gd

GAME DEVELOPER MAGAZINE

AUGUST/SEPTEMBER 1995

UBM Tech

Then There Was a Doughnut and a Snake...

I guess I'm getting older. While I used to dream about showing up for final exams totally unprepared (well, not so much "dream" as "remember"), the other night I had a new kind of anxiety dream. Apparently I'm more anxious about the future of digital entertainment than I thought.

Okay, my wife Tina and I are in my neighbor's backyard in full scuba gear. We're looking for the hole in my drysuit that flooded the previous weekend. I pick up the phone and it's this sleazy movie producer I know, and he's panicked.

"Val Kilmer has frozen to death! And they're after me next! You have to—" He cries before being cut off. I know, although Mike and I had been in a perfectly legitimate deal to make an interactive game, that he's somehow involved me in one of his more colorful movie-financing schemes. I tell Tina that we have to leave immediately because we're in imminent danger.

So I take her hand and we start moving down the row of chairs in an empty Candlestick Park. Samuel L. Jackson is there, reading the note left by the terrorists, who have him in their sniper scopes but let him go because they don't know where Bruce Willis is. It turns out that the chairs are surrounding the pool John Lilly used to do dolphin communication research in. Standing knee-deep in the pool is my brother, holding up his 16-month-old son, who's babbling but doesn't talk. A bottlenose dolphin swims by, and I wonder if the entropy of dolphin signalling shares characteristics with other signals with complex meanings, such as human language or DNA.

But it turns out that this is not John Lilly's dolphin pool, but an auditorium where they're previewing Windows 95 for $25 a ticket. That's just ridiculous, so Tina and I leave the auditorium for the lobby of our local Pacific Regency theater. Tina tells me to buy the popcorn while she goes ahead and gets us seats. But wouldn't you know it, just as I'm getting in line, Steven Spielberg walks by.

My keen journalistic instinct kicks in and I ask Steven if I could ask him a few questions sometime for the magazine. He agrees and I tell him I'll call him in a few days when I'm prepared. Steven shakes his head and says it's now or never. I get tongue-tied, but the screenwriter William Goldman taps me on the shoulder and says, "It's all changed since *Jaws*." Seizing on that (and forgetting that, given my druthers in real life I'd rather meet Goldman than Spielberg) I ask Spielberg, "Will a video game ever give the director the kind of control over the experience needed to make a *Jaws*? Or a *Schindler's List*?"

Here, unfortunately, is where my dream failed me. Spielberg answered me in depth, but apparently my subconscious gave up on second-guessing reality and I couldn't understand his sentences. I got so frustrated trying to understand him that I thrashed myself into waking.

Dreams are our most powerful fantasies, with the power to weave deep allegory into our mundane personal experiences. My dream was subjectively a complex pondering on the nature of communication and narrative. But even if you can follow the metaphorical thread of my dream, and the unusual all-star cast creates some resonance in you, another person's dream is at best an amusing and inefficient narrative. Why? The three most important reasons are the personal symbolic meaning, the dismissal of explanatory transi-

GAME DEVELOPER

Editor Larry O'Brien
gdmag@mfi.com
Senior Editor Nicole Freeman
76702.706@compuserve.com
Managing Editor Nicole Claro
71743.452@compuserve.com
Editorial Assistant Deborah Sommers
dsommers@mfi.com
Contributing Editors Alex Dunne
75010.2665@compuserve.com
Chris Hecker
checker@bix.com
David Sieks
dsieks@arnarb.harvard.edu
Wayne Sikes
70733.1562@compuserve.com
Editor-at-Large Alexander Antoniades
sander@mfi.com

Cover Photography Charles Ingram Photography

Publisher Veronica Costanza
Group Director Regina Starr Ridley

Advertising Sales Staff
West/Southwest
Yvonne Labat (415) 905-2353
ylabat@mfi.com
New England/Midwest
Kristin Morgan (212) 626-2498
kmorgan@mfi.com

Marketing Manager Susan McDonald
Advertising Production Coordinator Denise Temple
Director of Production Andrew A. Mickus
Vice President/Circulation Jerry M. Okabe
Group Circulation Director Gina Oh
Group Circulation Manager Kathy Henry
Circulation Manager Mike Poplardo
Newsstand Manager Pam Santoro
Reprints Stella Valdez (415) 655-4269

Chairman of the Board Graham J.S. Wilson
President/CEO Marshall W. Freeman
Executive Vice President/COO Thomas L. Kemp
Senior Vice President/CFO Warren "Andy" Ambrose
Senior Vice Presidents David Nussbaum, H. Verne Packer, 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

tions, and the personal nature of the dream experience. You dream alone.

Cinema is the second most powerful narrative force we experience. I'd give the nod to one-on-one storytelling, but raconteurs are sadly rare, and professional storytellers make their meager livings almost exclusively with children. Television affects us more, but as a mental massage, not as a narrative. Literature is broader, deeper, and subtler, but no artists have as much control over the audience as moviemakers. And there has never been a director more in tune with the nature of that power than Spielberg.

