
    // Update game with remote turn data
    SetTileState(ptTile.x, ptTile.y);

    return TRUE;
  }

  return FALSE;
}
```

ture and passed to DPReceiveTurnMsg. DPReceiveTurnMsg notifies the local game of the remote player's move, as shown in Listing 6.

The game-specific messages sent between players correspond to coordinates on the TicTacToe grid. These coordinates are used to specify each player's move. A POINT structure is used to pass this information in DPReceiveTurnMsg. DPReceiveTurnMsg receives this structure and sets the state of the grid tile to the appropriate value, X or O, by calling SetTileState.

SetTileState is the workhorse function for maintaining the state of the game. It is passed the X and Y values of the grid tile to be set. It first checks to make sure that the tile is empty. It then checks whether it is the local player's turn, in which case it sends the tile coordinates to the remote player to signify the move. This is handled by calling DPSendTurnMsg. DPSendTurnMsg simply calls the DirectPlay object's Send member function with the proper parameters. After updating the remote game, SetTileState updates the local game by changing turns, setting the tile state, and updating the window so that the new tile state is displayed. The source code for SetTileState is shown in Listing 7.

After setting the new tile state in both games, SetTileState proceeds to check for a win or draw by calling IsWinner and IsDraw. These two functions contain the logic for determining whether a player has won the game or whether the game is a draw. That's it!

You've seen first hand how you can use DirectPlay to create a fully functioning network game. Almost every aspect of using DirectPlay was touched on, along with sample code for you to reuse in your own games.

Although a practical network game implementation often gets messy, you have the building blocks required to frame up a network game so you can focus on synchronization details. You also have some pretty clean interface objects to use for working with DirectPlay. You have all you need to go write a cool network game for Windows 95! ■

Michael Morrison is the co-author of Windows 95 Game Developer's Guide to Using the Game SDK. *You can contact him via e-mail at gdmag@mfi.com.*

Listing 7. The CGame::SetTileState Member Function for TicTacToe

```
:                                   BOOL
CGame::SetTileState(UINT uiX, UINT uiY)
{
  ASSERT((uiX < 3) && (uiY < 3));
  CWave wavTile;
  if (m_tsGrid[uiX][uiY] == tsEMPTY)
  {
    // Send tile info to remote player via a turn message
    if (m_bMyTurn)
      if (!DPSendTurnMsg(CPoint(uiX, uiY)))
      {
        AfxGetMainWnd()->MessageBox("Error sending turn message.",
          AfxGetAppName());
        return FALSE;
      }
    // Change turns and set the tile state
    m_bMyTurn = !m_bMyTurn;
    m_tsGrid[uiX][uiY] = (m_uiTurns % 2) ? tsO : tsX;
    // Update grid
    AfxGetMainWnd()->Invalidate(FALSE);
    // Play the tile wave
    wavTile.Create((m_uiTurns % 2) ? IDW_O : IDW_X);
    wavTile.Play();
  }
  else
  {
    // Play the tile error wave
    wavTile.Create(IDW_ERROR);
    wavTile.Play();
    return FALSE;
  }
  // Check for winner/draw
  if (IsWinner())
  {
    // Determine winner and notify
    if (m_bMyTurn)
    {
      CWave wavLose(IDW_LOSE);
      wavLose.Play();
      AfxGetMainWnd()->MessageBox("Bummer, you lost!",
        AfxGetAppName());
    }
    else
    {
      CWave wavWin(IDW_WIN);
      wavWin.Play();
      AfxGetMainWnd()->MessageBox("Congratulations, you won!",
        AfxGetAppName());
    }
    // Start new game
    return NewGame();
  }
  else
  {
    if (IsDraw())
    {
      // Play draw wave
      CWave wavDraw(IDW_DRAW);
      wavDraw.Play();
      // Notify of a draw
      AfxGetMainWnd()->MessageBox("It's a draw!", AfxGetAppName());
      // Start new game
      return NewGame();
    }
  }
  return TRUE;
}
```

DirectSound Unplugged

Sound is a powerful, expressive medium—more powerful, I believe, than even our visual sense for conveying information and emotion. John Ratcliff, designer of Seawolf and 688 Attack Sub, has a favorite example of sound's impact: compare a tyrannosaurus rex scene in Jurassic Park both with and without the sound track.

My example is even more dramatic: imagine you watch the great opera La Boheme in New York City, but you wear earplugs. Now, although you may actually find the music tolerable under this condition, opera without sound is essentially just a bunch of fat mimes. And who wants to watch that for three hours?

So there's really no doubt about how much atmosphere sound can add to a game. Unfortunately, the Windows APIs traditionally have given short shrift to audio. Well, no longer—under Windows 95, DirectSound allows you to do everything you could do by accessing the hardware directly, and, as a bonus, provides a solid base for future sound technology developments.

In this article, we'll discuss everything you need to know to add DirectSound to your application. We've only got four thousand words to do it—which isn't a lot (my bad memories of writing class notwithstanding), so we're going to have to cruise. Buckled in?

Listing 1. A Function that Creates Awesome Secondary Buffers

```
HRESULT CreateDSBuffer(LPDIRECTSOUND lpDS, LPDIRECTSOUNDBUFFER * lplpDSB,
          DWORD SoundBytes, DWORD Frequency, int IsStereo, int Is16Bit)
{
  DSBUFFERDESC dsbd;
  PCMWAVEFORMAT fmt;
  fmt.wf.nChannels=(IsStereo)?2:1;
  fmt.wBitsPerSample=(Is16Bit)?16:8;
  fmt.wf.nSamplesPerSec=((DWORD)Frequency);
  fmt.wf.nBlockAlign=fmt.wf.nChannels*(fmt.wBitsPerSample>>3);
  fmt.wf.nAvgBytesPerSec=((DWORD)fmt.wf.nSamplesPerSec)*((DWORD)fmt.wf.nBlockAlign);
  fmt.wf.wFormatTag=WAVE_FORMAT_PCM;
  memset( &dsbd, 0, sizeof(dsbd) );
  dsbd.lpwfxFormat=(LPWAVEFORMATEX)&fmt;
  dsbd.dwSize=sizeof(DSBUFFERDESC);
  dsbd.dwBufferBytes=SoundBytes;
  dsbd.dwFlags=0;
In C++: return( lpDS->CreateSoundBuffer( &dsbd, lplpDSB, 0) );
In C:  return( IDirectSound_CreateSoundBuffer( lpDS, &dsbd, lplpDSB, 0) );
   }
// Sample use of the CreateDSBuffer function
LPDIRECTSOUNDBUFFER lpDSB;
if (CreateDSBuffer( lpDS, &lpDSB, TotalSoundBytes, 22050, 0 , 0) ) { // Open 22050, mono,
8 bit sample
// Use the DirectSoundBuffer
In C++: lpDSB->Release();
In C:  IDirectSoundBuffer_Release( lpDSB );
}
```

This is Not an Overview

Normally, the folks at *Game Developer* magazine respond to the word "overview" like a French chef would respond to a request for ketchup. So, to keep the editorial saliva out of my alphabet soup, we'll zoom through this section as quickly as we can.

First, the DirectSound API is based on the Component Object Model (COM). COM arrived with OLE, but it can stand alone as a standard way to present an API to an application. It lets C++ people access the API with nice object-oriented code, and it lets C people access the API with weird macros. We'll show both types of calling sequences in this article.

COM-based APIs are all used the same way. You call a `Create` function that returns a pointer to an object (C programmers read "structure"). This object contains the important data, as well as member functions (C programmers read "function pointers") that operate on the object. So, with COM, everything the API can do is accessed through an object.

In the DirectSound COM API, we find two objects: the `DirectSound` object and the `DirectSoundBuffer` object. You create the `DirectSound` object to gain

Jeff Roberts

You've long wanted direct access to the hardware within Windows, eh? Here 'tis. DirectSound prodvides a method for playing back and mixing digitally recorded audio within Windows 95.

access to everything that DirectSound can do. Once you have created this object, it can (among other things) create the `DirectSoundBuffer` object, which is the object that actually plays sounds (you knew that feature was in there somewhere, right?).

Make sense? If not, don't sweat it—just remember that we have to create objects to do anything in DirectSound (and in any other DirectX APIs for that matter).

DirectSound Objects

The `DirectSound` object is the key to using the DirectSound API. To create a `DirectSound` object of this type, you simply call the `DirectSoundCreate` function. Since this call is one of only two functions that aren't member functions of an object (the other is `DirectSoundEnumerate`), the calling sequence is the same for both C and C++:

```
LPDIRECTSOUND lpDS;
if (DirectSoundCreate(NULL, &lpDS, NULL)
== DS_OK)
  // lpDS is now a valid DirectSound
    object
else
  // the DirectSoundCreate call
    failed (lpDS is NULL)
```

The first `NULL` in the `DirectSoundCreate` call is the ID of the DirectSound device that you want to open—it will almost always be `NULL`. You can get a list of other valid IDs with the `DirectSoundEnumerate` function. The second parameter is a pointer to where you'd like the DirectSound pointer to be placed (a pointer to an object pointer). The final parameter must always be `NULL` to keep COM happy.

Once you have the `DirectSound` object, you can call any of the eleven member functions that it currently contains. However, there are really only three member functions that you will normally use: `SetCooperativeLevel`, `CreateSoundBuffer`, and `Release`. The other member functions are for infrequent tasks like querying capabilities, compacting onboard sound memory, and managing speaker configuration. Don't worry about them—I've never had to use them and you probably won't either.

You must, on the other hand, use the `SetCooperativeLevel` member function. If you don't call it after creating your `DirectSound` object, most of the other member functions won't work. This silly goof has burned me at least once, and, judging by the CompuServe message traffic, plenty of others. So, if you get a `DSERR_INVALIDPARAM` result from one of the DirectSound functions, check your code and make sure you have set your co-op level.

Since the `SetCooperativeLevel` call is our first member function, let's stop for a moment and discuss calling a COM member function from C++ and C. An example of a `SetCooperativeLevel` call in the two dialects is as follows:

In C++:

```
lpDS->SetCooperativeLevel
( YourMainHwnd, DSSCL_NORMAL );
```

In C:

```
IDirectSound_SetCooperative Level (
lpDS, YourMainHwnd,DSSCL_ NORMAL);
```

You can see how C++ treats a

COM object just like a normal C++ object—you call the function just like you would a normal C++ member function. In C, however, you must use macros to make calls to the member function. These macros serve to make the function calls cleaner and to mask any changes Microsoft may make to COM in the future.

All COM object macros follow the same naming convention: an uppercase "I," the name of the object, an underscore, and, finally, the name of the member function that you wish to call. For example, a `BillG` COM object with a Boolean member function would have a macro called `IBillG_IsLoaded()` that always returned `TRUE`.

OK, back to `SetCooperativeLevel`—the first parameter (besides the object pointer itself) is a handle to your application's main window. Why would DirectSound need an `HWND`? Good question! Microsoft considers sound a "system resource," so when a user flips away from your application, DirectSound mutes all your sound! Although this is correct behavior for most apps, I believe it should have been under our control—not the API's. DirectSound 2 is supposed to fix this lapse with support for background sounds.

Anyway (I'll make it through this function yet), the final parameter to `SetCooperativeLevel` is the priority level you are requesting. There are several different priority levels, but you will almost always use `DSSCL_NORMAL`, which signifies fully cooperative status (as opposed to grumpy, pain-in-the-ass status, I suppose). Actually, the other priority levels mostly create primary sound buffers, which you should rarely need to do. So, for our purposes, just use `DSSCL_NORMAL`.

The next function on my common list is `CreateSoundBuffer`. This function creates a `DirectSoundBuffer` object. We will discuss these objects in the next section—they're where all the action is, so let's finish up the `DirectSound` object first.

Listing 2. The Locking Process

```
HRESULT LoadSoundData(LPDIRECTSOUNDBUFFER lpDSB, char* SoundDataPtr, DWORD TotalBytes)
{
LPVOID ptr1,ptr2;
DWORD len1,len2;
HRESULT result;
  TryLockAgainLabel:
In C++: result = lpDSB->Lock( 0, TotalBytes, &ptr1, &len1, &ptr2, &len2, 0 );
In C:  result = IDirectSoundBuffer_Lock( lpDSB, 0, TotalBytes, &ptr1, &len1, &ptr2, &len2,
0 );
  switch (result) {
   case DS_OK:                    // The DirectSound buffer was locked successfully
  memcpy( ptr1, SoundDataPtr,len1);
  if (ptr2)
    memcpy( ptr2, SoundDataPtr + len1, len2);
In C++:   lpDSB->UnLock( ptr1, len1, ptr2, len2 );
In C:     IDirectSoundBuffer_Unlock( lpDSB, ptr1, len1, ptr2, len2 );
  break;
  case DSERR_BUFFERLOST:          // The DirectSound buffer was lost - try to restore
In C++:   result=lpDSB->Restore();
In C:     result=IDirectSoundBuffer_Restore( lpDSB );
     if (result == DS_OK )        // If the restore worked, go do the lock again
      goto TryLockAgainLabel;
     break;
  }
  return( result );
}
```

The final common `DirectSound` member function is `Release`. This function simply frees the `DirectSound` object. Call it at the end of your application to close `DirectSound`. You may notice that the `Release` function isn't shown in the DirectSound help file because `Release` is one of the standard COM member functions. It is there, though, and you should always call it when you're finished with DirectSound.

That wraps up the `DirectSound` object—doesn't do much, does it? It does, however, allow us to create `DirectSoundBuffer` objects, where the true coolness of DirectSound lies.

DirectSoundBuffer Objects

`DirectSoundBuffer` objects are containers for your actual audio data. They contain both the sound format (bit-depth, frequency, and so on) and a buffer for the sound data itself. There are two types of `DirectSoundBuffer` objects: primary and secondary. You will always create secondary buffers, unless you have a very unusual use for the primary buffer (I know of only one, which I'll talk about in a moment).

Secondary buffers are nice because you can have many open at once. During playback, each buffer is volume-scaled, pan-scaled, bit-depth adjusted, and mixed with other buffers completely on the fly. After the final buffer is mixed, the resultant sound data is placed into the primary buffer to be heard. You don't have to worry about converting, massaging, or mixing any of the data—you just let DirectSound deal with it. Pretty cool!

Which, indirectly, brings us to the only reason to use primary buffers—because all secondary buffers are mixed into the primary buffer, it is the primary buffer that governs the final sound quality. For example, if you play a 16-bit, 44 KHz secondary buffer, but the primary buffer is only 8-bit, 11 KHz, then your sound data will be scaled down to the primary buffer's format.

So, if your sound card is capable, you can create a primary buffer and change its output format to deal with this problem. Usually though, the pri-

mary buffer will be set in the best output mode for your particular sound card, so you'll never need to change it. Because of this fact, we'll focus on the more useful secondary buffers for the remainder of this article. If you really want to use the primary buffer and get stuck, e-mail me, and I'll try to help.

So how do we create these awesome secondary buffers? Well, the example function in Listing 1 does just that.

The first thing this function does is set up a `PCMWAVEFORMAT` structure that contains the type of sound data the secondary buffer will contain. Usually, you will simply load this structure from the header of a .WAV file. For a good example of loading and parsing .WAV files, check out an article titled, "Recording and Playing Waveform Audio" on Microsoft Developer Network (MSDN).

Next, the code sets up a `DSBUFFERDESC` structure that describes the requested secondary buffer. The `dwBufferBytes` field specifies how large the secondary buffer should be in bytes. This amount is usually extracted from the `DATA` chunk in a .WAV file.

The second important field in the `DSBUFFERDESC` structure is `dwFlags`. In this case, we are setting `dwFlags` to zero, but other useful options are `DSBCAPS_CTRLVOLUME`, `DSBCAPS_CTRLPAN`, and `DSBCAPS_CTRLFREQUENCY`. These options tell DirectSound that you will be adjusting the volume, pan, or frequency while the sound is playing. If you don't specify these options when you create the `DirectSoundBuffer`, then you won't be able to control these sound attributes at playback time.

The code then asks the `DirectSound` object to go ahead and create a `DirectSoundBuffer` object for us. If the function succeeds, the `lpDSB` variable will now contain our `DirectSoundBuffer` object pointer. As with the `DirectSound` object, once we're done with a `DirectSoundBuffer`, we must call the `Release` member function.

Now we know how to create a secondary buffer, but how do we get our sound data into it?

Loading Data into a DirectSoundBuffer

To load sound data into our secondary buffer, we have to use the `Lock`, `Unlock` and `Restore` member functions. The locking process is a bit complicated, so let's start with an example function as shown in Listing 2.

Geez, that's a lot of code just to load a buffer! It's pretty simple once we've walked through it though.

The `Lock` function gives us access to the `DirectSound` buffers. Its first parameter is the starting byte location of the lock you request—this will normally be zero unless you're streaming sound data into the sound buffer (we'll talk about this later). The next parameter is the number of bytes you are locking—this will almost always be the same amount that you used for the `dwBufferBytes` field when you created the buffer.

Two sets of pointers and lengths are filled in by the lock call. There are two sets of pointers and lengths because you could conceivably request a lock that wraps around the end of the sound buffer. If your lock parameters didn't cause DirectSound to wrap around its sound buffer, then `ptr2` will be `NULL`. With these two pointers, you can use `memcpy` or `memmove` to place your sound data into the `DirectSoundBuffer` object.

So far so good, but what does the other code do? Well, one of the trickier parts of DirectSound is the fact that you can "lose" your sound buffer. Losing a sound buffer means that the buffer that was holding your sound data has been appropriated for other DirectSound needs. (Even stranger, on some new video-sound combination cards, you can also lose your sound buffers to DirectDraw!)

Losing a buffer is usually no big deal—you just call the `Restore` member function and reload the sound data into the buffer. You can implement various strategies to deal with this: reload the sound files back off the disk, keep the sound in another system RAM buffer so that you can reload it at any time, or, best of all, use streaming buffers (we'll talk about streaming a bit later). The sample code above simply calls the `Restore` function if the buffer was lost, and then retries the lock.

Finally, after you've successfully locked the buffer and loaded your sound data, you must call the `Unlock` member function to give the buffer back to DirectSound. Notice that the `Unlock` function doesn't take pointers to the pointers and lengths (like `Lock` does), but accepts the pointers and lengths themselves. (Try saying that three times quickly.)

So, loading sound data isn't too bad at all. Just remember to have an easy way to reload it if your `DirectSound`

Listing 3. Code to Play a DirectSoundBuffer

