gd

GAME DEVELOPER MAGAZINE

APRIL/MAY 1996

UBM Tech

Network Games

Network gaming will recreate the computer entertainment industry and revitalize the cartridge market. Within a few years, online gaming will be the dominant form of computer recreation. Normally, I preach the "horizon of predictability"—beyond which nothing can be said with certainty—is an astonishingly short 14 to 16 months away. Anyone who predicts beyond that is like a six-year-old on a whale-watching expedition, shouting, "There's a whale," and pointing randomly. And network gaming isn't going to be a major force in the next 14 to 16 months (although by the end of 1996, early adopters will play exciting new games); it's more likely to happen within 5 years. But we're on the eve of adopting several technologies, which point to explosive growth in multiplayer gaming.

First and foremost, the world is getting wired. In other words, everyone will be able to at least "get to" game servers.

Second, 3D chips will take the market by storm. In the cartridge market, one need only look at the new generation of machines to be impressed, while the desktop video card market has been fairly bland for several years as the limitations of Windows 3.1 overshadowed improvements in card technology, color-depth, and onboard RAM. The past 14 months have seen an unprecedented boom in the high-end of the desktop 3D market—the technology gap Silicon Graphics has enjoyed for years has rapidly shrunk and, according to some, disappeared in the low-workstation price points. Consumer 3D technology is poised to enter the market, and card manufacturers will enjoy perfect timing as 3D graphics upgrades become the upgrade of the year.

Third, network connections will add voice capabilities. I don't think DVSD and ASDL modems will enjoy the explosive growth I predict for 3D video boards, but they'll become popular with niche, motivated buyers. Initially, this motivation will come from network-based telephony and multiplayer gaming of existing games. More importantly, ISDN should fill out the niche without jeopardizing bandwidth. ISDN, however, remains a technology with a considerable barrier to entry. Most people will wait for the big news. Which will be high-bandwidth connections, cable modems or ATM-to-the-curb.

This is the fourth, and furthest out element prepping us for an online gaming explosion. I can't predict whether to buy stock in cable companies or ATM manufacturers, since widespread availability of these technologies is years away. The bandwidth numbers of these technologies are incredible—more than enough to make believable the wildest ideas of network-based applications. If these technologies are really able to deliver and transmit several megabits per second, a revolution as profound as the arrival of the desktop PC will follow.

Finally, why did I say that network gaming will revitalize the cartridge market? Let's talk about the Java terminals computer magazines say have no market. They're right to say people won't download word processors and work on their resumes with an Internet terminal rather than their $3,500 home PC; kids will download the latest version of "Java Warriors," to their cartridge machines. Sun's second-tier MicroJava chip is scheduled for the first quarter of 1997, with a unit price of $25 to $50. Can an Internet terminal be built for $500? Let's see—a Java game cartridge, hooked up to a cable modem, on a Sony PlayStation or Ultra64. Do you think it would sell? ■

Larry O'Brien
Editorial Director

GAME DEVELOPER

Editorial Director Larry O'Brien
gdmag@mfi.com

Senior Editor Nicole Freeman
76702.706@compuserve.com

Managing Editor Diane Anderson
dianderson@mfi.com

Editorial Assistant Jana Outlaw
joutlaw@mfi.com

Contributing Editors Alex Dunne
76702.1142@compuserve.com

Barbara Hanscome
bhanscome@mfi.com

Chris Hecker
checker@bix.com

Mike Michaels
mike@irvine.com

David Sieks
dsieks@arnarb.harvard.edu

Editor-at-Large Alexander Antoniades
sander@mfi.com

Cover Photography Charles Ingram Photography

Publisher Veronica Costanza

Group Director Regina Starr Ridley

Advertising Sales Staff

Western Regional Sales Manager
Steve Nikkola (415) 905-2256
snikkola@mfi.com

Promotions Manager/Eastern Regional Sales Manager
Holly Meintzer (212) 615-2275
hmeintzer@mfi.com

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

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

Miller Freeman
A United News & Media publication

Applesauce

Alex Dunne

Those who do not learn from history are doomed to repeat it. At Apple, Amelio's got his work cut out. Let's jump in the time machine and look back on Spindler's reign.

If you had been at Macworld in San Francisco earlier this year, you probably wouldn't have seen any overt signs that Apple was auguring in. More than 70,000 people attended the show, you had to strong-arm your way through the crowds, and everyone seemed upbeat (even actor Gregory Hines of "White Nights" and "History of the World, Part 1" fame, who I saw on the show floor). But the bustling crowds and high energy at Macworld hid a frightening fact: Apple is in serious financial trouble, and despite Hines presence, there's no white knight coming to the company's rescue.

The latest rumor before we went to press was that Sun Microsystems was to acquire Apple. Sun joins an illustrious list of would-be suitors over the past few years. Unfortunately, even if this merger rumor had panned out, it probably would have been too little, too late. Apple's mismanagement has taken it to the brink. Pursuits such as Newton, eWorld, OpenDoc, Kaleida Labs, and Taligent have drained the company's coffers and diverted its attention away from its bread-and-butter computer business, which has suffered as a result.

Now it appears that the Wintel juggernaut has more than enough momentum to carry it past any would-be competition. Looking back a few years, it all seems so clear now how misguided Apple's strategies were. Let's set our way-back machine to the beginning of 1993 and roll tape.

1993: Spindler takes over.

January. Apple is enjoying outstanding sales of the immensely popular PowerBook notebook computers. However, because Apple is cutting its hardware prices to compete with the free-falling prices of Windows-based computers, first fiscal quarter earnings are actually down compared to 1992.

March. Second quarter results are again flagging, again due to price cuts. In response, Apple contemplates cutting its operating expenses through reorganization.

May. CEO John Sculley announces Apple is "shifting its focus away from hardware and concentrating on the system software that controls computers and communications, and even online information services." David Coursey of *PC Letter* calls the company "immensely confused."

June. Sculley steps down after a ten-year stint as CEO, and the board names president Michael Spindler to take his place. Spindler is charged with navigating the company out of its financial trouble and taking a greater hands-on role in management. Sculley stays on as chairman.

July. Third quarter results send shock waves through the industry. Apple reports a whopping $188 million loss, its largest-ever quarterly loss, due mostly to a $321 million restructuring. On the same day, Apple declares that 2,500 employees (over 15% of the company) will be laid off. Analyst Doug Kass, president of the Viewpoint Group consulting firm, sums it up when he tells the *San Francisco Chronicle*, "A well-run company shouldn't have to reorganize every couple of years. To bring out a ground-breaking product like the PowerBook and then stumble leaves us with little confidence

that they can navigate a new market with Newton. . . ." How prescient.

August. At Macworld in Boston, Apple unveils the Newton, its $700 dyslexic "personal digital assistant." Apple insists it is not betting the company on Newton. Good thing.

September. Apple announces it will license necessary technology to other companies interested in manufacturing Macintosh clones, reversing a decade of closed-architecture strategy. Donald Strickland, Apple's vice president for licensing, hints to the *Chronicle* that a big player in the hardware industry will be among the clone makers.

October. Robert Puette, president of Apple USA, leaves a day before fourth quarter results are announced that show profits down 97%. Apple's inventory of unsold products stands at an unhealthy $1.5 billion. Ian Diery, executive vice president of Apple's Personal Computer Division, takes over for Puette. At Seybold, Diery explains that Apple is centralizing its forecasting procedures and monitoring aspects of sales to prevent product shortages and overproduction. Sculley leaves his position as chairman and is replaced by Apple cofounder Mike Markkula Jr.

November. Dell Computer, AST Research, and Compaq allegedly are approached about licensing the Mac OS but all refuse Apple's offer.

1994: Trimming Down, But Losing Focus.

January. Apple's online service, eWorld, debuts at Macworld in San Francisco. Gary Arlen, a media researcher, quips, "Unless Apple has something new to offer, it'll be tough to hold its own against the existing players." eWorld is priced much higher than competing online services, such as America Online and CompuServe. Earnings for the first quarter are down 75% by the ongoing price war in the hardware industry, despite record shipments of Macs.

February. Apple unveils its own version of a TV set-top box for delivering movies and services like electronic shopping. The box, which will sell at about $300, gets shrugs from analysts.

March. Apple launches its line of PowerMacs, which are driven by Motorola's PowerPC chip. The launch at Lincoln Center in New York is the most anticipated event since Apple launched the Macintosh a decade earlier. Oracle CEO Larry Ellison considers buying Apple with help from convicted junk-bond king Michael Milken. Go figure.

April. Gaston Bastiaens, head of Apple's Personal Interactive Electronics (PIE) division responsible for the Newton, is fired because of the product's dismal sales. The Apple PIE division employees reportedly break into cheers upon hearing the news. Spindler blames poor Newton sales on retailers who weren't doing enough to explain the Newton to customers.

May. Rumors fly that Apple is trying to persuade IBM to produce Macintosh clones. Ten years earlier, IBM was portrayed as Big Brother in Apple's famous television commercial launching the original Macintosh. How times change!

July. System 7.5 is launched. Analyst Bruce Lupatkin of Hambrecht & Quist says the operating system is "a nice, evolutionary extension, but there is nothing dramatic about it."

September. Apple starts an aggressive campaign to license its operating system, targeting major U.S. companies.

October. AT&T holds talks with Apple about a possible takeover, according to the *Chronicle*. Motorola is also mentioned as a possible contender for Apple. None of the companies comment. Spindler reportedly talks with IBM about securing an equity investment in Apple and possibly leasing the MacOS.

November. Apple, IBM, and Motorola team up to design a common machine. Two weeks later at Comdex, IBM backs away from the alliance, citing concerns from its corporate customers who think IBM might be wavering in its support for OS/2 by working closely with Apple.

December. Power Computing, a company with only 20 employees, and Radius announce that they will be the first companies to sell Macintosh clones.

1995: The Downward Spiral

January. *Information Week* reports that Oracle, Philips Consumer Electronics, and Matsushita are about to undertake a hostile takeover of Apple. Oracle is apparently interested in the MacOS and Taligent software, while Philips and Matsushita will split the hardware business. All parties deny the rumor, and Apple reiterates it is not for sale.

Apple posts record revenues of $2.83 billion for the first fiscal quarter, due largely to outstanding Power Macintosh sales and cuts in operating costs.

February. The Supreme Court denies Apple's appeal in a seven-year old lawsuit against Microsoft. The suit accused the Redmond company of copying the Macintosh graphical user interface, dating back to Microsoft Windows 2.03. Apple dashes off a letter to U.S. District Judge Stanley Sporkin, warning Sporkin that Microsoft has bullied Apple and can't be trusted to abide by the proposed antitrust settlement. Gates chastises Spindler for Apple's legal tactics, and denies Apple's allegation that Microsoft threatened to stop producing software for the Macintosh.

March. Spindler acknowledges Apple is having problems filling orders for Power Macintoshes because of poor planning and component shortages. Salomon Bros. cuts its earnings estimates as a result, lowers its rating on Apple to underperform, and the company's stock drops 8% in one day.

April. Spindler announces another major shakeup at Apple. Four divisions are merged into two, and veterans David Nagel and Dan Eilers are given leadership over the divisions. Ian Diery, Apple's executive vice president and head of the PC Hardware division, resigns.

Meanwhile, rumors swirl that Canon Inc. is in talks with Apple about acquiring the company.

May. Apple Chief Financial Officer Joseph Graziano states that Apple is not for sale and that the impact of licensing the MacOS will be "modest" during 1995. Radius announces it has to wait until fall to ship its Macintosh clones in large quantities due to a shortage of parts.

June. eWorld celebrates its first birthday. With a subscriber base of only 80,000, prices slightly higher than competing services, and no internet connections other than e-mail, the service is described by one analyst as, "America Online with nicer artwork, no content, and no users."

July. Frank Seiji Sanda, the president of Apple's successful Japanese operation, quits unexpectedly. Apple's third quarter results fall short of expectations, as earnings are down due to supply problems. Markkula sells 400,000 shares of his Apple stock. Apple declines to explain why.

August. Apple tries to dismiss the hype surrounding the launch of Windows 95 with their "been there, done that" slogan. Senior Apple executives sell hundreds of thousands of more shares of Apple stock. In two months, Markkula has pared his stake in Apple by $44.1 million.

Apple executives have quietly drawn up "golden parachute" agreements for themselves, entitling management to large payments in the event they lose their jobs or are demoted following a takeover. Spindler's payout alone could be worth as much as three times his salary and bonus, or around $3 million.

September. In one day, the company delivers a double-whammy of bad news. First, it advises investors that its earnings for the fourth quarter will be "well below" Wall Street estimates. Second, it announces that it has stopped shipping its new PowerBook 5300 models, due to problems with their lithium ion batteries that have a tendency to overheat and ignite.

Following this announcement, business columnist Herb Greenberg slams Spindler. "Yesterday, Apple appeared to be spinning out of control, but Spindler was nowhere to be found. On a day when he should personally have been reassuring investors, the press, and customers that all is well, he was out of his office, and he wasn't expected to return until Monday. Terrible timing."

Apple lays out plans to revamp eWorld, by offering sections of the service to users of the World Wide Web, and allowing eWorld subscribers access to the web from within the service. eWorld, still lagging far behind other services with only 115,000 members, is in trouble.

Mike Markkula, Chair
Photo courtesy of Apple Computer Inc.

Micheal Spindler, CEO
Photo Courtesy of Apple Computer Inc.

An increasingly skeptical press and analyst community speculates whether Spindler is going to get the boot from Apple. Markkula remains strongly behind Spindler, however, and there's little chance that the board will act to fire Spindler without the Chairman's say-so.

October. Joseph Graziano, Apple's CFO for six years and a member of the board, leads efforts within the board to remove Spindler, reports the *Chronicle*. According to unnamed sources, Graziano says, "Either Spindler goes and the company is sold, or [I] will quit." Apple's board meeting held thereafter produces no changes, and Graziano announces his resignation, citing "differences of opinion" with Spindler as the reason.

Commenting on Graziano's departure, Kimball Brown of Dataquest tells the *San Jose Mercury News*, "The most dangerous job in high tech today is the No. 2 spot at Apple, because Spindler is not comfortable sharing power."

Approximately a week later, Apple undergoes another reorganization. In the shakeup, the media group, which used to report to senior vice president Dan Eilers, now reports directly to Spindler.

November. Dan Eilers resigns. Apple and IBM's joint venture, Kaleida, shuts down.

December. Apple announces it will probably lose money in the first quarter, citing its inability to meet production demands and its shrinking profit margin. Apple's stock drops 15% in the two days following this announcement, and as a result Standard & Poor's Corp. puts the company's debt on credit watch for possible downgrading.

Apple and IBM end their partnership in Taligent, an effort to advance object-oriented software technology. Taligent becomes a subsidiary of IBM.

1996: Chapter 11? Acquistion? Your Call

January. A busy month indeed. At Macworld, Apple makes no major news announcements and holds no news conferences for the press. Spindler doesn't even make an appearance at the show.

Another exodus of Apple executives takes place, as a stream of vice presidents depart: Barbara Krause (VP of Corporate Communications), Jim Groff (VP of Education Marketing), Peter Friedman (VP and General Manager of Internet Sources), Keith Fox (VP of the Home Division), and Don Strickland (VP of Business and Government Sales). The number of high-ranking executives that have left in the past year stands at 14.

Apple reveals that it lost $60 million during the traditionally strong Christmas quarter and that its profit margins have hit a record low of 15% — despite further price cuts. Apple's misjudgment of demand for the more powerful Power Macintoshes resulted in a

backlog and lost sales during the holiday season. Apple announces it will undergo a huge restructuring, cutting 1,300 jobs at the company, and jettison unprofitable businesses.

Radius sells its clone-making business to Umax Data Systems. Linley Gwennap, editor of *Microprocessor Report* opines to the *Chronicle*, "It sounds to me as if Umax and Radius decided it wasn't worth having both of them [making clones] and that they'd rather combine their efforts...[the demand for clones] is not taking off as quickly as people thought it would."

The Wall Street Journal reports that Sun Microsystems is in negotiations to acquire Apple, and Sun's stock falls 9%. Apple denies the rumor, states that it is not for sale, and says that Spindler has the full support of the board.

Markkula announces that Spindler is suffering from health problems and says Spindler's role at the company "could be lessened or redefined." Some speculate this may be a way to ease the CEO out of the company gracefully. Others demand Markkula's resignation, blaming him for the company's travails.

At the annual shareholder's meeting, insults fly at Apple management. As Spindler and Markkula sit in front of 500 investors and employees, there are calls for resignations, accusations of incompetence, and demands for improvement. Employees complain about the loss of competent managers. One employee almost breaks down crying, explaining that though he loves Apple, he's considering job offers from other companies.