I'm not saying that Hollywood is producing better movies than it has in the past or that Spielberg is artistically head-and-shoulders above every director, living or dead. But I do assert that audience control has never been as polished. I'll even commit sacrilege and say that Spielberg routinely plays the audience better than Hitchcock.

What about the potential of digital entertainment? The engagement of the player is so focused that peripheral elements seem to be useless — if it's not part of the gameplay, it's not part of the game. Putting the player in control seems central, but every other form of fiction relies on plots driven by the narrator, not the listener. And the best fiction relies on plot points that appear inevitable, but only in retrospect. With the possible exception of Myst, no game has achieved that.

It's becoming cliche to equate current interactive entertainment with the early days of film, when films were spectacle with little or no plot. The comparison is generally used to reassure us that we'll get those talents able to combine art and popularity—the Griffiths, Eisensteins, Hitchcocks, Wilders, Scorseses, and Spielbergs. But will we, even in a hundred years? Even my subconscious doesn't know.

Your Hour's Up

Before being wildly distracted by this dream, I had planned to write this editorial on the E3 show in Los Angeles, a show that couldn't contrast more with the Computer Game Developer's Conference I wrote about in our last issue ("Brain Goes Whoosh," Game Plan, June/July 1995). E3 was an awesome spectacle: 750,000 square feet of exhibition space devoted to the interactive entertainment industry. Of that, about 700,000 square feet seemed devoted to fighting games—now in flat-shaded, three-dimensional polygons! The lack of creativity in the cartridge industry was overwhelming. The home computer CD-ROM game is where all the innovation is. I felt like I was seeing Comdex in 1984—with mainframe vendors amused at the notion that a general purpose desktop computer could take over from their specialized hardware.

As Chris Hecker points out in this month's "Behind the Screen," the new generation of cartridge machines on display were technically less than awe-inspiring. My first reaction to Nintendo's decision not to unveil the Ultra 64 was that it was disastrous not to show a next-generation machine. But after seeing the competition (and after a couple of quiet meetings off the show floor), I think that Nintendo is the once and future king of the cartridge market.

It's not a great surprise that I ended up dreaming about Hollywood after attending E3; Hollywood and the cartridge industry are staring with incestuous lust into each other's eyes, totally missing the fact that the home computer is going to drive the U.S. gaming industry and that U.S. software developers are poised to become as dominant in the entertainment software industry as they are in the applications industry. Those developers aren't interested in shelling out tens of thousands of dollars for a development license, much less millions; they're aiming for the PC market, knowing that if they deliver a Doom, a Wing Commander III, or a Magic Carpet, the cartridge manufacturers will fight to get them, with their checkbooks open. Now there's a dream to lull you to sleep.

Finally, I'd like to once again thank those who made the first year of *Game Developer* such a success. Since our last issue, *Game Developer* was honored by the Computer Press Association as a runner-up for "Rookie of the Year" and won the WPA's "Best New Magazine-Trade" Maggie Award. Without the volunteer efforts of the editorial staff in the first year and the great writers who took a chance with us, we wouldn't have had a magazine, but without you, the readers, we wouldn't have had a success. The Maggie belongs to them. But I get to hold on to it. ■

Larry O'Brien
Editor

ERRATA

Okay, so we made a few errors in our last issue. They all appeared in "Perspective Texture Mapping Part II: Rasterization," Chris Hecker's June/July installment of Behind the Screen.

- Equation 1 on page 19 has a misplaced parenthesis that renders it incorrect. The equation should read as:

$$x_{\text{int}} = \left\lceil \left(\frac{x_1 - x_0}{y_1 - y_0} \right) (y - y_0) + x_0 \right\rceil$$

- On page 19, Chris Hecker wrote:

  "If we're exactly on an integer pixel center we will light the pixel, but if our x is at all greater than the integer—*to the left of the pixel center*—the ceiling will bump us up to the next pixel that's strictly inside the edge."

  It should read:

  "...to the right of the pixel center."

- Oren Patashnik and Ronald L. Graham were credited with having written *Concrete Mathematics*. D.E. Knuth was also an author of the book.

The Windows 95 Game Plan

Alex Dunne

The launch of Windows 95 is weeks away. At some point in the coming months it's going to alter your development plans. How will you prepare for game development's newest arena?

Characterizing your development efforts for the coming six months will require a little Q and A: Will you migrate all game development over to Windows 95 immediately, hold out and continue to work on DOS-based games, or take the middle-of-the-road approach as you divert some DOS development staff to Windows 95 efforts?

I talked to a sample of game developers in the industry, and found that there's no prevailing route. Also, I found no correlation between the choices companies made and the size of their company. One question many companies wanted an answer to, though, was how quickly Windows 95 would be adopted by the game market.

Moore's Launch Curve

There's a theory that might help answer this question. In a recent column in *Software Development* ("The Launch Chasm," Tools of the Trade, July 1995) Warren Keuffel describes Geoffrey Moore's concept of "the launch chasm." This phenomenon occurs after a product release in high-tech markets, and by studying it you get a sense of what will likely happen once Windows 95 becomes available.