```
    DWORD status;
    TryPlayAgainLabel:
In C++: if ( lpDSB->Play( 0, 0, 0 ) == DS_BUFFERLOST )
In C:  if ( IDirectSoundBuffer_Play( lpDSB, 0, 0, 0 ) == DSERR_BUFFERLOST )
     if ( LoadSoundData( lpDSB, SoundDataAddress, TotalSoundBytes ) == DS_OK )
       goto TryPlayAgainLabel;            // Try to play the buffer again
    GetAsyncKeyState(VK_ESCAPE);           // Clear the state of the Escape key
    for (;;) {
In C++:   lpDSB->GetStatus(&status);
In C:    IDirectSoundBuffer_GetStatus(lpDSB, &status);
     if (status!=DSBSTATUS_PLAYING)
      break;
     if (GetAsyncKeyState(VK_ESCAPE))       // If the Escape key is hit, stop the sound
In C++:    lpDSB->Stop();
In C:     IDirectSoundBuffer_Stop( lpDSB );
    }
```

Listing 4. A Streaming Example that can be Pasted into a DirectSound Application

```
typedef struct DSSTREAMTAG {
  int  Playing;               // This field will be non-zero while sound is streaming
  int  PleaseClose;           // Set this field to stop sound streaming
  char* CurrentPosition;      // The next sound address that will be mixed into the DS
buffer
  DWORD BytesLeft;            // How many bytes are left from CurrentPosition
  DWORD NoCallbacks;          // When this is non-zero the timer callback won't execute
  DWORD HalfBufferPoint;      // The size of half the DirectSound buffer (don't change)
  DWORD LastHalf;             // The pointer to the last half buffer that we were in
(don't change)
  int  CloseOnNext;           // Internal flag to mark the end of playback (don't change)
  LPDIRECTSOUNDBUFFER lpDSB;  // The DirectSound buffer that is handling the streaming
  char SilenceByte;           // The value for silence (different for 8 and 16 bit sounds)
} DSSTREAM;
static void StreamCopy(DSSTREAM* s, char* ptr, DWORD len)  // Copy from buffer into DS with
end of buffer handling
{
  DWORD amt;
  amt=(len>s->BytesLeft)?s->BytesLeft:len;     // Only copy what's left in the main sound
buffer
  if (amt) {
   memcpy(ptr,s->CurrentPosition,amt);
   s->CurrentPosition+=amt;
   s->BytesLeft-=amt;
  }
  len-=amt;
  if (len) {                  // Fill the remainder of the buffer with silence
    memset(ptr+amt,s->SilenceByte,len);
    s->CloseOnNext=1;         // Set the "done on the next buffer switch" flag
  }
}
static void StreamFillAHalf(DSSTREAM* s, DWORD half)  // fill a half of the DirectSound
buffer
{
  char* ptr1;
  char* ptr2;
  DWORD len1, len2;

 TryLockAgainLabel:
  switch (s->lpDSB->Lock(half, s->HalfBufferPoint, &ptr1, &len1, &ptr2, &len2, 0)) {
   case DS_OK:
     StreamCopy(s, ptr1, len1);  // Copy sound data into the first pointer

     if (ptr2)                  // Copy sound data into the second pointer if necessary
      StreamCopy(s, ptr2, len2);
     s->lpDSB->Unlock(ptr1, len1, ptr2, len2);
     break;
   case DSERR_BUFFERLOST:          // The DirectSound buffer was lost - try to restore
     if (s->lpDSB->Restore() == DS_OK)
    goto TryLockAgainLabel;
   break;
  }
}
static void CALLBACK StreamTimer(UINT id, UINT msg, DWORD user, DWORD dw1, DWORD dw2)
{
  DWORD playp,writep;
```

buffer is ever lost. I try to make a stand-alone function that I can call from anywhere in my application if my buffer disappears.

Now that we have sound data in our `DirectSoundBuffer` object, we're ready to play it!

Simple DirectSoundBuffer Playback

In comparison to the set up and loading of the `DirectSoundBuffer` object, playback is a piece of cake. The two playback control member functions are `Play` and `Stop`, and they do exactly what you'd guess. As an example, let's look at the code in Listing 3 which plays a `DirectSoundBuffer` until you press `Escape`.

The `Play` member function actually starts the sound. It takes three parameters—the first two are reserved and must be zero. The final parameter is a flag field. Currently, the only flag is `DSBPLAY_LOOPING` which tells DirectSound to keep looping the `DirectSoundBuffer` object over and over. The `DSBPLAY_LOOPING` flag is also used to set up a streaming sounds.

Notice that, again, you have to watch for the sound buffer being lost. If the buffer is lost, then this code simply calls the `LoadSoundData` function that you wrote earlier. This is a workable but clumsy solution, because you have to buffer the sound data twice—once in your own buffer and once inside the `DirectSoundBuffer` object. Alternatively, you could load the sound data off the disk to save the double memory use. However, as I alluded to earlier, the best solution is probably to stream the sound data.

OK, so once the sample begins playing, the above code simply waits until the `GetStatus` member function tells us that the `DirectSoundBuffer` object is finished. This will happen if the `DirectSoundBuffer` plays through to the end of the sample, or you hit the `Escape` key and cause the `Stop` member function to be called.

There are other member functions for controlling the volume, pan, frequency, and playback position of the `DirectSoundBuffer` object, but these are

all pretty self-explanatory so I'll let you experiment with them on your own.

And that's all there is to simple playback. No problem, right? Good, because our final discussion will be about streaming audio. It is a bit more complicated, but definitely worth understanding.

DirectSound Streaming

Sound streaming is the act of using a tiny buffer to play a large sample a little bit at a time. Streaming is generally used to play sound data off a hard drive or CD-ROM, but you can also use it to play a large piece of sound data into a tiny `DirectSound` buffer.

This is how you get around the lost buffer problem—you load an entire sample into system memory and then use DirectSound to play a little bit at a time. In this situation, if you lose the sound buffer, it's no big deal because you are not losing the entire sample—just a little piece.

As you can imagine, playing sound data this way is more complicated than just calling the `Play` member function. The nice thing is that this technique can be encapsulated into one function call fairly easily, so you can just use the same code over and over again.

Basically, DirectSound streaming is accomplished by creating a looping secondary buffer and placing data into it at the right time. DirectSound believes that it is playing the same sound over and over, but actually we're placing new sound data into the buffer each time it loops around to simulate one long seamless sound.

The easiest way to learn streaming is to start with an example that can be pasted right into a DirectSound application to implement streaming immediately. This example will play a sound sample that is loaded into system RAM, but you could easily modify it to play sound off a hard drive or CD-ROM. Let's check it out in Listing 4 (only the C++ calls are shown to make the code easier to read).

To start streaming with this example code, call the `StartStreaming` function with a stream structure, the sound

Listing 4. Continued from p. 44

