G A M E   D E V E L O P E R   M A G A Z I N E

AUGUST/SEPTEMBER 1996

UBM Tech

Back

Wild at Heart

Activision's seen it all. It was one of the first independent, third-party developers on the original Atari 2600 back in 1979, and went on to produce titles for Nintendo and Sega. It weathered hard times with the cartridge market in the early '80s, and today its diverse action and adventure games cover the Sony, Sega, and Nintendo consoles as well as Intel and Macintosh. This issue we look at one of their latest efforts, a futuristic sports game called HyperBlade which feels a lot like the 1975 James Caan classic, "Rollerball."

Activision collaborated on HyperBlade with WizBang!, a game development firm out of Seattle, using a homegrown development environment called ADLIB. WizBang! created this ADLIB development system, which let them create and manipulate the behavior of objects (such as your rollerblading opponents) at a high level. The beauty of ADLIB is that it lets creative but non-technical staff help develop a game. It's refreshing to see a company abstracting game design to a level where a wider range of people can participate in game development.

HyperBlade isn't your typical sports game, but then again Activision seems to revel in its unique line of games. For instance, in their upcoming Blast Chamber, you control a person trying to defuse a time bomb strapped to his body. In Interstate 76, another soon-to-be-released title, Activision transplanted their MechWarrior engine into a retro-style '70s car combat simulator (sort of a "Dukes of Hazard" on 'roids). The combination of far-out concepts and high production value is one of the reasons the company has been able to weather the ups and downs of the fickle game market. I asked Eric Johnson, Activision's vice president of marketing, why they pursue wild concepts.

"Activision believes that the 'me too' strategy is not effective over the long haul. Of course we've capitalized on sequels to our past successes like Zork and Pitfall, but we feel that if you're going to develop something new, develop something completely new," Johnson said. "I've seen a lot of Doom-style products, and there are quite a few Myst and Command & Conquer wannabes coming out. But we don't think that's a good strategy because it's usually the game that strikes first—the one that has an amazing story, fantastic graphics, or is a technological breakthrough—that becomes the market leader. And when you establish a lead over your competition this way, it's hard for your competitors to catch up because the lead time to develop a game is so damn long."

Activision's strategy is logical, consistent, and intelligent. They milk cash cows like Zork, MechWarrior, and Pitfall for reliable returns while the company simultaneously explores new avenues. It works.

Going out on a conceptual limb with a game has a downside though, as Johnson explained: "We want to create breakthrough titles, and we take risks to do so. The challenge is communicating with the public—that's one of our biggest hurdles on the marketing side. It's not easy explaining Blast Chamber to someone."

Successful development collaborations will continue to play a part in Activision's future. Johnson explained that Activision is continually on the lookout for new WizBang!s to work with, and that their business development unit continually scours the country in search of companies with innovative technologies and proven track records. So if you have a fantastic idea—and the talent and experience to back it up—you may have found yourself a potential publisher. Just remember: think wild concepts. ■

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
Western Account Manager
Barbara Wren (415) 356-3362
bwren@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

Back

Readers Want Flames, Believe It or Not

FLAME ON!

Dear Editor:

The article by David Sieks, "Where the Sun Don't Shine" (April/May 1996), was great coverage of lighting and creating mood. However, his article didn't mention anything about torchlight and other animated light forms caused by fire. Since I am currently working on a project that incorporates this kind of lighting, I would like to find out more about it. Do you have an idea of where to look?

Thomas Buytaert
Via e-mail

David Sieks replies:
When first writing the article, I had considered the issue of firelight. The problem I ran into is that approaches vary from one 3D package to the next, depending on the tools available. Unfortunately, I haven't seen very much written on flames or torchlight that I can recommend to you. One article that does come to mind is "Fun With Flames," a LightWave how-to by Bill Arbanas in 3D Artist *#17. It's got a lot of LightWave-specific detail, but many techniques should also prove of more general interest if you use another package.*

There's also a "Pyromania" CD-ROM of animated flames and explosions, though in a quick look around here I wasn't able to turn up the company that offers it. I bet a web search would help. I don't know what software you are using or what you have attempted so far in the way of flame effects, but the two chief effects you have to create are the flames themselves and the light cast by the flames. As far as the first item, you probably don't want to try to model a leaping torch flame in 3D. Instead, try making a short, looping 2D animation of a flame shape surrounded by an alpha key color (so that the negative space around the flame drops out). You can use a yellow-to-orange gradient to color the flames, but move the gradation slightly up and down over the course of the sequence so it doesn't look too static. Apply this flame sequence as an animated texture map to a flat polygon in your 3D program. Position this flame-painted polygon to face your camera. Many 3D packages let you link the two together so that even as the camera moves, the polygon turns to face it.

As for the flickering light of the torch, you'll need to place one or—preferably—more light sources on or very near the flame-painted polygon. Animate the intensity levels of the lights so that their brightness increases and diminishes and animate a color shift back and forth between yellow and orangey-red. Depending on how much your animated fire map "leaps," you may also want to animate a small range of movement for the lights.

Multiple light sources, each shifting hue (at a different rate) from yellow to orangey-red, with animated intensity levels never quite in sync, positioned slightly apart from one another and all casting shadows should give a really nice look. Of course, shadow-casting is a real time-sink, so you have to strike your own balance here.

If your flame is off camera and you just need the effect of flickering light and leaping shadows in your scene, it might be easier to use an animated projection map in front of your light source: make an abstract, looping 2D animation of yellows and oranges amidst patterns of light and dark and project this animated map onto your scene.

This is rather general, but I hope it's been some help. I'd be interested to hear of any refinements or new tricks you come up with as you continue with your project.

I HAVE A GREAT IDEA...

Dear Editor:

In a not so recent *Game Developer* article ("Let's Get To The [Floating] Point," February/March 1996), Chris Hecker derived some equations for properly taking the floor and mod of negative numbers. I just had an idea which may speed up the method he gave.

If you know that the number you want to operate in is never going to be less than -A where A is a positive integer, pick some integer B>A. Then, when you want to do a floor on x, add in B, do the floor and mod with the assurance that you are operating on a positive number, and then subtract B out of the floor. For example (note I am modding with 1):

x = -2.3
B = 100
floor(100+-2.3) = floor (97.7) = 97
97.7 mod 1 = .7
then, subtract B from the floor
(97-100) = -3

This gets rid of the jumps needed when checking for the positive and negative cases.

Tim DeBruine
Via e-mail

Chris Hecker replies:
That's a good idea! I've also seen cases where you can bias your entire coordinate system until it's positive to avoid the subtraction. This technique also works for the floating-point conversion from another article.

We Want Your Feedback!
Please direct all comments, questions, and suggestions to the *Game Developer* web site http://www.gdmag.com. Thanks!

Who's Connected?

Diane Anderson

Deals are going down all around town—between Ten and Netscape, QSound and Sonic Foundry, Rendition and Intel.

Online Gaming

Multiplayer gaming over the Internet is big business: Ten and Netscape have teamed up. Ten's front-end software package will include Netscape Navigator's Web browser. Thus, Netscape gains distribution through hit games, and Ten members get a web client. When Ten launches the final version of its service, the Netscape arrangement will take effect.

■ Ten
San Francisco, Calif.
(415) 778-3500
http://www.ten.net/

Sonic Youth

Sonic Foundry has teamed up with QSound Labs to develop a new plug-in for Sound Forge, the digital audio editor for Windows. QTools/SF combines three plug-in tools that add QSound Labs's patented audio technology to Sound Forges's palette of wave editing, modification, and effects for PCs. The set features static placement of mono sound files at a user-selected location along a 180 arc in front of the listener, processing of existing stereo image creating a dramatically widened 3D soundfield, and a high-definition sample rate converter.

■ Sonic Foundry
Madison, Wis.
(608) 256-3133
http://www.sfoundry.com
http://www.qsound.ca

Rendition, Intel, Microsoft

Rendition will support Intel's Accelerated Graphics Port (AGP) specification. The announcement is a boost for the AGP standard, which makes its debut in Pentium Pro-based PCs sometime in 1997.

The AGP specification intends to leapfrog existing 3D-graphics solutions by offering four times the bandwidth for graphics with real data throughput of over 500MB/second. The result, according to Intel, will be a new level of realism, speed, and detail for games. Rendition will support AGP in future versions of its Verite 3D graphics accelerator chip.

Verite's architecture consists of a flexible RISC core and pixel pipeline. Rendition's first-generation Verite chip reaches consumers with the release of Creative Labs's 3D Blaster PCI.

The Verite chip served as the hardware design reference platform for Microsoft's Direct3D API. Verite will also bring real-time Direct3D acceleration to the entire family of Direct3D, Windows 95 based titles from Microsoft. Pricing for the 3D Blaster PCI starts at $349.

■ Rendition
Mountain View, Calif.
(415) 335-5900
http://www.rendition.com

Mplaying

If you heard Brian Moriarty's "The Point Is" at the CGDC like I did, you've heard of Mpath Interactive. Mpath announced it garnered the support of 11 major software developers for its upcoming Internet gaming service, Mplayer. This list includes game giants such as Accolade, Blizzard Entertainment, Maxis, New World Computing, and Strategic Simulations and will provide Mplayer subscribers with many new titles to use with the service.

Mplayer's advantage over competing online gaming services is speed. Mpath's proprietary network architecture provides low latency, allowing twitch-style and other fast action games to be played over the Internet. The company has also developed interactive voice features, which will allow players to communicate with each other as they play.

■ MPath
Cupertino, Calif.
(408) 342-8800
http://www.mpath.com

DimensionX

Dimension X released a beta version of their Liquid Reality developers kit. It is the first platform-independent implementation of VRML 2.0 and the only VRML 2.0 toolkit coded entirely in Java. Liquid Reality with VRML 2.0 support is available on the Windows 95, Windows NT, Solaris, and Linux platforms with Macintosh and Pippin versions to follow.

Liquid Reality is integrated with the Microsoft ActiveX and DirectX technologies. It also includes support for 3D sound, compatibility with multiuser servers, an open API, 250 classes to support 3D content creation, and Java classes upon which developers can build a branded VRML 2.0 browser.

■ Dimension X
San Francisco, Calif.
(415) 243-0900
http://www.dimensionx.com

Back

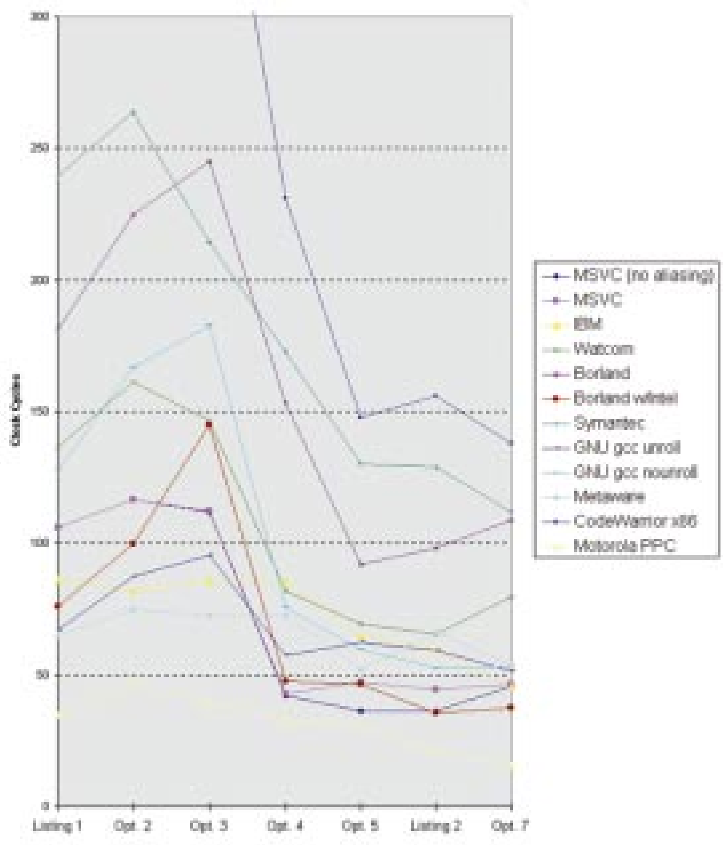
More Compiler Results, and What To Do About It

I sure am glad my full-time job isn't reviewing compilers, because their ubiquitous bugs and wacky user interfaces would drive me insane. However, evaluating compilers does have its moments, like when I found the following paragraph in the Watcom 10.6 compiler's help file under the heading, "What you should know about optimization":

"The C/C++ language contains features which allow simpler compilers to generate code of reasonable quality. Register declarations and imbedding [*sic*] assignments in expressions are two of the ways that C allows the programmer to "help" the compiler generate good quality code. An important point about the Watcom C/C++ compiler is that it is not as important (as it is with other compilers) to "help" the compiler. In order to make good decisions about code generation, the Watcom C/C++ compiler uses modern optimization techniques."

Considering Watcom did around the fourth worst overall in my simple performance test, and did a bad job compiling the texture mapper as well, it would behoove the Watcom compiler writers to refrain from reading their own help files and get back to work on those "modern optimization techniques." Of course, I shouldn't single out Watcom for abuse just because they handed me a convenient passage in their help files. Just like my last article "PowerPC Compilers: Still Not So Hot" (Behind the Screen, June/July 1996), all the compilers this time around deserve abuse, so let's get to it.

Figure 1. The Timing Results



The Contestants

This month, we'll finish up my two-part series on compiler optimizations. Last issue, I evaluated a bunch of C++ compilers for the Macintosh PowerPC platform, and this time I'll do the same for the current crop of x86 compilers. I hesitate to call these "reviews" since I'm not completely evaluating every compiler feature or every possible optimization. However, unlike most compiler reviewers, I'm actually focusing on the compiler—you know, that tiny part of the 200MB Integrated Development Platform and Suite of Accompanying Visual Applications that actually generates the computer code for your application. I'm assuming that since you're reading this magazine you're interested in fast code, and the compiler's the part of the above-mentioned 200MB that generates (or, as we'll see, doesn't generate) that fast code.

This month will be slightly more than an x86 version of last issue's column, however. I'll quickly recap the test results and then move on to what the results from the two articles mean to you as a performance-oriented game programmer.

Chris Hecker

Chris Hecker finishes up his two-part series on compiler optimizations and interprets what the test results mean to you as a performance-oriented game programmer.

I tested eight compilers this time around: Microsoft Visual C++ 4.0, the beta 4 of IBM's VisualAge for C++ for Windows V3.5, Borland C++ 5.0, Watcom C++ 10.6, Metaware C++ 3.32 for OS/2, Metrowerks CodeWarrior 8 in x86 cross-compilation mode, the Free Software Foundation's gcc 2.7.2 on Linux, and Symantec C++ 7.2.

Table 1 shows the eight compilers and their results on my test programs, plus one extra row for both Microsoft VC++ and gcc with different command-line switches. I also included one extra row for Borland using the Intel optimizing backend they supply, and just for kicks I've included the results from the Motorola PowerPC compiler from last time. The timing numbers are in clock cycles per iteration of the test loop, on my 133Mhz Pentium. The PowerPC results are on my 132Mhz PPC604, so it's a pretty fair comparison. I realized after writing the last column that a table full of numbers doesn't exactly tell the most interesting story, so Figure 1 is a graph of Table 1.