Moore said that the market for a given high-tech product can be represented by a bell curve made up of five distinct categories of consumers, each of which roughly corresponds to one standard deviation. These groupings are called (in order of adoption) innovators, early adopters, early majority, late majority, and laggards. As shown in Figure 1,

Figure 1. Moore's Launch Curve

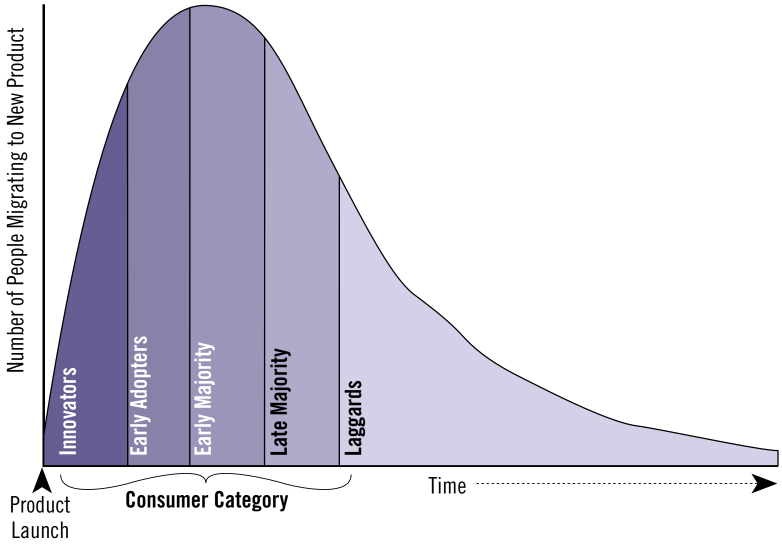
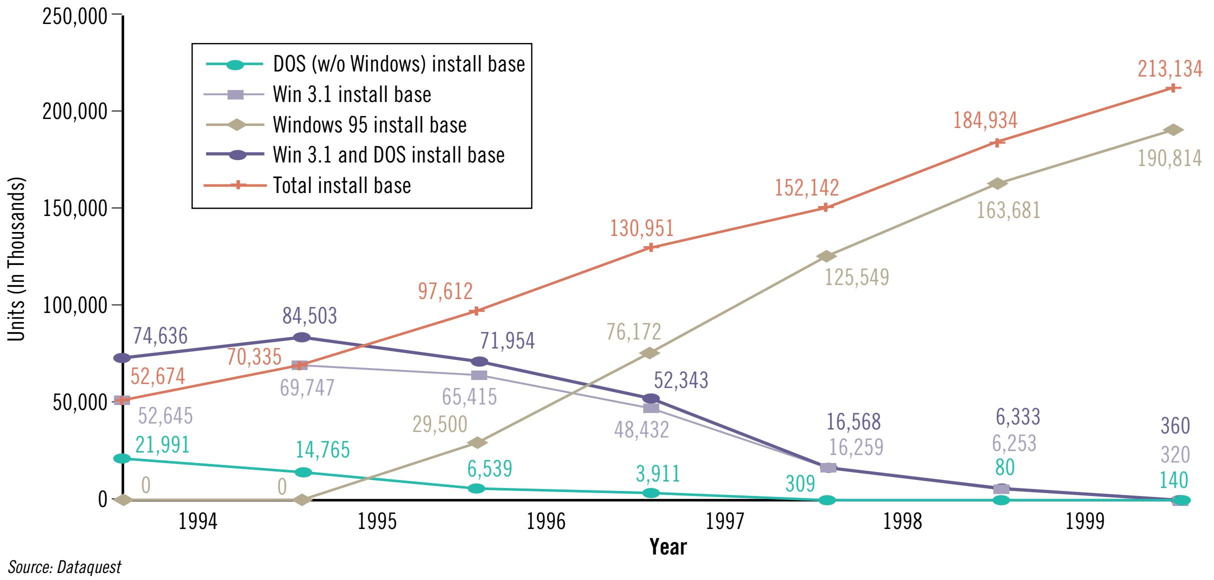


Figure 2. Software Installed Base



Source: Dataquest

the first two groups of consumers typically understand and are enthusiastic about new technology and actively seek it out. Unfortunately this group is relatively small. Beginning with the early majority, skepticism increases causing longer and longer delays before people adopt the new technology.

Leading the Way

Although game developers releasing Windows 95 products this year will initially face a market of limited size (compared to the DOS market), there's an upside: this smaller market of innovators and early adopters will consist of a proportionately high amount of hard-core gamers, as most devoted gamers typically live on the technological "cutting edge."

Companies moving development quickly over to Windows 95 will get a head start in a number of areas over those who stick with the tried-and-true DOS world. In addition to the skills developers learn, the benefits will propagate down to staff in marketing, sales, documentation writers, and most importantly, employees in technical support.

Bringing Up the Rear

The majority of consumers will take longer to adopt Windows 95. Skepticism, ignorance, apathy, dedication to the old way of doing things, or simply insufficient hardware will hold most people back for at least six months, if not longer. Dataquest has projected the growth of the Windows 95 installed base as seen in Figure 2. The company anticipates the installed base of Windows 95 surpassing that of Windows 3.1 and DOS combined sometime in mid-1996. Dataquest's projections for shipments of Windows 95 are depicted in Figure 3, which illustrates that shipments of Windows 95 will surpass shipments of Windows 3.1 around the end of this year.