February. Apple's board announces that Gilbert Amelio, former president and CEO of National Semiconductor, will replace Spindler and take over the Chairman position from Markkula. The appointment signals to the world that an acquisition by Sun Microsystems is off.

Where to From Here?

That's the saga as we go to press. Whatever its future, major changes will continue to rock the company. Some have suggested that by spinning off its hardware business and focusing solely on its more profitable software business, Apple would insulate itself from the cutthroat hardware price wars which have caused the company's profit margins to dive. It's not a bad idea, but I doubt Apple will consider it, given their strong feelings about keeping the company intact.

This past January, one Macworld attendee was resigned to all of the depressing reports about Apple: "Most Macintosh people are used to hearing bad news all the time." Unfortunately, the worst may be yet to come. ■

While Alex Dunne is allegedly a Mac fan, coworkers have never seen him use one. Contact him via e-mail at 76702.1142@compuserve.com.

Games Aren't Just for Kids Anymore!

SAY IT!

The shady staff of *Game Developer* would love to hear your comments, questions, and suggestions! Please send them to: *Game Developer* magazine, Sez U!, 600 Harrison St., San Francisco, Calif., 94107. For those of you who *do* have access to the Internet, send e-mail to joutlaw@mfi.com or go to the *Game Developer* web site at http://www.mfi.com/gdmag. Thanks!

SATISFACTION GUARANTEED!

Dear Editor:

Great issue! Particularly, "The Play's The Thing" by Barbara Hanscome (Dec./Jan. 1996). I especially enjoyed reading quotes from people like Corey Cole and Jonathan Knight. Please do that again and more often. I much prefer to hear developer people talk than reviewers pontificate. Insights are invaluable.

Courtland Shakespeare
Via e-mail

EAT, SLEEP, AND DRINK GAME DEVELOPER

Dear Editor:

My copy of Game Developer and a Snapple 6-pack were placed on my desk by a caring individual. Before looking at the cover, I tried to picture what it would look like. "Windows 95 and Game SDK" page 62, "The Game Companies in and Around the New York Area" page 94, and "The Best C/C++ Compilers for Windows 95" page 105. I understand that the last two are recent requests, and you could not have possibly known of their existence. But the first article—there are 100 million Windows users in the world, and half of them have Windows 95. DOS is dead, and Windows 95 is the future. I just realized my opportunity, and I was hoping that you would help us lone programmers get a jump start with Game SDK. The last two requests are desperately needed by a young C and assembly student hoping to get a junior position in any game company. Help me, you're my only hope.

Sean S. Whalen
Via e-mail

Editorial Assistant Jana Outlaw responds:
Stay tuned: upcoming issues cover game programming using the Microsoft Game SDK. Maybe the Multimedia Careers section of the magazine will point you in the right direction, in the meantime.

CHRIS HAS FANS!

Dear Editor:

Thanks for the excellent series on "3D Texture Mapping." In the discussion of the lines-of-constant-z approximation to perspective texture mapping, Chris Hecker writes, "walls and floors are special cases where lines-of-constant-z are vertical and horizontal," but explains why it is not satisfactory in the general case. Maybe the best answer is to use the lines-of-constant-z method for the special cases of walls and floors, and to use the subdividing affine method for polygons that require the general solution. A closer examination of the lines-of-constant-z method for this very common special case would be valuable, although maybe it is too simple to fill a whole column.

Steve Schonberger
Via e-mail

Chris Hecker responds:
Some people have vertical and horizontal rasterizers and they use the vertical ones for

polygons that are more wall-like and the horizontal ones for polygons that are more floor-like. This lets you subdivide less if you're doing adaptive subdivision (since your "canlines" are closer to the lines-of-constant Z, so there's less warp along them), but it means you have to integrate both rasterizers in your code.

ANOTHER SATISFIED CUSTOMER

Dear Editor:

The magazine is great! I don't read anything but *Game Developer*. It's my one-stop information magazine!!!

Anonymous

INQUIRING MINDS WANT TO KNOW

Dear Editor:

I was reading Michael J. Norton's article, "The Mode X-Files" in the Oct./Nov. 1995 issue of *Game Developer* magazine and was wondering how I would get in contact with him. He stated in the article that he was working on a book for programming the Windows 95 SDK. I'd like to find out when he thinks the book will be done and who will be publishing it. I've been looking for a book like this for a while now.

Robert Hildebrand
Via e-mail

Michael Norton responds:
I'm glad to hear there is interest in a Microsoft Windows 95 Game SDK book. This summer, Spells of Fury *by Waite Group Press, will be reaching the shelves of your local computer book store. This is a culmination of a year of work on a very exciting project. As a professional in the industry, I pulled together my resources to form a development team including programmers, artists, and musicians to put together ten fully functioning DirectX games. The chapters devoted to these games go through every inch of code with a magnifying glass. Source code in these game examples were evaluated by professionals at Microsoft and other software engineers in the game industry for accuracy.*

The subject matter in Spells of Fury *will satisfy a broad range of programmers, from the novice to the professional developer—no stone is left unturned.*

ERRATA

Quality is job one here at *Game Developer* magazine. We strive to correct any mistakes we've made and give any additional information that may be helpful in providing you with the best product we can.

- In "Bit Blasts" (Feb./Mar. 1996), we told you that you could order a trial demo CD of Caligari's TrueSpace2 for $14.95. One reader, David Scarbrough, informed us that at ftp.caligari.com, you can download it for free. The directory is /pub/trueSpace, and the two files are ts2trial.txt and ts2trial.zip. Anonymous login is allowed; you can use your e-mail address for your password.
- In Chris Hecker's "Let's Get to the [Floating] Point" (Feb./Mar. 1996),there were two exponent typos. On page 20, it says, "so we can represent numbers as large as 2^{1000} and as small as 2^{1000}." The second exponent should be 2^{-1000}. On page 22, it also says, "What happens if we add in 223...." The number should read 2^{23}.

Our Readers

This month, our readers search for books, examine lines-of-constant-z, and explain their feelings about Game Developer magazine and its staff.

What's the Buzz?

Diane Anderson

What's new in the world of developing games? Check out products that capture motion, that model animation, and that accelerate it. And a word from the Gossip Lady mourning the loss of Eden at Apple.

At press time, we are looking forward to the Computer Game Developer's Conference. Here are some products to check out while you are there.

Capture Polhemus

Polhemus is building on its motion capture system; Ultratrak Pro is its new version. It incorporates DSP technology and is targeted toward game developers, movie studios, and production houses involved in animation and special effects. It uses real-time, six-degree-of-freedom data from numerous sensors. Data can be processed in real-time for live broadcast or virtual set applications. Using the Long Ranger transmitter, Ultratrak Pro provides a working area of more than 700 square feet. Drivers are available for most major software packages including Alias/Wavefront, Softimage, Side Effects, 4DVision, and Hash Animation. Ultratrak Pro is an integrated system free of DIP switches, in a 19-inch rack mountable chassis. Data is transmitted to a workstation over an Ethernet or SCSI interface or may be saved to the hard drive. Pricing depends on receiver and Hz configuration and ranges from $23,500 to $71,500.

■ For more information contact:
Polhemus Inc.
1 Hercules Dr.
Colchester, Vt. 05446
Tel: (802) 655-3159
Fax: (802) 655-1439

Stealth 3D 2000

Diamond announces Diamond Stealth 3D 2000, a new multimedia accelerator that delivers 3D animation, 2D graphics, and digital video playback acceleration for PCs running Windows 95. It uses S3's triangle-based polygon rendering engine and features perspective-corrected texture mapping, bi-linear filtering, MIP-mapping, alpha blending, depth-cueing, and fogging. It comes standard with 2MB of DRAM that is split between display memory and Z-buffer memory, but can be upgraded to 4MB of DRAM for better Z buffering, texture mapping, and better acceleration of 3D applications at higher resolutions. The Diamond Stealth 3D 2000 features full-motion digital video playback at 30 frames-per-second by off-loading functions from the host CPU. The Diamond Stealth 3D 2000 with 2MB EDO DRAM for the PCI-bus costs $300. Upgrades, such as the MPEG Video Player 1100 daughtercard or the Diamond DTV 1100 TV tuner, are available from Diamond as well.

■ For more information contact:
Diamond Multimedia
2880 Junction Ave.
San Jose, Calif. 95134-1922
Tel: (408) 325-7000
Fax: (408) 325-7070
Web: http://www.diamondmm.com

Softimage 3.0

Microsoft is now shipping a Windows NT version of its 3D modeling and animation software, which offers the same animation environment as Softimage 3D for Silicon Graphics. Softimage 3D for Windows NT requires Windows NT 3.51 (with Service Pak 2 installed), a minimum of 64MB of RAM, 1GB of hard-disk space, and a supported OpenGL

graphics accelerator card. Softimage for Windows NT is optimized for several turnkey hardware configurations. Softimage 3D for Windows NT costs $7,995 through the Softimage network of VARs. Owners of version 2.66 or earlier can upgrade for approximately $1,995 (hardware not included). Softimage also announced support for Silicon Graphics Indigo2 Impact workstations, adding to the list of certified configurations.

■ For more information contact:
Microsoft
1 Microsoft Wy.
Redmond, Wash. 98052
Tel: (800) 576-3846
Fax: (206) 936-7329
Web: http://www.softimage.com

Diane Anderson is managing editor of Game Developer *magazine.*

Schmooze news...

Multimedia Shakeout Intensifies...

Digital Pictures axed 30 people a couple of weeks ago. Sanctuary Woods drop-kicked their CEO (Scott Walchek) and laid off 20 of their 100 people. Guess Director games didn't sell too well over Xmas! After lulling Compton's game people with promises that it would continue producing "quality" games, SoftKey fired virtually the entire game development crew. Reports are that the few who received offers to stay opted to go looking for greener pastures anyway.

Death Threats from Disk Pirates in South China

The organization that represents the global music industry has closed its Canton, China operation after staff learned that local CD pirates had hired hit-men in an imaginative anti-anti-piracy move. Hopefully U.S. and Canadian pirate shareware publishers won't get ideas!

The Miles Drivers Boogie

RAD Software acquired John Miles's AIL sound library, renamed it to "Miles Sound System" and intends to extend it to the Mac and PowerMac.

Speaking of Macs...

Apple Computer Inc. seems to be withering on the vine. With a huge loss for the last quarter and CEO Spindler's departure, things are looking pretty bleak for the hardware and software manufacturer. One comment heard on the news: "Is the Macintosh going to become the next eight-track tape player?" One thing is certain: now is a REALLY lousy time to be an Apple employee. Layoffs are looking likely. Industry analysts are predicting up to one quarter of the Apple employees will receive pink slips in a major reorganization. The moral of the story here is: "Evolve or die."

The sad part is that if Apple had had smarter management, they'd be in Microsoft's position right now. Instead, they're in the situation that if they had sold off their manufacturing division and invested the proceeds in Microsoft stock, they would have had a much better year. The Gossip Lady predicts that Apple will be sold in the next year, and that they'll become a superb software division for a major hardware company.

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

Perspective Texture Mapping, Part V: It's About Time

Chris Hecker

Finally! The moment game developers have been waiting for. At long last, Chris Hecker unveils the denouement of his texture mapping series. What a long, strange trip it has been.

Check the date on the *Game Developer* in your hands. Does April/May 1996 mean anything special? How about the same issue last year: April/May 1995? What, you have no idea what I'm talking about? That's understandable, since it was so long ago. I'll refresh your memory with a small quote from my column in that 1995 issue: "My next two articles should fix this lack of documentation, first by giving an easy-to-understand mathematical foundation…and sample code to implement the naive algorithm. In the next article, we'll speed it up to interactive performance."

Yes, you guessed it, the April/May 1995 issue contained the first installment of my "two-part" perspective texture mapping series. Now, a year later, we're finally going to finish the two part series with this issue, part five. True to my software engineering background, I can't estimate how long it will take me to do something to save my life. However, I feel the somewhat lengthy trip has been worth it.

A Long, Strange Trip

Unlike the first installment, this article will not be a journey through the elegant theory and math behind perspective texture mapping. Instead, this article will romp through the myriad optimization tricks and techniques we can use to squeeze every last bit of texture mapping performance from the Intel Pentium processor. While the rest of the series was platform independent, we have to turn to assembly language to get the most out of modern processors, and the Pentium is the market leader (and the CPU I know best). You'll still get a lot from reading this regardless of your chosen platform, but the code is specific to Intel. However, I got a new Macintosh clone from Power-Computing with a PowerPC 604 chip in it, so don't be surprised to see a PowerPC version of this texture mapper at some later date (as soon as I get used to the concept of 32 general purpose registers)!

To quickly bring us up to date, we decided to use a subdivided affine approximation to the true perspective curve (Behind the Screen, Dec./Jan. 1995). This approximation uses short linear spans with perspectively correct endpoints to approximate the rational perspective curve. I modified the `DrawScanLine` function to do the affine subdivision in C++ and the texture mapper got three times faster than the version with the divides. I uploaded a sample application that contains the texture mappers we developed so far. Of course, I was late in uploading it (see above comment about saving my life), and I apologize to anyone who looked and couldn't find it. The sample is up now though—see the end of this column for information on where to find it.

I can provide an overall outline for the optimizations we'll cover before going into detail. Our main optimizations will take advantage of the dual integer pipelines on the Pentium to implement a very fast fixed-point linear texture mapper for our affine spans, and we'll use the floating-point unit's ability to overlap execution (especially those costly perspective divides) with integer instructions to calculate the interpolation values for the next span as we render the current span.

Listing 1. The C++ Inner Loop

```
for(int Counter = 0;Counter < AffineLength;Counter++)
{
  int UInt = U>>16;
  int VInt = V>>16;

  *(pDestBits++) = *(pTextureBits + UInt +
      (VInt * TextureDeltaScan));

  U += DeltaU;
  V += DeltaV;
}
```

In addition, we'll pull a bunch of cheap tricks along the way to give it that extra kick.

Carry Me Away

Listing 1 shows the C++ linear inner loop for our `DrawScanLine` function. While this is better than our previous loops with their divides, it's still not great because it's doing a multiply and a bunch of shifts and adds for each pixel. If we look at why it's doing a multiply, we see it's calculating the source texture offset for each new coordinate, even though we're just doing a normal linear interpolation through the texture for each span. By definition, a linear interpolation increments by the same amount each step, so we can take advantage of this coherency to speed up our loop.

Let's ignore the V coordinate for the moment and see how we can take advantage of the U coordinate's coherency. As you can see from the loop, the U fixed-point variable is shifted down to extract its integer portion for each pixel, which is then added into the texture pointer. However, since we're linear, the integer portion of `DeltaU` is going to stay constant for the span—always incrementing the integer part of U by the same amount—so there's really no need to keep the integer portion of U around at all. If we think only of the U increments, we can keep the source pointer at the current texture pixel, and we can find the next texture pixel by just adding in the integer part of `DeltaU`—calculated outside the loop—to the pointer at each step. The only problem with this plan is the fractional part of U plus the fractional part of `DeltaU` will sometimes carry into the integer part of U. We need to know when this happens and add an extra step to our source pointer.

This carry problem uncovers a major hole in C and C++ from the standpoint of integer optimizations: there's no carry bit. In other words, in assembly language, it's trivial to know when two added numbers overflow because the carry bit will be set, but in C++ there's no way to know without doing a bunch of cumbersome tests. So, in pseudocode, we want to do this:

```
UFrac += DeltaUFrac
pTexture += DeltaUInt + Carry
```

Where the variable `Carry` is set to 0 if the fractional addition didn't carry into the imaginary integer part (that we're not storing anymore) and is set to 1 if the addition did carry. This pseudocode is trivial in all the assembly languages I've ever seen, for example, in x86 assembly:

```
add     ebx,ecx
adc     esi,edx
```

Assuming the given registers contain the right values, the `adc` (add with carry) will add in the step and any carry from the previous addition. Implementing this code in C++ would be a mess.

That's it for the U coordinate, but we conveniently ignored the V coordinate because it's a good deal trickier. Like U, the V coordinate is linear for our span, so we can precalculate our increment and leave the integer part of V out of our loop. However, as you can see from Listing 1, the integer part of the V coordinate is scaled to step vertically in our texture bitmap. This doesn't present a problem for the normal V step, but when the fractional part of V carries into the integer part, the source pointer no longer steps by 1, it steps by the width of the texture bitmap. Not even assembly language has an instruction to add in an arbitrary number—like the `TextureDeltaScan` in Listing 1—on carry.