```
  DWORD whichhalf;
  DSSTREAM* s=(DSSTREAM*)user;
  if (s->NoCallbacks++==0) {
   if (s->PleaseClose) {          // Programmer requested Close - shutdown immediately
    ShutDownStreamingLabel:
     timeKillEvent(id);
     timeEndPeriod(62);
     s->lpDSB->Stop();
     s->lpDSB->Release();
     s->Playing=0;
     return;
   }
    s->lpDSB->GetCurrentPosition(&playp,&writep);   // Get the current position and figure
the current half
   whichhalf=(playp < s->HalfBufferPoint)?0:s->HalfBufferPoint;
   if (whichhalf != s->LastHalf) {
      if (s->CloseOnNext)           // If we previously used up our sound data, then do a
shutdown
       goto ShutDownStreamingLabel;
     StreamFillAHalf(s, s->LastHalf);   // Fill the buffer half that we just left
     s->LastHalf=whichhalf;
   }
  }
  s->NoCallbacks--;
}
void StartStreaming(DSSTREAM* s, void* addr, DWORD len, LPDIRECTSOUND lpDS, LPWAVEFORMATEX
format)
{
  DSBUFFERDESC dsbd;
  if (s) {
   memset(s,0,sizeof(DSSTREAM));
   if ((addr) && (lpDS) && (format)) {
     memset( &dsbd, 0, sizeof(dsbd) );
     dsbd.lpwfxFormat=format;
     dsbd.dwSize=sizeof(DSBUFFERDESC);
     dsbd.dwBufferBytes= ((format->nAvgBytesPerSec/4)+2047)&~2047;
     dsbd.dwFlags=0;
     if (lpDS->CreateSoundBuffer( &dsbd, &s->lpDSB, 0) != DS_OK)
      return;
     s->NoCallbacks=1;      // Don´t let the callback do anything until we´re fully setup
     timeBeginPeriod( 62 );
     if (timeSetEvent( 62, 0, StreamTimer, (DWORD)s, TIME_PERIODIC )==0) {
      timeEndPeriod( 62 );
      s->lpDSB->Release();
     } else {
      s->HalfBufferPoint=dsbd.dwBufferBytes/2;
      s->CurrentPosition=addr;
      s->BytesLeft=len;
      s->SilenceByte= (format->wBitsPerSample==16) ? 0:128;
      StreamFillAHalf(s, 0);
      StreamFillAHalf(s, s->HalfBufferPoint);
        s->CloseOnNext=0;     // Clear the close flag, so that the first two buffers are
played
      s->lpDSB->Play( 0, 0, DSBPLAY_LOOPING );
      s->NoCallbacks=0;
     }
```

Listing 4. Continued from p. 45

```
   }
  }
}
// Streaming test code:
volatile DSSTREAM s;
StartStreaming( (DSSTREAM*)&s, SoundDataAddress, TotalSoundBytes, lpDS, &SoundFormat );
GetAsyncKeyState(VK_ESCAPE);    // Clear the state of the escape key
while (s.Playing) {         // wait until the sound is done or the user hits escape
 if (GetAsyncKeyState(VK_ESCAPE))
   s.PleaseClose=1;
}
```

data address, the sound data length, the `DirectSound` object to use, and the format of the sound data. From there, everything is handled automatically, and sound will start immediately.

If you'd like to stop the streaming, just set the `PleaseStop` field to non-zero. You can monitor the playback with the `Playing` field: non-zero means that the stream is still playing, and zero means that the stream has been stopped and its resources have been freed. Finally, to track the playback position, use the `CurrentPosition` field (it is a pointer that increases from your starting address as playback proceeds).

Now let's shift from describing how to use the streaming example to how it actually works. We'll begin with the `StartStreaming` function.

The `StartStreaming` function only has to set the appropriate values in the `DSSTREAM` structure and start up the timer callback. First it creates a small `DirectSoundBuffer` that will handle one-fourth of a second of audio data. It then sets a timer to call the `StreamTimer` function and assigns all of the initial streaming values. Then the `StreamTimer` function will be called 16 times per second (every 62 milliseconds) to process all of the sound data.

The `StreamTimer` callback contains most of the logic for streaming. The first thing it does is to check to see if the `PleaseClose` flag is set; if so, it closes the streaming for this sound. Next, the timer checks where the current play position is with the `GetCurrentPosition` member function. If DirectSound has moved from one half buffer to the next, then the old half buffer is ready for new sound data.

The `StreamFillAHalf` function is used to load sound data into one half of the `DirectSound` buffer. It handles the locking, restoring, and unlocking of the `DirectSoundBuffer` object. The `DirectSoundBuffer` object, in turn, calls the `StreamCopy` function to move the data from your large sound buffer into the tiny `DirectSound` buffer.

One bit of semi-tricky logic is found when the `StreamCopy` function determines that the end of your sound data has been reached. `StreamCopy` can't just close the stream immediately, because you wouldn't hear the last little bit of sound, so it sets a flag called `CloseOnNext`. This flag is checked on the next buffer switch in the `StreamTimer` function which lets you hear the last buffer's worth of sound.

This example code is rather simple—integrating it into your own application should be a snap. A few cool features to tack on: the ability to pause the playback, a smarter callback that handles multiple streams (instead of one callback per stream), and the ability to stream from a disk file. Go crazy with it—after all, you have the source code!

This is Not a Summary

Well, you made it! You now know almost everything there is to know about DirectSound.

After you've used it a while, I think you'll agree that Microsoft really did a great job on DirectSound: it is a terrific, low-level API to play digital sound with little to no latency response. As this article demonstrates, however, DirectSound is *not* a high-level API. A DirectSound application must handle buffer management, callbacks, streaming, start and stop control, and such on its own behalf.

For those who don't want to deal with the low-level coding that DirectSound requires, there are several good libraries that combine DirectSound's awesome playback abilities with a true application-level API to give you the best of both worlds.

Personally, I think the best thing about DirectSound is that my complaints about the old days of Windows programming are getting better and better.

"Back when I was a Windows programmer, we had to walk to work in two feet of snow every morning, and hexadecimal hadn't been invented yet, and we didn't have DirectSound, and my mouse was a real dead mouse with wires shoved up its ...." ■

Jeff Roberts is a programmer at RAD Software, publisher of Smacker and the Miles Sound System. He can be reached via e-mail at gdmag@mfi.com.

DirectSound for the Impatient

For those of you who don't want to read this whole article, just follow the following easy steps to add DirectSound to your application in no time:

1. **Create a DirectSound object.**
2. **Set the cooperative level to `DSSCL_NORMAL`.**
3. **Create a secondary sound buffer.**
4. **Lock the sound buffer.**
5. **Fill the buffer with your sound data using the two pointers and lengths returned by `Lock`.**
6. **Unlock the sound buffer.**
7. **Play the sound buffer.**
8. **Release the sound buffer.**
9. **Release the DirectSound object.**

Playing with Waves

You've just blown away a room full of bad guys. You then hear a door open and a squad of gurgling demons behind you. A breeze carries snatches of tinny post-apocalyptic singing forced through a wheezing old FM radio. "The population is greatly decreased…" You head toward it, hoping to find a friend.

There's no substitute for the pleasure audio gaming experiences bring to players' ears. Without sound, games lack essential life-saving audio cues, humor, character, and magic. I've spent much time discussing cross-platform graphics and user interaction, and it's about time I rounded things off with a bit of audio.

In this article, I will leave you with a short demo that runs without change on top of a small core of Windows- and Macintosh-specific code. You will see how to implement a simple PlayWave function that will allow up to four simultaneous sounds to play using system services, in this case the Macintosh Sound Manager and Windows' DirectSound.

Basic Wave Theory

The algorithm for mixing two waves isn't very difficult. Sound waves are additive: two waves played at the same time produce a combined wave that is their sum, as approximated in Figure 1. The goal of a software mixer is to allow a single wave synthesizer to play multiple waves simultaneously, so its job is to perform the addition itself before playing the combined wave, instead of leaving the job to natural physics.

Electronic sound devices usually play digitally sampled waves. A digital description of a sound wave is created by recording the amplitude of a wave input at a constant frequency using a specific number of bits to measure the wave at each time slice. Today's hardware generally handles 8 or 16 bits per sample at frequencies of 11,025, 22,050, or 44,100 samples per second, with higher frequencies and resolution producing a more accurate replica of the original while requiring additional memory and processing power.

A one-second, 8-bit wave sampled at 11,025 hertz is represented by a block of 11,025 bytes, each of which describes the amplitude of the wave at a specific point in time with a number from -127 to 128. To mix two such waves into a third buffer, the mixer steps through each wave's 11,025 samples and adds them together, storing the result in a third 11,025-byte block to be played out through the speaker. This is the elemental mixing operation: adding two waves.

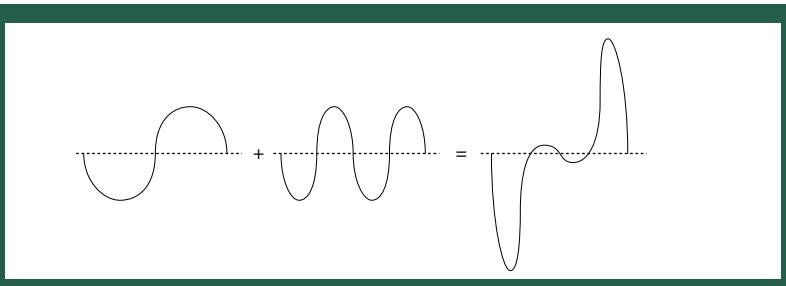


Figure 1. Two waves playing simultaneously, approximately.

You can perform other operations on sampled sound data to produce special effects. You can change the volume of a sound by multiplying or dividing every sampled value by a constant. You can fade a sound by dividing each sample value by a number that increases or decreases across the wave. You can create an echo by mixing a wave with itself with a slight delay. You can play waves backwards, slow them down, distort them, or do whatever mathematical transformations you like.

Usually, a game has simple run-time needs: it has to mix waves with different starting and ending times into a single continuous audio stream. When you fire two shots in quick succession, you want to hear the second even though the first has started playing, and you still want any active background music or other noises to play through.

Generally, a game doesn't have the luxury of mixing the two waves together before playing them as one combined wave because they don't start or end at predefined times. Someone has to mix new sounds into an active audio stream by mixing in the new wave starting just after the point from which the sound hardware is playing.

There are other things a mixer has to worry about, too. For instance, the

mixer doesn't have infinite memory at its disposal, so an application may ask to play a sound that's too large to fit in the buffer the mixer is using, so the mixer may have to break the sound into pieces. Or maybe the application wants to mix two waves sampled at different rates or with different resolutions. A general purpose mixer has to convert them to a common format.

Books have been written on the subject of signal processing, and numerous software mixers and sound editing tools abound. You can dig into them, but I have neither the space, the time, nor the expertise to write another volume on sampling theory. As long as you understand the basics of what your wave mixer does, you don't have to deal with the gritty details. Unless you want to.

Pitfalls

Both DirectSound and Sound Manager cope with playing whatever sound in whatever format of whatever length you throw at them. You can take their abilities for granted. However, it's important to understand the basics of wave mixing in order to understand a fundamental performance difference between the two.

Programmers who write wave mixers must inevitably answer the following question: What happens when the sum of any two samples from the waves to be mixed exceeds the resolution of the sample? How do you mix two 8-bit samples when their sum is greater than 128 or less than -127? A byte simply can't handle that information. Digital technology has let you down.

At this point, you have two basic choices. You can divide every sample in half, essentially restricting each wave to the -63 to +64 range to guarantee the sum to fall between -127 and 128, or you can continue as usual, replacing any sum greater than 128 with 128 and less than -127 with -127. I suppose you can always ignore the problem, too, but I don't recommend that option.

We know already that dividing samples in a wave by a constant (in this case 2) lowers the volume of the wave. So basically, the first technique is to turn down every wave to be mixed so that they're guaranteed never to exceed the maximum volume. You divide each sample by the number of waves played before adding them together. That's the same as adding them all together and dividing by the number, so I call this averaging the waves. I'm going to break my impartial reporter front for a second to say that this is the wrong way to mix waves for games!

The second technique guarantees that the waves you play will always be played at the expected volume, but if the sum of all the waves played exceeds the limits of the digital representation, the highs and lows will get chopped off. If you know you're going to be mixing four sounds, you may choose to author your sounds in a reduced range so they never distort, but that's up to you. This mixing technique is called clipping the waves because it clips off the highs and lows.

Figure 2 shows these two mixing methods at work. Looking at these diagrams, you can see how averaging distorts the shape of the wave, squashing it into silence. When you've only got eight bits to describe a sound sample, you don't want to throw any of them away by division! On the other hand,

Jon Blossom

DirectSound and Sound Manager handle the process of mixing audio differently. Knowing each platform will prevent your game's sounds from unexpected clipping as well as average to low volumes.

Listing 1. PlayWave for the Macintosh