One interesting aspect about Dataquest's predictions is its positive outlook for Windows 95 through the end of the decade. Windows 95's successor, Cairo, is scheduled for release in two to three years, which will take some of the wind from Windows 95's sails. Beginning in 1998, I'd predict fewer sales than Dataquest projects.

For now however, DOS is with us, and I suspect it will remain with many consumers for years (my mother still uses an Apple IIE, so I'm sure that some people will be working on 386s in the year 2010). Many companies are banking on the DOS market for the foreseeable future, and I have no doubt that this market will continue to be lucrative for years. The majority of consumers who initially shun Windows 95 may not be the "digerati" who consume games on a weekly basis, but many are consistent, if not frequent, game buyers. In addition, Windows 95's backwards compatibility keeps both markets open to your product: you'll sell to both the early adopters and the hold offs.

Staying the Course

This is the route that Nova Logic is taking. I spoke with David Seeholzer, vice president of software, and asked him how many Windows 95 games the company would be releasing this year.

"Zero," Seeholzer replied. "We're a pretty small company and don't have a group of R&D people sitting around and playing with stuff. All of our people are pretty much working on our core products that are on the verge of getting released within the next couple of months or next year. As we have programmers come free from projects in the next couple of months, we'll have one or two of them play around with Windows 95 and evaluate it. We're always looking at new opportunities to do games in a Windows environment. But right now we just haven't spared the manpower to look into it."

I inquired whether this meant that DOS would continue to be Nova Logic's primary platform for the foreseeable future. "Yes, though I think we recognize that DOS isn't going to last forever," Seeholzer said.

"There are things that we like about DOS in terms of doing high performance games, it just doesn't get in your way. And the original Windows [3.x] got in your way big time. WinG was an effort to bridge the gap, which was successful on a few fronts, but not on as many as we would have liked. Windows 95 is trying some additional things, but it still has some major issues that we're concerned about. It's our assumption that the trend will continue, though–that eventually, there will be a Windows environment that we'll feel comfortable doing our kind of games on. Not to mention the fact that DOS will probably become an unviable way to publish.

"We know that there's change ahead, but we're currently selling our products to the huge installed base of people who can run DOS, whether or not they also run Windows... It's quite possible that we would have something next year. Because we are small, we're able to move pretty quickly on things, and it's not at all unlikely, as we finish our current round of products that we'd look into it, figure out what we could do, and start to [develop for Windows 95]," Seeholzer said.

However, if your shop is one that will continue a solid effort in the DOS development world, you may be sacrificing an important technological lead to those companies that dive into Windows 95 and train their staff early. Don't give away a head start if you don't have to; small pilot projects might be the answer.

Getting the Feet Wet

One game developer that seems to be taking this route, though the company doesn't consider its game a "pilot project" as such, is LucasArts. Tom Sarris of LucasArts described the company's first Windows 95 title, "Indiana Jones and His Desktop Adventures."

"I hate to use this word, but it's a 'simpler' title than perhaps what you're used to seeing from us," Sarris remarked. "Here's a game that you can play in between a half hour and an hour. You can load it onto your PC and goof around with it when you have an extra half hour or so. It's actually just on one high-density floppy disk. It's a top-down view, much like [the 16-bit Nintendo game] 'Legend of Zelda,' and has over one billion game play scenarios."

LucasArts's "Indiana Jones and His Desktop Adventures" is the only Windows 95 title that the company is planning to release this year, and may turn into a series of games.

Taking the Plunge

Finally, I spoke to a company that's decided to dive straight into the Windows 95 arena. San Francisco-based 47-Tek released a sophisticated 3D fighting game last year called Sento, which was optimized for Matrox video accelerator cards. Mark Hirsch, the president of 47-Tek, said that the company would finish up 1995 with one more DOS game and then is calling it quits with that platform.