Quickly adding in the vertical source step on V's carry is where 99% of the programming brainpower is spent on linear texture mapping optimizations. I've seen about five or six ways of doing it myself, but by far the coolest, fastest, and most elegant way I've seen was invented by Michael Abrash. However, before I describe it, I'm going to address the optimization a lot of the experienced texture mappers in the audience think I'm going to use here.

If you go out on the Internet and look for affine texture mappers, you'll undoubtedly run into a lot of very optimized x86 code that only works with power-of-two source texture sizes, and specifically two to the eighth power (or 256-bytes wide for 8bpp textures), because if you keep your textures to a power-of-two width, you can very easily handle the V carry we're discussing using some special x86 instructions that operate on 8-bit portions of the full registers.

Let's run through an example, where our source texture is 256 by 256. We'll use the x86's `ebx` register, and its corresponding "byte registers," `bh` and `bl`. The byte registers are part of the 32 bit `ebx` register, and `bl` (b-low) is the lowest 8 bits—bit 0 through 7—and `bh` (b-high) is the next higher 8 bits—bit 8 through 15. If we keep the U coordinate in `bl` and the V coordinate in `bh`, we can use the following code to increment both U and V (assuming `ecx` and `edx` have the current `UFrac` and `VFrac`, respectively, and `esi` contains the texture pointer):

```
add     ecx,[DeltaUFrac]
adc     bl,[UIntStep]
add     edx,[DeltaVFrac]
adc     bh,[VIntStep]
add     esi,ebx
```

Notice the second `adc`. It adds in the carry from the `VFrac` addition, but I just got finished saying how this wouldn't

Listing 2. The x86 Asm Inner Loop

```
add     edx,[DeltaVFrac]        ; add in dVFrac

sbb     ebp,ebp                 ; store carry
mov     [edi],al                ; write pixel n

mov     al,[esi]                ; fetch pixel n+1
add     ecx,ebx                 ; add in dUFrac

adc     esi,[4*ebp + UVStepVCarry]      ; add in steps
```

work because V's carry needs to add in the width of the texture. The trick is that bh is actually already multiplied by the width of our texture—256—by virtue of its bit position in `ebx`. Neat, huh? Now, given such a cool trick, why wouldn't we use it? There are two reasons: first, restricting yourself to power-of-two textures isn't very flexible and is bad for cache coherency (see Behind the Screen, Oct./Nov. 1995). More importantly, with the Pentium Pro, this code will run slowly due to a new pipeline stall called the Partial Register Stall (PRS). The PRS happens when you modify one of the byte registers and then try to use the encompassing 32-bit register, much like the above code. The instruction `add esi,ebx` will stall for a very long time on the Pentium Pro. Why did Intel let this happen? I have no clue, although they say it will let them increase the clock speed more than if they had prevented the stall. Regardless, it's there, and we'll need to live with it.

So, given that we're not going to use the power-of-two texture trick, how do we write our code so it can carry an arbitrary value into the pointer when `VFrac` overflows? Enter Abrash's code snippet shown in Listing 2. This is the code for a pixel from the middle of an unrolled loop, so there's a bit of setup not shown here, but imagine this same snippet concatenated with itself a bunch of times. See if you can figure out how it works and then read on for the description of this tour de force of optimization.

Hit the Pipe

There are so many cool things about this code it's hard to know where to start describing it, but, since we were discussing the V carry, we'll start with how the code addresses that problem. The first half of the solution is these two instructions:

```
add     edx,[DeltaVFrac]
sbb     ebp,ebp
```

The first instruction adds in the fractional step as usual, but the second instruction saves the carry flag, using a neat trick involving the `sbb` (subtract with borrow) instruction. The `sbb` instruction is like the opposite of `adc`, it subtracts its source from its destination, but also subtracts the carry bit, so `sbb ebp,ebp` will subtract the `ebp` register from itself, giving 0 if there was no carry, or -1 if there was a carry. Thus, the carry bit from the `VFrac` addition is stored as a 0 or a -1 in `ebp`.

The second half of the solution comes with these instructions:

```
add     ecx,ebx
adc     esi,[4*ebp + UVStepVCarry]
```

The first instruction is the `UFrac` addition, so after it completes, the carry bit is set appropriately. The next instruction is where all the action occurs. It's an `adc`, so it adds in the carry from the `UFrac` addition as you'd expect. However, it's an `adc` from memory, and it uses a two `dword` array to accomplish its magic. `UVStepVCarry` is the address of the second `dword` in the array, and the 0 or -1 in `ebp` from the `VFrac` carry will select either the second `dword` if there was no V carry, or the first `dword` if there was a V carry (since 4 • -1 will subtact 4 bytes from the array address). The only thing left is to make sure the array has the appropriate steps in it, including the U and V integer steps and the V carry step for the first element in the array.

As if the basic operation wasn't good enough, the pipelining on this code is amazing as well. The order of instructions perfectly fills both pipes on the Pentium and manages to run the two additions from memory—both two-cycle instructions—in the Pentium U and V pipes at the same time (remember, the next pixel's code will come right after this pixel, so the `add edx` and the `adc esi` will run at the same time). The instructions are also far enough away from each other that there are no Address Generation Interlocks. Overall, it's a beautiful piece of code.

Walking and Chewing Gum

Regardless of how amazing our integer affine inner loop is, we'll still be slower than we need to be if we're waiting for the floating-point unit to calculate the perspective-corrected texture coordinates before starting each span. This is where the floating-point overlap I hinted about in the last issue enters in. Most modern processors can execute floating-point instructions at the same time as integer instructions, and some, like the Pentium, can execute multiple floating-point instructions at the same time. As an example of integer and floating-point overlap, the following code will take 36 cycles on a Pentium in double precision mode:

```
fdiv    [Number1]
fst     [Number2]
```

The division takes 33 cycles, the store takes two cycles, and there's a one-cycle stall for trying to store the result of the division right after it's completed. Guess how long the following code takes:

```
fdiv    [Number1]
rept 33
        add     ebx,ecx
        add     edx,eax
endm
fst     [Number2]
```

To guess correctly, you need to know that the `rept` macro repeats the contained code 33 times, so there are actually 66 instructions between the `fdiv` and the `fst`. This is actually a trick ques-

tion because this code takes the same 36 cycles as the first snippet, but we got to execute 66 integer instructions for free!

Well, we don't really get just any 66 instructions for free, but we do get 33 U and V pipe slots in which we can try to get some work done before using the result of the division. Some instructions, like integer multiplies, won't overlap with the floating-point unit, and you can't really do many other floating-point instructions at the same time as floating-point division, but we can start up the perspective divide for our next span and have it calculate as we're processing the current span, making it almost free.

Listing 3. Naive Adding

```
fld   [a1]        ; 1
fadd  [b1]        ; 3
fstp  [a1]        ; 2+1
fld   [c1]        ; 1
fadd  [d1]        ; 3
fstp  [c1]        ; 2+1
fld   [e1]        ; 1
fadd  [f1]        ; 3
fstp  [e1]        ; 2+1
```

Short Stack

The second floating-point technique I mentioned, executing multiple floating-point operations simultaneously, is slightly more convoluted. The Intel floating-point architecture is stack based, which means almost all the floating-point instructions will only operate on the top of the stack. This made it hard for Intel to pipeline the floating-point unit for the Pentium since all the instructions were vying for the same register—the floating-point top-of-stack register. So, instead of breaking all the existing floating-point code by making a bunch of new instructions to randomly access the floating-point registers, Intel decided to make it possible to move operands around on the stack very quickly. I actually wish they'd broken the code and made random register access easy, but Intel doesn't usually ask my opinion on these things, so I'll quickly describe the stack-based solution.

Listing 3 shows the obvious way to add some numbers together, along with the cycle counts for each instruction. It's implementing this C++ code:

```
a1 += b1; c1 += d1; e1 += f1;
```

The code executes in 21 cycles, including the three stalls (the 2+1 `fstp`

Listing 4. Quick Adding

```
fld   [a1]        ; 1
fadd  [b1]        ; 1
fld   [c1]        ; 1
fadd  [d1]        ; 1
fld   [e1]        ; 1
fadd  [f1]        ; 1
fxch  st(2)       ; 0
fstp  [a1]        ; 2
fstp  [c1]        ; 2
fstp  [e1]        ; 2
```

timings) for storing the results of an operation immediately following its completion. Listing 4 shows an alternate implementation of the same code, which executes in 12 cycles, or almost twice as fast. The instructions can pipeline if you don't access their results before the instruction is finished (the `fld` instruction pushes its operand onto the stack, and the previous `fadd` continues on its operand even as it's moved down one stack position). In our example, the `fadd`s take 3 cycles each in Listing 3, but only a single cycle each in Listing 4.

The second thing to notice is that the `fxch` instruction is free in Listing 4. This is Intel's offering to the angry God of Processor Architecture, who threatened to smite Intel dead if it didn't pipeline the floating-point unit. The almost-free `fxch` instruction makes it possible—not easy, just possible—to pipeline your floating-point code even though most of the instructions only operate on the top-of-stack register. Using `fxch`, you can move things around while they're still calculating, like the `e1+f1` addition in Listing 4. I called it "almost-free" because there are some restrictions you have to obey to keep it free; for example, the following instruction must be a floating-point operation, as it will stall a cycle if the following instruction is an integer operation. Intel's AP500 Application Note, available on their www.intel.com site and the Intel Architecture Labs CD, describes this technique in detail.

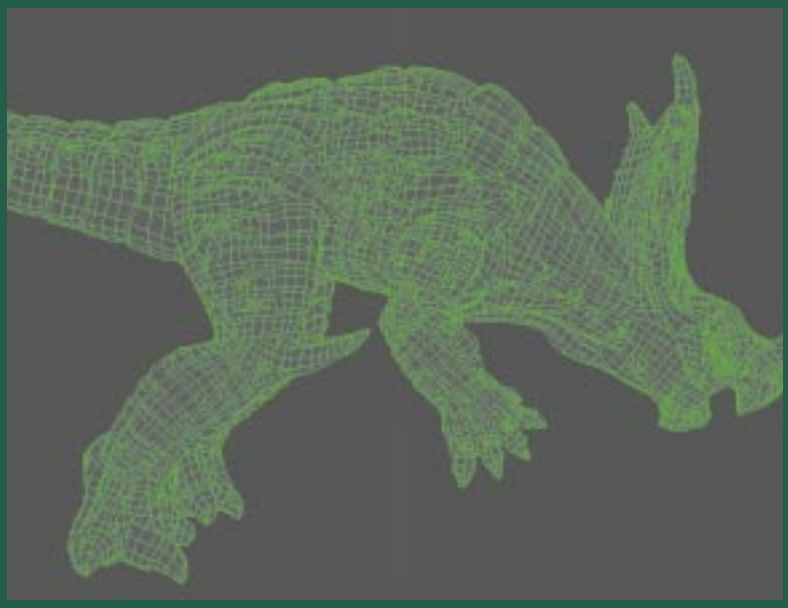
Finally

There are more tricks in the assembly texture mapper that deserve a mention, but they're all minor and I'm out of space. You can find them in the code itself. As with most assembly code, the texture mapper is way too long to include here in the magazine. You can, however, get it in the texture mapping archive on the *Game Developer* web site, on its ftp site (ftp.mfi.com/pub/gamedev/src), or on my homepage at http://ourworld.compuserve.com/homepages/checker.

How fast is it? Well, I must admit, I'm not finished optimizing it yet as I write this (again, see the comment about estimating how long it takes me to do something at the beginning of this article), but it's already two times as fast as the C++ subdividing affine texture mapper, and I hope to make it another two times faster by the time you read this and are able to pick up the code. It's currently drawing 4.5 million pixels per second on a Pentium 133, which is 5 times faster than our original texture mapper, and fast enough to do 70 frames per second at 320 by 200 if you're not doing anything else except texture mapping. That's definitely fast enough for a high-end 3D game, and I think we can safely say we've met the goals we set for ourselves at the beginning of this series, even if we did meet them a bit late. ■

You can e-mail Chris Hecker at checker@bix.com. Don't be surprised if it takes him a while to respond, although he'll assure you it will only take a second....

Data Capture of 3D Models



A wireframe model of Armadon, one of the beasties in the Primal Rage game. Every character in the game started out as a sculpture, which was then digitized. A coordinate mapping system ensured realistic movement.

Afficionados of the popular arcade video game, Primal Rage, are well acquainted with the combatants—Sauron, Talon, Blizzard, Vertigo, Diablo, Chaos, and Armadon. These seven prehistoric fantasy creatures are some of the largest fighting game characters, and each has a distinct personality and fighting style.

When Time Warner Interactive Inc. in Milpitas, Calif. decided to recreate Primal Rage for play on home video game equipment, it was imperative that nothing about the well-known characters changed as they made the transition to new hardware. Maintaining consistency across platforms was a challenge, however, because graphic imaging requirements for each game device differ. For example, the resolution available in a CD-ROM version, with its prerendered images, is far superior to that of the arcade game which renders scenes in real-time. Consequently, CD-ROM creatures can display more detail and move more realistically.

To ensure that Sauron, Talon, and company made an accurate transition to the new PC CD-ROM version of the game, Time Warner hired 3Name3D, a Santa Monica, Calif.-based geometric modeling company. 3Name3D sculpted each of the Primal Rage creatures, then digitized the sculptures using the Faro Arm, a portable coordinate measuring machine from Faro Technologies in Lake Mary, Fla.

According to 3Name3D CEO, Sandeep Divekar, this approach assured his company of getting truly realistic recreations of the creatures. “Objects that must be replicated exactly must be digitized or scanned rather than modeled on screen using CAD or other modeling software,” says Divekar. “We chose a coordinate measuring machine to control where we captured the x, y, z coordinates. We chose the Faro Arm because the Primal Rage sculptures we created were up to three-feet long, and we needed its large working envelope.”

Three Architects

3Name3D was founded by three architects—Oscar Yglesias, Steve Wallock, and Divekar. The company creates digi-

Caren D. Potter

How did Time Warner move critters to a new platform? A portable coordinate measuring machine was key to replicating the arcade game characters for a PC CD-ROM version of Primal Rage.

tal models for clients in a variety of industries and offers a collection of ready-made computer models.

For the Primal Rage project, Time Warner supplied 3Name3D with the original nine-inch latex models that had formed the basis for the arcade game characters. To create the arcade game, these fully jointed models had been filmed using the stop-motion technique, in which a series of painstakingly small movements are recorded. Stop-motion would not work for the PC CD-ROM game, according to Divekar, because it would not supply the sufficient level of detail or the highly realistic motion Time Warner wanted.

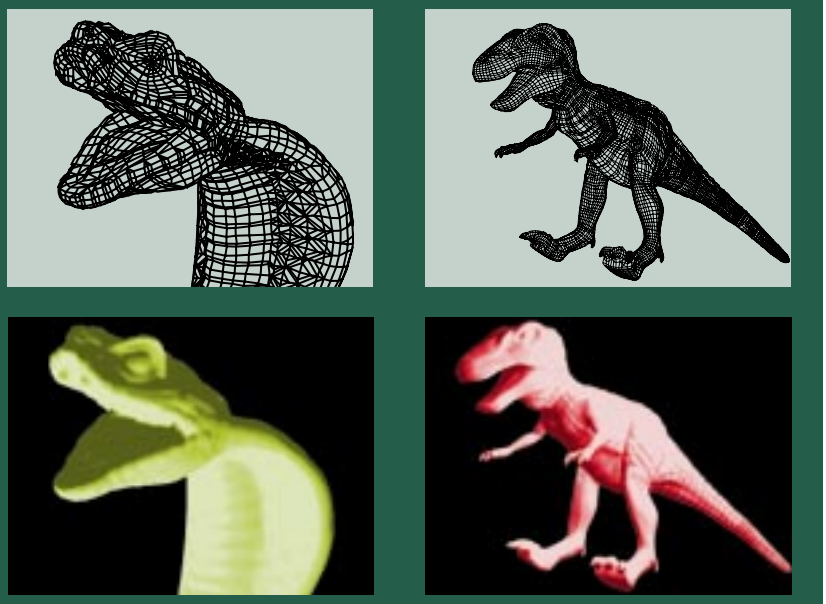
3Name3D's first step was to create its own set of sculptures of the Primal Rage creatures. They made them bigger to capture more detail but otherwise they were identical to the original latex models. With the exception of Blizzard (a gorilla), the new sculptures were constructed of the modeling clay, Sculpey. Sculpey couldn't support Blizzard's long arms, however, so Blizzard was sculpted in clay. A latex mold was made of the clay, then Blizzard was cast in plaster of paris with armatures supporting his arms.