The Test

To test the x86 compilers, I used the same simple product of a three-by-three matrix and an array of three element vectors that I used on the PowerPC compilers. Listing 1 shows the first attempt at the code and corresponds to the first column of Table 1. As you move across the table, each column represents a new optimization I applied to the base code in an attempt to coax reasonable output from the compilers. The compilers did pretty poorly on Listing 1; most were two to three times slower on it than on their fastest code, which was usually attained on the function in List-

Table 1. The Timing Results

	Listing 1	Opt. 2	Opt. 3	Opt. 4	Opt. 5	Listing 2	Opt. 7
MSCV (no aliasing)	105.9	116.3	111.7	42.1	36.3	36.3	45.6
MSVC	106.1	116.5	112.1	42.8	47.1	44.4	46.4
IBM	86	81.9	85.7	85.8	64.5	59.6	45.6
Watcom	136.4	161.4	146.4	81.9	69.5	65.6	79.6
Borland	181.1	224.9	245.2	153.1	91.9	98.1	108.84
Borland w/Intel	75.9	99.7	145	47.7	46.4	35.4	37.5
Symantec	239.7	263.8	214.3	172.5	130.2	129	111.9
GNU gcc unroll	67.2	87.4	95.5	57.4	62.3	59.3	51.6
GNU gcc no unroll	127.6	166.9	182.6	75.7	59.6	52.6	53
Metaware	65.1	74.8	72.4	72.6	51.9	66.4	53.9
CodeWarrior x86	309.3	331.3	400.3	230.9	147.7	155.8	137.9
Motorola PPC	34.5	47.4	39.5	33.2	30.8	20.6	15.5

Note: For source code and optimizations, refer to the June/July 1996 issue.

ing 2 and whose results are recorded in the fifth column.

I'm not going to explain the different test programs in detail because I covered that last time. For the complete story and an explanation of the weird variable names in Listing 2, pick up the June/July 1996 issue. The final column of Table 1 shows the results of applying the optimization mentioned in the last paragraph of that article to Listing 2.

In brief, the same criticism I leveled on the PowerPC compilers applies to the x86 compilers: you have to spoon-feed them already optimized code to get reasonable results.

The biggest difference between the PowerPC compilers and x86 compilers is that while you can coax a bad PowerPC optimizer (such as Symantec's PowerPC compiler) into producing almost-optimal code, the same was not true of the bad x86 optimizers (such as Symantec, Borland, and CodeWarrior). I believe generating good PowerPC floating-point code is relatively straightforward compared to generating good x86—and especially Pentium—floating-point code. The wackiness of the Pentium Floating-Point Unit (FPU), with its FXCHes, stack-based operands, and stalls, makes optimizing difficult for the x86 compilers. Of course, you'll notice the difference in cycle counts between the PowerPC and x86 tests. My 132Mhz PPC604 is twice as fast as my 133Mhz Pentium at running this code. The speed difference is due to the flat register-based, floating-point architecture of the PowerPC, combined with a quaternary multiply-accumulate instruction. "Quaternary" instructions have four operands (d = a * b + c in the case of a multiply-accumulate); contrast this with the pathetic stack-based binary x86 instructions, where the compiler is forced to constantly move operands around, and you can see why there's a huge difference. Too bad about that annoying market share thing, huh?

It's always a good idea to try to calculate the optimal cycle count for your functions to see what kind of performance improvements are possible, so let's do that for the x86 and the PowerPC. We'll ignore loop overhead and any stalls and assume maximum throughput for this estimate to give ourselves a lower bound.

For the x86, my estimate for the minimum clock cycles to do our matrix multiply is 30 cycles: 9 multiplies at an optimistic 1 cycle each, 6 additions also at 1 cycle each, 3 stores at 2 cycles each, and 9 loads for the source vector because the binary x86 instructions don't let you keep an untouched copy of it in registers. For the PowerPC, we can load the whole matrix into registers before we start, so I count 15 cycles: 3 loads, 9 multiply-additions, and 3 stores. Both estimates are close to the best times we achieved, so we can be pretty sure we're not missing anything major in our analysis.

I suppose, if forced to pick a winner, I'd choose the Microsoft compiler. It seemed to do what it was told most of the time, so if you give it highly optimized code it does an okay job. The Borland compiler with the Intel backend did okay as well, but its quality seemed slightly more random (note the spike in Figure 1). I should also note the Intel backend wouldn't compile my texture mapper correctly, while Borland without the Intel backend compiled it correctly but generated the code quality you'd expect from Borland's position on Figure 1. The IBM and the Metaware compilers were the most consistent of the bunch, meaning they did better than most on the unoptimized functions, as in Listing 1. To me, this indicates both compilers recognize optimization opportunities at a high level but can't generate tight x86 machine code at the low level. Watcom was the most disappointing of the bunch, simply because the conventional wisdom says Watcom generates great code. I didn't see great code from Watcom in my tests.

The main point here is that you cannot expect the compiler to do much work for you beyond a rote translation of the code you write into native machine code. (With the incredible code generation bugs I've found, you sometimes can't even expect this.) If you write a loop that does one simple thing and you express it with 10 inefficient operations, the compiler will faithfully translate all ten operations for you, performance be damned.

With that in mind, let's discuss the kinds of optimizations you should be able to expect from the compiler but currently have to perform yourself.

Transformers, More Than Meets The Eye

When a compiler optimizes your program it (supposedly) does work at a number of different levels. At the lowest levels, it obviously needs to generate the fastest instruction sequence for a given atomic high-level language operation: a C addition of two integers shouldn't turn into much more than a machine code addition with a possible load or store. At a higher level, the compiler puts your code through a series of program transformations which turn the code you wrote into something more amenable to the lower-level code generator. These transformations aren't algorithmic changes. For example, the compiler won't change your O(n^2) bubble sort to an O(n log n) quick-

Listing 1. The Initial Code

```
void TransformVectors0( 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 = 0.0f;
                        for(j = 0;j < 3;j++) {
                                Value += pMatrix[i][j] * pSourceVectors[j];
                        }
                        *pDestVectors++ = Value;
                }
                pSourceVectors += 3;
        }
}
```

sort; that part is up to you (and algorithm changes are still the most important part of optimizing with the sole exception of profiling your application to make sure you know where to optimize). There are a number of these transformations available to the compiler, but we'll discuss what I think are the five most important ones: alias analysis, code motion, common subexpression elimination, strength reduction, and loop unrolling. You can perform these transformations on your code better than the current crop of compilers, once you know how they work.

Alias Analysis

As we've seen in previous articles, memory is slow compared to registers, so it would be really nice if the compiler could keep all your active variables in registers and operate on them there. If it could do this, it wouldn't have to keep touching memory to reload everything after every store. With pointers, however, it's not that simple. If your code performs reads and writes through two pointers, the compiler needs to decide whether one pointer can point to the same object as the other, a phenomenon called pointer aliasing. For example, think about what would happen in Listing 1 if `pDestVectors` pointed into the middle of `pMatrix`; it certainly wouldn't behave the same as Listing 2. According to the ANSI standard, the compiler needs to be pretty conservative and assume the worst for pointers to variables of the same type. So, one of the first transformations I made to Listing 1 was to use local temporary variables to make explicit to the compiler where I could alias pointers. The compiler knows a write to a temporary cannot affect anything else if you've never taken the address of the temporary. I initially declared a temporary array (as you saw in the previous issue), so I didn't have to unroll the matrix multiply loop, but none of the compilers used this temporary array to eliminate spurious reloads. Apparently today's compilers can't do alias analysis on arrays. I also looked for a compiler switch to make the compiler assume I wasn't aliasing pointers. Most compilers have these switches, and I turned them on when I found them.

Table 1 contains results for the Microsoft compiler both with and without the "assume no aliasing" switch turned on, and you can see the switch makes a big difference on Listing 2. From looking at the disassembly, it looks like the speed increase is due to the compiler now having the ability to move the stores to `pDestVectors` around to better schedule the code. It didn't have this freedom when it had to assume writes to the destination could be writing into one of its source operands. However, you can also see it makes little difference on the unoptimized code; it would be hard to make that code any slower.

On the other hand, you don't necessarily want to copy all of your active variables into temporaries, as I found out the hard way. The final column in Table 1 shows the results of code that copies the entire matrix into temporaries before entering the loop. On the PowerPC, you can see this gave me a 25% speedup because the compiler could copy the matrix into registers and reduce the number of loads in the inner loop. On the x86, however, most compilers slowed down on this code because they actually implemented the copies to temporaries. This behavior is related to alias analysis, I believe. If the matrix is in stack-based temporaries in the source code, the compiler needs to prevent writes through `pDestVectors` from changing the matrix elements, so it makes a copy of the matrix in the generated machine code. The PowerPC compiler didn't have to do this because it knows `pDestVectors` can never point into the floating-point registers, where it's keeping the matrix. The x86 compilers couldn't put the matrix in the floating-point registers, so they needed to copy it. This is a particularly bad example, because it means our C level optimizations aren't portable across machines: the PowerPC version got faster while the x86 versions got slower on the same code.

As an aside, I wish the ANSI C++ standard would loosen up their requirements for compilers to support pointer aliasing so pointers to `const` could be assumed to not be aliased by pointers to non-`const` in a function. However, I'm sure this would break a ton of code that relies on aliasing, so it's not likely to happen. I'd say this code is poorly written and deserves to be broken, but aliasing is a complex issue and I might be missing a legitimate use of it.

Code Motion

When you move loop invariants out of the loop, you're performing code motion. A loop invariant is something that doesn't change during the life of the loop, so it makes sense to calculate it once outside the loop and store it rather than calculate it every time. Code motion can also mean rearranging code so that it pipelines better or accesses memory sequentially for better memory bandwidth.

Common Subexpression Elimination

A common subexpression is an operation that appears multiple times in your code. For example, if you compute x + y in two places, and neither x nor y can change between those two places, then x + y is a common subexpression. Usually it's faster to compute the expression once and store its result than to compute the result multiple times. Of course, there's an exception to every rule, especially in these days of wicked fast processors and slow memory systems. Computing something and caching it might be slower than just computing it multiple times. Time your code, as always. By the way, I've seen the acronym "CSE" applied to both Common SubExpression and Common Subexpression Elimination.

Strength Reduction

The classic example of strength reduction is turning a multiply or divide by a power-of-2 into a shift. I'm not sure why it's called strength reduction, but the basic idea is to convert an expensive operation into a cheap one or a series of cheap ones. Taking the classic example a step farther, you can break up more complicated multiplies into simpler ones (for example, x * 6 = x * 2 + x * 4), which can again be strength-reduced to some shifts and adds. Some architectures might further benefit from strength-reducing shifts by 1 to an addition. Another subtle but powerful example of strength reduction is a Bresenham line drawer, or any kind of forward differencing algorithm. These algorithms convert linear or even arbitrary degree polynomial equation evaluations into a bunch of additions by computing the forward differences outside the loop.

Replacing a divide with a multiplication by the reciprocal is an optimization that could arguably be called strength reduction, but it could also be considered an example of a related transformation called algebraic identification. You can guess from the name what that means.

Loop Unrolling

Finally, we come to everyone's favorite optimization—loop unrolling. Here, we try to mitigate some loop overhead and perhaps open up possibilities for pipelining by duplicating the loop body and reducing the loop count to compensate. Of course, you have to deal with some setup issues if your loop count doesn't divide evenly by your unroll count.

I got another large speedup in our test code by unrolling the loop in Listing 1 on my way to Listing 2, and this speedup was particularly surprising because it's a no-brainer. Alias analysis and code motion are hard. Unrolling a loop is basically a cut-and-paste operation.

Listing 2. The Optimized Code

```
oid TransformVectors5( float  *pDestVectors,
const float  (*pMatrix)[3],
const float  *pSourceVectors, int NumberOfVectors )
{
    int Counter;
    float Value;
    float _Krr1;
    float _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;
    }
}
```

The biggest thing to watch out for when unrolling a loop, besides the setup overhead mentioned above, is code bloat. You can actually make your unrolled code slower by causing it to be so big that it doesn't fit in the cache or kicks other important code or data out of the cache. Jumps aren't as expensive as they used to be, so amortizing the loop overhead isn't a big win since there's less overhead to amortize. Jumps on the Pentium, for example, are only a half cycle under the right circumstances. Cache misses are a lot more than a half cycle.

The gcc compiler supplies a command-line switch to force unrolling, so I used it in the row labeled "GNU gcc unroll." As you can see, it's not always faster than the gcc without unrolling.

Final Output

If you want to learn more about compiler technology, the bible is *Compilers: Principles, Techniques and Tools*, by Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman (Addison-Wesley 1986), affectionately called "The Dragon Book" because the cover illustration is a dragon bearing the words "Complexity of Compiler Design" being trounced by a knight with a sword that says, "LALR Parser Generator." I agree its goofy, but it's a classic.

You should also spend time on the web; there's tons of compiler information out there. For example, http://www.nullstone.com/htmls/category.htm has a series of 40 understandable examples of program transformations. For some aggressive compiler optimizations on a supercomputer compiler, check out http://www.astro.ku.dk/~aake/optimize/options.html. It covers optimizations that don't even preserve the original meaning of the code, which we didn't get into.

I would be remiss if I didn't mention that my friend Mike Phillip of Motorola actually managed to equal my final optimization using only compiler switches and Listing 1; you'll remember I mentioned Mike at the end of the last article. He ran the code through KAP *twice* with the no alias switch and then through the Motorola PowerPC compiler. It just goes to show you that knowledge of how something works (in this case Mike's knowledge of how the compiler worked) is always a good thing. ■

Don't believe the hype—Chris Hecker can use all the help he can get at checker@bix.com or gdmag@mfi.com.

Back

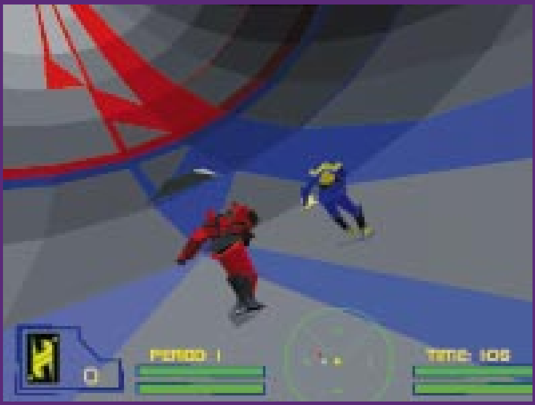
Bringing Life to HyperBlade

Game developers have rushed headlong into the third dimension producing increasingly realistic and immersive graphical environments. But, as everyone is quickly discovering, this additional dimension has presented more than an incremental challenge to overall game design. Meanwhile, players' expectations have soared as they continue to invest in newer and more powerful platforms. But what are those challenges and expectations? In this article, we'll discuss two elements of 3D game design that were addressed in the development of HyperBlade:

1. Fluid and responsive character behavior
2. Interactive cinematography.

As the two principals of WizBang! Software Productions (the creators and developers of HyperBlade), we have worked extensively in the military simulation and training systems industry and are intimate with virtual reality technology and the user experience it produces. As contributors to the SIMNET project (an R&D project for DARPA—Defense Advanced Research Projects Agency—that developed large scale distributed network simulations for warfare training), our software involved hundreds of participants engaged in war games on a common virtual battlefield. We observed soldiers training on our systems, spending hours cooped up in M1 tanks and other vehicle simulators playing out their roles as part of a larger military force, attacking or protecting a piece of rendered terrain. Palms sweating, soldiers eventually emerged from their fiberglass mockups, victors or victims in a simulated turf war that, after a time, became indistinguishable from reality.

As different as our game is from a battlefield simulation, it's that transcendent experience that we reached for in HyperBlade. What brought the experience to the brink of reality for the trainees was not the graphics—they were modest by today's standards. Knowing that the other vehicles were controlled by independent beings capable of thinking and reacting unpredictably is what made the experience so compelling and engaging. In fact, some of the forces were a mix of real soldiers and automatons.


HyperBlade, the game WizBang! developed for Activision, is a futuristic variation of roller hockey.

Populating HyperBlade with believable virtual characters—interspersed with real players in a network environment—was therefore a major goal. An equally important and related goal was to develop an architecture and authoring approach that would support many interactive objects (or "entities") without the accompanying geometric increase in complexity at the authoring level. Providing an authoring environment that would significantly reduce actual code development and be accessible to nonprogrammers was also a major requirement for our design.

Creating Character and Object Behavior

HyperBlade was built using a development system we call ADLIB (Authoring and Design Language for Interactive Behavior). With ADLIB, we could approach the game design from the bottom up as well as from the top down. That is, we were able to add characters and game objects incrementally, experiment with behaviors and interactions, and observe the overall effect on game play.