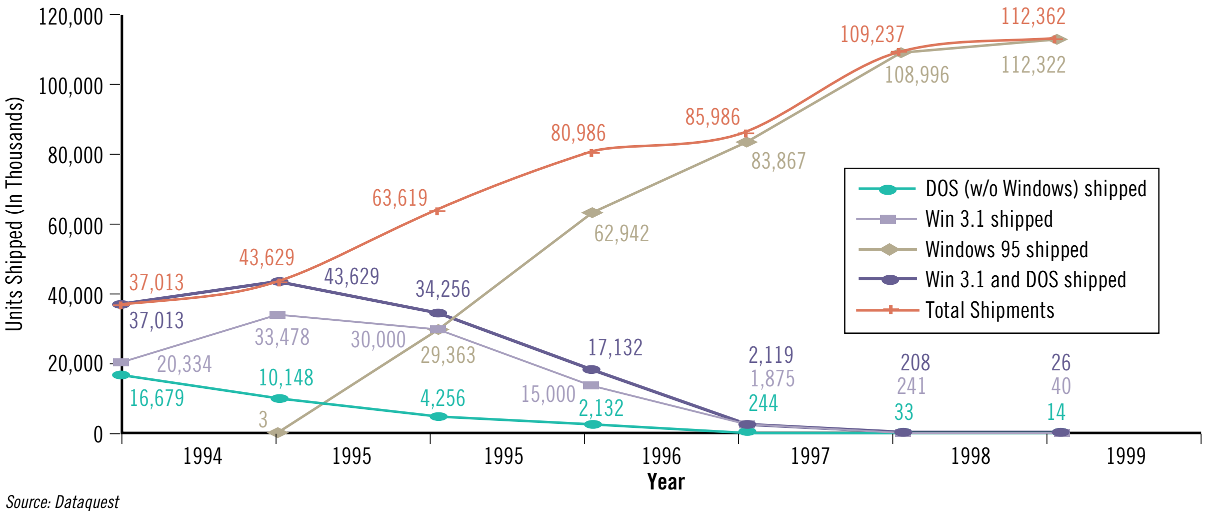
"We're going to do three titles next year and we'll do a Windows 95 version of at least two of them," Hirsch said. "The other three will go on the [Sony] PlayStation, [Sega] Saturn, maybe [Nintendo] Ultra64."

When asked if he considered DOS dead with the release of Windows 95, Hirsch gave a straight and simple, "Yes." His company is using the Reality Lab 3D API contained in the Windows 95 Game SDK, and will design upcoming titles to support accelerators that conform to the 3D DDI standard.

Both 47-Tek and Nova Logic are small companies, so it makes sense for them to concentrate limited resources on specific technologies. The fewer platforms they are divided among, the more quickly they'll exploit the power of those platforms. LucasArts, on the other hand, has a larger staff of developers and can afford to divert resources in more directions without as large a sacrifice to their overall development efforts.

What are your plans for game development over the next six to twelve months? Drop me a line and tell me about your experiences so far or strategies your shop is using to prepare for the move to Windows 95. ■

Figure 3. Software Shipments



Source: Dataquest

Back

Poser, Dude!

Nicole Claro
Barbara Hanscome

New Bit Blasts features! "The Buzz" is a roundup of pertinent corporate news. "Schmooze News" is industry gossip (you just might see yourself in there). Watch this space for more new and exciting things.

We all know how difficult it can be to create lifelike humans in games and multimedia applications (we've discussed tough hair situations here before, you might recall). Fractal Design Corp. recently introduced Poser, an application that lets you do electronic "life drawing."

Poser is designed to work closely with other two- and three-dimensional applications and provides stylized models—both male and female—that can be moved, modified, shaped into any pose, and viewed from any angle. It features a choice of body sizes (from infant to adult to superhero) and direct figure manipulation—which means when you move a body part, all connected parts will automatically move accordingly. Once you've shaped your model, you can apply multiple light sources and bump and texture maps to create a fully rendered human model. The software runs on Macintosh and Power Macintosh and will be available through August 31, 1995 for $99.

■ For more information contact:
Fractal Design Corp.
335 Spreckels Dr.
Aptos, Calif. 95003
Tel: (408) 688-5300

Multimedia galore

MFactory has announced mTropolis, a fully object-oriented development system for multimedia applications. Object orientation lets artists rapidly create complex environments and store, share, and reuse any object or combination of objects, predefined or user created. It has a fully integrated debugger and multiplatform portability for authoring and playback. mTropolis also has an open architecture, which lets you transparently and smoothly enhance your system. mFusion, the scalable technology at the core of mTropolis allows you to create titles that execute extremely fast, even on lower-end platforms. mTropolis will initially be available as an editor for Macintosh and Power Macintosh and a player for Macintosh, Power Macintosh, and Windows 3.x. Suggested retail price is $4,995 for single-seat units.

■ For more information contact:
mFactory Inc.
1440 Chapin Ave.
Suite #200
Burlingame, Calif. 94010
Tel: (415) 548-0600

Lightning Fast

Criterion Software has announced RenderWare Lightning. The real-time three-dimensional graphics games library was specifically created for development on many platforms including DOS, Windows 95, and Power Macintosh. It features fast software rendering on PCs, support for three-dimensional accelerators on PCs and three-dimensional consoles, an easy-to-use API, powerful importing tools, and automatic cross-platform support. RenderWare Lightning costs $995 for a personal license and $10,000 for a commercial license.

■ For more information contact:
Criterion Software
17-20 Frederick Sanger Rd.
Guildford, Surrey
GU2 5YD, U.K.
Tel: (408) 749-0493

the BUZZZ

- Interplay is all over the map. The company has announced an agreement with GEnie Online Services to develop two games available exclusively on GEnie for six months. In August, the first title, a deluxe version of Descent, originally developed by Parallax studios, will makes its online debut on GEnie. In May, Interplay acquired Shiny Entertainment, makers of the huge hit Earthworm Jim. This marked Interplay's first major acquisition.
- Nintendo's got two new deals going. The company has entered into an ecxlusive agreement with Multigen, under which Multigen will create three-dimensional development tools for Nintendo's 64-bit machine, the Ultra 64. One of the users of those tools will probably be Rare, a company based in the U.K., which will develop a series of new 16-bit and 64-bit games for the Ultra 64 and other NIntendo platforms as well as games for Game Boy and Virtual Boy. Rare, which collaborated with Nintendo on the 1994 game Donkey Kong Country, is the first video game development company outside of Japan that Nintendo has invested in.
- Leo the Lion has gone digital. Major film studio MGM recently announced the inception of MGM Interactive, a new multimedia arm of the organization. The new division will produce multiplatform and online titles (some new, some based on existing material) on its own and in conjunction with various other companies.