Once the sculptures were complete and approved by Time Warner, the process of recreating them as digital models began. First, grid lines were placed on the sculptures to indicate the locations of the points (x, y, and z coordinates) to be entered into the computer. Coordinate location was a critical issue because the models were going to be animated. "This meant they had to be broken at joints, with each joint positioned precisely so the movement would look realistic," explains Divekar. (Imagine a knee positioned mid-way up the thigh. The motion of that leg would not look very realistic.)

Because grid line placement affected the location of the joints and 3Name3D needed precise control over these lines, laser scanning was ruled out as a method of capturing the coordinates. "If a model is going to sit off in the corner of a digital set and not move, it doesn't need joints and, in that situation, laser scanning is fine," says Divekar. "But laser scanners pick up point locations in a regular pattern. A jointed, moving model requires irregular grid lines. With a coordinate measuring machine, grid lines can be placed in response to the animator's needs. The user tells it where to capture coordinates by pointing at them and then clicking the probe."

Although on previous projects 3Name3D had used another company's small 3D digitizer, its probe was limited to a sphere of 12 inches. The Primal Rage models were three-feet long. To get the reach they needed, 3Name3D bought a portable coordinate measuring machine with a six degrees of freedom arm and a spherical working envelope of eight feet.

Called a "liberated" coordinate measuring machine, the Faro Arm used by 3Name3D is not restricted to three axes. Unlike some other digitizing methods, there are no line-of-sight limitations or restrictions on digitizing metal objects. Although generally used in labs, engineering offices, and manufacturing facilities, computer artists are now starting to use it. The Faro Arm features



Before and after pictures of Vertigo (left) and Sauron (right). The wireframe images appear on top and the rendered versions are below.

direct serial port input into AutoCAD and other modeling packages and supports standard output formats such as IGES, ASCII, DXF, and ACL.

One-Person Digitizing

Often, digitizing large objects is a two-person job—one person operates the probe while the other watches the computer screen to make sure data is captured correctly. However, 3Name3D needed only one person to digitize the Primal Rage creatures because the software they used to capture input from the arm, Mira Imaging's HyperSpace, incorporated audio feedback.

Explains Divekar, "HyperSpace, which we run on a Macintosh computer, uses audio cues to provide feedback to the operator. When a point is entered, it signals the computer to emit one kind of sound. If the point has to be locked on to a previous point, the computer makes a different sound when the two points are identified. The person who did the digitizing of Primal Rage sculptures, who was also the artist who had created them, spent very little time looking at the screen."

HyperSpace was also the software used to create the surfaces over the points. After the points representing a certain area of the sculpture were entered, the software was given the command to place a skin over them.

When the model was completely digitized and surfaced, the HyperSpace file was transferred to Wavefront's Advanced Visualizer software running on a Silicon Graphics workstation. There, it was cleaned up and broken down into logical groups to make joints. The last step was translating the Advanced Visualizer file to 3D Studio file format. (3D Studio was the modeling and animation software used by Time Warner.)

It took 3Name3D about four days per creature to complete the process of digitizing a sculpture and adding surfaces to the computer model. The entire operation, from making a sculpture to creating a fully surfaced, jointed digital model, took about 10 days per creature, and 3Name3D modeled five of the seven Primal Rage creatures.

A final screenshot of Talon vs. Blizzard.

Digitizing the models.

For 3Name3D, the Primal Rage project has led to additional work, in part due to the use of a digitizing arm. "There's no way we could have done this without one," says Divekar. Game artists interested in rapidly creating complex shapes would do well to consider the advantages of sculpting and digitizing models rather than creating them entirely digitally. The availability of large envelope, six-degree-of-freedom digitizing arms makes this a valuable cost-cutting option.

The PC CD-ROM version of Primal Rage was released on August 25, 1995 along with versions for other home game equipment such as the Sega Genesis, Super Nintendo Entertainment System, and Nintendo Game Boy. Time Warner has already shipped more than one million units of these new releases, and appears to have another hit on its hands. ■

Caren D. Potter is a freelance writer living in McKinleyville, Calif. She has been writing about computer technology, in its various forms, for 10 years. Contact her via e-mail at cpotter@northcoast.com.

The Z Buffer Algorithm

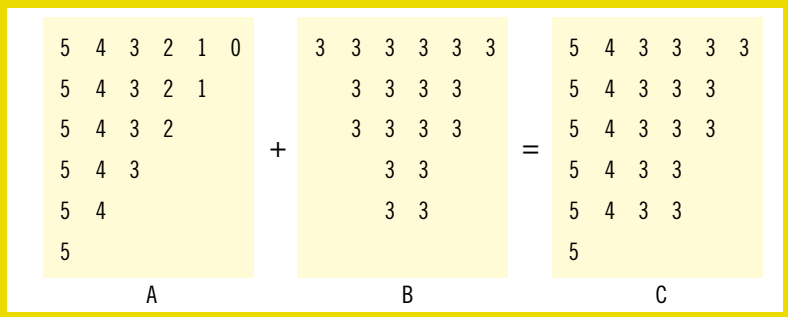


Figure 1. The result of Z buffering polygon A and B is shown in Box C.

Years ago—when I got my feet wet in the virtual world of 3D computer graphics—I stumbled upon an algorithm. This algorithm, in case the title of this article hasn't completely spoiled the surprise, is popularly known as the Z buffer algorithm and deals with the slippery problem of visual surface determination. Visual surface determination and its converse, hidden surface elimination, are two approaches to solving the age-old problem of correctly rendering 3D surfaces on a 2D screen.

In the real world, mere mortals cannot see through opaque surfaces. However, in the world of 3D computer graphics, no such limitation exists. In fact, we 3D programmers go to great lengths to impose this limitation on our 3D worlds so as to model the real world more accurately. The only other alternative to imposing this limitation on our 3D worlds is drawing all polygons in the dataset in an arbitrary order, ignoring their 3D properties, which will almost guarantee that some distant surfaces will show through closer surfaces, producing a surrealistic effect that is generally not desirable.

One method of dealing with this annoying problem of distant objects showing through closer ones is to draw all surfaces (polygons) in a back-to-front order. If done properly, this assures us that the resulting view will be rendered accurately. Known as the Painter's or Depth Sort Algorithm, this approach has three serious flaws: first, the amount of time it takes to sort the surfaces increases exponentially with regard to the number of polygons in the dataset; second, precious time is wasted rasterizing pixels that will be over-written further down the list; finally, surfaces cannot penetrate each other (such as a boat in water, or aircrafts soaring through clouds). Though you can probably overlook the second and third deficiencies, the first one is tough to swallow. In large datasets, the amount of sorting and testing that must be performed is prohibitively complex—rendering this method useless for many modern applications. Thus, many developers of 3D software have turned to two more flexible and faster approaches: BSP (Binary Space Partition) trees and Z buffering.

A BSP tree is a collection of polygons listed in an order based on the relative position and orientation of all polygons to each other. Typically, the creation of this BSP tree is rather time-consuming, and, therefore, a BSP tree compiler creates it in advance. Though this method provides extremely quick sorting, it suffers from other problems of the Painter's Algorithm and imposes a limitation of its own—it restricts the

dataset to static objects. The static requirement of the BSP tree has prompted a number of developers to search for ways to integrate moving objects into a BSP tree without performing time-consuming calculations, most of which are less than successful.

With the rise of faster processors, more memory, and video cards with onboard Z buffers, the Z buffer algorithm is starting to look attractive to many developers simply because it overcomes all flaws of the Painter's Algorithm (save number two, where a number of adaptations may be used while maintaining visual accuracy). The Z buffer algorithm is an image-precision algorithm, meaning it deals with visual surface determination at the image level, requiring an increase in memory for an increase in resolution. With 8MB of memory standard on today's machines and 16MB common, this memory requirement is somewhat less than restricting, especially considering the number of problems the Z buffer eliminates.

The Z buffer is, as implied by its name, a linear array of Z (the depth coordinate) values. Each pixel in the viewport corresponds to an element in the Z buffer. This array is designed to hold the Z values of all the visible polygons in the viewport. When rasterizing a polygon, before each pixel is written to the viewport, the corresponding Z value in the Z buffer is checked against the Z value of the pixel being rasterized. If the Z value of the current pixel is closer to the viewpoint than the Z value previously stored in the Z buffer, the current Z value is stored in the Z buffer and the pixel rasterized; if not, the process continues. (See Figure 1 for illustrations.) As you might have guessed, this means the Z buffer must be cleared (reset) every frame to some number (the farthest possible Z value), though a later optimization removes this requirement.

The only apparent difficulty we might have is determining the Z (depth) at each point, which is not as hard as it sounds. The equation for a plane or polygon is as follows:

$$Ax + By + Cz + D = 0$$

Solving for Z gives us:

$$Z = (-Ax - By - D) / C$$

However, we can compute Z incrementally for coplanar surfaces, which is fortunate for our sakes because it saves us from solving the above at each pixel. Once the value of Z has been determined (which we'll call Zc), the value of Z at the following pixel is:

$$Zc - (A / C)$$

A single subtraction per pixel! As fate would have it, however, we'll mostly be interpolating $^1/_z$ values instead of direct Z values. The reason is simple: $^1/_z$ is linear in screen space, and can be interpolated using a linear equation, whereas Z is linear in 3D space, and cannot be linearly interpolated in screen space. Since this means our Z values are inverted, we must clear the Z buffer to zero each frame, which represents a literal infinity: the farthest possible $^1/_z$ value. On the positive side, interpolating

John De Goes

If you are involved with 3D graphics, you are familiar with the problem of visual surface determination. The Z buffer algorithm can help you get around the problem.

Listing 1. Classes to Make a Z Buffer (Visual Surface Determination Algorithm Implementation)

```
a#include <Math.H>
#include <Windows.H>

// The precision to the right of the imaginary decimal point:
const ZPREC = 26; // s00000.00000000000000000000000000 = 0.000000015

class ZBuffer {
protected:
  long *ZBuff, ZTrans;
  WORD Init, Code;
  unsigned int Width, Height;
public:
  enum { NoMem = 0, Success };

  // Sets defaults:
  ZBuffer () : Init ( 0 ), Code ( NoMem ),
               ZBuff ( NULL ), ZTrans ( 0 ) { }

  // Accepts width and height for Z buffer creation:
  ZBuffer ( int W, int H ) : Init ( 0 ), Code ( NoMem ),
                             ZBuff ( NULL ), ZTrans ( 0 )
                             { Create ( W, H ); }

  // Deallocates memory:
  ~ZBuffer () { delete [] ZBuff; }

  // Initializes the Z buffer:
  inline BOOL Create ( int W, int H );

  // Width and height member functions:
  unsigned int GetWidth () { return Width; }
  unsigned int GetHeight () { return Height; }

  // Returns the 1/Z translate value:
  long GetZTrans () { return ZTrans; }

  // Returns the error code:
  WORD GetCode () { return Code; }

  // Returns a pointer to the Z buffer (use with caution):
  long *GetPtr () { return ZBuff; }

  // Performs a "safe", slow Z buffer write:
  // (Dest must be a 256 color buffer equal in dimensions
  //  to the Z buffer.)
  void Write ( WORD X, WORD Y, long Z, BYTE Val, BYTE *Dest )
     {
     unsigned int Index;
     if ( ( Init ) && ( X < Width ) && ( Y < Height ) )
        {
        Index = X + Y * Width;
        if ( ZBuff [ Index ] < Z )
           {
           ZBuff [ Index ] = Z;
           Dest  [ Index ] = Val;
           }
        }
     }

  // Performs a "safe", slow Z buffer read:
  long Read ( WORD X, WORD Y )
     {
     if ( ( Init ) && ( X < Width ) && ( Y < Height ) )
        return ZBuff [ X + Y * Width ];
     return 0;
     }

  // Performs an "unsafe", fast Z buffer write:
  // (Dest must be a 256 color buffer equal in dimensions
  //  to the Z buffer.)
  void FastWrite ( unsigned int Index, long Z, BYTE Val, BYTE *Dest
)
     {
     if ( ZBuff [ Index ] < Z )
        {
        ZBuff [ Index ] = Z;
        Dest  [ Index ] = Val;
        }
     }

  // Performs an "unsafe", fast Z buffer read:
  long FastRead ( unsigned int Index )
     {
     return ZBuff [ Index ];
     }

  // Clears the Z buffer to zero:
  // (Optionally accepts the number of completed
  //  frames to aid in the clear reduction algorithm.)
  BOOL Clear ( unsigned int FrameCount = 0 );
};

inline BOOL ZBuffer::Create ( int W, int H )
  {
  // Function creates a Z-buffer - can be called
  // in succession:
  delete [] ZBuff;
  ZBuff  = NULL;
  Init   = 0;
  ZTrans = 0;

  if ( ( ZBuff = new long [ W * H ] ) == NULL )
     {
     Code = NoMem;
     Width = 0; Height = 0;
     return 0;
     }
  Init = 1; Code = Success;
  Width = W; Height = H;

  // Clear the Z buffer:
  return Clear ();
  }

class ZEdge {
protected:
  long Z, ZStep;
public:
  ZEdge () : Z ( 0 ), ZStep ( 0 ) { }
  // Clip function for clipping a coordinate to boundary:
  void Clip ( int C, const B )
     {
     // Takes advantage of the fact that
     // ( ( a * ( b * c ) ) / c ) is equal to ( a * b ):
     if ( C < B )
        Z += ZStep * ( B - C );
     }
  // Init function for stepping along polygon edges:
  inline void Init ( float OneOverZ1, float OneOverZ2,
                     int Length, ZBuffer &ZBuff );
  // Init function for stepping along scanlines:
  inline void Init ( ZEdge &Left, ZEdge &Right,
```

Listing 1. (Continued from p. 40)

```
                    unsigned int Width );
  // Step operators:
  void operator ++ ( )     { Z += ZStep; }
  void operator ++ ( int ) { Z += ZStep; }
  // Function returns the current fixed-point Z value:
  long GetZ () { return Z; }
};

inline void ZEdge::Init ( float OneOverZ1, float OneOverZ2,
                          int Length, ZBuffer &ZBuff )
   {
   // Calculate the steps for a polygon edge:
   long FixedZ1, FixedZ2;
   if ( Length )
      {
      FixedZ1 = long ( OneOverZ1 * float ( 1 << ZPREC ) );
      FixedZ2 = long ( OneOverZ2 * float ( 1 << ZPREC ) );
      Z       = FixedZ1 + ZBuff.GetZTrans ();
      ZStep   = ( FixedZ2 - FixedZ1 ) / Length;
      }
   else {
        Z = 0; ZStep = 0;
        }
   }
inline void ZEdge::Init ( ZEdge &Left, ZEdge &Right,
                    unsigned int Width )
   {
   //Calculate the steps for a scan-line:
   if (Width)
      {
      Z  = Left.GetZ ();
      ZStep=(Right.GetZ () - Z) / Width
      }
   else {
        Z=0; ZStep = 0
        }
   }
```

Listing 2. Code for Z Buffer Classes

```
#include "ZBuffer.HPP"

BOOL ZBuffer::Clear ( unsigned int FrameCount )
   {
   // Function clears -- if necessary -- the Z-buffer to
   // zero (infinity):

   // Calculate the number of frames before each Z buffer
   // clear:
   unsigned int MaxWait = pow ( 2.0F, ( 31.0F - ZPREC ) );
   unsigned int N, Length, Rem;

   // Determine if a clear is in order:
   if ( MaxWait == 0 )
      Rem = 0;
   else Rem = FrameCount % MaxWait;

   Length = Width * Height;
   if ( Init )
      {
      if ( Rem == 0 )
         {
         // If clear reduction is enabled, there is no need
         // for an assembly Z buffer clear function as the
         // Z buffer is cleared only once per so many frames.
         for ( N = 0; N < Length; N++ )
             ZBuff [ N ] = 0;
         ZTrans = 0;
         }
      // Else no clear - adjust translate value:
      else ZTrans += ( 1 << ZPREC );

      // Return success:
      return TRUE;
      }

   // Return failure:
   return FALSE;
   }
```

$^1/_z$ allows us to use fast linear equations, while at the same time providing information necessary for perspective texture mapping.

Optimizations are also important. Two of the most blindingly obvious performance reducers of the Z buffer are the test per pixel and the actual clearing of the Z buffer. The clearing of the Z buffer cannot be avoided, but you can reduce the number of Z buffer clears you must perform by using 32-bit integers for the Z buffer. I call this optimization the clear reduction algorithm, and it has saved many a processor cycle on my own applications. Since the values in the Z buffer are never greater than 1, and because 1 is the closest possible value to the viewport, it is possible to fool the Z buffer into thinking that each frame is closer than the previous frame by adding a translation value to the $^1/_z$ terms, thus eliminating the need to clear the Z buffer every frame. You can think of it as a translation of all polygons, bringing them closer to the viewport each frame by an amount proportional to the maximum extents of the dataset. Though this may sound a bit difficult, it adds but a few lines of code.