The ADLIB system has three parts:

1. A language for describing characters' behaviors and interactions
2. An authoring environment
3. A run-time system that executes these descriptions.

The descriptions of characters and their motivations are called Plans, and Plans are capable of controlling, with clarity and ease, virtually every aspect of a 3D virtual scene, including aspects

Stuart Rosen &
Robert Duisberg

When WizBang! needed a high-level authoring tool to give lifelike behavior to skaters in Activision's HyperBlade, it didn't shop around for a tool. It built it from scratch.

beyond character animation such as camera work, context-sensitive input device interpretation, adaptive music accompaniment, and placement and synchronization of 3D audio environments.

The ADLIB Plan language is a declarative, object-oriented language with a extensible vocabulary that can describe, at a very high level, the entities, circumstances, and interactions in the game—everything such as characters, environments, obstacles, rooms, fighting sequences, and animations (Listing 1 contains sample ADLIB code). A Plan consists of a collection of Behavior declarations and interactor declarations (which determine how a character reacts in various situations). So at its very simplest, ADLIB can define a typical finite-state-machine AI decision tree, where behaviors represent discrete states in the machine and the interactions represent discrete transitions. For example, if a game were to contain a vending machine, the initial behavior of that machine would be to simply wait for someone to insert a coin. The interaction of a coin insertion would cause a transition to another behavior which would tally the total value of the coins inserted. When the tally is sufficient to make a purchase, that interaction will cause a transition to a behavior that waits for the customer to make his selection, which interaction leads to a behavior to make changes and so on.

ADLIB, however, supports richer (and subtler) lifelike behavior than finite-state machines. ADLIB Behaviors are not discrete. Rather, they are composed of numerous simultaneous Activities, such as looking back to wave at a friend while riding a bicycle. A change in a Behavior is a merge of the Activities comprising the new Behavior. Behavior changes aren't necessarily discrete, but what the user sees may look more like a blending of Activities.

The decisions to make behavioral changes aren't necessarily discrete ether. An interaction such as a collision or a message can invoke a decision function to determine whether to change behaviors, and if so, which behavior to change to. These decision functions may consist of simple `if...then` statements or invoke neural networks or fuzzy logic. Although these technologies are considered slow and expensive, the decisions made in response to a particular interaction are typically very limited and specific. Fuzzy technologies can therefore be used without appreciably slowing down the game.

The Authoring Environment

ADLIB's authoring environment is a suite of tools that define, develop, and maintain databases of models, animations, and the vocabulary for manipulating and interacting with them. Three-dimensional models are imported in a standard format output from leading high-end modeling tools, such as Alias/Wavefront or Softimage, and are converted into a proprietary format honed for real-time performance. Another proprietary tool allows motion-captured animation segments or even just keyframes to be viewed, assembled, and seamlessly blended, with independent speed and directional control for each segment and transition. Right now, authors type in the text of the language in our production setting, but a goal is to provide "smart editing" which will be syntax- and template-driven and contain tools to test and check for plan completeness and consistency.

An important feature of ADLIB Plans is that behavioral specifications that one would normally consider procedural instructions (for example, "Pick up the ball" or "Move to the goal") have effectively been recast as declarative data. This means changes to the game don't require you to recompile your code. Rather, the p-code compiler is integrated into the game itself so that changes in Plans can be made and tested immediately, without quitting the game. This rapid turnaround makes our designers and developers more productive.

Many libraries of activities, behaviors, interactions, and decision criteria are being distilled out of this development experience. The goal is to provide libraries general enough to let even a hobbyist or non-programmer assemble a credible, interactive character. But, because serious developers need to code unique character behaviors at a low level, facilities are provided for defining new actions, decisions, and messages. The framework of templates to develop these extensions consists of structured script windows in a Hypercard-like authoring environment (with the significant distinction that these code elements run as compiled C++ rather than interpreted Hyperscript).

Achieving Performance

The ADLIB run-time system is designed for speed. Serious AI has a well-earned reputation for being slow, especially on a PC. However, events that require some computational reasoning in HyperBlade are localized and limited in scope so they have a negligible impact on frame rate. Other performance improvements are achieved via several stages of compilation:

1. The author's high-level text is compiled into a compact p-code representation.
2. This p-code description is used to construct a network of linked objects representing the current behaviors and interactions when a character adopts a given Plan.
3. Once a Plan has been instantiated, there is a minimum of overhead in switching and blending behaviors in response events.

All this is expressed in the Plans in a manner similar to the sample ADLIB code shown below. Interactions are perceived and processed in a running ADLIB system at rates of several dozen per second.

The fine granularity of interaction is driven by the idea that perception is the precursor to action. For example, if you hit another skater on the way to the goal, that is certainly an interaction. But so is the message that skater receives from the falling animation indicating that he or she has taken a spill in the thirty-fifth frame, which may cause that player to slide to a halt, incur damage, yelp, and message the team manager to consider a substitution.

The performance provided by the high levels of abstraction that ADLIB supports are of paramount importance in networked gaming where the bandwidth limitation for communicating complex behaviors and interactions is critical. Consider what it might require to transmit all the details of complex behavior over a network. With our ADLIB system, you send indices into duplicate plan instruction streams on either side of the network so behaviors are executed by the local run-time engine and yet remain completely in sync across the net.

We also see great potential for ADLIB in client/server network applications and web sites that wish to support interactive 3D. After downloading the compact p-code representation of a Plan quickly, a game can take advantage of local behavior libraries in response to the terse network packets which index interactions and behaviors in the Plan.

Listing 1. Sample ADLIB Code

```
DECLARE_BEHAVIOR Named: "GoForTheGoal" BehaviorType: "GlideToTarget"
SET_BEHAVIOR_PARAM Target: "TheGoal"
SET_INTERACTION Message: "YerHit"
            //Sent by the collision detection mechanism
RESPONSE
SetAnimation: "DoubleBackFlip"
            //which has a trigger set at the 35th frame
SET_INTERACTION Message: "YerDown"
            //Sent by the trigger in the previously set animation
RESPONSE
SetFriction: "High"a
SetUserControl: "Disabled"
PlaySoundType: "Yelp"
DecrementDamageBar: 20
SendMessage: "TeamManager" "ImDown" "MyDamageLevel"
ChangeToBehavior: "GetUp"
```

Interactive Cinematography

Producing a dynamic, "intelligent" camera to follow the action was a major challenge in the development of HyperBlade. The over-the-shoulder view we wanted as the primary viewing mode was particularly problematic. Contributing to this challenge were the ellipsoidal play surface, the players' speed combined with their turning and jumping capabilities, high speed collisions, plus the need to keep track of the Rok (game ball). In real sports, it's known that athletes rely heavily upon their peripheral vision, yet a computer screen provides a relatively narrow field of vision so orienting yourself and knowing where the action is on the field can be difficult. A variety of camera scanning modes were useful; they allow the player to look for other objects (such as teammates or the Rok) while keeping the first person character in view.

The cameras are defined as ADLIB objects and have their own sets of Plans and Behaviors. We could easily change the camera behavior based on certain situations and events during game play. It was easy to move the camera from your player in a passing situation, to follow the Rok in flight, and to ultimately assume a position behind the player receiving the pass.

Racing around the HyperBlade drome at high speeds could, as you might imagine, produce jolting or disorienting camera affects, especially with in the over-the-shoulder view. We tried to ameliorate this problem by tethering the camera to the player using various situationally adjusted, parametric models of bungees and springs. Nevertheless, with the complete freedom of motion available to the players, making it work for the large number of camera situations took months of tuning. During this tuning phase, the rapid turnaround time provided by ADLIB plans and the ability of non-programmers to modify camera plan parameters (changing spring constants and damping factors, for instance) proved to be critical.

An often heard lament in the industry is that Hollywood and the game development communities simply speak different languages and are concerned with utterly different domains. In other words, there's a communication gap. Storytellers often don't appreciate the engineering constraints technology imposes upon production, and developers often don't grasp aesthetic nuances central to artistic imperative. Yet the future dictates that the two worlds will learn to speak to each other and learn to work together. The ADLIB language and authoring systems aspire to address this need. Given today's high production costs of electronic entertainment, any language or technology which can significantly streamline the process of communicating content into working code should fill a real need and find itself in demand. ■

Stuart Rosen and Robert Duisberg are two principals of WizBang! Software Production Inc. who created the original Hyper-Blade game concept and developed it for Activision. WizBang! demoed their Hyper-Blade prototype to Activision at the Game Developers conference two years ago.

Stuart worked for Atari in the early 1980s as a program manager. He also worked in the military simulation industry for 10 years.

Robert Duisberg holds doctorates in Computer Science and Music Composition and is a research assistant professor in the School of Music at the University of Washington. He researches real-time gestural control and neural networks.

Back

Building a Scene Using Retained Mode Direct3D

Microsoft's new Direct3D package comes in two flavors—Immediate Mode and Retained Mode (RM). Immediate mode provides down-to-the-bone access to 3D accelerated graphics hardware and is ideal for developers who are tied to an existing 3D engine they want to port to Windows. Retained Mode, on the other hand, provides a higher level of services and allows application developers to create 3D scenes with varied lighting effects. The Retained Mode engine provides the rendering functions you need to create fast and efficient 3D Windows applications. This article will give you a brief outline of the Retained Mode engine and what you might do with it.

Direct3D Retained Mode is a complete 3D run-time rendering package that runs very fast on Windows 95 and will run on Windows NT in the near future. The Retained Mode (RM) interfaces allow you to create 3D Windows games that take full advantage of the hardware acceleration features of your target machine's graphics card. Figure 1 shows where the RM engine fits into the overall Direct3D architecture.

As you can see, the RM engine plays a similar role to Microsoft's OpenGL engine. The primary difference between using the RM engine and OpenGL is performance. OpenGL was designed for very general use; RM was designed for high-performance applications, such as games.

RM applications can run in either a window or in full-screen mode. If a game is run in a window, the current screen mode determines the pixel format. If the user has a machine set up to run in 256-color mode, then your game also runs in that mode. RM supports 8-, 16-, 24-, and 32-bits-per-pixel modes, so don't worry whether your game will run. If you elect to run the application in full-screen mode, you gain several advantages. In full-screen mode, you can choose almost any screen mode you like. If a game player is running in 1024-by-768-by-8 bpp when your game is launched, it can switch to 320-by-240-by-16 bpp if that's what you want. When the game is suspended or terminated, the screen returns to its original mode.

Your application can choose one of two lighting modes. If performance is your primary goal, then you can run in mono mode (sometimes called ramp mode). In mono mode, lights only have grey levels, which helps speed up the lighting computations. If you want higher fidelity in your application, you can use the RGB mode, which gives you full-color lights. You can also choose to dither the final rendered result for a slightly better look on 8 and 16 bpp displays at the expense of some performance.

Direct3D applications can be written in either C or C++. However, writing your application in C++ is slightly simpler because all the Direct3D interfaces are COM (Component Object Model) based. If you haven't used COM interfaces before, you'll have to do a little homework to make sure you understand how COM objects use reference counts to control their lifetimes and how to use the `QueryInterface` func-

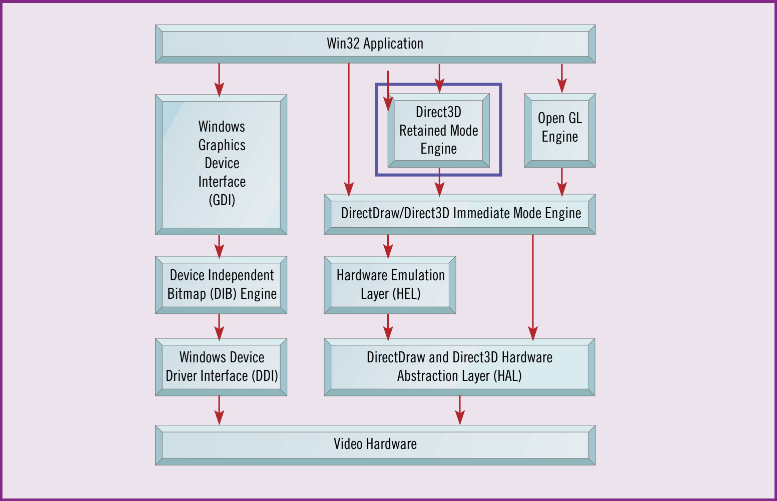

Win32 Application
Windows Graphics Device Interface (GDI)
Direct3D Retained Mode Engine
Open GL Engine
DirectDraw/Direct3D Immediate Mode Engine
Device Independent Bitmap (DIB) Engine
Hardware Emulation Layer (HEL)
Windows Device Driver Interface (DDI)
DirectDraw and Direct3D Hardware Abstraction Layer (HAL)
Video Hardware

Figure 1. The Direct3D Architecture. Note that the Retained Mode engine sits above the Immediate Mode engine and is therefore more abstracted.

Nigel Thompson

Microsoft's Direct3D comes in two flavors: Immediate Mode and Retained Mode. In his article, Nigel Thompson takes you through the development of a scene using the more abstracted Retained Mode.

tion to obtain an interface to a COM object. For my work, I created simple C++ classes that encapsulate the Direct3D COM objects and provide a more familiar environment. My C++ class wrappers handle all of the COM interfaces, reference counting issues, and so on—which leaves me free to get on with creating the application.

Creating a Retained Mode Application

Creating the initial scene for an RM application roughly consists of these steps:

1. Set up the window (although it may be full screen).
2. Create the viewport and camera.
3. Set the background color or image.
4. Set the ambient light level and add additional lights.
5. Add 3D shapes and sprites.
6. Set up motion parameters.
7. Update the scene regularly (such as when the application is idle).

Let's go through these steps and look at what each does. There isn't space to look at all the code required, so you'll need to read through the sample code in the Direct3D SDK (or the code from my book samples once it's published). I've included some code fragments from my own applications to give you a feel for what the application code might look like, but remember that the actual Direct3D interface calls are mostly hidden in the implementation of my C++ classes.

Window Setup

Even if you want your game to run in a full screen, you still need to create a window object. This prevents the Windows GDI from interfering with what you're doing to the display. Once you have a window, you can create the RM engine interface. This can be done several ways depending on how much control you want over what gets created for the RM engine to use. Whichever method you use, you end up with a front buffer surface, a back buffer surface, and a Z buffer surface. If you're running on an 8 bpp display, then you also get a palette object. If the application is running in a window (rather than full-screen), then you also get a clipper object which prevents the RM engine from drawing outside the window area.

You also need to choose the lighting model (RGB or mono) and how you want solid objects filled and shaded. The RM engine supports wire frames and solid-filled objects. Fills can be done in a variety of ways ranging from flat-unlit solid color to Gouraud shading (a later version will also support Phong shading). Most applications will probably select the mono lighting model with Gouraud shading. This combination gives you the best performance with a good degree of realism.

There is actually quite a lot of code required to set up the initial configuration. In the Direct3D SDK, the samples directory provides this code. For my own work, I created a C++ class (`C3dWnd`) that lets me easily create a 3D scene in a window. Here's a part of my startup code:

```
// Create the 3D window
if (!m_wnd3d.Create(this, IDC_3DWND)) {
        return -1;
}
```

OK, so this oversimplifies things significantly, but I'm sure you get the point—once you have written the startup code, you can just use it and forget it.

Once the various surfaces, palettes, and so on have been created, you can forget about them because you don't access them directly. You make requests to the RM engine, and it uses the surfaces to draw your scene.

Creating the Viewport and Camera

The viewport and camera components determine how your scene looks from the viewer's point of view. You can use either a simple orthographic projection mode or the more common perspective projection mode. You can alter the effective camera angle, the camera position, and direction to get whatever view you need. You can move the camera in a scene like any other object, so it's very easy to fly through a scene by simply moving the camera along a path. Figure 2 shows the viewing frustum that determines what is visible in a scene.