Yamaha Breaks the Bank

Yamaha's new YGV612 three-dimensional graphics controller chip features texture mapping, Gouraud shading, and Z buffering. Geared toward DOS and Windows 95 game developers, the new chip lets your games run in 640-by-480, 64-bit colors with high resolutions. It supports a DRAM frame buffer, direct PCI bus support, and integrated DAC. It is currently available in sample quantities for $40. You can't beat that with a stick!

■ For more information contact:
Yamaha Systems Tech. Inc.
100 Century Center Ct.
San Jose, Calif. 95112
Tel: (408) 467-2300

Genre Busters

The untapped market of female teens might not go for the traditional shoot-em-up game, but will it go for the "social adventure"? Or intensive personality analysis? That's what two game developers targeting female teens are hoping for.

At press time, Games for Her—a new division of American Laser Games—was planning the release of its first title, McKenzie and Co. According to the company's press materials, the title is aimed at female gamers aged 9 to 15 and will treat players to the "real life dilemmas that make going to an all-American high school such an adventure." With the help of a CJ-17 jeep named McKenzie and a band of wacky pals, the player must successfully juggle the demands of school, romance, friends, jobs, and family by avoiding "bad" decisions throughout the game, such as cutting class or lying to parents. The player also gets to establish a relationship with the boy of her dreams and shop for a prom outfit with an elaborate "shopping engine" featuring 400 clothing options provided by Urban Outfitter and Limited II stores.

Reading the game description, you can't help but wonder if McKenzie and Co. could be the CD-ROM equivalent of the Brady Bunch, feeding kids the illusion of an ideal teen world that doesn't exist. Folks sensitive to gender stereotyping might be appalled at the game's shopping engine. And where are the real teen dilemmas of the 90s—drugs, sex, teen pregnancy, interracial dating, even the complex issues surrounding tattoos and nipple rings?

Games for Her's director of marketing, Patricia Flanigan, argues that while these issues are real for today's teens, they aren't necessarily what young women want in a game. The "social adventure" themes featured in McKenzie and Co.—school, romance, career aspirations, and friends—were determined from intensive focus group research with girls from a wide range of socio-economic backgrounds in Albuquerque, N.M. "It's what the girls tell us they like," says Flanigan. "The world is tough for teens, and I think something like this allows them to escape and to have fun." Flanigan says the social adventure is just one of four genres Games for Her has developed for teen women: mystery is another; Flanigan will not reveal the other two at this time.

Flanigan hopes to explore the heavier side of teen life in Games for Her's new online service Her OnLine, a joint venture with teen book publisher Daniel Weiss Associates Inc. The new service, which at press time was scheduled to premiere in late summer, will feature chat rooms and opportunities for teen women to hook up with electronic pen pals and other McKenzie gamers.

Flanigan is also being aggressive in trying to open new marketing channels for girls games, placing the title in outlets other than computer stores. McKenzie and Co.'s launch will begin with a national mall tour and will include several alliances and point-of-purchase tie-ins with retailers

Schmooze news:

Game Developer's Conference Doubles again... More than 2,400 attendees converged on the Santa Clara Westin April 22-25 for the 7th Annual Computer Game Developers Conference (CGDC). Rival operating system vendors Microsoft (Windows 95) and IBM (OS/2 Warp) were both out in force, giving goodies to developers. Apparently, the next killer game may propel its operating system into top position and both companies are doing their best to woo game developers. It was a great deal for people who wanted to load a few extra operating systems on their computer.

Logitech again threw the opening night bash with the ever popular velcro wall people-throw, munchies, and libations. It was difficult to find friends as the room was dark, the spotlights blinding and the music deafening. Industry veterans moved out into the corridor and to the hotel bar to schmooze where they could be seen (and heard!)

The unscheduled but traditional jam session at the CGDC spilled over into an extra night with our talented digital musicians going analog to the delight of partying gamers.

On stage were Dave Schultz (DBS Music) and Dave Albert (Sega). Brian Moriarty (Mpath) and George Sanger (Big Fat) provided vocals. Michael Land (LucasArts) and Mark Miller (Sega) played bass. Neal Grandstaff (Dynamix) was on drums and vocals.

Guitarists included Craig Utterbach, Charlie Albert, David Albert (Sega), and Jim Donfrio (Dijon). Burke Trieschmann blew harmonica. Rob Wallace (Wallace Sound & Music) and Alexis Utterman (DeMaria Studios) traded off at the keyboard. Don Griffen (Computer Music Consulting) did the honors on trumpet.