To implement this optimization, for every new frame you should add 1 more to each of the $^1/_z$ terms than you added the previous frame, taking care not to exceed the limit of the data-type you are dealing with. By definition, the $^1/_z$ terms will never exceed a [0-1] range, meaning that adding an offset to these terms will effectively translate the dataset closer to the viewpoint by an amount equal to the translate value. The number of frames you can wait before you clear the Z buffer (and reset the translate value) will depend on whether or not you're using fixed-point or floating-point math. If you use fixed-point math, the number of frames will depend on where you place your imaginary decimal point. Traditionally, you'll have a `ClearZBuffer` function in your main loop, perhaps such as:

```
ClearZBuffer ( ZBuffer );
```

Using this optimization, that same portion of code might become:

```
if ( ( FrameCount % MAX_WAIT ) == 0 )
   {
   ClearZBuffer ( ZBuffer );
```

```
    ZTrans = 0.0F;
    }
else ZTrans += 1.0F;
```

Obviously, this example uses floating-point math, which is a definite no-no as far as fast 3D graphics are concerned. The `ZTrans` term is the value that is added to the $^1/_z$ starting terms at each of the polygon's vertices preceding rasterization. `MAX_WAIT`, is of course, the maximum amount of frames before the Z buffer is cleared, and `FrameCount` is the total number of frames that have been completed.

If much of what has been discussed regarding Z buffers has been review for you—with the exception of the clear reduction algorithm—you're probably wondering what you can do to eliminate the test per pixel that is required for an academic Z buffer. The solution is theoretically simple, but a pain to implement: Z buffer at the scan-line level instead of the pixel level. Using this approach, you never have to write a pixel more than once, and the Z buffer never has to be cleared. Note that, in such an implementation, the Z buffer becomes a linked list of scan-line information.

It's beyond the scope of this article to discuss the subject of proper rasterization, so I will provide a set of 32-bit, operating system independent classes instead of a full rasterizer. The classes can easily be added to just about any rasterizer. The classes, whose declaration appears in Listing 1 and whose non-inline code appears in Listing 2, implement a Z buffer algorithm in fixed-point mathematics and include all the necessary code for the clear reduction algorithm (see above). Of particular interest is the `ZBuffer` class's `Clear()` member function. This function optionally accepts the number of frames that have been displayed since the program's initialization.

If you do not wish to use the clear reduction algorithm, it is not necessary to provide this argument; however, providing it lets the function reduce the number of Z buffer clears that occur without sacrificing visual quality. In the `ZEdge` class, you will notice that one of the initialization functions accepts two $^1/_z$ values. You can calculate these values by taking the inverse of the Z values at the vertices of the polygon. The other initialization function accepts two `ZEdge` classes, which represent the sides of the polygon at a given scan line.

Typically, you will use one `ZBuffer` object per window, and three `ZEdge` objects for rasterizing polygons. One `ZEdge` object will represent the right side of the polygon being rasterized, another, the left side, and the third, the scan line being rasterized. If you perform polygon clipping at the image level, you will find the `Clip()` member function most helpful, which accepts a coordinate and the boundary for that coordinate. (The coordinate will be an X or Y point, depending on which axis you are clipping on; an X point for scan-line clipping, and a Y point for the top of the polygon clipping.) The `ZEdge` class overloads the `++` operator, so each time you step in a coordinate (that is, stepping down the left or right edge of the polygon or across the scan line), you can simply use this operator.

> You can be assured that the application's problems will not be related to the Z buffer classes.

For those curious souls among you, I decided to put multiple Z buffer read and write functions in the `ZBuffer` class because while debugging an application, you can use the slow, safe functions; when optimizing, you should be able to easily convert to the fast, unsafe functions. Your application can use all the help it can get during debugging, and since the safe functions cannot do anything harmful to your system memory (barring unequally sized Z buffers and screen buffers), you can be assured that the application's problems will not be related to the Z buffer classes.

One last word on the provided source code: the Z buffer write functions will actually write to the video buffer, but the buffer must be equal in dimensions to that of the Z buffer, and each byte in the buffer must represent a single pixel (with a range of 256 colors). In mode 13h, for instance, this is the case, as it is for many windowed applications running under Windows 95 or Windows NT.

For a discussion on proper rasterization, I suggest checking out the highly regarded *Computer Graphics: Principles and Practice* (Addison-Wesley, 1990) by Foley, van Dam, Feiner, and Hughes, or back-ordering Chris Hecker's excellent series of *Game Developer* articles on texture mapping, which also covers correct rasterization. For more information on the clear reduction algorithm, along with general information and source code for Z buffering, perspective texture mapping, Phong shading, intensity interpolation, Gouraud shading, and related information, you can pick up my up-coming book, *Cutting Edge 3D Game Programming* (Coriolis Group, 1996). There are many resources out there for anyone interested in the topic. ■

John De Goes is a freelance C/C++ programmer currently working on high-performance 2-D and 3-D graphics software for individuals and companies wealthy enough to afford his outrageously high rates. He can be reached at 75404.2752@compuserve.com for a mere $10 a message (plus tax).

Full-Screen Graphics for Macintosh and Windows 95

The best entertainment meticulously crafts an experience for its audience. Just as the best film directors carefully control the viewer's experience, the best game designers carefully control the player's experience. In simulation and strategy games, this often involves a system of overlapping windows the player can arrange to access maps, charts, and tools they need to play effectively. Most adventure and action games immerse the player in the game world by controlling every visible pixel. In such games, window elements like caption bars, window borders, and other applications in the background waste space and intrude on the experience.

Games that use windows can benefit from a cross-platform windowing library like the one we've seen so far in XSplat. For games that want the whole screen, however, XSplat as it stands solves the wrong cross-platform problem.

This month, I'm going to step back a bit from the XSplat windowing system and head off in a different direction. We're going to look into enabling full-screen graphics on Windows and Macintosh, leaving their integration with XSplat for another day.

Once again, I'm going to describe a bit too much code to print in a magazine article, so you'll find the complete source code on the *Game Developer* ftp site.

Direct to Windows

The amount of hype and number of new trademarks surrounding the Windows 95 Game SDK Featuring DirectX has been incredible. The praise is also deserved: the DirectX team at the 'soft bent over backwards to provide Windows 95 applications low-level hardware access comparable to what has traditionally been available only under DOS. Microsoft even provides a consistent interface to eliminate the nightmare configuration programs, buggy VESA drivers, and technical support calls over IRQ and DMA settings. Or such is the religion it preaches.

We'll start our DirectX experimentation with a function that sets up a complete full-screen, double-buffered environment for us, called `BeginFullScreen`. You tell this function the display settings you want, and it sets everything up for you. Listing 1 shows the complete source code for `BeginFullScreen`, if you want to follow our negotiations with DirectDraw.

We need a window. Any window will do because it's just an anchor for DirectDraw, so we'll just use a static text window. The fun begins when we hook up with DirectDraw. `DirectDrawCreate` returns a pointer to a DirectDraw COM object, essentially a pointer to a table of `DirectDraw` functions. You can think of it as a C++ object except that when you're done with it, you call `Release` instead of using the delete operator.

Once connected to DirectDraw, `SetCooperativeLevel` tells the system that we want to take over the video hardware for a while, and `SetDisplayMode` makes the actual resolution switch. If the requested mode isn't available, `SetDisplayMode` will return an error code (something other than `DD_OK`), and we'll bail out of the setup function. We're going to run our sample application at 640-by-480-by-8, a mode available on 99.99% of PC video hardware and a required mode in all Macintosh monitors other than the full-page portrait display.

Jon Blossom

DirectDraw and cross-platform compatibility? Is it possible? Not only is it possible, we've got the code to prove that it's possible!

After switching the video mode, the static text window we created fills the screen. DirectDraw only uses the window as a link to the operating system, handling all the graphics through DirectDraw surfaces. The window could be 1-by-1 and we would still have full-screen graphics, though we'd have problems receiving mouse messages occurring over most of the screen.

DirectDraw uses low-level surfaces, not high-level windows, so we need to create a primary surface representing the visible portion of video memory, the whole display. To create a surface, you use `CreateSurface`, passing a `DDSURFACEDESC` structure describing the surface you want. The primary surface can have secondary (off-screen) surfaces attached to it to enable hardware page-flipping, so we'll try to take advantage of that.

We begin by setting `dwBackBufferCount` to 2 and including the `DDSCAPS_FLIP`, `DDSCAPS_COMPLEX`, and `DDSCAPS_VIDEOMEMORY` flags in our primary surface request. This request asks DirectDraw to create a primary surface and to allocate two additional buffers in off-screen video memory that we can page flip into visible memory. If this fails, we try again for a primary surface with only one off-screen buffer in video memory, and if this fails, we just accept a standard no-frills primary surface.

You may be wondering why we go for two offscreen buffers if we want only a double-buffered library. Well, DirectDraw's page flipping function can take place asynchronously, temporarily preventing access to the two surfaces being switched. It may wait for a vertical retrace before flipping or blting, which means that an attempt to lock the offscreen surface immediately after a call to `Flip` or `Blt` may fail, forcing you to burn cycles waiting for the video hardware to catch up. The flip or blt doesn't involve the third buffer, though, so one surface will always be available for drawing. If you can triple buffer, your application will never have to wait on a lock.

With a primary surface in hand, we now need to go for a secondary surface. If DirectDraw succeeded in creating a page flipping surface, `GetAttachedSurface` will return a pointer to the off-screen buffer. If we had to settle for a simple primary surface, we have to create a second surface manually. The `PageFlip` global will tell us later what type of surface we have.

Finally, we have to set up a color environment by setting up an array of `PALETTENTRY` structures, passing it to `CreatePalette`, and passing the resulting `DirectDrawPalette` pointer on to `SetPalette`.

If all this succeeds, we'll have the full-screen graphics environment we always wanted (and always had under DOS!).

Managing the Macintosh Display

Until recently, Macintosh systems required a specific dot pitch in their monitors. You could change the bit depth of the display, but if you wanted more pixels, you bought a larger monitor. With the introduction of the PowerMacs, Apple made a number of changes to their software architecture, including the Display Manager, which worked its way back to the 68K Macs in system 7.5.1. With the Display Manager installed, it's possible for Mac appli-

Listing 1. Setting Up a Full-Screen Environment Using DirectDraw (Continued on p. 47)

```
static HWND Window;
static HINSTANCE ghInstance;

static LPDIRECTDRAW pDirectDraw;
static LPDIRECTDRAWSURFACE pPrimarySurface;
static LPDIRECTDRAWSURFACE pOffscreenSurface;
static LPDIRECTDRAWPALETTE pPalette;

static int PageFlip;

int BeginFullScreen(int Width, int Height, int Depth)
{
    // Create a Width x Height popup window using the STATIC
    // control class so we don't have to implement a window
    // procedure right now
    Window = CreateWindow("STATIC", "FullScreen", WS_POPUP,
        0, 0, Width, Height,
        0, 0, ghInstance, 0);
    if (!Window)
        goto Failure;

    ShowWindow(Window, SW_SHOWNORMAL);
    UpdateWindow(Window);

    // Connect to DirectDraw, if not already connected
    if (!pDirectDraw)
        DirectDrawCreate(0, &pDirectDraw, 0);
    if (!pDirectDraw)
        goto Failure;

    // Set up DirectDraw for full-screen exclusive mode at the
    // requested resolution
    HRESULT DDReturn;
    DDReturn = pDirectDraw->SetCooperativeLevel(Window,
        DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN);
    if (DDReturn != DD_OK)
        goto Failure;

    DDReturn = pDirectDraw->SetDisplayMode(Width, Height, Depth);
    if (DDReturn != DD_OK)
        goto Failure;

    // Create the primary surface
    // Try to get a triple-buffered one we can page flip
    PageFlip = 1;
    DDSURFACEDESC SurfaceDesc;
    ZeroMemory(&SurfaceDesc, sizeof(SurfaceDesc));
    SurfaceDesc.dwSize = sizeof(SurfaceDesc);
    SurfaceDesc.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
    SurfaceDesc.dwBackBufferCount = 2;
    SurfaceDesc.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
        DDSCAPS_FLIP |
        DDSCAPS_COMPLEX |
        DDSCAPS_VIDEOMEMORY;
    DDReturn = pDirectDraw->CreateSurface(&SurfaceDesc,
        &pPrimarySurface, 0);
    if (DDReturn != DD_OK)
    {
        // We couldn't get a triple buffer, try a double-buffer
        SurfaceDesc.dwBackBufferCount = 1;
        DDReturn = pDirectDraw->CreateSurface(&SurfaceDesc,
            &pPrimarySurface, 0);
        if (DDReturn != DD_OK)
        {
            // We couldn't get a page-flip-able surface at all.
            PageFlip = 0;

            // Just go for a no-frills primary surface
            SurfaceDesc.dwFlags = DDSD_CAPS;
            SurfaceDesc.dwBackBufferCount = 0;
            SurfaceDesc.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
            DDReturn = pDirectDraw->CreateSurface(&SurfaceDesc,
                &pPrimarySurface, 0);
            if (DDReturn != DD_OK)
                goto Failure;
        }
    }

    if (PageFlip)
    {
        // Get the attached page-flip-able surface as the
        // offscreen buffer
        DDSCAPS caps;
        caps.dwCaps = DDSCAPS_BACKBUFFER;
        DDReturn = pPrimarySurface->GetAttachedSurface(&caps,
            &pOffscreenSurface);
    }
    else
    {
        // Create a second surface for the offscreen buffer
        ZeroMemory(&SurfaceDesc, sizeof(SurfaceDesc));
        SurfaceDesc.dwSize = sizeof(SurfaceDesc);
        SurfaceDesc.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
        SurfaceDesc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
        SurfaceDesc.dwWidth = Width;
        SurfaceDesc.dwHeight = Height;

        DDReturn = pDirectDraw->CreateSurface(&SurfaceDesc,
            &pOffscreenSurface, 0);
    }

    if (DDReturn != DD_OK)
```

Listing 1. (Continued from p. 46)

```
        goto Failure;

    // Set up a palette - a grey wash in 0..255
    PALETTEENTRY PaletteColors[256];
    int i;
    for (i=0; i<256; ++i)
    {
        PaletteColors[i].peRed = i;
        PaletteColors[i].peGreen = i;
        PaletteColors[i].peBlue = i;
        PaletteColors[i].peFlags = PC_RESERVED;
    }

    DDReturn = pDirectDraw->CreatePalette(DDPCAPS_8BIT,
        PaletteColors, &pPalette, 0);
    if (DDReturn != DD_OK)
        goto Failure;

    // Attach the palette to the surface
    DDReturn = pPrimarySurface->SetPalette(pPalette);
    if (DDReturn != DD_OK)
        goto Failure;

    // Success!
    return 1;

Failure:
    return 0;
}
```

cations to exert low-level control over display settings.

Display Manager 1.0 now ships as part of System 7.5.1, but it sports a clunky programming interface and is not always reliable. Display Manager 2.0 is available as an extension for these systems and is integrated into System 7.5.2. It's cleaner and simpler, so we'll be writing for version 2.0.

If you don't have Display Manager 2.0, you can download it from Apple's ftp site, at ftp://ftp.info.apple.com/Apple.Support.Area/Developer_Services/Development_Kits/Display_Manager_Development_Kit.

Listing 2 shows the complete **BeginFullScreen** procedure for the Macintosh so you can sing along.

The first step to taking over the Macintosh display is to get rid of that pesky menu bar. I've written a **StealMenuBar** function to recover the screen space used by the Macintosh menu bar. Once it's gone, we'll create a window of the requested size and set up a palette as I discussed in previous articles.

Now the Display Manager comes into play. The first thing to do is to store away the current display settings so we can restore them when we're done harassing the video hardware. A call to **GetMainDevice** gives us a handle to the main display device, and **DMGetDisplayMode** fills us in on its current settings.

On the Macintosh, a display mode is defined by two numbers: **csMode** and **csData**, which together describe a unique color format, resolution, and refresh rate setting. In order to change the display mode, we have to find out the appropriate secret numbers for the mode we want. The Display Manager will provide us with a list of available modes through **DMNewDisplayModeList**, and we'll scan through that list until we find an appropriate one.