The camera angle determines where the edges of the viewing frustum (truncated pyramid) are. The front- and back-clipping planes truncate the pyramid. The area inside the truncated pyramid determines which objects will be visible in the scene. Given a pointer to the viewport interface, altering viewport parameters can be as simple as this:

```
m_pIViewport->SetBack(rvBack);
m_pIViewport->SetFront(rvFront);
m_pIViewport->SetField(rvField); // cam-
era field of view
```

It's tempting to set the back-clipping plane at some huge value and the front-clipping plane directly in front of the camera position. If you do this, you'll find you occasionally get weird rendering artifacts, because setting the front-clipping plane right next to the camera means all the Z-buffer math occurs in a very small numerical range, which can introduce math errors as the visibility of object surfaces are being determined. Keep the clipping planes close to your objects to give a dynamic range for the math. If you need to do a flying scene where objects can get very close to the camera, move the front-clipping plane dynamically so that it stays just in front of your front-most object.

The RM engine uses a left-handed coordinate set with Y up, X to the right, and Z into the screen (away from the viewer). If you have a passion for a right-handed coordinate set, you can apply a simple transform to your right-handed coordinates to make them work in RM's left-handed world.

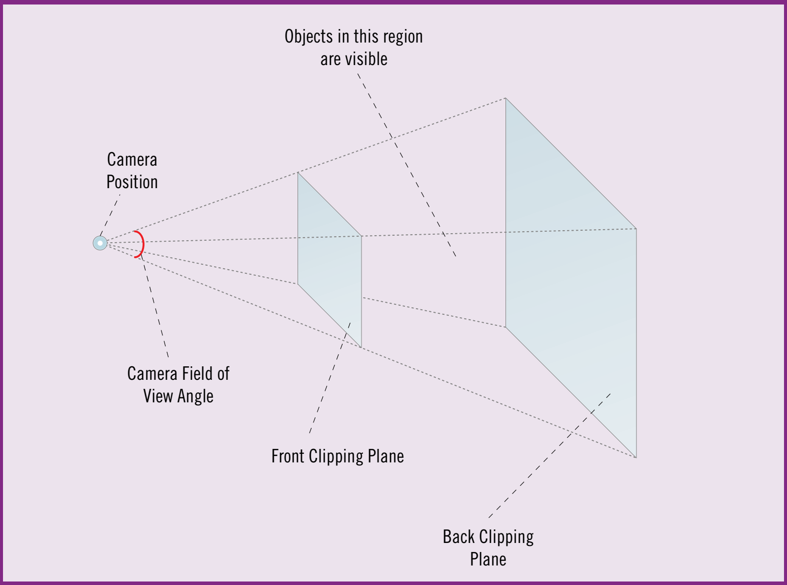

Objects in this region are visible
Camera Position
Camera Field of View Angle
Front Clipping Plane
Back Clipping Plane

Figure 2. The Viewing Frustrum determines what can be seen.

Setting the Background

Backgrounds can consist of a simple flat color or a texture map. If you use a texture map you need to be aware of two things. First, the texture map will be stretched to fit the shape of your window, so the aspect ratio might not be what you intended if you let the user resize the window to an arbitrary shape. Second, texture maps can really eat up palette entries on an 8 bpp system, so texture maps are generally restricted to a limited number of colors and shades of those colors. You might need to experiment with how you create your backgrounds and how many shades you use in a scene to get the best compromise between your background and your other scene objects.

I created a C++ class to handle all the image functions I needed so that adding a background to a scene can be as simple as:

```
m_imgBkgnd.Load(IDB_BKGND);
m_pScene->SetBackground(&m_imgBkgnd);
```

In this case, the background image is a Windows bitmap attached to the application as a resource.

Setting the Lighting

Each scene has an ambient light level and as many additional lights as you choose. Each light you add slows down the scene; some light types are expensive, so you need to trade image fidelity for performance again. The RM engine supports lights that vary from a simple infinite distance source (like the sun) to spotlights that have a bright center cone and a dim outer cone. You can even set the attenuation characteristics of some lights—but, of course, more complexity in the math means slower rendering.

You need to add at least a directional light (which has the lowest cost-to-performance ratio) to your scene to

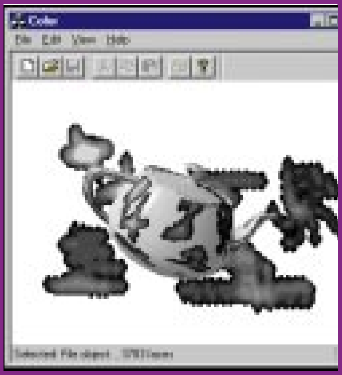


Figure 3. An example of an application that uses a chrome wrap to simulate reflections from an object.

get any 3D effect. Using only an ambient light produces a very flat scene. Remember that colored lights require the RGB lighting model, whereas the mono lighting model only uses the brightness of each light, not its actual color. My applications generally use the mono lighting model, and I use one directional light source in addition to the ambient lighting. Since lights are regular objects in the 3D scene just like any other object, they can have position and direction—although the position or direction of some light types doesn't affect how they illuminate a scene. Directional lights, for example, have no position—just a direction. The RM engine does not generate shadows, but a mechanism to simulate shadows using

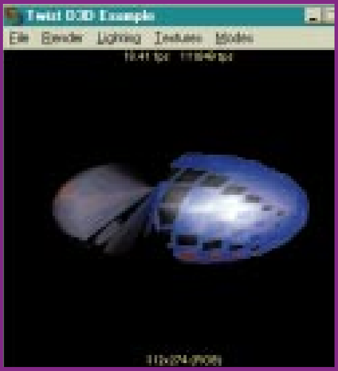


Figure 4. The DirectX II SDK comes with a Direct3D example called Twist, which lets you tweak rendering options in real time and see the impact on frame rate.

flat dark objects is supported for special cases.

Adding Shapes

The Direct3D RM engine does not create 3D shapes. You create shapes either by using a tool like AutoDesk's 3D Studio and converting them to the Direct3D file format (the SDK has a conversion tool for 3D Studio shapes) or by creating your own meshes from a list of vertices. If you create your own shapes, you can control exactly how they look by applying color and texture maps to individual faces or to the entire object. You can determine how a single texture map is applied to an object by using different wraps. The RM engine supports flat, cylindrical, spherical, and chrome wraps. Chrome wraps give the illusion of reflections from an object's surfaces.

To define the mesh for an object, you typically create two or three lists. The first list contains all the vertices of the object in model coordinates. So, for a cube, you'd typically have 8 vertices. (I say "typically" because there are a few cases where you need to duplicate the vertices in an object because of the way texture maps are used.) The second list describes the faces of the object in terms of its vertices. All faces must be listed with the vertices in clockwise order. The third list is optional and consists of a set of normals to be used with the vertices. These normal vectors control how the object will be shaded. The RM engine includes a function to generate a set of normals from the face descriptions if you just want a smooth surface.

In practice, a 3D object consists of at least two parts: a mesh and a frame. The mesh is the set of vertices, texture maps, and so on that describe what it looks like. The frame is a mathematical transform that determines how the object will be placed in the scene. The RM engine allows frames to have child frames so you can set up a hierarchy of frames to describe complex object systems. For example, a tank can have a frame to describe the overall object's position and direction with child frames that describe the position and direction of it's turret and gun. In this way, you can control the angle of the gun and rotation of the turret relative to the body of the tank. If you alter the tank's frame to move the tank, then you retain the relative positioning of the gun and turret.

Setting Motion Parameters

The RM engine supports some simple motion capabilities that can be applied to any frame. The frame could describe the position and orientation of an entire object or group of objects, or the frame might just describe the position of a wheel on a car. Individual frames can be set to rotate about a given axis and can have a fixed linear velocity along a given axis. These simple parameters can provide some elementary animation of object parts, such as spinning a radar antenna or orbiting moons around a planet.

You can achieve more complex movement in a variety of ways. The RM engine supports animation sequences using key points to determine position and orientation. When the key points have been set, the object can be moved to any point interpolated along the path defined by the key points. The path can be set up to loop so the object follows the path continuously.

You can, of course, move an object yourself using the frame interface's `SetPosition` and `SetOrientation` functions before the next scene is rendered. This gives you complete flexibility in how your object moves.

Updating the Scene

In many games, you'll want to update the scene as often as possible to give the illusion of continuous movement. The RM engine provides interface functions to apply all the current motion parameters to compute the next position for each object and then to render the next scene state to the back buffer. Your code then flips the back and front buffers to show the next scene and starts work on the next one.

If you're running in a window, then you'll most likely use the built-in `Blt` function to copy the contents of the back buffer to the front buffer. In full-screen mode you'll probably use the `Flip` function to swap the back and front buffers for the fastest possible screen update.

Using 2D Sprites

The RM engine supports texture maps with transparent areas for use as 2D sprites (or "decals," as the RM documentation calls them). Sprites can be a very efficient way to create game characters in a 3D scene. They are much faster to render than 3D mesh objects and have the advantage of always appearing to face the viewer—just right for those bad guys.

You can combine all the phases (different views) of a sprite into one image strip and then use just a part of the image strip as the active image. This lets you load the entire set of sprite images in one go and avoids having lots of redundant headers wasting memory. The RM engine tries to load your texture maps into video memory for the best possible performance.

Like 3D objects, sprites can be moved around in a scene, so you can mix sprites and 3D mesh objects to give the best combination of performance and realism.

Interacting with the User

Given a screen position, The RM engine allows you to obtain a list of 3D objects that lie directly behind that position. The list is sorted in Z order so you can find the object closest to the viewer, which gives you a simple hit testing mechanism. Once you have determined which object has been hit, you can go on to find which face of the object was hit and the coordinates of the point on the object where the hit occurred. This might be overkill in your average shoot-'em-up game, but it can be used to implement sculpting or 3D surface painting applications.

Selecting and moving objects is something you need to do entirely for yourself. The engine provides only the hit detection mechanism—it's up to you to determine how your objects react to a user's attempts to drag them in a scene.

Microsoft's Direct3D Retained Mode interfaces provide you with a way to implement a fast and efficient 3D application for the Windows environment. By allowing you to run full-screen, it gives you complete control over screen resolution and pixel format while retaining excellent portability. Now you have no excuse not to port that 3D game to Windows! ■

Nigel joined the multimedia group at Microsoft in 1989 where he led the team that created the Multimedia Extensions for Windows. He went on to port the extensions to NT and spent several years writing for the Developer Network group under the name Herman Rodent. Nigel published his first book Animation Techniques in Win32 *in 1995. He has just completed his second book for Microsoft Press, which is entitled* 3D Programming in Windows 95.

Back

A Portrait of DirectDraw

DirectDraw is the component of the Windows 95 Game SDK that bypasses the Windows Graphics Device Interface (GDI) and gives you direct access to the video card and video memory using a well-defined set of methods. In this article, I'll to try to hit all the important areas of DirectDraw and, more importantly, show some code samples.

DirectDraw provides a number of benefits to game developers. It allows you to change the resolution and color depth of the display on the fly, draw directly to the screen, copy from off-screen DirectDraw surfaces to the display, and page flip an off-screen surface to the display. Finally, DirectDraw takes advantage of hardware-transparent blitters to provide fast transparent bit-blitting with source and destination color keys.

Here is the typical sequence of events you step through using DirectDraw:

1. Acquire exclusive screen access.
2. Set up the desired display mode.
3. Get access to some working video RAM and the display RAM.
4. Set up any desired color keys.
5. Draw on the working video RAM.
6. Move it to the display with a bitblit or a page flip.

Creating a DirectDraw Object

If you're proficient with Microsoft's Component Object Model (COM), you know that you normally use a COM API and a COM `CLSID` to create a COM object. However, to create the DirectDraw object you ought to use the `DirectDrawCreate()` function, because it wraps `CoCreateInstance()`. If you're not proficient with COM and don't know what a CLSID is, don't worry: the developers of DirectDraw have provided APIs to do the work for you and return the correct object pointers.

The `DirectDrawCreate()` function takes a pointer to a DirectDraw object pointer. If successful, it sets the pointer to the newly created DirectDraw object and returns `DD_OK`. Listing 1 shows a typical sequence of calls used to create a DirectDraw object. Once we've got a pointer to a DirectDraw object, we can use it to create DirectDraw surfaces, clippers, and palettes.

Determining Capabilities and Setting the Display

Once you have created the DirectDraw object, it's a good idea to check the capabilities of the DirectDraw display driver, the hardware, and capabilities of the DirectDraw Hardware Emulation Layer (HEL). To do this, you use the `IDirectDraw::GetCaps()` interface, which returns the capabilities in two `DDCAPS` structures.

Once you've verified that the hardware or emulation layer can meet your application's requirements, you should take exclusive control of the display, especially if your application is a full screen, non-Windows GUI game. To do this, you use the `IDirectDraw::SetCooperativeLevel()` interface and pass the desired cooperative level in the third parameter, `dwFlags`.

Four flags correspond to the various display options that DirectDraw sup-

Figure 1. The Fox and Bear demo is an example of DirectDraw performance. This benchmark, which is included with the DirectX II SDK, displays frame rates in various screen modes.

ports. First, the DDSCL_NORMAL flag indicates that the application wants to share the display with other applications and operate as a normal windows application. Second, the DDSCL_EXCLUSIVE flag indicates that the application is requesting exclusive control of the display screen. If DirectDraw has already granted exclusive control of the screen to another application, this call will fail with an error code DDERR_EXCLUSIVEMODEALREADYSET. The third flag, DDSCL_FULLSCREEN, indicates that the application will be drawing the entire screen, so that GDI may be safely ignored by the DirectDraw display driver. (Note: the DDSCL_FULLSCREEN flag requires the DDSCL_EXCLUSIVE flag.) Finally, the DDSCL_ALLOWMODEX requests that the application be allowed to set mode X screen resolutions of 320-by-240-by-8 and 320-by-200-by-8.

Once the cooperative level has been set, check the current display mode using the IDirectDraw::GetDisplayMode() interface. This function takes a pointer to a DDSURFACEDESC structure and changes the appropriate members to the preferred screen resolution and pixel bit depth.

If the current display mode isn't correct, you can check the display modes that the display adapter and its DirectDraw driver support using the IDirectDraw::EnumDisplayModes() interface. This function takes a pointer to a DDSURFACEDESC structure which describes the types of surfaces to enumerate. Pass a NULL for this parameter to enumerate the supported display modes. Once you've verified that your preferred display mode is supported, use the IDirectDraw::SetDisplayMode() function to change the current display mode. This function takes a width, height, and pixel depth in bits per pixel.

With the display set in the correct mode, use the IDirectDraw::CreateSurface(), CreatePalette(), and CreateClipper() functions to create some DirectDraw objects. The CreateSurface()

Listing 1. Creating the DirectDraw Object

```
LPDIRECTDRAW         pDirectDraw;
HRESULT    hResult;
hResult = DirectDrawCreate( NULL, &pDirectDraw, NULL );
if ( FAILED( hResult ) )
         return FALSE;
DWORD dwFlags = DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN | DDSCL_ALLOWMODEX;
if ( bWindowed )
         dwFlags = DDSCL_NORMAL;
hResult = pDirectDraw->SetCooperativeLevel( hWndMain, dwFlags );
if ( FAILED( hResult ) )
         return FALSE;
if ( !bWindowed )
{
         hResult = pDirectDraw->SetDisplayMode( nWidth, nHeight, 8 );
         if ( FAILED( hResult ) )
         return FALSE;
}
```

Bob Provencher

DirectDraw lets you access video cards and video memory. Using it, you can speed up animation, change color depth, and alter resolution on the fly.

interface gives access to the primary surface and creates working off-screen surfaces.

Before we go any further, I need give you a quick definition two terms. The term, "surface," is a DirectDraw term for an area of memory that represents a drawing area. A surface can be the current video display, or it can be an area of off-screen memory. A color key is simply a color that you wish to treat as transparent when bitblitting.