```
Global channel information initialized by BeginSound
SoundHeader ChannelHeader[4];
SndChannelPtr pChannel[4];
void PlayWave(int SampleSize, int SampleRate,
short BitsPerSample, short ChannelCount,
char unsigned* pSample)
{
   // Look for a channel to use to play this sample
   int ChannelToUse = -1;
   for (int Count = 0; Count < 4; ++Count)
   {
      // An available channel is
      // recognized by a null sample pointer
      if (pChannel[Count] && !ChannelHeader[Count].samplePtr)
      {
        ChannelToUse = Count;
        break;
      }
   }
   if (ChannelToUse > -1)
   {
      // Found an unused channel...
      // Set up buffer information
      ChannelHeader[ChannelToUse].length = SampleSize;
      ChannelHeader[ChannelToUse].loopStart = SampleSize;
      ChannelHeader[ChannelToUse].loopEnd = SampleSize;
      // Allow only 11025Hz samples
      // This is just to save space. The code  on
      // ftp.mfi.com allows 22050 and 44100 as well
      ChannelHeader[ChannelToUse].sampleRate = rate11025hz;
      ChannelHeader[ChannelToUse].baseFrequency = rate11025hz;
      // Allow only 8-bit samples
      // Again, see the code on the ftp site
      // The stdSH code indicates 8-bit samples
      ChannelHeader[ChannelToUse].encode = stdSH;
      // Set up the sound data to indicate
      // that the channel is playing
      ChannelHeader[ChannelToUse].samplePtr =
        (char*)pSample;
      // Play the sound!
      SndCommand Command;
      Command.cmd = bufferCmd;
      Command.param1 = 0;
      Command.param2 = (long)&ChannelHeader[ChannelToUse];
      SndDoCommand(pChannel[ChannelToUse], &Command, false);
      // Queue up a callback to reset the channel
      // header when finished. The command gets passed
```

Listing 1. Continued from p. 50

```
        // as an argument to the callback function.
        // In this case, param2 will contain a pointer to
        // the memory to be zeroed when the wave terminates.
        Command.cmd = callBackCmd;
        Command.param1 = 0;
        Command.param2 =
(long)&ChannelHeader[ChannelToUse].samplePtr;
        SndDoCommand(pChannel[ChannelToUse], &Command, false);
    }
}
pascal void SoundCallBack(SndChannelPtr pChannel, SndCommand*
pCommand)
{
    // This function gets called when we queue up a
    // callBackCmd above, to indicate that the sample
    // has finished playing.
    //
    // The command's param2 points to the
    // ChannelHeader.samplePtr of the channel that finished,
    // which we zero to indicate that it is no longer playing.
    *((Ptr*)pCommand->param2) = 0;
}
```

clipping can distort the tips of the wave, creating harsh highs and lows like the ones you might hear escaping speakers that have been pushed to a volume they can't support.

For reasons I can neither explain nor imagine, Apple decided that Sound Manager should average waves, even if mixing them normally wouldn't cause clipping. This guarantees you will never hear clipping from waves Sound Manager produces, but it also guarantees that individual sounds drop in volume as other sounds begin to play.

DirectSound clips waves that exceed the playback resolution. This guarantees that your sounds will be played at full volume, but it also leaves your sounds susceptible to clipping.

Playing the Mac

Now we're going to make some noise, starting with the Macintosh. Two platforms worth of code is too much to fit in one article, so I've only printed the highlights here. Check out the *Game Developer* web site for the full source code.

In spite of my exhortation against mixing waves by averaging (shudder), I'm going to show you a simple way to use the Sound Manager.

Sound Channels are the essential communication pipes to the Sound Manager. They're essentially queues of commands to be processed, including commands to play a buffer from memory, loop over a specific piece of a sample, or perform other simple sound operations.

To play a sample from memory, we have to set up a channel, initialize a `bufferCmd` command that points to the wave data to be played, and call `SndDoCommand` to pass the command down the pipe. By following that with a `callBackCmd`, we can have the Sound Manager call us back when it's finished playing the sample.

That's all it takes. Every time you send a `bufferCmd`, the Sound Manager mixes all the active sounds into one audio stream and plays it out the speaker.

For our purposes, we'll allow four sounds to be played at once. The `BeginSound` function creates four sound channels using `SndNewChannel`, registering the `SoundCallBack` function as the target of a `callBackCmd` command for that channel. It initializes a `SoundHeader` structure to describe the sound wave installed in each channel. Each header initially contains null as the pointer to its sound, indicating the channel is unused. `EndSound` cleans up this work when we're done playing.

The `PlayWave` function, shown in Listing 1, implements the heart of the system. It searches the four channels to find an unused one, as indicated by a `ChannelHeader` whose sample pointer is null. If it can't find one, it refuses to play.

If it does find a free channel, `PlayWave` fills in the associated `ChannelHeader`

Listing 2. PlayWave for DirectSound