DMGetIndexedDisplayModeFromList iterates through the list for us. You give this function a pointer to a callback function and a pointer to some data of your choice, and it invokes the callback on each element in the list. **NewDMDisplayModeListIteratorProc** (whew!) creates a universal procedure pointer to use as your callback (this is similar to the Win16 **MakeProcInstance** function). We'll pass along a custom **DisplayModeRequest** structure that tells the callback what resolution to look for and includes space for it to return the **csMode** and **csData** codes if it finds the requested resolution.

Once out of the loop, **ModeRequest** will contain valid information if the Display Manager reported a mode we can use. If so, **DMSetDisplayMode** will make the switch for us.

Finally, we can create an offscreen **GWorld** for the window as we did before, and we have total double-buffered control at the desired resolution.

Offscreen Access

A few more functions round out our full-screen needs. **EndFullScreen** will undo everything done by **BeginFullScreen**. **SwapBuffer** will either copy or page flip the offscreen buffer onto the screen. **SwapRect** will copy a portion of the offscreen buffer onto the screen. **OffscreenLock** and **OffscreenUnlock** will give us access to the bits of the offscreen image, and between these two calls, **GetOffscreenBits** and **GetOffscreenStride** will provide us with the information we need to draw on the surface.

On the Macintosh, we don't have to learn anything new. Sure, we've used the Display Manager to tweak the monitor to the settings we want, but otherwise everything's fundamentally the same: we still have a window, a **GWorld**, and palette. We'll still use **CopyBits** to get the **GWorld** to the screen, and we'll still lock and unlock the bits of the **GWorld PixMap** to gain access to the offscreen buffer, as I demonstrated before.

Under Windows, however, things have changed. No more **DIBSections**, no more WinG. We're now cooking with DirectDraw surfaces, and they require some special handling. The **OffscreenLock**

Listing 2. Setting Up a Full-Screen Environment Using Display Manager 2.0 (Continued on p. 49)

```
static WindowPtr Window;
static PaletteHandle hPalette;
static GWorldPtr pOffscreenGWorld;
static GDHandle DisplayDevice;

static unsigned short csPreviousMode;
static unsigned long csPreviousData;

// This structure encapsulates the data sent to the Display Manager list
// enumeration callback function. We fill in the desired values, pass
// this on through the enumeration, and it fills in the csMode and
// csData info we need.
struct DisplayModeRequest
{
    // Returned values
    unsigned short csMode;
    unsigned long csData;

    // Provided values
    long DesiredWidth;
    long DesiredHeight;
    long DesiredDepth;
};

// This function filters through the display modes reported by the
// Display Manager, looking for one that matches the requested
// resolution. The userData pointer will point to a DisplayModeRequest
// structure.
pascal void DisplayModeCallback(void* userData, DMListIndexType,
    DMDisplayModeListEntryPtr pModeInfo)
{
    DisplayModeRequest *pRequest = (DisplayModeRequest*)userData;

    // Get timing info and make sure this is an OK display mode
    VDTimingInfoRec TimingInfo = *(pModeInfo->displayModeTimingInfo);
    if (TimingInfo.csTimingFlags & 1<<kModeValid)
    {
        // How many modes are being enumerated here?
        unsigned long DepthCount =
            pModeInfo->displayModeDepthBlockInfo->depthBlockCount;

        // Filter through each of the modes provided here
        VDSwitchInfoRec *pSwitchInfo;
        VPBlock *pVPBlockInfo;
        for (short Count = 0; Count < DepthCount; ++Count)
        {
            // This provides the csMode and csData information
            pSwitchInfo =
                pModeInfo->displayModeDepthBlockInfo->
                depthVPBlock[Count].depthSwitchInfo;

            // This tells us the resolution and pixel depth
            pVPBlockInfo =
                pModeInfo->displayModeDepthBlockInfo->
                depthVPBlock[Count].depthVPBlock;

            if (pVPBlockInfo->vpPixelSize ==
                pRequest->DesiredDepth &&
                pVPBlockInfo->vpBounds.right ==
                pRequest->DesiredWidth &&
                pVPBlockInfo->vpBounds.bottom ==
                pRequest->DesiredHeight)
            {
                // Found a mode that matches the request!
                pRequest->csMode = pSwitchInfo->csMode;
                pRequest->csData = pSwitchInfo->csData;
            }
        }
    }
}
int BeginFullScreen(int Width, int Height, int Depth)
{
    // Hide the menu bar
    StealMenuBar();

    // Create a window of the requested size
    Rect WindowRect = { 0, 0, Height, Width };
    Window = NewCWindow(0, &WindowRect, "\pFullScreen",
        TRUE, plainDBox,
        WindowPtr(-1), FALSE, 0);
    if (!Window)
        goto Failure;

    // Set up a palette with a gray wash
    hPalette = NewPalette(256, 0, pmExplicit | pmAnimated, 0);
    if (!hPalette)
        goto Failure;

    RGBColor Color;
    int i;
    for (i=0; i<256; ++i)
    {
        Color.red= i << 8;
        Color.green = i << 8;
        Color.blue = i << 8;

        SetEntryColor(hPalette, i, &Color);
    }

    // Force 0 and 255 to White and Black
    SetEntryUsage(hPalette, 0,
        pmExplicit | pmAnimated | pmWhite, 0);
    SetEntryUsage(hPalette, 255,
        pmExplicit | pmAnimated | pmBlack, 0);
```

Listing 2. (Continued from p. 48)

```
SetPalette(Window, hPalette, TRUE);

// Store information about the current display settings
DisplayDevice = GetMainDevice();
if (!DisplayDevice)
    goto Failure;

csPreviousMode = -1;
csPreviousData = -1;

VDSwitchInfoRec DisplayInfo;
OSErr MacError = DMGetDisplayMode(DisplayDevice, &DisplayInfo);
if (MacError != noErr)
    goto Failure;

csPreviousMode = DisplayInfo.csMode;
csPreviousData = DisplayInfo.csData;

// Get the display ID for the main display
DisplayIDType DisplayID;
DMGetDisplayIDByGDevice(DisplayDevice, &DisplayID, FALSE);

// Use it to get a list of available modes from the
// Display Manager
DMListIndexType DisplayModeCount = 0;
DMListType DisplayModeList;
MacError = DMNewDisplayModeList(DisplayID, 0, 0,
    &DisplayModeCount, &DisplayModeList);
if (MacError != noErr)
    goto Failure;

// Create a callback function pointer to filter available modes
DMDisplayModeListIteratorUPP uppDisplayModeCallback =
    NewDMDisplayModeListIteratorProc(DisplayModeCallback);
if (!uppDisplayModeCallback)

{
    // Aborting - let go of the mode list
    DMDisposeList(DisplayModeList);
    goto Failure;
}

// Go through the list, comparing each available mode with
// this mode request
DisplayModeRequest ModeRequest;
ModeRequest.csMode = -1;
ModeRequest.csData = -1;
ModeRequest.DesiredWidth = Width;
ModeRequest.DesiredHeight = Height;
ModeRequest.DesiredDepth = Depth;

for (short Count = 0; Count < DisplayModeCount; ++Count)
{
    DMGetIndexedDisplayModeFromList(DisplayModeList, Count,
        0, uppDisplayModeCallback, (void*)&ModeRequest);
}

// Done with the list
DMDisposeList(DisplayModeList);

// Done with the callback
DisposeRoutineDescriptor(uppDisplayModeCallback);

// If we found a mode fitting the request, switch to it!
if (ModeRequest.csMode == -1 || ModeRequest.csData == -1)
    goto Failure;

DisplayInfo.csMode = ModeRequest.csMode;
DisplayInfo.csData = ModeRequest.csData;

unsigned long Mode = DisplayInfo.csMode;
MacError = DMSetDisplayMode(DisplayDevice,
    DisplayInfo.csData, &Mode,
    (unsigned long)&DisplayInfo,
    0);
if (MacError != noErr)
    goto Failure;

// Create a matching GWorld using current device and window
CGrafPtr CurrentPort = (CWindowPtr)Window;

PixMapHandle CurrentPixMap = CurrentPort->portPixMap;
CTabHandle ColorTable = (*CurrentPixMap)->pmTable;

NewGWorld(&pOffscreenGWorld, (short)Depth,
    &CurrentPort->portRect, ColorTable,
    DisplayDevice, noNewDevice);
if (!pOffscreenGWorld)
    goto Failure;

// Success!
return 1;

Failure:
    RestoreMenuBar();
    return 0;
}
```

Listing 3. Loading a DirectDraw Surface

```
static char unsigned *pBits;
static long Stride;

int OffscreenLock(void)
{
 int ReturnValue = 0;

 pBits = 0;
 Stride = 0;

 DDSURFACEDESC SurfaceDesc;
 ZeroMemory(&SurfaceDesc, sizeof(SurfaceDesc));
 SurfaceDesc.dwSize = sizeof(SurfaceDesc);

 // Loop until an error occurs or the lock succeeds
 HRESULT DDReturn = DD_OK;
 while (1)
 {
     // Attempt the lock
     DDReturn = pOffscreenSurface->Lock(0, &SurfaceDesc, 0, 0);

     if (DDReturn == DD_OK)
     {
         // Successful lock - store bits and stride
         pBits = (char unsigned *)SurfaceDesc.lpSurface;
         Stride = SurfaceDesc.lPitch;
         ReturnValue = 1;
         break;
     }
     else if (DDReturn == DDERR_SURFACELOST)
     {
         // Attempt to restore the surfaces
         DDReturn = pPrimarySurface->Restore();
         if (DDReturn == DD_OK)
             DDReturn = pOffscreenSurface->Restore();

         if (DDReturn != DD_OK)
         {
             // Surfaces could not be restored - lock fails
             break;
         }
     }
     else if (DDReturn != DDERR_WASSTILLDRAWING)
     {
         // Some other error happened - fail
         break;
     }
 }

 return ReturnValue;
}
```

Listing 4. A 640x480x8 Cross-Platform Demo

```
void DemoMain(void)
{
 if (BeginFullScreen(640, 480, 8))
 {
     // Go for 10 seconds
     char unsigned Color = 0;
     long unsigned Time = GetMillisecondTime();
     while (GetMillisecondTime() - Time < 10000)
     {
         if (OffscreenLock())
         {
             // WARNING: YOU CANNOT USE A DEBUGGER IN BETWEEN
             // LOCK..UNLOCK CALLS WHEN USING DIRECTDRAW!

             // Draw a wash offscreen using memset
             char unsigned *pSurface = GetOffscreenBits();
             long Stride = GetOffscreenStride();

             for (int i=0; i<480; ++i)
             {
                 memset(pSurface, (Color + i)%256, 640);
                 pSurface += Stride;
             }

             OffscreenUnlock();
             // OK TO USE A DEBUGGER NOW

             // Next time through we'll use a different color
             // (Allow this to overflow back to zero)
             ++Color;
         }

         SwapBuffer();
     }
 }

 // Undo everything done by BeginFullScreen.
 // Even if BeginFullScreen failed, it may have partially succeeded
 // and may require cleaning up
 EndFullScreen();
}
```

and `OffscreenUnlock` functions just map onto calls to DirectDraw's `Lock` and `Unlock`, but these may fail if the surface is busy.

If an asynchronous blt or a flip has been started but hasn't finished, we'll receive a `DDERR_WASSTILLDRAWING` code from most DirectDraw functions, indicating that we have to wait until the hardware has completed its task. If we receive a `DDERR_SURFACELOST` notification, DirectDraw has given our video memory to someone else, and we have to restore all of our surfaces before we can use them again. Although this shouldn't happen when we've set ourselves to `DDSCL_EXCLUSIVE`, we should be ready for it. We have to wait in a `while(1)` loop trying to lock the surface until we receive a `DD_OK` or a fatal error code.

A similar situation occurs during a `Flip`, a `Blt`, or a `BltFast` call: it may be necessary to wait until the surface becomes available. Passing `DDFLIP_WAIT`, `DDBLT_WAIT`, or `DDBLTFAST_WAIT` will force DirectDraw to wait until the operation completes before returning, eliminating the possibility of contention, but that may waste time you could spend doing some other processing.

Listing 3 shows the DirectDraw implementation of `OffscreenLock`. The other functions use a similar loop to wait until the operation succeeds, bailing out if an error occurs.

This Month's Cheesy Demo

Of course, to prove this works, I have to write a simple cross-platform program using these functions. Unfortunately,

there's no means of user input in this month's code, though by combining the full-screen code presented here with the message processing architecture in XSplat, you can easily build an interactive system.

This month's demo has the simple goal of setting up a 640-by-480-by-8 graphics environment and drawing palette washes for ten seconds, starting each wash with a different index. The resulting pattern will make you feel like your old black-and-white has lost vertical synch. The code is shown in Listing 4. If you're looking at this with a page-flipped DirectDraw surface, you'll see how incredibly smooth page flipping can be.

The `GetMillisecondTime` function was presented in my last article, if you're wondering about that. It maps onto `timeGetTime` on Windows and `TickCount` (or `Microseconds`) under MacOS.

Thoughts and Warnings

Before I return to my cage for the next couple of months, there are a few hairy details you should hear about before plunging in with this full-screen API. Here's my laundry list of warnings:

- When you lock a DirectDraw surface, DirectDraw grabs the Win16 lock, the semaphore that prevents reentrant execution of 16-bit system code including GDI and USER. This means that essential system components shut down as long as you have a locked surface. If you attempt to step through code with a GUI debugger while a surface is locked, you will hang the system.
- `SetCooperativeLevel` uses the contents of the `GWL_USERDATA` bytes of a window to store information used to restore the display state. This means that the `HWND-to-CXSplatWindow` mapping used in my previous articles can not coexist with DirectDraw. If you want to use this association technique, set `cbWndExtra` to `sizeof(CXSplatWindow*)` when registering the "XSPLAT" window class and use 0 instead of `GWL_USERDATA` when setting or retrieving the data.
- On many banked display cards, DirectDraw emulates direct video access using a virtual device driver called VFLATD.386, which maps video memory banks onto selectors and uses a page fault handler to switch video banks when you cross a boundary. Only one display memory bank can be active at any time, so if you write across one of these boundaries, the fault handler will thrash as it attempts to write to both pages at once. This will hang your system. Using only 32-bit aligned writes will guarantee that you never write across one of these boundaries.
- Because the `SwapBuffer` function may use `flip` instead of `blt`, there is no guarantee of the contents of the offscreen buffer after a swap. After a `flip`, the buffer will contain the former contents of the display. After a `blt`, it will remain the same. If you need to preserve the contents of your offscreen buffer, always use `SwapRect` instead of `SwapBuffer`.
- DirectDraw preserves color 0 as black and color 255 as white. Changing the Macintosh display resolution and creating a full-screen window doesn't cause the Mac to release colors 0 (white) and 255 (black) either, so the color zero problem remains. Don't forget!
- When you shrink the resolution of the screen, Display Manager 2.0 will reposition the open windows and icons on the desktop. When you set the resolution back, it may not replace them, leaving your desktop looking very strange.
- The display mode you request through `BeginFullScreen` may not be available. You can ask for a 100x609x13 display if you want, and `BeginFullScreen` will try to accommodate you. A more robust system would provide a list of available modes rather than leaving you to cross your fingers, but do you expect a magazine article to do all your work for you?

That's it! You can find the complete source code referenced in this article on the *Game Developer* ftp site. ■

Jon Blossom sometimes wishes operating systems would go away so he could stop tweaking with Macintosh, Windows, OS/2, and X. He can be reached at blossom@slip.net or through Game Developer *magazine.*

Fuzzy Logic in Games

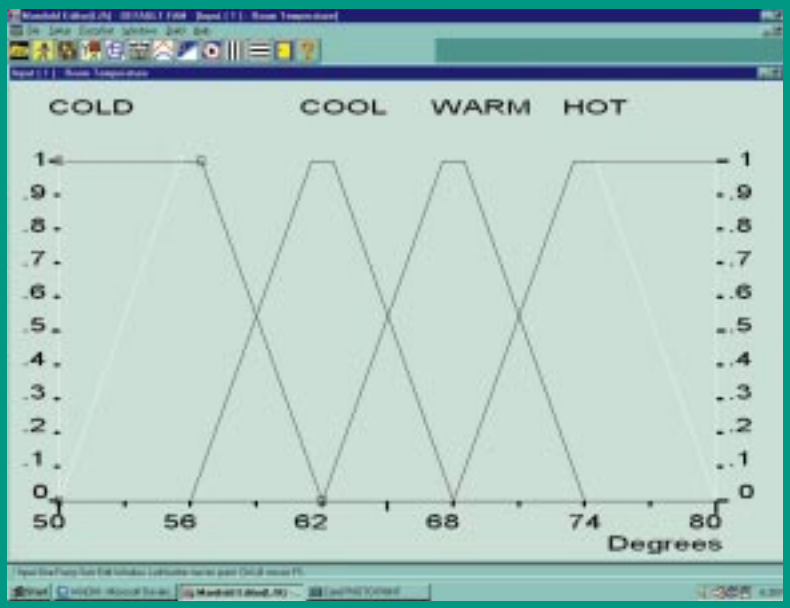


Figure 1. Trapezoids map precise variables such as temperature into linguistic variables such as "Warm."