DirectDrawSurface Object

Let's begin working with DirectDraw surfaces. We first need to define the DDSURFACEDESC structure, which describes a DirectDraw surface. Listing 2 is the C definition of DDSURFACEDESC. The definition shows the flags that you commonly use to create the surface, such as dwSize, dwFlags, dwHeight, and dwWidth. I recommend checking the flags field to see if the field you are interested in is valid. During output, when examining a surface description, you should check the flags field to see if the field you are interested in is valid.

To create a DirectDrawSurface object you use the IDirectDraw::CreateSurface() interface, which takes three parameters. The first is a pointer to the DDSURFACEDESC structure which describes the surface. The second is a pointer to a DirectDrawSurface object, which is set to the newly created surface if the function succeeds. The final parameter is to support future COM aggregation features. For our purposes we will pass NULL for this one.

You use IDirectDraw::CreateSurface() to access to the primary surface (which represents the visible display screen), to create off-screen surfaces (for work areas or to hold sprites), or to create a group of surfaces called complex structures. An example of a complex structure is the complex flipping structure, which you create in order to page flip using DirectDraw.

Listing 3 shows IDirectDraw::CreateSurface accessing the primary surface and creating off-screen surfaces. In the first section, a DirectDrawSurface object is created. First we will fill in the size field of the DDSURFACEDESC structure, to tell DirectDraw what version of DirectDraw our application was written for. Then we set the DDSD_CAPS bit to tell DirectDraw to check the ddsCaps field of the structure. You don't need to set any other bits, because the primary surface already has a height and width, and we can't change it. Finally we set the capabilities flag to indicate that we are interested in creating a primary surface.

When we pass the address of the surface description structure and the DirectDrawSurface object to IDirectDraw::CreateSurface(), the function will create a new surface for us and set the surface pointer to the new surface if successful. This function returns the standard COM HRESULT return code.

The second part of Listing 3 shows how to create an off-screen surface. For this surface, you must set the DDSD_HEIGHT and DDSD_WIDTH flags to tell DirectDraw to check those fields to determine the size of the surface you are creating. For this surface, you also set the DDSD_CAPS bit and set the ddsCaps.dwCaps field to DDSCAPS_OFFSCREENPLAIN. Again, if successful, the function sets the surface pointer for you.

If you want to do page flipping in your application, instead of creating the various surfaces separately, you can create them together in a complex flipping structures as seen in Listing 4. Note that DDSD_CAPS and DDSD_BACKBUFFERCOUNT are set to indicate which fields in the DDSURFACEDESC structure are valid—that is, those fields you'll be filling in. You don't set the height and width fields because the height and width of the various surfaces in a flipping structure are the same size as the screen. In the capabilities field, you set DDSCAPS_COMPLEX to indicate that you are creating a structure of surfaces. Set DDSCAPS_FLIP to indicate the type of complex structure you're creating, one that you'll use for page flipping.

Finally, you tell dwBackBufferCount the number of back buffers you will need in the flipping structure. I have specified two back buffers in this example so that I can write on the next back buffer, even during a flip (when the front buffer and the previous back buffer are busy).

Having filled out the structure, you call the CreateSurface function and check the return code. You then call the IDirectDrawSurface::EnumAttachedSurfaces() interface to get pointers to the back buffers so you can draw on them.

Once a surface is created, it's a good idea to check the surface capabilities by calling the IDirectDrawSurface::GetCaps() interface. This function simply fills in a DDSCAPS structure with the capability bits of a surface. Another good idea is to get a full description of the surface by calling the IDirectDrawSurface::GetSurfaceDesc() interface. This function fills a passed in DDSURFACEDESC structure with the description of the surface object through which the interface was called.

Listing 2. The DDSURFACEDESC Structure typedef struct _DDSURFACEDESC

```
{
    DWORD           dwSize;             // size of the structure for version checking
    DWORD           dwFlags;            // what fields are in use?
    DWORD           dwHeight;           // height of the corresponding surface
    DWORD           dwWidth;            // width of the surface
    LONG            lPitch;             // bytes between scan lines
    DWORD           dwBackBufferCount;  // number of back buffers attached
    DWORD           dwZBufferBitDepth;  // pixel bit depth for Z buffer
    DWORD           dwAlphaBitDepth;    // pixel bit depth for alpha channel
    DWORD           dwReserved;         // reserved
    LPVOID          lpSurface;          // pointer to the surface memory when locked
    DDCOLORKEY      ddckCKDestOverlay;  // color key for destination overlays
    DDCOLORKEY      ddckCKDestBlt;      // color key for destination blits
    DDCOLORKEY      ddckCKSrcOverlay;   // color key for source overlays
    DDCOLORKEY      ddckCKSrcBlt;       // color key for source blits
    DDPIXELFORMAT   ddpfPixelFormat;    // format of the surfaces pixels
    DDSCAPS         ddsCaps;            // capabilities of the surface
} DDSURFACEDESC;
```

If you're not going to be using full-screen exclusive mode DirectDraw, you'll probably need to attach a DirectDrawClipper object to your surface. I won't describe the creation of this object in this article, but be aware that you can create DirectDrawClipper objects and attach them to surfaces to do clipping for you.

If your DirectDraw application expects a pixel depth of 8 bits, you're going to need a DirectDrawPalette object, which describes a table of colors that your application uses. You create a DirectDrawPalette object with the IDirectDraw::CreatePalette() interface, and attach it to a surface with the IDirectDrawSurface::SetPalette() function. Usually, you will only have to attach a palette to the primary surface.

Another useful IDirectDrawSurface interface is SetColorKey(). As you might expect, the SetColorKey() interface defines a color key and the type of color keying for a surface. This interface accepts two parameters. The first is a bit field that represents the types of color keying that the color key will be used for. The valid flags are DDCKEY_COLORSPACE, DDCKEY_DESTBLT, and DDCKEY_SRCBLT. The first flag indicates that the color key specifies more than one color, or a color space. The second and third flags indicate whether the color key should be used for destination or source blitting.

Lost Surfaces

If the display mode changes or another application acquires exclusive cooperative level, it is possible for all surfaces to lose their display memory. The surface object still exists, but the memory that it used was reorganized and returned to the system. When this happens, it's up to you to call the IDirectDrawSurface::Restore() interface and to reconstruct what the surface contained before it was lost.

When a surface is lost, all subsequent interface calls through that surface will return DDERR_SURFACELOST. This can make error checking quite complex, requiring you to repeatedly check for this error after every operation and attempt to restore surfaces if they become lost.

Fortunately, there is a way to detect a lost surface before you attempt to use it, using the IDirectDrawSurface::IsLost() interface. This function returns DD_OK if the surface is not lost, or DDERR_SURFACELOST if the surface has become lost.

Another set of useful functions when working with surfaces are the IDirectDrawSurface::GetDC() and ReleaseDC() interfaces. The GetDC() interface returns a GDI-compatible handle to a device context, or HDC, for the surface. With this HDC, you can write to the surface using a GDI function or a TrueType font, or load your bitmap sprites to off-screen surfaces via the Windows SetDIBits() API. The GetDC() function does an implicit lock, so unless you've created the surface using the DDSCAPS_OWNDC capability, you probably shouldn't leave it locked for an extended period.

Listing 5 shows a sample application using the GetDC() and ReleaseDC() calls on a surface. In this example, we use the GetDC() function to obtain an HDC for the surface. Then, we use the GDI functions PatBlt() and TextOut() to white out and put some text on the HDC and the working text surface, and then blit the surface to the display.

The next pair of useful IDirectDrawSurface interfaces are the Lock() and Unlock() pair. Lock() obtains a pointer to a surface's physical memory so that you can directly draw either on a rectangular portion or the entire surface. The Lock() interface takes the following parameters: a pointer to a Windows RECT structure which specifies the portion of the surface to lock, the address of the DDSURFACEDESC structure through which the physical memory pointer is returned, a flags field, and an optional event handle to trigger when the Lock() has been obtained.

Busy Surfaces

A DirectDraw surface can be busy for a number of reasons. For example, suppose the hardware is capable of an asynchronous bitblit operation. After a bitblit is requested, the function immediately returns and does not wait for the blit to be completed. Until the blit is completed on this surface, subsequent operations will return the error DDERR_SURFACEBUSY. The primary surface will also be busy during the vertical retrace blank and during a page flip operation.

Fortunately, there are several ways to wait for a Lock() when a surface is busy. The first way is to pass the DDLOCK_WAIT flag, which tells DirectDraw to return from the Lock() call when the surface is finally available. The second method uses the DDLOCK_EVENT flag. DDLOCK_EVENT instructs DirectDraw to signal an event when the Lock() has been obtained, using the WaitForSingleEvent() API.

The Unlock() function, which unlocks a locked surface, takes one parameter: the memory address of the surface that was returned when the lock

Listing 3. Creating Primary and Off-Screen Surfaces

```
LPDIRECTDRAWSURFACE      pPrimarySurface;
LPDIRECTDRAWSURFACE      pOffSurface;
DDSURFDESC               ddPrimSurfDesc;
DDSURFDESC               ddOffSurfDesc;
//  Get access to the primary surface
ddPrimSurfDesc.dwSize                    = sizeof( DDSURFACEDESC );
ddPrimSurfDesc.dwFlags                   = DDSD_CAPS;
ddPrimSurfDesc.ddsCaps.dwCaps            = DDSCAPS_PRIMARYSURFACE;
hResult = pDirectDraw->CreateSurface( &ddPrimSurfDesc, &pPrimarySurface, NULL );
if ( FAILED( hResult ) )
          return FALSE;
//  create an off-screen surface
ddOffSurfDesc.dwSize                     = sizeof( DDSURFACEDESC );
ddOffSurfDesc.dwFlags                    = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddOffSurfDesc.ddsCaps.dwCaps             = DDSCAPS_OFFSCREENPLAIN;
ddOffSurfDesc.dwWidth                    = nWidth;
ddOffSurfDesc.dwHeight                   = nHeight;
hResult = pDirectDraw->CreateSurface( &ddOffSurfDesc, &pOffSurface, NULL );
if ( FAILED( hResult ) )
          return FALSE;
```

Listing 4. Creating a Complex Flipping Structure

```
ddFlipSurfDesc.dwSize                           = sizeof( DDSURFACEDESC );
ddFlipSurfDesc.dwFlags                          = DDSD_CAPS |  DDSD_BACKBUFFERCOUNT;
ddFlipSurfDesc.ddsCaps.dwCaps                   = DDSCAPS_COMPLEX | DDSCAPS_FLIP | DDSCAPS_PRIMARYSURFACE;
ddFlipSurfDesc.dwBackBufferCount                = 2;
hResult = pDirectDraw->CreateSurface( &ddFlipSurfDesc, &pFlipSurface, NULL );
if ( FAILED( hResult ) )
        return FALSE;
hResult = pFlipSurface->EnumAttachedSurfaces( NULL, EnumAttachedSurfacesCallback );
if ( FAILED( hResult ) )
        return FALSE;
```

occurred. This parameter is required since you can independently lock any number of non-overlapping rectangles on a surface.

The final set of calls that we'll use with DirectDraw surfaces are `Blt()`, `BltFast()`, and `Flip()`. Both `Blt()` and `BltFast()` move rectangular areas from one surface to another. `Blt()` accepts both source and destination rectangles and can do arbitrary stretching and shrinking. `Blt()` also accepts some flags as parameters and an optional `DDBLTFX` structure. `BltFast()` is a streamlined version of `Blt()`. It always attempts an asynchronous `Blt()`, and it does not do any stretching, shrinking, or clipping.

Finally, `Flip()` instructs the hardware to perform a page flip on a complex flipping structure. `Flip()` accepts two parameters. The first specifies which alternate buffer will become the new front buffer, and the second is a flags field.

Flipping operations are exceedingly simple to implement with DirectDraw. Once the complex flipping structure has been created, you only need to maintain a pointer to the first back buffer and the root surface. When the `Flip()` is performed on the root surface, DirectDraw automatically shifts the memory around so that the first back buffer `DirectDrawSurface` object points to the new first back buffer. In this way, the first back buffer pointer is always the next surface to be displayed. Listing 6 shows a sample application of `Blt()`, `BltFast()`, and `Flip()`.

In this example, a text surface is copied to the background working surface. Then, depending on whether the application has been configured for page flipping or not, the back buffer is blitted or flipped to the front.

Terminating an Application

You've created your surfaces, palettes, and clippers, your application has run, and now it's time to terminate. If you've been page flipping, use the `IDirectDraw::FlipToGDISurface()` interface to ensure that the GDI surface is the visible surface when your application terminates. If you've changed the display mode, use the `IDirectDraw::RestoreDisplayMode()` interface to restore it (don't use the `SetDisplayMode()` function to restore a display mode that you've saved away yourself). Finally, use the `IUnknown::Release()` function to decrease the reference count of all the DirectDraw objects you've created. Once the reference count reaches zero, COM will destroy the objects and delete all of the memory associated with them.

That covers the overall framework of a typical DirectDraw application. You now know enough to create a working DirectDraw application. I've written a sample application, TILETEST, that is available on the *Game Developer* web site and uses the various functions I've described. TILETEST differs slightly in the code samples shown above, in that I've wrapped the various DirectDraw COM objects into some C++ classes. TILETEST also shows some techniques for DirectDraw programming that I did not cover in this article, particularly debugging helpers and how to draw to a window with DirectDraw. The point of the sample is to show how to use DirectDraw; therefore, it isn't particularly speedy. If you're so inclined, the TILETEST `TileFloor()` function can be optimized by doing the drawing manually, and possibly by using RLE decoding of the DIBs to the surface. But I'll leave that as an exercise for you. ■

Bob Provencher is one of the principles of Cog Interactive, a start-up game development company dedicated to developing real-time action and strategy games for Windows 95 and the Internet. He can be reached via e-mail at mail18080@pop.net.

Listing 5. GetDC() and ReleaseDC()

```
char*s = "Hello World.";
HRESULT hResult;
HDC hDC;
hResult = pTextSurface->GetDC( &hDC );
if ( FAILED( hResult ) )
        return FALSE;
PatBlt( hDC, 0, 0, TEXT_WIDTH, TEXT_HEIGHT, WHITENESS );
TextOut( hDC, 0, 0, s, strlen( s ) );
hResult = pTextSurface->ReleaseDC( hDC );
if ( FAILED( hResult ) )
        return FALSE;
```

Listing 6. Blt(), BltFast() and Flip()

```
hResult = pBackSurface->BltFast( 0, 0, pTextSurface, NULL,  DDBLTFAST_WAIT | DDBLT-
FAST_SRCCOLORKEY );
if ( FAILED( hResult ) )
   return FALSE;
if ( bPageFlip )
{
   hResult = pFlipSurface->Flip( NULL, DDFLIP_WAIT );
}
else
{
   hResult = pFrontSurface->Blt( &rcDest, pBackSurface,  &rcSrc, DDBLT_WAIT, NULL );
   }
if ( FAILED( hResult ) )
   return FALSE;
```

Back

Optimizing CD-ROM Performance under DOS/4GW

You're building the coolest, most realistic game with a half hour of gorgeous full-screen video and a phenomenal soundtrack featuring famous voices. You're using the Watcom compiler, with DOS/4GW, because you want the game to run on both DOS and Windows machines. There's just one problem: the video segments don't play smoothly on anything less than a 6X CD-ROM drive.

Let's look at some CD-ROM quirks you should know about, no matter which PC platform you're targeting, and how to read a CD-ROM drive efficiently from a DOS/4GW program. Then we'll discuss two games that exemplify intelligent use of a double-speed (2X) CD-ROM drive: Rebel Assault II: The Hidden Empire from LucasArts Entertainment, and Loadstar: The Legend of Tully Bodine from Rocket Science.

CD-ROM Fundamentals

In most games that read data off a CD-ROM in a continuous stream, the CD-ROM drive's data transfer rate will be the biggest bottleneck.