```
// Global channel information initialized
// by BeginSound
static LPDIRECTSOUNDBUFFER pChannel[4];
static LPDIRECTSOUND pDirectSound = 0;
void PlayWave(int SampleSize, int SampleRate,
short BitsPerSample, short ChannelCount,
char unsigned* pSample)
{
  // Look for a channel to use to play this sample
  int ChannelToUse = -1;
  for (int Count = 0; Count < 4; ++Count)
  {
      if (!pChannel[Count])
      {
        // This channel isn't in use
        ChannelToUse = Count;
        break;
      }
      else
      {
        DWORD Status;
        HRESULT DSResult =
                  pChannel[Count]->GetStatus(&Status);
        if (DSResult == DS_OK &&
                  !(Status &
                  (DSBSTATUS_PLAYING | DSBSTATUS_LOOPING)))
        {
                  // This channel has finished playing -
                  // it's OK to free it and use it now
                  pChannel[Count]->Release();
                  pChannel[Count] = 0;
                  ChannelToUse = Count;
                  break;
        }
      }
  }
  if (ChannelToUse > -1)
  {
      // Found an unused channel...
      // Set up buffer information
      WAVEFORMATEX WaveFormat;
      WaveFormat.wFormatTag = WAVE_FORMAT_PCM;
      WaveFormat.nChannels = ChannelCount;
      WaveFormat.nSamplesPerSec = SampleRate;
      WaveFormat.wBitsPerSample = BitsPerSample;
      WaveFormat.cbSize = 0;
      WaveFormat.nBlockAlign = WaveFormat.nChannels *
        (WaveFormat.wBitsPerSample / 8);
      WaveFormat.nAvgBytesPerSec = WaveFormat.nBlockAlign *
        WaveFormat.nSamplesPerSec;
      // Set up a DirectSound buffer
      DSBUFFERDESC BufferDesc;
      ZeroMemory(&BufferDesc, sizeof(BufferDesc));
      BufferDesc.dwSize = sizeof(BufferDesc);
      BufferDesc.dwFlags = DSBCAPS_STATIC | DSBCAPS_CTRLDEFAULT;
      BufferDesc.dwBufferBytes = SampleSize;
      BufferDesc.lpwfxFormat = &WaveFormat;
      // Create a new buffer using the settings for this wave
      HRESULT DSReturn =
        pDirectSound->CreateSoundBuffer(&BufferDesc,
                  &pChannel[ChannelToUse], 0);
      if (DSReturn == DS_OK && pChannel[ChannelToUse])
      {
        // Lock the buffer and copy in the data
        BYTE* pData;
        DWORD DataSize;
        if (pChannel[ChannelToUse]->Lock(0, SampleSize,
                  &pData, &DataSize, 0, 0, 0) == DS_OK)
        {
                  memcpy(pData, pSample, SampleSize);
                  // Unlock the buffer
                  pChannel[ChannelToUse]->Unlock(pData,
                            DataSize, 0, 0);
                  // Actually play it!
                  pChannel[ChannelToUse]->Play(0, 0, 0);
        }
      }
  }
}
```

with the sample characteristics, points it at the specified wave data, and sends a `bufferCmd` command to the appropriate channel to start the wave playing. In the listing, I've restricted `PlayWave` to 8-bit 11,025Hz samples, but the code available on the *Game Developer* web site allows for others.

Before leaving, `PlayWave` queues up a `callBackCmd` command, which will result in a call to `SoundCallBack` when the wave has finished playing. `SoundCallBack` zeroes the sample pointer in the appropriate `ChannelHeader`, making the channel once again eligible to play a wave.

The Well-Tempered PC

The Sound Manager requires you to create channels only for the sounds you want to mix, implying a specific playback buffer into which all channels are mixed. But DirectSound has no such default. A primary buffer represents the sound moving through the hardware, and the application must create a primary buffer before it can play any sound through the hardware.

The Windows `BeginSound` implementation handles the set-up of the primary buffer. Because a primary DirectSound buffer must be associated with a window, `BeginSound` creates a simple static text control and sets up DirectSound for exclusive audio access through that window before creating the primary sound buffer. `EndSound` reverses all that and frees the additional buffers created in the process of playing waves.

Once the system is set up, the Windows `PlayWave` implementation follows a pattern similar to the one described for Sound Manager. Specifically, it looks for an unused channel among the four allowed, creates and sets up a buffer for the requested sample, and calls `Play`.

Listing 2 shows the source code for this function. Notice that every call to

PlayWave creates a new DirectSound buffer to hold the wave you're playing. DirectSound doesn't play waves directly from memory like Sound Manager does, so PlayWave has to lock the buffer, copy in the wave data, and unlock it. That may be time-consuming and may even involve downloading a wave to the sound hardware. A better system could avoid that by keeping waves to be played in preallocated and precopied DirectSound buffers.

The Echo Chamber

I've included a demo that uses the standard C file package to open a file called sample.wav, assumed to be in the .WAV file format used by Windows and read in sampled wave data. Then, it calls PlayWave eight times with that data, leaving 3/4 of a second between each call, waits another five seconds, and terminates.

Since .WAV is a PC format, containing data in Intel byte ordering, the demo uses two functions to adjust them to the run-time format. I've declared Swap16 and Swap32, which swap bytes into Motorola format on a Mac and leave bytes intact on a PC. For a 16-bit sample, every 16 bits of the wave data will have to be swapped around as well, a factor that made me decide to leave this demo in 8-bit land.

I urge you to compile and run these on a Mac and on Windows 95 if you can. You'll immediately hear the difference between averaged and clipped mixing. The

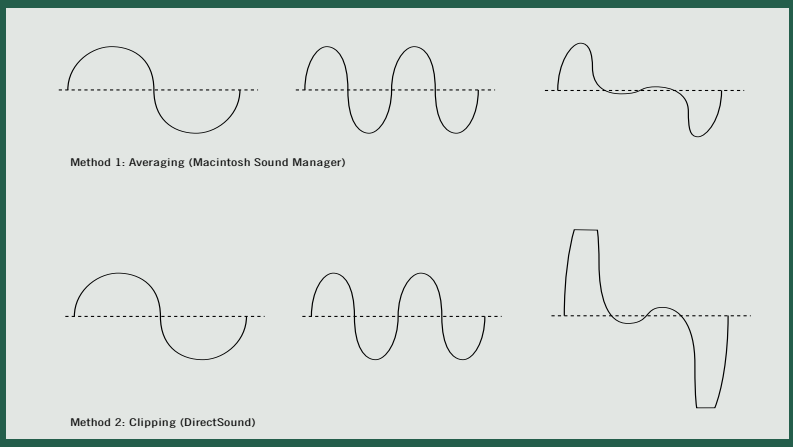


Figure 2. Two mixing methods.

Listing 3. Code to Play a Wave Eight Times