Fuzzy logic is a powerful artificial intelligence (AI) technique appropriate in many gaming situations. The basic fuzzy technique is to specify a situation using fuzzy linguistic variables (FLVs), which, when cross-referenced on a fuzzy associative matrix (FAM), create a "fuzzy set" on a third FLV. A fuzzy manifold can either be manipulated as its own type of object or it can be defuzzified into a more traditional crisp value. Similarly, crisp values can be fuzzified into FAMs. If that jargon didn't explain it all, the accompanying source code should make it concrete.

A fuzzy linguistic variable is an array of labels such as "Hot," "Warm," "Cool," "Cold." Associated with this array is a mapping of a crisp values such as 68° F into the array. This mapping is done with trapezoidal membership definitions, as illustrated in Figure 1. For each slot in the array, the logic designer chooses four numbers representing the begin low, full membership low, full membership high, and begin high values. Listing 1 shows a simple and quick function for assigning a membership value to a value.

As Figure 1 shows, a single crisp value can have some amount of membership in several slots in the FLV. For instance, the way I've designed the room temperature manifold, 68° is an inflection point, where the set changes from some membership in Cool as well as Warm to some membership in Hot as well as Warm. The greater the overlap you put in your manifolds, the more continuous will be the fuzzy logic response curve—in effect, your critters will be less decisive. By decreasing the overlap, you can tune your AI for the opposite effect—more capriciousness. This is the essence of fuzzy logic—manifold tuning, which can be done by a nonprogrammer on a spreadsheet, can create dazzlingly complex results.

The complexity stems from the fuzzy associative matrix, a matrix in which two FLVs combine to specify membership in a third, as shown in Figure 2. Adjusting the FAM provides gross adjustments in fuzzy logic, while manifold tuning provides the richness. A

Larry O'Brien

Forget classical expert systems. Fuzzy logic makes your game AI subtle and complex, but it is easy to program and lightning fast at run-time. Here's a look at FLVs and FAMs.

fuzzy system is simply one that sets up input FLVs and uses the resulting output FLV to specify behavior.

As a concrete example of the power of fuzzy logic, let's say you were working on a real-time strategy game of the Command & Conquer school. It's usually very difficult to get to work on the AI for such a game early in the design process, but fuzzy logic lets you do so. The AI designer can start with general rules such as, "Distance to the enemy and relative strength determine unit movement." After the rules have been determined, the FLVs associated with the antecedents (distance to the enemy and relative strength) and the consequent (movement) are roughed out. For simple purposes, you might choose, "Proximate," "Near," "Separated," and "Far" as your distance FLVs "Dead Meat," "Overmatched," "Equal," "Undermatched," and "Steamroller" for relative strengths, and "Run away," "Surrender," "Retreat," "Fall Back," "Hold," "Probe," "Assault," and "Overrun" for movement. Behavior can then be roughed into a Fuzzy Associative Matrix, such as that shown in Figure 3.

In a classical expert system or with naive state-based AI, the richness of this rule is severely reduced by the three-fold

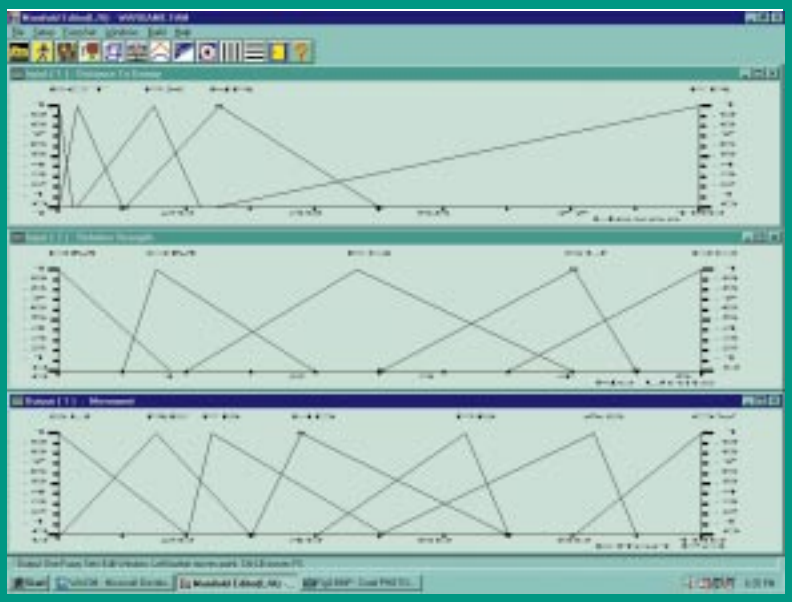


Figure 2. The two top fuzzy sets define the inputs to fuzzy output set at the bottom. Notice asymmetries introduced to make AI more aggressive.

Listing 1. A Quick Function for Determining Membership in a Trapezoidal Fuzzy Set

```
class FuzzySet{
public:
 ...
 float Membership(const float& inVal);
private:
float lowMin, lowTrue, highTrue, highMin;
}

float FuzzySet::MemberShip(const float& inVal){
   if(inVal > lowMin && inVal < highMin){
        if(inVal > lowTrue){
                if(inVal < highTrue){
                        membership = 1.0;
                }else{
                        //must be on high edge
                        membership = (inVal - lowTrue) / (highMin - lowTrue);
                }
        }else{
                //must be on low edge
                membership = (inVal - lowMin) / (lowTrue - lowMin);
        }
   }else{
        membership = 0.0;
   }
   return membership;
);
```

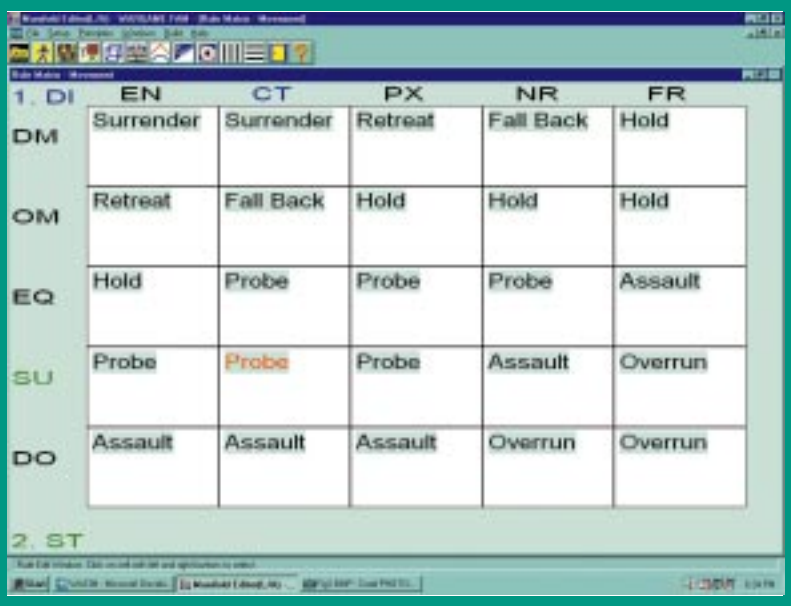


Figure 3. Being able to see and edit an entire matrix of rules simultaneously is one of the strengths of fuzzy logic.

pigeonholing of crisp values. We all know games where opponent behavior is clearly discontinuous—stand at one spot and the guards will ignore you, move another step closer and they start swarming at you in their predictable manner. Fuzzy logic creates richer, continuous responses by evaluating all the rules, but gauging them on a spectrum ranging from totally false to totally true, with infinite gradations (rejecting the law of the excluded middle and the law of contradiction, for you fans of dead Greek mathematicians). Since it evaluates all the rules, the performance of a fuzzy logic inference engine degrades proportional to the total number of rules, and it is not subject to the catastrophic nonlinear degradations that classical expert systems are prone to.

After filling in the FAM and the FLV array for the fuzzy expert system, evaluation is the next step. Each cell in the FAM represents a rule so, for instance, the upper-left corner of the matrix in Figure 3 represents the fuzzy rule, "If the forces are engaged, and we're dead meat, surrender." To evaluate the rule, we determine the membership in the input FLVs as described previously and illustrated in Listing 1. While distance is a crisp value, membership in the Relative Strength FLV is probably the result of another fuzzy rule. Assume that membership in the "dead meat" FLV is 28%. The degree of membership in the output is set equal to one of these two values—the minimum if the rule "and"s the antecedents, the maximum if the rule "or"s them. In this case, the rule is intended to be read "If the forces are engaged, and we're dead meat, surrender," and not "If engaged or dead meat, surrender," so we set the membership in "surrender" for this rule to 28%.

Since fuzzy logic allows various shades of truth, the entire rulebase is evaluated, setting various memberships in each of the output FLVs. For an "and" FAM, the maximums of the output values are chosen, if an "or" is chosen, the minimums are taken. A typical result is shown in Figure 4. This output fuzzy set can be used as input to another rule or, if necessary, defuzzified by the

simple technique of calculating the centroid of the polygon.

Fuzzy logic is an extraordinarily powerful and flexible artificial intelligence technique that is much easier to program than alternatives such as classical expert systems. Unlike neural networks, fuzzy logic systems can be tuned manually, and unlike genetic-based systems, fuzzy systems work "out of the box." For continually improving performance, you can use machine-learning techniques to modify the parameters of the FLVs. Of course, "improved" is a fuzzy concept itself, and judging the efficacy of a move can be as hard or harder than making the move in the first place. Strategic analysis techniques, one of the oldest and deepest veins in the field of artificial intelligence, will have to wait for another day, however. ■

Editorial Director Larry O'Brien specializes in artificial forms of intelligence.

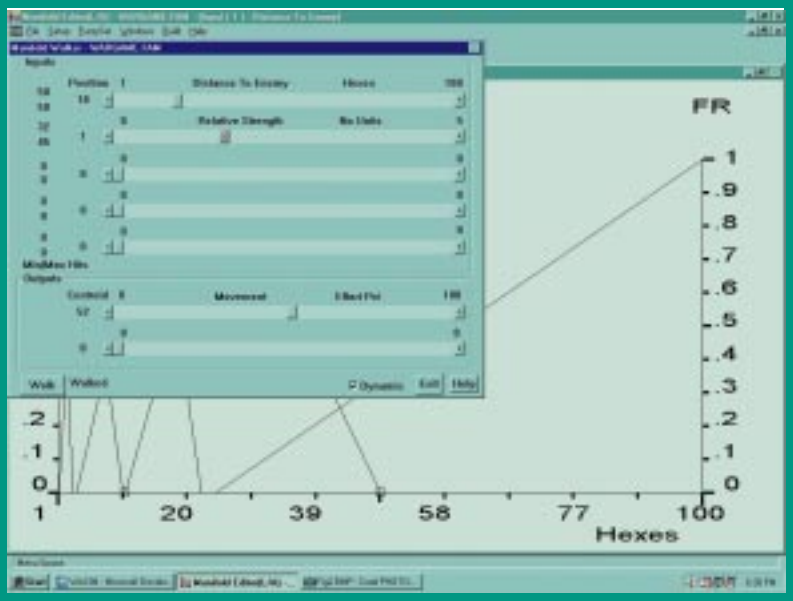


Figure 4. A moderate distance and slightly inferior enemy make for a cautious advance.

Blizzard of '96

Mike Michaels

The secret to gaming success is getting users hooked on your game. Discover what the folks at Blizzard Entertainment did to make Warcraft II such an addictive game.

Welcome to Chopping Block. In this issue, we'll look at Warcraft II: Tides of Darkness, the sequel to last year's sleeper hit, Warcraft: Orcs & Humans. Warcraft II is a real-time strategy war game with a fantasy theme. The object is to survive long enough to decimate all opposing forces and destroy all they hold dear.

Like its predecessor, the heart of the game is resource management. Before you can train your knights and archers, you must have a barracks to train them in. Before you can build a barracks, you must mine enough gold and harvest enough trees. Before you can mine gold and harvest trees, you must have peasants to do the work for you. How you allocate your limited resources can mean the difference between victory and defeat.

The game is played from an overhead, isometric viewpoint. Unlike other games in this genre, the viewpoint is close enough to the ground that both the buildings and the creatures are rendered with a high degree of detail. This propensity of detail stems from the game objects (creatures, buildings, vessels) being first modeled (using 3D Studio or the like) and then rendered at the required sizes. The knights you control in the game are rendered from the same model used to generate the knights in the cut-scenes.

The game is tile based, and all game objects are an integral number of tiles large. Don't make the mistake of assuming that the game looks blocky. The programmers did a wonderful job when they created their map editor. I won't venture a guess as to what algorithm they used to create their coast-

In Warcraft II: Tides of Darkness, the player has to learn to manage resources effectively.

lines, but I was extremely impressed by how natural and asymmetric it all is. A creature can move without too much hint of tile transitions. Obviously, enough frames of animation were provided to make the tile transitions seamless. The only time you realize this underlying tile system is when a creature must go around another creature or object. When this occurs, the creature makes a 45-degree turn and moves to a diagonal tile. Then the creature finds the nearest tile it can move to that reorients it on its goal. The overall effect is that the creatures move in a very unnatural manner (angular rather than a more natural, circular path).

The interface is very intuitive. You can control creatures using either the mouse or the keyboard (or a combination of both). Simply clicking the cursor on a creature selects that creature and brings up a series of buttons representing actions the creature can perform. Moving the cursor over one of the buttons brings up a description of the action and, if appropriate, the amount of resources that must be paid to perform the action.

Distinguishing itself from it's predecessor, Warcraft II provides sea and air units along with shipyard and aviaries to train these units. The beauty of the interface is that there is essentially no difference between controlling a soldier and controlling a battleship. The same options are available for both. Just choose an option and choose a target. It's that simple.

Warcraft II lets players choose either single-player or multiplayer scenarios. Multiplayer support is available via null modem, modem, and IPX network. With a network connection, up to eight simultaneous players are possible, including computer opponents.

In Blizzard Entertainment's game, you strategize to survive.

Warcraft II: Tides of Darkness

Blizzard Entertainment
P.O. Box 18979
Irvine, Calif. 92713
Tel: (800) 953-7669
Web: http://www.blizzard.com/
Price: $54.95

System Requirements: 486DX/33, 8MB RAM, 29MB free hard-drive space, SVGA graphics, Sound Card (Redbook Audio, General Midi, Sound Blaster, PAS, Gravis, and compatibles), and a double-speed CD-ROM drive.

I didn't get a chance to try the direct connect or modem options, but I can attest that the game plays smoothly over a standard IPX network. I installed it at work on two Pentium machines running Windows 95 with standard network support. No extra effort was required, the game picked up connection immediately, and the speed of game play seemed to be identical to that of single-player mode.

If you choose to play the single-player game, fourteen human and fourteen orc campaigns are provided. Things don't really start getting difficult until around the sixth or seventh campaign, at which point they get very challenging. Unfortunately, this challenge isn't really due to increasing the game's artificial intelligence (AI). Rather, the challenge comes from increasing the amount of resources your enemy starts the scenario with. It can be a little discouraging to be attacked by dragons when you can't even build flying units yet.

But other than the numerical advantage, the computer opponent doesn't appear to cheat. It seems to be following the exact same rules you have to follow. On the other hand, the AI doesn't appear to be particularly smart either. There doesn't appear to be an overriding intelligence in the computer opponent. As a human player, I jump around the map keeping track of what is going on in many sectors of the game at once. The computer artificial intelligence doesn't have this advantage. In general, it looks like each creature has its own AI routine to determine what its next action should be. With no omniscient intelligence, the computer opponent makes the same mistakes repeatedly.

For example, assume there is a gold mine out of visual range for any of the enemy's fighting units. You can sit a few soldiers and archers around the mine and pick off enemy peons as they come to mine the gold. If this happened to a human player, units would be dispatched almost immediately to get rid of the troublemakers screwing up the gold supply. The computer opponent doesn't seem to notice that the peons it is sending to collect gold never seem to return.

The map editor is included with the game so that you can create your own scenarios. The map editor requires Windows 95 to execute. Building maps is a snap. It's as simple as selecting an object and clicking on the map screen. There's even an animate feature that will let you give your scenario a test run.