Today, single-speed drives are obsolete, and double-speed (2X) drives are the norm. A 2X drive, at best, can read 300K of data per second; a quad-speed (4X) drive, 600K. Considering that movie-quality video runs at 30 frames per second and 30 frames of uncompressed, 640-by-480 pixel, 16-bits-of-color video plus one second of CD-quality audio require 18,176K per second, today's CD-ROM drives are woefully inadequate for movie playback. To beat the bandwidth problem, you'll have to cut a few corners.

To reduce your game's data rate to sustainable levels, you must sacrifice some quality. For example:

- Compress the data.
- Play fewer video frames per second.
- Reduce the displayed resolution (perhaps to 320-by-240 pixels or fewer).
- Use only 8 bits of color per pixel.
- Use lower-quality audio.

Some games copy data to the hard disk, which is capable of a higher (but still limiting) transfer rate. This technique works well for small amounts of frequently accessed data, but it increases the game's hard-disk space requirements and slows down the installation.

Even if your data rate is sustainably low, you must ensure that it's steady. Otherwise, your video will skip, and your sound will stutter. If you want to display 15 frames of video per second, you either have to read and process each frame in less than $^1/_{15}$ of a second or use a buffer to smooth out spikes in the data rate.

If you read non-sequentially from the disc, the data rate will drop to zero whenever the laser moves to a new location. In designing your game, you must somehow work around this problem. It's possible to cover up the seeks by playing previously buffered sound and video, but you could also display a static screen (text, for example) to distract the player.

CPU and the MPC Spec

You can guess the data rate of a drive just by knowing if it's rated 2X, 4X, or 6X. Most 2X drives can deliver 300 ± 25K, most 4X drives can deliver 600 ± 50K, and so on. However, a few drives are optimized for a particular access pattern: they live up to their ratings when you use DOS `copy` on them, but fail miserably under the more demanding conditions of gameplay. We've also seen drives that didn't live up to their rating because they (or their controller cards) were defective.

Data rate isn't the only consideration in CD-ROM programming. After reading the data, you need enough CPU cycles left over to decompress it, display the video, play the audio, and execute your game's logic. Unfortunately, the percentage of CPU cycles consumed varies among CD-ROM models, largely due to the efficiency of the device drivers. Some move the data from the drive's hardware buffer to system memory using programmed I/O, while others use direct memory access (DMA). Some drivers transfer data a byte at a time, while others wait for an interrupt that signals a full sector is ready to be transferred.

Since DOS is inherently single-threaded, drivers monopolize the CPU even when they are simply waiting for a hardware event. Both the "waiting" algorithm and the "transfer" algorithm affect how the CD-ROM will respond to your efforts to reclaim spare CPU time.

A computer that meets the MPC II specification is guaranteed to deliver 150K with at most a 40% CPU load, while an MPC III computer (running DOS) can deliver 550K at 40%. But 150K is too slow for most applications, and MPC III systems are not yet common. Further, a drive's rated CPU load is a laboratory average. If you take over the CPU at times which are inconvenient for the CD-ROM driver, your mileage may vary. Thus, for most uses, the MPC

Dan Teven
and Vincent Lee

It pays to know how to read a drive efficiently from a DOS/4GW program. Rebal Assault II and Loadstar show us how to intelligently use a double-speed CD-ROM drive.

specification serves only as a vague indicator of probable system performance.

Expected Variation

The variance in CPU load is greatest among 2X drives. The best drives approach 300K using 10% of the CPU; the worst suck down 90% of the CPU. A game designed for a full 300K data rate will starve to death on the poorer 2X drives because it won't have any processor time to spend on decompression, output, and game logic. The practical limit for a game intended to run on a 2X drive is around 260K, while 225K is common. CPU load is more consistent among 4X systems. If your game needs a 4X drive, you can probably assume the performance guaranteed by MPC III.

We're not intending to embarrass drive manufacturers in this article, but Mitsumi deserves special recognition for selling a ton of 2X drives with truly awful device drivers. More than one best-selling game contains special workarounds for Mitsumi drives. The drives are O.K., but the MTMCDAS.SYS driver is truly brain-damaged. It synchronizes itself to the drive with delay loops, and it doesn't check the drive status within the loops. If you interrupt a delay loop (for example, because you're doing preemptive multithreading), the driver still ties up the CPU by counting down to zero after you switch back to it—even though the drive could already be done with its operation. The MTMCDAE.SYS driver is only slightly better: it synchronizes itself to the drive with an interrupt, but its interrupt handler leaves interrupts disabled for so long that other interrupts (timer, sound card, threading) get missed.

Variation in Seek Times

If you want to avoid a noticeable delay when you open a different file or jump around within a file, you should buffer up at least a half second of sound and video to be played while the seek takes place. MPC II guarantees an average seek time of 400ms or less; MPC III guarantees 400ms for a notebook computer, 250ms for a desktop. Your code should anticipate seeks that take longer than these averages.

Seek times generally increase in proportion to the distance the laser has to move. Ironically, the worst seek times we've seen on any 2X drive (up to 1 second!) were for seeks of less than 256 sectors, because that particular drive used a different algorithm for very short seeks.

Some CD-ROM drives defer seeks until they're forced to actually read data. This trick makes for faster servers, but it can cause delays in your game if you're not prepared for it.

Programming Interfaces

Now that you know what you're up against, it's time to get down to programming. There are up to four different layers of system software between you and the CD-ROM drive: DOS, MSCDEX, the MS-DOS device driver provided by the drive manufacturer, and possibly a lower-level driver (for instance, a SCSI).

DOS affords the simplest programming interface and one you already know how to use. You can open a file on the CD-ROM drive with a call to `fopen()` and perform random-access reads by combining calls to `fseek()` and `fread()`. Although DOS is a real-mode interface, the protected-to-real-mode translation is done automatically by DOS/4GW.

DOS adds a little overhead to every CD-ROM request, and the DOS extender adds a little more. DOS/4GW has to copy the data read by `fread()` into extended memory, and it will break any read of more than 8K into multiple calls to DOS. You can't eliminate the DOS overhead. You can eliminate DOS/4GW's by allocating your own buffer in low memory and associating that buffer with the file pointer for the CD-ROM, like this:

```
FILE *fp;
// pointer to the file on CD-ROM
RealPtr p;
// see listing for RealPtr, AllocateLow
p = AllocateLow (16 * 1024);
// allocate 16K buffer in DOS memory
setvbuf (fp, p.ptr, _IOFBF, 16 * 1024);
```

The biggest remaining problem with this scheme is that DOS reads are synchronous; your program will have to sit and wait for the read to complete. Even if the CD-ROM drive has a low CPU load, your program won't be able to reclaim those spare CPU cycles for anything except its interrupt handlers.

The MSCDEX Interface

Since DOS calls MSCDEX when it's asked to read from or seek on a CD-ROM drive, you might ask how the MSCDEX interface is different. It's still a real-mode, synchronous interface, but you ask MSCDEX to read with `INT 2Fh/AX=1508h` instead of `INT 21h/AH=3Fh`.

More significantly, MSCDEX deals with 2,048-byte sectors, not files. When you open a file in DOS, DOS calls MSCDEX to return the directory entry structure for the path you specify. This structure contains the sector location and size of the file. If you know this information beforehand, you can avoid rereading the directory entry (and eliminate an unnecessary seek or two). MSCDEX always reads entire sectors, so you must do your own buffering if your request isn't sector-aligned.

Driver Level and Below

The next rung down on the ladder is the MS-DOS device driver, which is called by MSCDEX according to a very standard protocol. Even though a driver may be implemented in an idiosyncratic way, certain functions are required if the driver is to work with DOS. Hence, the device driver interface is the lowest level at which you can interact with a CD-ROM drive without knowing what kind of drive it is. Unfortunately, it's yet another real-mode, synchronous interface.

The device driver entry points are a pair of real-mode functions in the same code segment as the device driver header. The offsets to those functions are contained in the header. You make a far call to the first function, called the strategy routine, passing the address of a low memory data block in `ES:BX`. You then make a far call to the second function, known as the interrupt routine, and the driver performs the requested operation. It's a little tricky to set all this up from a DOS/4GW program, but MSCDEX provides a shortcut: put the data block address in `ES:BX`, put a drive identifier in `CX`, and issue real-mode `INT 2Fh/AX=1510h`. You can find an example of this technique online at http://www.gdmag.com.

By talking directly to the driver, you can bypass any disk caches and get the most consistent performance with the least overhead. You can also confuse the higher-level system software, so your game may not run correctly in a multitasking environment. If you decide to program to this level, it's a good idea to eject the disc at the end of your program to reset the state of the drive.

There's little to be gained from going below the MS-DOS device driver interface. Some device drivers might support asynchronous operation; some might even be callable from protected mode. Each one is different, and you'd have to support them all to have a game worth selling commercially.

Synchronous APIs

We've examined the sensible programming interfaces for CD-ROMs and found that, unfortunately, none of the choices is asynchronous. How, then, do we reclaim the spare CPU cycles, which are currently being used to turbocharge the wait loops in our device driver?

On some drives, we can improve the situation considerably by reading small blocks from the disc on a regular basis, interleaving the reads with other work so the CD-ROM drive's internal buffer has time to refill before the next read. In fact, this technique is the key to getting reasonable throughput from the MTMC-DAS driver. You can try varying the size of the blocks and the length of the delay between them to find the sweet spot of a drive, but you may not need to if you aren't after the highest possible data rate.

Not all drives perform well with small reads. To maximize throughput on most drives and reclaim seek time, we need preemptive multithreading. Preemptive multithreading is the only way to reclaim the CPU (to, say, render a frame) during one of those synchronous calls into the CD-ROM driver.

The implementation of a multithreading system for DOS/4GW is beyond the scope of this article, but commercial libraries are available. Different drives respond differently to different access patterns, so you'll need to experiment with the thread duty cycle (percentage of time given to the CD-ROM driver), thread switch frequency (size of each time slice), and the size of the blocks and the length of the delay. Keep in mind that many CD-ROM device drivers are not reentrant, so only one thread in your program should access the CD-ROM.

Case Study: Rebel Assault II

Rebel Assault II: The Hidden Empire from LucasArts is the sequel to the action-arcade game Rebel Assault. Set in the Star Wars universe, it features 15 chapters of play and uses high-quality cinematic video sequences to advance the story and mood. The game play features various flying, dodging, and shooting sequences set in front of interactive streamed backgrounds.

The minimum platform for Rebel II is a 486/50 with a 2X CD-ROM drive. To achieve acceptable performance and image quality on this platform, LucasArts wrote a custom animation system. This

Rebel Assault II combines gameplay and breathtaking cinematic video.

system, the INteractive Streamed ANimation Engine (INSANE), is a collection of code libraries designed primarily to compress and play back video sequences. The system is modular, easily portable, and will be used in a majority of LucasArts's upcoming titles.

In Rebel Assault II, noninteractive sequences are 320-by-200 pixels, while interactive sequences are rendered in 424-by-260 resolution. Both use 8-bit, 256-color imagery and appear full screen. For higher-end machines, optional interpolation up to 640-by-400 resolution is available. High resolution is more CPU-intensive, so this may result in a slower frame rate than low resolution, even on a moderately-powered system. To account for this, the system was designed to elegantly handle a less-than-optimal frame rate.

Each frame of video typically consists of 13K of video and 2K of audio. With a data rate of 225K per second, this allows a frame rate of 15fps. Due to the large quantity of video generated for the game, it would have been unreasonable to generate multiple copies of the video streams, each running at a different frame rate. Instead, all video sequences are designed to run at the machine's maximum speed, capping the rate at an optimal 15 frames per second. For high-end systems, the extra CPU time can be used to run in high resolution.

To account for possible synchronization problems due to variable frame rates, two approaches were taken. For sequences without onscreen speech, music and sound effects are linked to specific key frames and designed to accomodate up to a 15% variance in frame rate. For sequences with onscreen speech, rigid synchronization is used. For these sequences, every other frame of video can be optionally omitted, saving decompression and display time and allowing the animation engine to catch up to lip-synched audio.

For some interactive sequences, smooth branching must occur. To achieve this, the system allows video segments to be interlaced into the data stream and preloaded before a possible branch point. When the branch point is reached, the preloaded segment is played to cover up the seek delay to the new animation.

The INSANE library performs reads through DOS for portability. To achieve smooth, uninterrupted animation, it uses a hybrid preemptive cooperative multitasking system, in which data reads are performed within a mainline DOS thread; decompression and game logic run in time slices granted via the timer interrupt. Decompression time can vary from frame

Loadstar's game required smooth transitions between video segments.

to frame depending on the layers of imagery and compression options used in a particular frame. To achieve best overall performance on all video sequences, the system dynamically varies both CPU time-slice allocation and decompression frame rate based on CD-ROM read performance and decompression time.

Case Study: Loadstar

Rocket Science's game Loadstar: The Legend of Tully Bodine is a fast-paced arcade shooter set against a movie backdrop. Loadstar has received accolades for its speed and production values. Much of the action takes place within a network of tracks on the surface of the moon, and players must simultaneously navigate this maze, avoid damage to their ships, conserve power for shields, and shoot down attacking gunships.

The programming of Loadstar was guided by the following principles:

- The game should be playable on any 486/25 with a 2X drive but take full advantage of faster machines. It should play back as rapidly and smoothly as possible, without the player having to tweak it for his particular machine.
- The video images are 320-by-200 pixels, with a 256-color palette. For best color reproduction, the palette is updated on every frame.
- The narrative segments must play back at 24 frames per second, for smoothness. Some frame dropping is acceptable. The interactive segments (which are more expensive because of sprites and sound effects) should play back at 24fps if possible, with 12fps being the minimum acceptable rate.
- Video and audio must be perfectly synchronized.
- Even though every branch in the maze represents a jump to a new movie segment on the disc, there must be no delays no matter which direction the player goes.

To accomplish these ambitious goals, Rocket Science developed a game compiler that ensures related data is grouped close together on the master CD-ROM, that the data rate required throughout the game is known, and that the data rate stays almost constant. Rocket Science also spent considerable engineering effort figuring out the fastest way to read an arbitrary CD-ROM drive.

Loadstar profiles the machine it's running on and scales the size of the video rectangle, the quality of the sound effects, and several aspects of gameplay to ensure that the game will play smoothly. On machines with very slow CD-ROM drives, it will select 12fps data streams instead of the usual 24fps streams. It always bypasses DOS and MSCDEX because there's no room for overhead when your data rate is 260K.

Once it figures out the optimal access pattern for a drive, Loadstar uses a background thread to read data into a big queue. The foreground thread empties that queue, decompresses the data, superimposes sprites, and copies the composited data to another queue in video memory. The balance of time between the threads is adjusted dynamically, based upon the amount of data in both queues.

An interrupt handler synchronized with vertical nondisplay empties the video frame queue and updates the palette. If a frame is ready before its predetermined time, the extra time is given to the CD-ROM drive. If a frame is ready late, it's displayed as soon as possible, but the next frame is thrown away to give the game a chance to catch up.

The background thread is also responsible for seeks. Rocket Science's game compiler arranges the data stream on the CD so the first half second of data for every possible branch is read into memory before the branch actually takes place. Then, no matter which branch the player takes, the action continues seamlessly until the laser reaches its new location and a new block of data gets read.

The Need for Speed

Now you know how to make efficient use of a CD-ROM. Remember: the easiest way to speed up some models is to speed up everything else. Getting a decent data rate out of the Mitsumi FX001D, for example, forces you to give up at least 70% of your CPU cycles. If the rest of your game needs only 30% of the power of a 486/25 or scales across a range of processors, your job will be easier.

Protected-mode, multithread-aware CD-ROM drivers are another advantage Windows 95 and OS/2 have over DOS. The MPC III specification guarantees 550K at just 7% of the CPU on those operating systems. We look forward to the day you can design a CD-ROM game without worrying too much about the low end of drive performance. ■

Vincent Lee is a project leader and designer at LucasArts Entertainment. He was project leader and lead programmer for Rebel Assault and Rebel Assault II. He can be reached via e-mail at gdmag@mfi.com.