```
// The simple Wave Mixing API
int BeginSound(void);
void EndSound(void);
void PlayWave(int SampleSize, int SampleRate,
short BitsPerSample, short ChannelCount,
char unsigned* pSample);
// Byte-swapping functions
short unsigned Swap16(short unsigned value);
long unsigned Swap32(long unsigned value);
// The demo
void DemoMain(void)
{
   int SampleSize =0;
   int SampleRate =0;
   short BitsPerSample =0;
   short ChannelCount =0;
   char unsigned* pSample =0;
   // Load a wave file
   FILE* pFile = fopen("sample.wav", "rb");
   if (pFile)
   {
      // We're just going to assume this file is valid
      // Skip the 'RIFF' tag and file size (8 bytes)
      // Skip the 'WAVE' tag (4 bytes)
      fseek(pFile, 12, SEEK_SET);
      // Now read RIFF tags until the end of file
      unsigned long Tag;
      unsigned long Size;
      while (!feof(pFile))
      {
         // Read, watching for file end
         if (fread((char*)&Tag, 1, 4, pFile) == 0)
            break;
         Tag = Swap32(Tag);
         fread((char*)&Size, 1, 4, pFile);
         Size = Swap32(Size);
         if (Tag == 0x20746D66) // The 'fmt ' tag
         {
            // 16-bit PCM flag - assume PCM format
            fseek(pFile, 2, SEEK_CUR);
            // 16-bit Channel Count
            fread((char*)&ChannelCount, 1, 2, pFile);
            ChannelCount = Swap16(ChannelCount);
            // 32-bit Sample Rate
            fread((char*)&SampleRate, 1, 4, pFile);
            SampleRate = Swap32(SampleRate);
            // Skip Average bytes per second - (4 bytes)
            // Skip padding - (2 bytes)
            fseek(pFile, 6, SEEK_CUR);
            // 16-bit Bits Per Sample
            fread((char*)&BitsPerSample, 1, 2, pFile);
            BitsPerSample = Swap16(BitsPerSample);
            // Skip whatever's left
            if (Size > 16)
               fseek(pFile, Size - 16, SEEK_CUR);
         }
         else if (Tag == 0x61746164) // The 'data' tag
         {
            // Allocate space and read in the wave
            pSample = (char unsigned*)malloc(Size);
            if (pSample)
            {
               SampleSize = Size;
               fread((char*)pSample, 1, Size, pFile);
            }
         }
         else
         {
            // An unknown tag - just skip it
            fseek(pFile, Size, SEEK_CUR);
         }
      }
      fclose(pFile);
   }
   // Now play the wave!
   if (pSample && BeginSound())
   {
      long unsigned Time;
      // (Attempt to) play 8 times
      int PlayCount = 0;
      while(PlayCount < 8)
      {
         PlayWave(SampleSize, SampleRate, BitsPerSample,
            ChannelCount, pSample);
         ++PlayCount;
         // Wait 3/4 of a second between plays
         Time = GetMillisecondTime();
         while (GetMillisecondTime() - Time < 750)
            ;
      }
      // Wait 5 seconds then quit
      Time = GetMillisecondTime();
      while (GetMillisecondTime() - Time < 5000)
         ;
      EndSound();
   }
   // Clean up
   if (pSample)
      free(pSample);
}
```

Sound Manager makes the demo sound like an echo chamber, repeating the wave over and over at progressively lower volumes while the DirectSound demo provides eight crisp new shots. Try increasing the number of channels allowed to 8, swap in a loud wave, and listen as all your sounds are reduced to one-eighth their original volume by the Sound Manager and clipped to distortion by DirectSound.

There are ways to combat the Sound Manager's dynamic volume adjustments. You can guarantee that four sounds will always be playing by forcing the unused channels to loop over a buffer full of zeroes. You can artificially turn up the volume on the channels as new waves are mixed in. Or you can write your own mixer that plays through a single Sound Manager channel, but I won't be held responsible for the results! ■

Jon Blossom can be reached through Game Developer magazine.

Players Bored? Storyboard!

David Sieks

A low-tech tool can help pack your high-tech animations with visual power. Don't underestimate the power sketches can bring to your projects.

Each of us interprets differently what we read, so that even the author cannot really know what pictures his or her words might paint in the reader's mind. This is a wondrous, pseudo-magical thing about the written word as a means of creative expression. It's also the reason a written script generally proves insufficient as a tool for cinematographers, animators, and game developers, who are working largely with graphical concepts. When the visual element is this important to the end result, it is crucial that everyone involved has the same picture in mind. In most cases, the script remains a necessity, but the very flexibility that leaves its phrases open to personal interpretation makes text too inexact for the planning and sharing of visual concepts. This is where the storyboard comes into play.

In case any reader is unfamiliar with the term, a storyboard is a graphical reinterpretation of the script, using a series of rough sketches to convey setting and action from scene to scene, sometimes even from one movement to the next. Filmmakers and animators have used these visual devices for decades. As digital graphics grow in sophistication and importance, developers too rely increasingly on this tool to communicate and plan a game's visual element.

In general, the storyboard is a visualization aid. It helps establish the setting and the flow of action and pinpoints the positions of "actors" as well as the vantage point of the viewer. During game development, the storyboard is used to plan in detail the cinematic sequences used for game intros and cut scenes. Storyboarding is also useful in mapping out gameplay routines, such as character movement cycles. The information captured on the storyboard then serves as a visual shorthand for the artists who must bring the animation to completion.

Though indispensable as a tool for collaboration, the storyboard is of equal importance to the lone artist. Its usefulness is not just to share a visual concept but to plan the whole sequence out ahead of time. As I've noted before, it is tempting for the artist to plunge into an animation, but planning ahead is crucial to achieve the best possible results. Don't assume that at some point later in production you'll work out those issues left unresolved when you began. Animation is not an improvisational art. It's much easier to make changes before you begin animating. The storyboard is the place for that to happen.

Keep It Simple

Storyboarding isn't rocket science, but some approaches are generally more useful than others and some fairly well-established misconceptions need to be avoided. In this column, I am talking about the storyboard as a rough tool for planning and sharing visual information. I'm not concerned with presentation-quality storyboards often used to pitch a project or sell an idea. When called for, such presentation storyboards are created easily enough by making a prettified copy of your working storyboard. It's counterproductive, for several reasons, to add polish to your sketches prior to this.

The first reason is that the storyboard should be considered a work in progress, not a work of art. It holds every

Storyboarding has long been a staple of TV and movie production. Artists at Tom Snyder Productions have just three weeks to turn out each new half-hour episode of the award-winning Comedy Channel animated sit-com "Dr. Katz: Professional Therapist." Animator Mark Usher observes that the show's stock characters, sets, and a signature, minimalist animation style they call Squigglevision mean that storyboards can be quickly created by sketching in changes over preexisting frames from earlier episodes. The storyboard also guides the editor in piecing together the work of several artists for the final edit.

aspect of a sequence up for scrutiny before investing time and effort in animation. It is successful when it elicits change: changes represent a problem or weakness discovered and fixed at an early stage or a good idea replaced with an even better one. Since, due to these changes, many sketches may need to be scrapped and reworked, it's best not to have invested unnecessary time and effort into making them look pretty only to discard them later.

Which leads to the second reason to keep detail to a minimum in storyboard sketches: ego. A storyboard that survives review without changes probably wasn't looked at critically enough. Suggested changes are not a personal indictment of the artist's talent. This is hard for artists to accept, though, if they have prepared a storyboard filled with painstaking drawings. Better to think of the storyboard as visual shorthand and keep detail to a minimum. A lovingly detailed storyboard sketch is akin to a beautifully carved ornamental wooden matchstick: it's so pretty you can't bear to use it as intended.

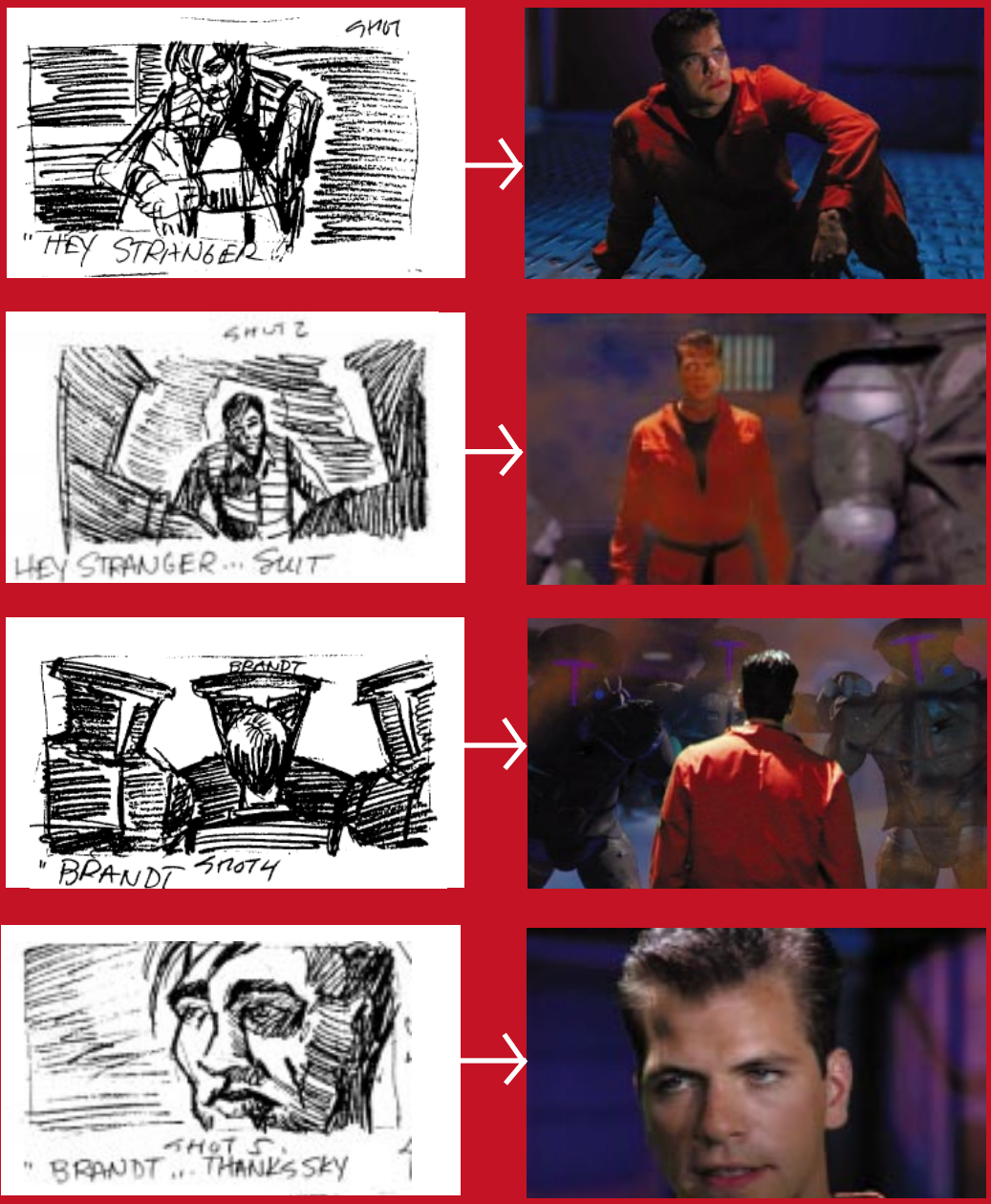
An aside to those working with a storyboard artist: you have entered the Realm of Creative Endeavor. Watch where you step: there are unexploded egos all around.