Like it's predecessor, Warcraft II is extremely addictive. The graphics are vibrant; the sound is great (hilarious at times); and the ability to play the game against many opponents makes this a sure bet at bringing the office network to a standstill.

That's it for this month. I'm still interested in hearing your opinions about the direction that you'd like to see this column progress. If you have any idea's or opinions, just drop me an e-mail at the address provided below! ■

Mike Michaels is doubtful that anyone reads these little bios (unless they're looking for his e-mail address to flame him) and therefore chose not to write one this month. He may be reached via e-mail at mike@irvine.com or through Game Developer *magazine.*

Where the Sun Don't Shine

David Sieks

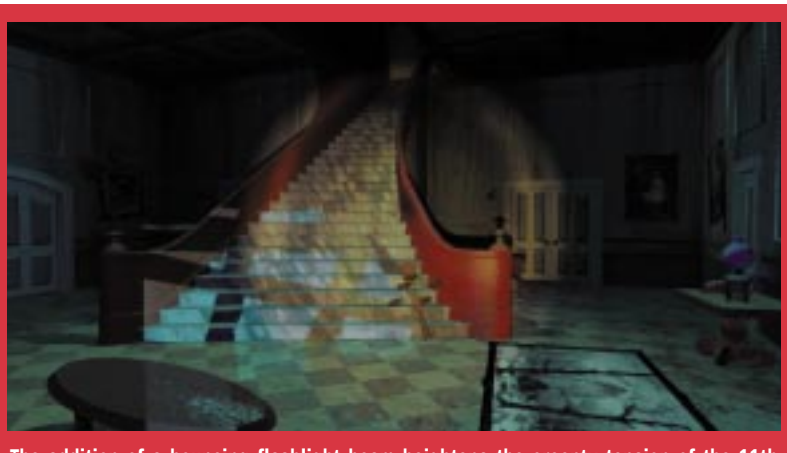
Lighting is vital to making your 3D games look realistic. Sieks takes us through the ins and outs of hard and soft lighting—learn how to really set the mood.

Game graphics used to be easy: you'd put row after row of blocky space aliens at the top of the screen and a clunky space cannon at the bottom of the screen, all against a black background (because, did I mention, this takes place in space) and ta-dah...you got game graphics. At this level, graphics could routinely be handled by the programmers themselves—no need to involve an artist.

Now, of course, it is a different matter. The visual component seems to have taken precedence over game play and originality. (Hey, that's not my beef: this is the Artist's View, man.) The market demands that games of all genres look as good as they can get, which provides plenty of opportunity and challenge for talented artists.

Even relatively straight-ahead shooters have gorgeous, rendered backgrounds; it's all but mandatory to include cinematic intros and cut sequences in everything from flight sims to fight-fests. Since the coming of Myst, adventure games must (it is a law) try to outdo one another in visual atmosphere. When you try to make eye-popping visuals at this demanding level, when your task is to capture an audience's interest and spur its imagination, you need to make use of every tool at your disposal.

Artists working with three-dimensional graphics now need to acknowledge something that photographers and cinematographers have held as a central tenet of their art forms throughout the century and that realist painters have practiced for longer still: control of a scene's lighting determines much of its impact. One cannot discern color or form without illumination, but, in the context of a scene, lighting takes on added significance. Used judiciously, areas of light and dark create the overall composition, direct the



The addition of a bouncing flashlight beam heightens the creepy tension of the 11th Hour, sequel to The 7th Guest, from Virgin Entertainment and Trilobyte Inc.

audience's attention, and can work to cue expectations in the viewer and to underscore the narrative.

Yes, 3D artists should realize the importance of lighting. It is all too easy to focus on the obvious challenges of modeling, texture mapping, and animation and pay only perfunctory notice to the demands of lighting a scene whose other components we have worked on so painstakingly. To do justice to every element, you need to spend equal time on and pay equal attention to each. A careless approach to lighting, however, can negate much of the hard work put into every other aspect by creating a flat, dull scene—or one that looks instantly and unmistakably computer generated.

There are two basic approaches to lighting a scene: the realistic and the expressionistic, or, to throw out a real film-weenie term, the diegetic and the nondiegetic.

Effect over Fact

When tackling the task of lighting a scene, one must think in terms of contrast—that is, the juxtaposition of light and dark. The effect is the important thing: not how many lights you place or where or what the logical source of such illumination would be, just how it all looks. A common pitfall is to create lights in the 3D scene that are, as nearly as possible, strictly analogous to the real-world light sources. Unfortunately, in the realm of 3D graphics, good lighting isn't as easy as flicking a switch.

First, the particular characteristics of computer graphics lights usually make it necessary to work with placement, intensity, and so on to finagle the desired effect. For example, you might need to place additional lights to simulate the effects of radiosity (the dispersal and propagation of reflected light). Or you might need to situate a light somewhere other than its apparent source in the scene: for example, positioning a light outside the dimensional model of the light fixture to avoid an unwanted hot spot (a bright, central highlight) on a nearby surface. The guiding precept should be to forget about what makes strict sense and place the light where it gives the best effect. Artists are only constrained by reality on rent day.

Despite the fact that they are capturing images of the real world, photographers and cinematographers (who are also rent-paying artists) go to great lengths to achieve the lighting effects they want. Even when the end result is intended to be perceived as realistic, they make use of multiple lights, reflectors, diffusers, filters, and so on. As Anthony Burgess notes in *A Clockwork Orange*, the colors of the real world never look so real as on the screen.

Pick a Palette of Light

There are two basic approaches to lighting a scene: the realistic and the expressionistic, or, to throw out a real film-weenie term, the diegetic and the nondiegetic. Diegesis refers to the narrative, so diegetic lighting is that which, for the most part, seems consistent with the setting and the action of the narrative. Subtle finagling with additional lights, diffusers, reflectors, and filters (or their computer graphics counterparts) still falls within this first category as long as a convincing look is the intent.

On the other hand, expressionistic, nondiegetic lighting is that which throws all pretense of realism out the window (at least for the moment) and goes purely for effect, using sometimes subliminal, psychological connotations of light, shadow, and color to make its point: bathing a scene of violence in red light or shining a heavenly light from above on a redeemed character. Nondiegetic lighting can tend to be clichéd and even campy, but, used with restraint, can also be very effective in getting a point across quickly and surely, as most people in your audience will intuit the intended meaning.

Two basic ways of applying light to a scene exist: hard and soft lighting. Hard lighting refers to sharply defined areas of light and shadow, whereas soft lighting uses a more diffused light, with gentler shadows and blended areas.

Though you may not have thought about it in such, well, black and white terms before, it's easy to see how hard and soft lighting fit into the basic lighting styles and how they relate to the different moods or genres commonly associated with them. Low-key lighting, often used for scenes of mystery or suspense, makes use of moody, atmospheric pools of light and dark. High-key lighting is bright but not harsh, even illumination with minimal shadows, generally used for lighthearted subjects. And high-contrast lighting juxtaposes dramatic streaks of light and shadow for a stark, edgy look full of tension.

These lighting conventions are not ironclad rules. Of course, you don't have to use high-key lighting for a lighthearted scene. They are, however, conventions that come with baggage we have carried for millennia: our ancestral memory tells us shadows are frightening; you'd better be prepared to work around that if you're determined to use low-key lighting in the splash scene for The Frolicsome Pony's Interactive Learning Adventure For Ages 6 to 9.

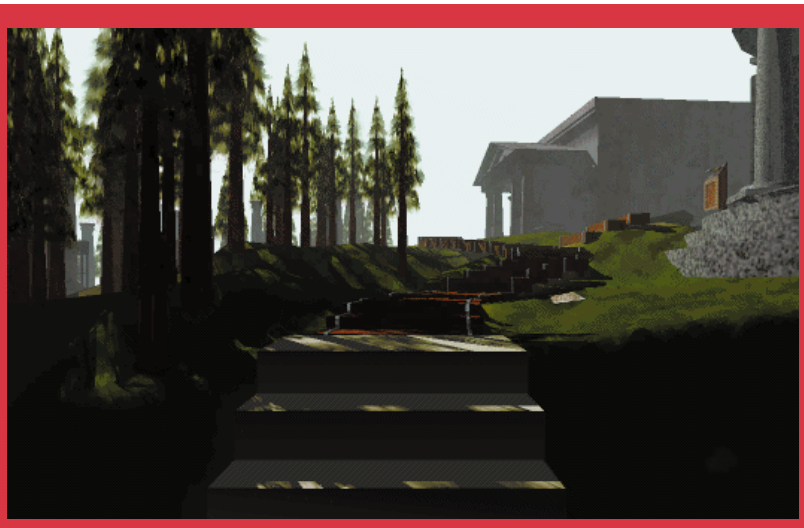
Can't Tell the Players Without a Program

The hump, of course, is figuring out how to control the lighting of a scene to use contrast knowingly and to your advantage. Though computer artists have many challenges unique to our medium, we can still learn much by observing the traditional approaches to lighting practiced by film artists. There we find not only a long, practical experience with the topic, but an established professional jargon that facilitates its discussion.

The classic Hollywood approach to lighting generally involves at least three light sources—namely the key light, fill light, and backlight. Of primary importance among one's lighting tools is the key light. This is the chief light source for the scene and casts the dominant shadows. A fill light is then used to "fill in" or soften shadows created by the key, while the backlight serves to separate the figure from the background. Typically, each major character (or other object of focus) will be assigned a key, fill, and backlight, though often one light can serve more than one figure.

The purpose of lights is to define form by painting the figure with light and shadow. To that end, the key light is generally positioned so its rays strike the figure diagonally from the front, near but off to one side and typically higher than the camera; this is known as "modeling light" because it tends to model the form well and be the most flattering. More direct, frontal lighting is to be avoided, as it will work to minimize shadows and thus flatten the appearance of form. A "skim" or "raking" light is a key light that comes from an oblique angle to the viewpoint; that is, one set more to the side than at a diagonal. It creates strong, long shadows and is useful for capturing surface detail. The key light will be the brightest in the scene and should be placed first.

Try a test render with just the key light in place. Next, place a fill light to soften and deepen the shadows cast by the key. The fill light should be lower than the key in intensity and is also often positioned lower in the scene, to light the figure from slightly underneath. Or, sometimes fill light is set to gently illuminate a portion of the background to diminish a cast shadow.



Thanks in large part to its moody, carefully lit 3D scenery, Myst—from Broderbund Software and Cyan Inc.—showed the world how good-looking a game could be.

One great advantage computer artists have over film artists is their ability to use shadow casting selectively. All 3D programs seem to have some provision for turning shadows on or off, so that either a light doesn't cast them or an object doesn't receive them. Take advantage of this ability. A fill light that softens existing shadows without adding more shadows to the scene can be a great tool. Just because a real-world light would cast shadows doesn't mean every light in your 3D scene has to.

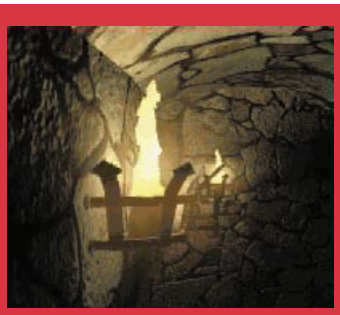
A kicker or backlight—also known as a rim or separation light—provides depth cues by separating the figure from the background. The intensity of illumination should fall between the key light and the fill lights. If there is not sufficient difference in the intensities of these lights, the tonal variations will not work well to suggest form, and the result will be a flatter, less dimensional look. Backlighting works best when directed from above and, obviously, behind the figure. It can be more or less to one side or the other as preferred.

At this point in your lighting setup, it is a good idea to balance the intensity levels of these main lights to achieve adequate illumination for the scene, remembering that to get a good tonal range the lights should decrease in brightness from key light to backlight to fill light. Do test renders and change light levels till the overall brightness and mix of light and dark is to your liking. You have "roughed out the canvas," so to speak. Now look at the scene with an eye to detail and really begin to paint with light.

See Spot Shine

The spotlight is the lighting technician's sable hair brush. Spotlights can be accurately aimed to illuminate objects or areas, to highlight details and direct the audience's attention. (3D Studio lets you place a highlight where you want, then it positions the source light accordingly.) One characteristic of spotlights is falloff: the extent to which the light diminishes in intensity away from the center of the beam. A light with little or no falloff appears as a sharply defined circle (assuming a circular spot), where a light with greater falloff has less distinct edges with a brighter center: the hot spot. A tightly focused spotlight that highlights a very specific area is known as an accent light. A spot aimed so the hot spot "misses" the figure—thus achieving a less intense highlight on the subject itself—is called a feathered light.

You can use multiple spots of varying size and falloff to highlight and brighten key features, but be aware of the amount of illumination you add to



From guttering torches to glowing force fields, atmospheric lighting effects set the mood in Buried In Time, by Sanctuary Woods and Presto Studios.

the scene. As you add more lights to your scene, you'll want to adjust the intensity levels of existing lights so you don't wash out details by over-illuminating them. Avoid overwhelming the scene with shadows that come from all directions because of the myriad lights set about. For most accenting purposes, spotlights can best be set to cast light without shadows.

One interesting lighting tool computer artists have over film artists is the darkon, a light with a negative intensity value. By setting the intensity level below zero, the darkon's rays in fact remove light color from areas they contact. In addition to a darkening effect, the darkon can force a color shift in the affected area. What it's doing is sucking light color from everything it touches in the scene, but doing so selectively. Even white light is composed of RGB values (translated to hue, luminance, and saturation or HLS values); by removing more of the red channel, for example, the darkon can cause the remaining light to appear greenish. It's a neat special effect, but it can be tricky to control. If the darkon is neutral in color—that is, has a zero saturation setting—it reduces the RGB color channels evenly and has a darkening effecting without changing color.

Imitating Mother Nature

Effective use of light and shadow not only helps the look of a scene, it can contribute to its narrative strength while saving the artist time and effort. A very useful tool toward this end is a 2D transparency map used as a gobo. A gobo is a screen that blocks light—or, more usefully, partially blocks light. In computer graphics terms, the gobo is used as a projection light: the opaque areas cast shadows on the scene. With a gobo, it's easy to create the appearance of shadows cast by offstage objects. You can create the effect of dappled sunlight through a forest canopy without having to model a forest full of trees or suggest a jail cell with the stripey shadows of cell bars.

Your gobo can be animated, too, which adds even more detail and depth to the scene with minimal effort: it can be much easier to create a 2D animation in silhouette than to model and animate a comparable scene in 3D. A 3D program that doesn't have projection lights per se but that does have raytracing capability can still make use of the gobo

technique by painting a planar object with the transparency map and positioning it in front of the light source so that it casts the desired shadow.

The gobo technique can also be used with subtler grey scale or color gradations as a computer graphics analog to the photographer's diffuser to soften the effect of a light or break up the evenness of its illumination. This results in a less rigid, more natural appearance. When striving for realistic lighting effects—especially for outdoor lighting—don't be constrained by the limits of the computer screen. The closer lights are positioned to the objects in a scene, the more apparent the effect of falloff will be, and the more flare any cast shadows will display.

Because of its great distance from the Earth, sunlight produces both an even illumination and shadows we perceive as running parallel. Simulate this by placing "sun" light sources at a distance from your scene. Also for simulated sunlight, avoid using attenuated light levels—those that diminish with distance from the source—with the possible exception of a sunset effect.

What Do They Expect?

With these tools in your arsenal, you can use light and shadow to greater effect in your scenes. Beyond making an image look good, lighting gives you the opportunity to play on the psychological connotations with which dark and light are freighted. There are pretty clearly understandable reasons why our primitive ancestors feared the darkness, which have only been reinforced by the symbolic use of light and dark through the ages. In darkness lurks all that is unknown and unsafe—dread, evil, death. By dispelling darkness, light promises truth, brings reassurance, and suggests virtue, happiness, safety.

Likewise, lighting color has significance as well. Orangey-red lights suggest fire, violence, or danger, while yellow light is warm and welcoming. Bluish light can be morbid and is good for suggesting nocturnal scenes. Greenish hues seem unnatural and are just plain creepy. Feathered accent hits of colored lights in an outwardly natural looking lighting scheme can add a nice undercurrent of implied meaning to your scene.

Of course, these expectations can be played with, too. You can build tension in the shadows without fulfilling it, and you can rain horror down upon your scene from a clear blue sky. The point is that, used knowingly, lighting effects can set your audience up just where you want them; what you hit them with is up to you. ■

Dave Sieks is a contributing editor to Game Developer. *You can contact via e-mail at 103302.301@compuserve.com or through* Game Developer *magazine.*