Microsoft Woos Game Developers... Microsoft rented Great America for its game developer guests. After dropping over The Edge a few times, developers were softened up for Microsoft's Game evangelist, Alex St. John, whose dynamic presentation (with backing from the Game SDK programmers acting as a sort of Geek Chorus) was designed to convince developers that Windows 95 will propel the PC in to game ascendancy. The day after this awesome party, Microsoft held a seminar for developers who want to use the Microsoft Game SDK. A great time was had by all and Microsoft proved it could relate to game developers on their own level after all.

Nintendo Postpones 64-bit Gameplayer... Nintendo's Ultra 64 next-generation machine won't be released as planned for Christmas season, but the delay could hurt competitors as much as it could Nintendo, with a number of buyers likely to wait until all the new machines are available for comparative shopping. The game system is intended to sell for about $250 and to offer performance approaching that of $20,000 workstations. The major concern I've heard about the Ultra 64 is the cost of the individual game cartridges: because of the large amount of RAM in the cartridges, they will be substantially higher than the competition's games, which will ship on CD-ROMs.

The Word from E3... The first Electronic Entertainment Expo in Los Angeles had more than 4,000 attendees. With a surprise announcement at E3, Sega began shipping the Saturn to American markets at the beginning of E3 weekend. Countering with a surprise of its own, Sony announced that while the Playstation will not ship until September, it will retail at $299, a solid slap at the $399 price tag of Sega's Saturn.

Nintendo's introduction of the Virtual Boy met with skepticism from many in the game development community, who are uncertain about the success potential of a three-dimensional game platform that can only display two colors—red and black.

One major problem with E3 was the inability to hear anything—the companies turned up the sound on their demos in an unbelievable "stereo war." To quote one game developer, "I knew we were in trouble when I was nearly run over by the high school marching band because I couldn't *hear* them!"

Windows 95 on August 24... Microsoft, fighting rumors that Windows 95 might turn into Windows 96, has named August 24 as the date the long anticipated software will ship to retail stores. Secret documents leaked to this column via the Internet indicate that Microsoft may plan to achieve this 1995 ship date by buying the current year, renaming it "Year M" and postponing the year 1995 until next year.

Got gossip? E-mail The Gossip Lady *at 71501.3553@compuserve.com.*

and manufacturers of products popular with female teens, including Sassaby Cosmetics, Sam & Libby shoes, and Limited II clothing stores. Packaged with the game will be a music CD and a special title called "Sure She Can," featureing the stories of teen women around the country who have made a difference.

The Most Important Person

GirlGames in Houston, Texas, is taking a different approach to reaching female teens: "Being You," the company's first title, is geared to young women 14 and older and will focus less on finding a guy and more on finding yourself. Subtitled "The CD-ROM for girls to learn more about who they are and who they can be." It will explore their interests and goals using hundreds of activities based on personality profile tests. Players will download new content quarterly from an online service and have access to a resource list of organizations covering a wide range of subjects—from astrophysics to ceramics—that young women can contact for more information.

GirlGames president Laura Groppe feels the key to this market isn't a specific genre or hook, but titles that put power into the users hands. "Our player will get to the next quiz or activity because of decisions she has actively made. She's figuring out what makes her tick, and that is very empowering." Groppe is keeping quiet about who the game's publisher will be and any other details about the product, but you can bet boy chasing won't be part of it. "Girls are approaching the 21st century without a lot of guidance, and technology is going to permeate their lives from every angle," says Groppe. "We want to equip them with the tools they need."

Whether the squeaky clean hijinks featured in McKenzie and Co. or the personal explorations featured in Being You will succeed or fail is yet to be seen. Both Flanigan and Groppe say they are devoted to sticking with this market regardless: "We're in this for the long haul," says Flanigan. ■

Nicole Claro is managing editor for Game Developer.

Barbara Hanscome is managing editor for Software Development.

Back

Perspective Texture Mapping, Part III: Endpoints and Mapping

Chris Hecker

If you think we've covered everything on perspective texture mapping, you're wrong. In Part III of this ongoing series, we get a close look at the math involved in endpoints and mapping.

By the time you read this article, the Electronic Entertainment Expo (E3) will be long over, but in the time warp of magazine article submission deadlines it was just last weekend in Los Angeles. E3 is the game industry's attempt to break from the huge toaster, car stereo, and microwave oven event that is the Consumer Electronics Show. Whether this breakaway was successful remains to be seen, but one thing is certain: the new generation of video game consoles garnered a lot of attention and floorspace. Atari, Sega, 3DO, and Sony battled for developers' attention, each hoping to wow people with its machine's high-end features and get the really cool games developed for its platform—the Jaguar, the Saturn, the Multiplayer, and the PlayStation, respectively.

The reason I bring this to your attention is the one feature advertised above all others for each machine is—you guessed it—texture mapping. Each company claims its system has the most realistic texture mapping, or the fastest texture mapping, or the least expensive texture mapping.

I'll mention one important caveat before I lay into this generation of hardware with technical criticism. It's completely unclear what relation, if any, exists between texture mapping quality and overall game quality (and certainly sales). Super Mario Bros., for example, has absolutely no texture mapping, but it sure is a great game, both from a playability and profitability standpoint.