Dan Teven specializes in 32-bit systems programming for extended DOS and Windows 95. He has consulted on threading and CD-ROM issues for numerous projects, including Loadstar. He can be reached via e-mail at gdmag@mfi.com.

Back

Introduction to Vector Math

It was a warm, sunny Spring afternoon. Birds were singing, and beautiful flowers gently swayed in the wind. My brother and I—both under age ten at the time—really didn't care how it was outside; we were spending our time inside the house absorbed in a tough problem: the entrance to our basement hideout had been obscured by large, heavy boxes. To us, the short hole that led beneath the stairway was more than just a storage facility—it was our top-secret hideout where we could discuss our plans of sister hijacking and such.

Despite repeated efforts on my part, all I could do was dent the book-filled boxes when I tried to move them, a situation I was sure Mother wouldn't be too proud of. My failure prompted me to call my younger-but-larger brother Brian in for help. In the process of our two-man herculean effort, we discovered an interesting principle which, although quite new to us, was based upon well-known geometric principles that were discovered long before we were born. We discovered that moving a square box forward can be accomplished by applying force to an opposite and adjacent side at 45° angles (see Figure 1). The principles behind this little trick not only helped us uncover our hideout, they also have applications in fields as diverse as aviation and computer games. These principles revolve around vectors and scalars in 3-space.

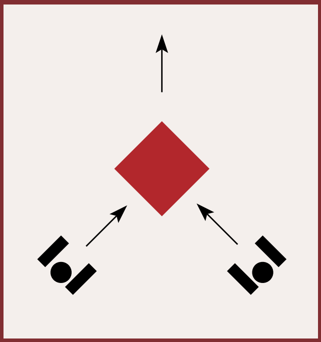


Figure 1. A square box can be made to move forward by applying force to two of its sides.

When describing a state or location, we often resort to using a direction and a magnitude. A magnitude can mean a velocity, a distance, a force, or just about anything that a single number can fully represent. For example, it's fairly common to describe a location using a direction (such as north or south) and a magnitude (such as 37 miles). Accurately describing how a box is being moved (as described in Figure 1), the location of your nearest supermarket, and the velocity and direction of a train all require the use of a magnitude and a direction. For example, we could describe a moving train using its magnitude (which would take the form of a distance or a velocity, whichever we preferred) and its direction (which direction the train is moving). Describing all these situations is so common that we might not be surprised to find out there is a mathematical name for the entity that posesses both a direction and a magnitude: a vector. A vector is a directed line segment whose magnitude is equal to its length and whose direction is measured by an angle (in two dimensions) or angles (in three dimensions). If an object requires only a single number to fully describe its state (such as temperature, where only a magnitude is necessary), it is called a scalar.

You can visualize two- and three-dimensional vectors, such as the vector shown in Figure 2, by graphing them on a piece of paper. Figure 2 shows two two-dimensional vectors in which each vector has a head and a tail. Vector tails are always located at the origin of the coordinate system, which for two-dimensional vectors is the point (0,0), and which for three-dimensional vectors is (0,0,0). The head of the vector specifies its direction and magnitude. By drawing a line from the origin of the coordinate system to the head of the vector, we obtain a line segment that visually represents both the direction and the magnitude of the vector.

English is a rich and varied language, carrying with it a number of words imported from other languages. With all its variety, you'd think it would be possible to describe almost everything using everyday words and expressions—and you'd be right. However, English can often be overkill, especially when an elegant and precise way exists to describe a situation. Fortunately for our sakes (or unfortunately, if you really hate math), such a short-cut exists for vectors.

Vectors are always represented with a single, upper-case letter, such as the letter V. The magnitude of a vector is represented by $|V|$, where V is the vector. The notation $V = <a,b,c>$ is used for a three-dimensional vector with its head at point (a,b,c), where a is the distance along the x axis of a Cartesian coordinate system, b is

John De Goes

the distance along the y axis, and c is the distance along the z axis (the z coordinate represents depth, of course). The vector <1,0,0> has a magnitude of 1, with a direction in the positive x direction, and it is always called the unit vector i. The vector <0, 1, 0> is pointing in the positive y direction and is called the unit vector j. Finally, the vector <0, 0, 1> is pointing in the positive z direction and is called the unit vector k. Any vector can be written in terms of unit vectors i, j, and k. For instance, a vector with its head at point (a,b,c) can be written: V = ai + bj + ck. The magnitude of any vector V can be found by using the Pythagorean Theorem ($a^2 = b^2 + c^2$), and the direction angle or angles can be found by using trigometric functions (more on this in a moment).

Vector Mathematics

Multiplication is defined for vectors, as is addition. You can multiply a vector by a scalar by multiplying all components of the vector by the scalar. This operation can be used to reverse the direction of a vector (multiplication by -1) or to change the length of a vector. Multiplying a vector by a scalar does not change the direction of the vector, only its magnitude.

You can multiply a vector by a vector two ways, one is called the dot product, the other the cross product. The dot product of two vectors (written U • V, and pronounced “U dot V”) is a useful scalar in certain types of calculations involving physics, forces, and orientations. In mathematical terms, this equation is written:

$$U \bullet V = (Ui + Uj + Uk) \bullet (Vi + Vj + Vk) = Ui \bullet Vi + Uj \bullet Vj + Uk \bullet Vk.$$

By performing a little algebraic manipulation (and by using the Law of Cosines), we can obtain the angle between any two vectors with the equation:

$$ø = \cos^{-1} \frac{(U \bullet V)}{(|U||V|)}$$

Magnitude and direction are both important in describing states and locations. Vector math is an important concept that game developers should know about.

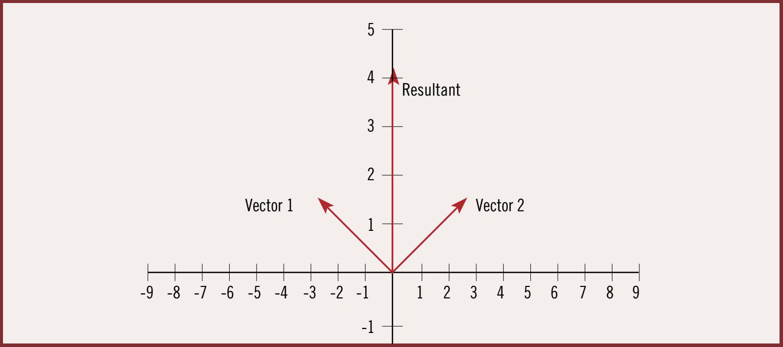


Figure 2. Force and direction are at work in these two vector representations of Figure 1.

Listing 1. Vector .HPP (A Vector Object) Targeted to 32-bit OS

```
// Note: All functions are inlined for speed

#ifndef __VECTOR3D__
#define __VECTOR3D__
#include <Math.H>
#include <Stdio.H>
#include <Stdlib.H>
class Vector3D {
protected:
  float i, j, k;
public:
  // The constructors:
  Vector3D () { ( *this ) = 0.0F; }
  Vector3D ( float Ni, float Nj, float Nk )
     {
     Set ( Ni, Nj, Nk );
     }
  // Set/Get member functions to provide access to
  // i, j and k:
  void Set ( float Ni, float Nj, float Nk )
     {
     i = Ni; j = Nj; k = Nk;
     }
  float Geti () { return i; }
  float Getj () { return j; }
  float Getk () { return k; }
  // The following two functions return the magnitude of
  // the vector:
  operator float ()
     {
     float Mag = sqrt ( ( i * i ) + ( j * j ) + ( k * k ) );
     return Mag;
     }
  float Mag ()
      {
      return float ( *this );
      }
  // The following function returns the dot product of
  // this vector and another:
  float Dot ( Vector3D &V )
      {
      float DP = ( i * V.i ) +
                 ( j * V.j ) +
                 ( k * V.k );
      return DP;
      }
  // The following function returns the angle
  // (a <= 3.14 >= 0) between this vector and another (angle
  // returned in radians):
  float Angle ( Vector3D &V )
      {
      float Rad = acos ( this->Dot ( V ) /
                       ( Mag () * V.Mag () ) );
      return Rad;
      }
  // The following function returns the cosine of the
  // angle between this vector and another:
  float CosTheta ( Vector3D &V )
      {
      float CosA = this->Dot ( V ) / ( Mag () * V.Mag () );
      return CosA;
      }
  // The following operators are overloaded for common
  // operations; some work with both scalars and vectors:
  Vector3D &operator = ( float Scalar )
     {
     i = j = k = Scalar;
     return *this;
     }
  Vector3D operator + ( Vector3D &V )
     {
     Vector3D R;
     R.i = i + V.i;
     R.j = j + V.j;
     R.k = k + V.k;
     return R;
     }
  Vector3D operator - ( Vector3D &V )
     {
     Vector3D R;
     R.i = i - V.i;
     R.j = j - V.j;
     R.k = k - V.k;
     return R;
     }
  Vector3D &operator += ( Vector3D &V )
     {
     i += V.i; j += V.j; k += V.k;
     return *this;
     }
  Vector3D &operator -= ( Vector3D &V )
     {
     i -= V.i; j -= V.j; k -= V.k;
     return *this;
     }
  // The following functions calculate the cross-product
  // of this vector and another:
  Vector3D operator * ( Vector3D &V )
     {
     Vector3D R;
     R.i = ( j * V.k ) - ( k * V.j );
     R.j = ( k * V.i ) - ( i * V.k );
     R.k = ( i * V.j ) - ( j * V.i );
     return R;
     }
  Vector3D &operator *= ( Vector3D &V )
     {
     float oi=i, oj=j, ok=k;
     i = ( oj * V.k ) - ( ok * V.j );
     j = ( ok * V.i ) - ( oi * V.k );
     k = ( oi * V.j ) - ( oj * V.i );
     return *this;
     }
  // The following functions multiply a vector
  // by a scalar:
  Vector3D operator * ( float Scalar )
     {
     Vector3D R;
     R.i = i * Scalar;
     R.j = j * Scalar;
     R.k = k * Scalar;
     return R;
     }
  Vector3D &operator *= ( float Scalar )
     {
     i *= Scalar;
     j *= Scalar;
     k *= Scalar;
     return *this;
     }
  // The following function normalizes a normal to
  // a length of 1:
  void Normalize ()
     {
     float OneOverDist = 1.0F / Mag ();
     ( *this ) *= OneOverDist;
     }
};
#endif
```

Taking the cross product of two vectors (written U x V, and pronounced "U cross V") produces a third vector perpendicular to the plane formed by the two vectors, or zero if the two vectors are parallel. We shall not examine the equation for the cross product, but you can see it in source form by examining Listing 1.

Adding two vectors results in a single vector that has the same effect as a combination of the two vectors; this resulting vector is called, intuitively enough, the resultant. (See Figure 3. If you're especially observant, you'll notice it describes the cause for the mysterious box movement my brother and I discovered.)

To add two vectors, you add the components of the first vector to the corresponding components of the second vector (in Figure 3, you add (2,3) to (4,2) to get (6,5)). To subtract one vector from another, you multiply the vector to be subtracted from the scalar -1 and then add the vectors; this is called vector subtraction. You can calculate the vector between any two points by subtracting (via a vector subtraction) the head point from the tail point. Finally, a vector whose elements are zero is called a zero vector; this vector has no direction or magnitude and is often written as 0 (thus, V + 0 = V).

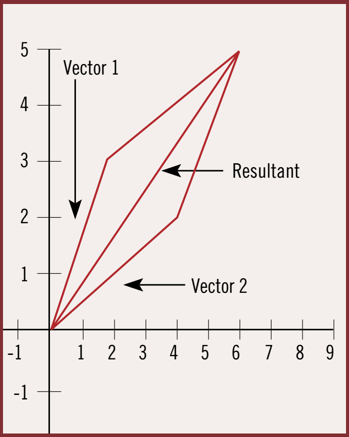


Figure 3. The addition of any two vectors is called the resultant; the resultant has the equivalent effect of the two vectors.

Applying Vectors to Polygons In Games

One important application of vectors lies in describing the orientation of polygons, as shown in Figure 4. Describing polygon orientation using vectors allows you to perform lightsourcing, backface culling, and visible surface determination. You can create a vector that describes the orientation of a polygon (often referred to as a surface normal) by taking the cross-product of two adjacent, coplanar vectors. These two vectors can be created by using two adjacent polygon edges. For instance, if a coplanar polygon has four points, labeled A, B, C, and D, you can create two vectors U = A - B and V = C - B for use in the cross-product operation, which will result in the surface normal.

Vectors can also be used in applications involving force and physics. For instance, let's suppose you're creating a game in which water crafts predominate, and you want to calculate the effect the current has on the crafts moving about in the water. The water can be represented with a vector, as can the water craft. Adding these vectors will give you a vector that describes the true heading of the water craft; indeed, you can even factor water resistance into the vector by multiplying the resultant by a scalar. These same principles can easily be extended to both aircraft and land craft, where forces such as wind constantly alter their heading and resist the movement of all objects. The position and speed of both the boat and the current form two distinct vectors.

Vectors can be used for simulating gravity (a gravity vector "pulls" on all polygons), for collision detection between a polygon and an object (by calulating the angle between the polygon and the object's direction you can determine the "bounce" vector), reflection mapping, ray-tracing, ray-casting, and almost everything else related to the third dimension, not to mention the important uses vectors have in such roles as simulating torque, resistance, and exterior forces.

In my explanation of vector principles, I've resisted using source code. The source code implements all the major vector operations, and, because it's written in C++, the mathematical operators have been overloaded so you can use them as you would any other data type. Therefore, I shall leave much of the explanation of Listing 1 to my comments contained therein. However, I want to mention the overloaded multiplication operator in passing, simply because seeing it may cause some confusion. Defining the operation for the multiplication operator gave me quite a headache: I could have designed the operator to perform a dot product, which is in fact one way to multiply a vector by a vector, or I could have designed the operator to perform a cross-product, another way of multiplying one vector by another. I chose the latter approach because the result was a vector, adding the member function `Dot` to perform a dot product.

In closing, I would like to recommend the ninth edition of *Calculus* by George B. Thomas and Ross L. Finney (Addison Wesley, 1995) for readers interested in the proofs behind the equations presented in this article. Vectors are a complex, advanced subject and deserve entire chapters, but hopefully this article and source code will help you take the plunge into the wide and wonderous world of vectors and scalars in 3D space. ■

Disgusted by the one or two flames he received from his last article, John De Goes requests further flames be directed to gdmag@mfi.com, with apologies to Larry O'Brien, editorial director of Game Developer.

Max Delivers on NT Promise

David Sieks

With 3D Studio Max, the PC platform's familiar 3D workhorse application makes the move to a new operating system...and a new level of sophistication.

Autodesk's 3D Studio has brought high-end computer graphics within reach of the plebeian desktop personal computer. Its rich feature set has been augmented by the availability of more than 250 functionality-enhancing IPAS routines (third-party plug-ins), giving users a wide range of creative tools and special effects modules to choose from. This, combined with network rendering capabilities on the affordable PC platform, has made 3D Studio a viable choice for the demanding production environment. Countless games ranging from Trilobyte's 7th Guest to The Daedelus Encounter by Mechadeus have cited its use.

For the past three years, Autodesk's Multimedia division, now renamed Kinetix, has been crafting the next step in the evolution of this popular 3D application. Marking a departure from the linear naming convention that brought us through 3D Studio Release 4, this past spring saw the much anticipated unveiling of 3D Studio Max. As the software documentation advises veteran 3D Studio users, this is not just an upgrade but an entirely new program.

One major change is the switch from DOS to Microsoft's no-nonsense Windows NT as the requisite operating system. NT combines the user-friendly (and almost universally familiar) Windows operating system with undiluted 32-bit multitasking, multithreading workstation-class power. Max has been designed to take advantage of that power with transparent support for multiple processors and 3D hardware acceleration.