The final reason to build the storyboard from quick sketches is something else I've talked about in this space before: flow. Anything that slows down the process of translating written script to visual information threatens to stifle the creative flow. When storyboarding for animation, you are planning something that ultimately will move and have a palpable pace to it. You need to convey that dynamism even in the static panels of the storyboard. If you get bogged down in the details of a single drawing, you lose that momentum: the end result is likely to be disjointed and unsatisfactory. Keep it simple and energetic.

Another worthwhile observation about detail, or lack thereof—it is a good practice to excise repetitive information from storyboard sketches. To establish setting, you will want at some point to indicate the background or portray the general color scheme. But it is unnecessary to duplicate this information in sketch after sketch if it has not changed from one to the next. Storyboarding captures what is dynamic in the scene: spend minimal time and effort on things that remain static.

In keeping with the quick and disposable nature of storyboard sketches, artists probably do not want to use those storyboard layout sheets with columns of neat preprinted rectangles on each page. Avoid these because it makes changes more difficult when, for example, there are eight sketches on a single piece of paper and you need to replace two of them. It also makes it near impossible for the artist to remain unconcerned about the "quality" of the image while drawing, especially once there are already a couple of acceptable sketches on the page and any slip-up threatens the work already done.

Instead, use single sheets of small paper—numbering them, if necessary, to keep them in sequence. Group the sheets on a large board afterwards or as you work, securing them with tacks or repositionable adhesive. With single sheets, it is easy for the artist to discard a drawing that isn't coming out right or to implement a different approach to the scene. Also, on a storyboard made up of single sheets it is a simple matter to remove sketches during a



Terra Nova: Strike Force Centauri, the new title from Looking Glass Technologies, features numerous cut scenes blending live action video with computer graphics sets and actors. By planning these scenes on the storyboard, artists were able to focus design and modeling efforts where they were needed and not waste time on areas that would go unseen. The storyboard also proved useful on the set for staging actors around virtual props that had not yet been created. Pictures courtesy Looking Glass Technologies Inc., Cambridge, Mass. Terra Nova, Looking Glass, and the distinctive logos are trademark of Looking Glass Technologies.

meeting to mark areas that need further attention.

The storyboard needs to consist of a series of sketches that establish setting and action and guide the creative team through the task of animating the sequence. Each change of viewpoint must be shown; each new character that appears onscreen must be indicated. However, you need not dwell on the details of characters, which should already be worked out on model sheets (for more on model sheets, see "A Question of Character" in the Feb./Mar. issue).

Of course, it is important to register actions as well: movement within the scene is one chief reason we require a storyboard rather than a single sketch. However, don't let the flow from sketch to sketch slow down excessively by indicating actions in great detail. More than one sketch can certainly be used to express a complex action, but not too many more. For the purposes of the sequence storyboard, your aim is to capture the extremes of the movement and leave it at that. A separate action storyboard can show in

detail how a character walks or hops or falls down.

Explanatory text will sometimes be unavoidable on the storyboard, but keep it to a minimum. There will be a written script of some sort to start with. The storyboard is a complement to this, not a replacement for it; nor should a storyboard repeat most of the script. The action depicted in the storyboard should be self-explanatory. If not, it's probably worth reconsidering the depiction you have chosen. Text that indicates the accompanying dialogue or narration, however, can help communicate the sequence's pace and is routinely included in a separate block below each sketch.

Consider This

Even more important than the form the storyboard takes or the appearance of its individual sketches are the considerations that go into taking a written script and turning it into an animation. The storyboard will act as the animators' map on this journey. The challenge is not just to convey the plot and action described in the script, but to settle upon the very best way to tell that story visually.

In laying out the storyboard, one is dealing with the foundations of good visual storytelling. Great modeling, rendering, and fluid animated movement can seem strangely hollow if the story is not told with verve and style. The groundwork for these elements is your storyboard layout. Your task may be to simply make a splash sequence for a fighting robot game: the script calls for the robots to stomp around, fly through the air, and shoot rockets at each other, and then the title pops up. Simple enough, on the surface. But how can you make that compelling?

You want to transport your audience, cast a spell over them, and even for a few moments take them somewhere beyond their computer screen—make them believe and care about what they see on that screen. You don't have to add to the plot or action beyond what is presented in the script, but you should challenge yourself to make the most of the material. Maybe you show the action from inside the cockpit of one of the giant robots, or from down at ground level to emphasize their great size, or distorted by a wide-angle lens, or with a series of quick takes from all these angles and more.

How you tell the story affects how the audience perceives and responds to it. What is shown, what is only implied, what angles you use, what actions or moments are emphasized—these decisions contribute to the overall impact of the sequence. Always be wary of the easy way out: avoid the obvious, hold cliché at bay. Whether the aim of the sequence is to amuse, frighten, or thrill, look for opportunities to show the audience something different and unexpected. The storyboard phase is the opportunity to consider all these things, to try out different ideas, and to settle on an approach before getting down to the work of animating.

When the storyboard is laid out in front of you, take a critical look at everything that makes up the sequence: justify the presence of each sketch. Your sequence will be composed of a number of "shots." Even if it all takes place in one room, it is shown first from one angle, then another; this action occurs, then that. Each is a shot, and your storyboard must establish every shot. Do they all really need to be there? Is there something interesting about every shot in the sequence? If not, you either eliminate the shot, or you find a way to give it more punch. Cut scenes within games are by necessity too short to support any dead weight.

With all that in mind, it is often important to plan an animation with an eye toward economy. How long will it actually take to animate the sequence as storyboarded? What 3D models will need to be created, in what degree of detail? What texture maps, backgrounds, and special effects are called for? The answers depend on how the sequence has been laid out in the storyboard. Practical limitations of budget and deadline may mean that the best way to tell the story from a creative standpoint just isn't realistic from a production standpoint.

The storyboard provides you with an opportunity to realize limitations ahead of time and plan around them. For example, one might frame shots so that it's only necessary to model the head and shoulders of a character rather than the entire body. Position certain figures at a distance, or show them in stark silhouette so that less-detailed models can be used. Or, as I suggested in my article on lighting in the previous issue, use a cast shadow gobo to represent a figure and thereby avoid actually modeling it at all. A carefully considered storyboard helps you balance at the earliest possible stage the twin demands of what-looks-best and what-time-allows.

Even if there won't be any fancy animated cut scenes in your game, the storyboard is a valuable tool for mapping character movement routines or action sequences. You'll find that most considerations outlined above will still apply. Make the movement dynamic for the time being, keep detail to a minimum, and above all don't be boring. Even if you're just animating a running sprite for a side-scrolling game, use storyboarding to lay out different approaches and find a way to make that run look interesting.

Storyboarding will probably be the least exotic tool you use in creating digital animation, but it will also prove one of the most valuable. I doubt any artist who has learned to appreciate the opportunities it provides for planning animations, for optimizing animations, and for sharing visual concepts with a team would trade storyboarding for all the special effects plug-ins ever made. ■

Dave Sieks is a contributing editor to Game Developer. *You can contact him at gdmag@mfi.com.*