Which is not to say that Max requires a multiprocessor system or a 3D chipset. Its Adaptive Degradation feature allows the user to specify a balance between on-screen graphics and playback speed to maintain responsive screen redraw rates for each system's particular capabilities. You can set each viewport to display objects in anything from solid shaded mode to simple bounding boxes; you can customize animation playback so that you can, if needed, sacrifice realtime playback in order to avoid dropping frames or can insist on realtime playback at the expense of dropped frames.

However, Max definitely does demand a more serious hardware commitment than its DOS-based precursors. Minimum system requirements for 3DS R4 called for a 386 CPU with math coprocessor and 8MB RAM. That was certainly a bare bones setup, and saintly patience would be an additional requirement of anyone attempting to use 3DS on a 386 machine. The bare bones requirement for Max is much steeper, though: nothing less than a 90Mhz Pentium with 32MB RAM and 100MB of available hard-disk space.

What Max really wants is every bit of juice it can get, and a bag of chips. For optimum performance, Kinetix recommends RAM in the neighborhood of "64-128MB or more" and hard-disk swap space of "200-300MB or more." Mere mortals using more mundane hardware configurations have commented that Max performs noticeably slower than R4 on the same machine. Though it will run on lesser machines—for most of this review I used a 90Mhz Pentium with 32MB, and Kinetix has shown off their new baby running well on a Pentium laptop with 48MB—a well-stocked multiprocessor Pentium Pro with Glint chip driven hardware acceleration seems to be the ideal system for making serious use of Max.

Kinetix isn't apologizing for that. Their stated goal with Max is to provide a production-ready application for professional 3D animators. Along with the many improvements in the 3D Studio interface and feature set, designing Max for NT and "personal workstation" caliber systems has enabled them to realize that objective. With over 65,000 installations prior to the release of Max, 3D Studio has long been numbered among the big boys. It's really grown up with Max.

Plug-Ins Unplugged

Professional game developers must be prepared to invest in the equipment needed to do the job, but I will allow a slight sympathetic pang for those users who despair of ponying up the dough for an NT workstation. Generous student discounts have long been a laudable Autodesk policy and still are: while Max retails for $3,495, the student price is only $1,295. Not exactly a giveaway, but along with the negligible system requirements of previous versions, such pricing enabled a lot of aspiring young computer-artists-on-a-budget to get their hands on the same pro-quality tool being used to create graphics for their favorite games. Many graphics pros today probably owe their jobs in the game industry to that early hands-on experience. Even with PC prices dropping, a minimally equipped NT workstation carries about a $5,000 price tag, which will likely put Max out of the reach of many hobbyists and amateurs.

But save your tears for the folks who kept the plug-in companies in business. Over the years, a lot of great plug-in modules came along to beef up 3D Studio's features: things like particle effects, lens flares and glows, metaball modelers, and much more, many selling for hundreds of dollars. Well, put 'em away: they won't work with Max. Which is not to say that Max can't use plug-ins. It's actually specifically designed to facilitate the integration of new plug-in modules.

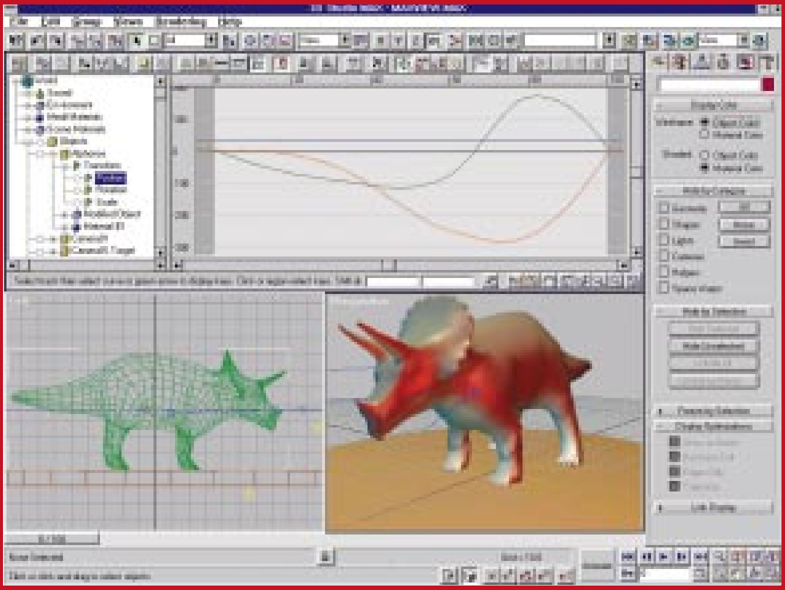
Kinetix will provide every registered user with the Max Software Developer's Kit, enabling those so inclined to create their own plug-ins (with Microsoft Visual C++) that will fit seamlessly with the Max interface and be as accessible as its out-of-the-box features. If you're more interested in developing your own games than your own graphics tools, rest assured that most of the third-party developers responsible for the old crop of 3D Studio plug-in modules are hard at work making add-ons for Max. Digimation—makers of the popular LenZFX and Bones Pro IPAS routines for 3D Studio—are even planning a plug-in that will let you use your old DOS plug-ins.

Max has incorporated some of the old 3D Studio IPAS routines as part of its standard feature-set. Particle effects that simulate rain, snow, drifting confetti, or spray from a garden hose—all with user-definable parameters—are part of the basic Max package. Combined with deflection, gravity, and wind routines, these can be used to create a wide range of realistic effects.

Also, Kinetix plans to release their own plug-ins. Some are to be free, added features for registered users—such as the Combustion plug-in for fire and explosion effects made available shortly following the official release of Max, or a patch planned for release this summer that will enable the use of 32-bit Photoshop plug-ins. Others will be optional extras, like the hotly anticipated Character Studio, which is to feature footstep-driven character animation and "muscle-bulging, tendon-stretching, and vein-popping" skeletal deformation capabilities.

Meet Max

Registered 3D Studio users had plenty of incentive (namely limited-duration upgrade offers) to make the move to


The 3D Studio Max workspace, with toolbar at top and command panel (showing Display option) to right. Multiple view panels can be selected by the user—here, the Track View is opened across the top half of the view area to display function curves controlling object movement. The camera view in the lower right has been set to display in solid shaded mode, showing light color and specular highlights, updated interactively.

Volumetric lighting adds atmosphere *a la* Ridley Scott.

Max in the months following its release. Therefore, I'm writing this article for those who aren't 3D Studio users but wonder if Max is something you or your company should be using. Well, I don't know. Why weren't you using 3D Studio?

Personally, I didn't use previous versions of 3D Studio as much as I might have because I found the interface and compartmentalized structure awkward. Despite that, I did use the program because of the wealth of features it made available for 3D graphics on the PC, but I never found working in 3D Studio to be the fluid experience the creative process demands. If you're familiar with DOS versions of 3D Studio, you will notice some similarities between it and Max, but the latter really is, as Kinetix says, a whole new program.

One important step in a positive direction is that all of Max's features are integrated in a single unified environment. Everything—from the creation of two-dimensional shapes to modeling and animating 3D objects, from tweaking animation timelines to video post compositing effects—takes place in the same workspace. Many elements of the interface can be customized to fit individual work styles.

The view onto the 3D scene can consist of a single large window or up to four smaller windows. Any window can display perspective, camera, or orthographic views, and the user can easily toggle between full-screen and multi-window configurations. Each window can be configured on the fly to display in wireframe or solid shaded mode.

Commonly used tools, such as Object Select, Move, Scale, and Rotate are always present in the toolbar at the top of the screen. Other tools appear in a command panel to the right of the screen. The command panel is arranged by notebook-style divider tabs into folders that group toolsets by function. As you work, the information and available tools in the command panel are automatically updated to correspond with the currently selected object or action: only what relates to the situation at hand is presented. This is important in helping to keep Max's many features and adjustable parameters from cluttering up the screen.

Max contains such a large number of features that even displaying them selectively can overflow the command panel. Kinetix, however, has managed this cornucopia quite well. Some functions are contained in collapsible rollout areas in the command panel, so they can be closed yet remain present when not in use. Other tools are reached via a More button, which indicates the availability of additional features. This may sound reminiscent of the nested menus-within-menus found in previous versions of 3D Studio, but I found this new command panel far more user-friendly. Best of all, the user chooses which tool buttons appear onscreen and which are relegated to the More file: customized buttonsets can be created and easily swapped. Customized keyboard shortcuts can also be created to quickly invoke oft-used features.

A range of view manipulation tools below the command panel lets the user pan, rotate, or zoom in on an area of detail in any window and zoom back out with one mouse click so the entire scene fits within the window. Cameras placed in the scene can be easily positioned and aimed by dragging the mouse in the view window. Also, lights and cameras can be "hidden" from sight: though still present in the scene, they can't be seen onscreen with other objects, which can help eliminate clutter in busy scenes. Actually, any object, or even a selected portion of an object, can be hidden for the same purpose.

The user selects objects for manipulation by clicking on them or by selecting them from a dropdown list. This latter ability makes it much easier to pick the item you want from a crowded scene. Unfortunately, the selection list is chronological, according to the order in which objects were created, rather than alphabetical, so find-ing an item in a long list of objects is not very fast.

Many icons gracing the tool buttons are cryptic in their meaning. Even when I knew what the tool was, sometimes I couldn't quite determine the significance of the icon. Obviously, though, the tools become familiar with use. New users can hold the mouse pointer over a button momentarily to cause a tooltip to pop up, identifying its function. Max also features Windows 95 standard online Help and the traditional Autodesk heap o' manuals.

The manuals are excellent, especially the tutorial manual, which presents 23 detailed lessons to introduce the new user to Max's wide-ranging features. The tutorials are especially successful because they not only show how to use Max, in many areas they attempt to familiarize the reader with common pitfalls or misconceptions and really explain how and why the program works. This leads to a much deeper appreciation and understanding of the product than simply being led by the nose through the sample exercises.

Modeling With Max

A plethora of cool modeling features give the user great versatility in the creation of objects. Objects can be built from geometric primitives, which can be modified as polygonal meshes, sliced and segmented, or smoothly molded by Bezier patches. Faces, edges, and vertices can be edited singly or in groups. Boolean operators can fuse objects together or subtract one object from another. You can even create an object completely by hand, placing vertices one at a time. There are also many modifiers with adjustable parameters to bend, twist, taper, and otherwise abuse objects.

For models to be used in real-time 3D graphics or for the sake of economy, Max offers an Optimize modifier that reduces polygon count interactively. You can also delete and fuse edges selectively to reduce geometry with complete control. Topology can also be animated over time, so that an object can be composed of simplified geometry until greater definition is needed later in the sequence.

One modeling feature crammed with potential is the Displace modifier. This creates a three-dimensional mesh object from a bitmapped image: light areas become 3D peaks, dark areas become valleys. You can use this to create anything from a landscape to a detailed face and then use the Optimize modifier to simplify its geometry to an acceptable level.

Perhaps the most liberating aspect of object creation in Max is the Modifier Stack. In life, we make decisions, we choose paths, and we rarely if ever get to see how things might have been had we chosen differently. The Modifier Stack in 3D Studio Max stores each object's entire construction history, and allows the user to go back to any point in that history to make adjustments. You can see what would have happened had you applied a Bend to the object before you performed a Boolean operation on it, instead of after.

Or suppose you sweat over a flying logo animation for your new game and at the next meeting everyone loves the sequence but hates the title, and the decision is made to change the name. With the Modifier Stack, you can go back to the creation parameters of the very letters that make up the title and change them for the new title. Everything else in your animation will remain the same. All you have to do is re-render the sequence to see the identical animation featuring the new game title. Playing "what if" has never been easier.

Move It, Max

Animation is easy to record and edit in Max, and almost everything can be animated—from position, scale, and rotation to material properties and geometry. You can check your work as you go by playing back animation in the workspace in solid shaded or wireframe mode. Simply press the Play button and watch it happen, or scrub through the action at your own pace with the time slider.

When you want to edit your animation, you open the Track View. This lays out on a timeline all the animation keys for the scene and can be opened into the workspace just like a view window. On one level, the Track View is similar to the Windows File Manager-type hierarchical menu, so navigating through the different objects and their various animatable features is quite intuitive.

To the right of the scene menu in the Track View are the individual tracks containing keys for all animated features. At first, I found this difficult to use because of the amount of space between the list and the tracks themselves (it was often hard to match the two up). But you can, I discovered, drag the two as close together as you wish, which makes the Track View much easier to use. The ruler at the top of the Track View can also be dragged down to reference any track more directly.

In the Track View, the user can assign preset movement tangents to adjust the pace of an animation profile: to make a bouncing ball slow near the top of its arc, for example. You can also create custom Bezier tangents and use control handles to affect the curve of the animation profile interactively.

To help synchronize animation with audio files, the Track View can display wave forms, and Max can play a .WAV file along with your animation right there in the workspace.

Animation in Max can make use of both forward and inverse kinematics, allowing you different ways to link objects that are to be moved in concert. Adjustable joint parameters and friction damping settings provide realistic limitations to the linked objects' range of movement.

Another feature that might as well be mentioned under the umbrella of animation is what in Max are called Space Warps. Space Warps are non-rendering objects that act on other objects in a scene, causing these objects to ripple or deflect or even explode. This is a great and very flexible way to simulate physics within a scene. Max Space Warps include Bomb, Deflector, Displace, Gravity, Ripple, Wave, and

Wind, and the names alone give you a good idea of their function.

Render, Max

Rendering is the payoff for all your hard work, and Max has all the usual features to bring your scene to life: animated backgrounds, shadow maps or raytraced shadows, and a versatile range of light capabilities. Objects can be excluded from the effect of lights, allowing very selective lighting within the scene. Lights can also be precisely positioned by situating the specular highlight on the target object, and spotlights can be used to project images into a scene.

These are all good features that most computer graphics artists are quite used to, but it's good to know they're there. On the neater side, a new Max feature is volumetric lighting. This creates the effect of streaks or columns of light in the air, like sunlight illuminating dust motes or a flashlight casting its beam into the evening fog. It's a very sexy feature, sure to appeal to the Ridley Scott in everyone, and I'll go out on a limb and predict that it will soon give lens flares a run for their money as most overused effect.

Max also has a range of atmospheric fog effects you can add to your scene. Layered fog creates a creepy, low-lying pool of fog, while volume fog creates wispy blankets of mist you can blow around the scene with variable wind strength settings. You can even fly through wisps of volume fog in your scene.

The Material Editor provides a staggering array of options for adding surface detail to objects in your scene. This depth is not without its price—there's a lot of features for the user to assimilate to make full use of the Material Editor. If you're like me, you'll spend some time wandering lost in its intricacies, but you're unlikely to feel shortchanged by Max's material mapping capabilities.

A video post feature is also integrated with Max to facilitate editing tasks, including compositing with alpha channels, fades, wipes, and motion blur. Max customers already using a professional editing tool will probably not be inclined to abandon that for Video Post, but it does provide good, built-in functionality as part of the basic package.

For the busy production environment, Max's network rendering capabilities are indispensable. Only one registered version of Max with its hardware lock is needed to distribute large rendering jobs over a network. The network must be configured with the TCP/IP protocol and each machine must meet Max's minimum system requirements and be running NT. You can also monitor progress or even initiate a queue assignment from a remote machine that is properly configured to communicate with your network.

It's A Max, Max, Max, Max World

For more information and opinions, you might check out the Kinetix forum on Compuserve ("go kinetix"). This forum is administered by Autodesk and also serves as a good source for technical support, both from Kinetix staff and other users...a group, I expect, which will be growing very large indeed. ■

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

3D Studio MAX

3D Studio MAX for Windows NT
Autodesk Inc.
111 McInnis Pkwy.
San Rafael, Calif. 95903
Tel: (800) 879-4233
Web: http://www.ktx.com
Price: $3,495
System Requirements: 90 MHz Pentium PC, Windows NT 3.51, 32MB RAM, 100MB swap space, PCI or VLB graphics card, screen resolution of 800x600x256 colors, CD-ROM drive.