Keeping that in mind, the texture mapping on these machines sure does suck.

How do they screw up texture mapping? Let me count the ways. First and most noticeable is that all the texture mapping hardware in this generation is affine. Affine texture mapping, as we discussed in "Perspective Texture Mapping Part I: Foundations" (Under the Hood, April/May 1995), assumes the equation to map screen coordinates to texture coordinates is linear. This results in really nasty texture warping when the linear equation and the true equation start to differ by a substantial amount. The ironic part about affine texture mapping is these two equations differ most when the textures are very close to your viewpoint, which makes the problem easy to spot.

You can clearly see this for yourself in almost every game produced for these machines. Check out the floors in some of the fighting games or the walls in walkthrough or driving games. Get real close and prepare for a stomach-churning texture dance.

Second, and particularly germane to today's discussion, some of these machines only support integer-texture coordinates, that is, the vertices of the polygons can only correspond to integer coordinates in the source bitmap. This wouldn't seem so bad until you realize one of the ways to combat the affine problems I've mentioned is to subdivide your polygons until the linear equation is a closer fit (we'll cover this technique in the near future). The subdivision points are not likely to fall on integer texture coordinates, so this hardware forces you to snap to the nearest integer, resulting in jitter that's plainly visible in the games.

Finally, a few of the machines only support integer screen-space polygon vertices. In other words, if your polygon comes out of your three-dimensional transform pipeline with noninteger endpoints (as it's very likely to do) you've got to snap the vertex to an integer pixel location, which causes even more jitter. Conveniently for the purposes of this article, this is the exact jitter problem we introduced into our own texture mapper when we converted from floating-point to integer rasterization ("Perspective Texture Mapping, Part II: Rasterization," Behind the Screen, June/July 1995). Of course, we haven't spent millions of dollars on developing a piece of hardware and marketing it, so we can fix our jitter problem pretty easily.

Figure 1. Equations 1 Through 3

$$x_{\text{int}} = \left\lceil \left(\frac{x_1 - x_0}{y_1 - y_0} \right) \left(y - y_0 \right) + x_0 \right\rceil \qquad (1)$$

$$\left\lceil \frac{a}{b} \right\rceil = \left\lfloor \frac{a-1}{b} \right\rfloor + 1 = \left\lfloor \frac{a-1}{b} + 1 \right\rfloor = \left\lfloor \frac{a-1+b}{b} \right\rfloor \qquad (2)$$

$$\frac{a}{b} = \left\lfloor \frac{a}{b} \right\rfloor + \frac{a \bmod b}{b} \qquad (3)$$

Jitter Bug

We don't have the space to do a total review of the work we covered in my first two columns on texture mapping. However, the five-second summary is as follows. In the first column, we derived the perspective texture mapping equations, including the equations for perspective projection and those for stepping the texture coordinates across the destination polygon (these step values are called the gradients). We also looked at how to correctly sample with subpixel accuracy. This last topic caused us to investigate how to get rid of the cost of this subpixel accuracy while retaining its advantages, and in the second column we showed how to do this using a digital differential analyzer (DDA). We converted our original floating-point rasterizer to an integer DDA to realize the savings, but we uncovered a nasty jitter as our polygon moved and animated.

This jitter was introduced because our triangle gradients are calculated from the endpoints of the triangle, and those endpoints, when restricted to be integers, change by a relatively large amount from frame to frame. (The mathematically inclined among you will notice that the gradients are calculated from two times the signed area of the triangle [which is also the cross product]. When the endpoints are truncated to integers this area changes, altering our gradients and causing the jitter.)

What we need is better precision on the endpoints, but we want to keep the advantages of using a DDA rasterizer. Enter fractional endpoints.

Fractional Endpoints

When I say fractional endpoints, the first thing that comes to mind is fixed-point math. While we are going to be using fixed-point numbers to represent our vertices and to give us the extra precision we need to avoid the integer jitter, you'll see we're not going to be rasterizing the edges using the familiar fixed-point increments. As usual, to pack all the information we need into this article, I'm not going to be able to describe the basics of fixed-point math. For a description of fixed-point math that's easy to understand, I suggest reading Michael Abrash's *Zen of Graphics Programming* (Coriolis Group, 1994) or the *PC Game Programmer's Encyclopedia*, which is a neat freeware programming book available via ftp on x2ftp.oulu.fi.

We'll use 28.4 fixed point for our endpoints. I'm going to use the integer-dot-fraction notation for fixed-point numbers, so 28.4 means we have 28 bits of (usually signed) integer and four bits of fractional precision.

We'll use this format for two reasons. First, four fractional bits is enough to eliminate the jitter. Second, I happen to know that the Windows NT polygon rasterizer can be set up to do correct top-left 28.4 rasterization, and it always helps to have a proven version against which to test (although I won't show it here, we can write a program that rasterizes a polygon with our code, then rasterizes the polygon with Windows NT's rasterizer, so we can check for differences to test our rasterizer). Once you see how the math works you'll be able to derive a rasterizer for whatever fixed-point format you like best.

As I hinted before, instead of using a fixed-point or floating-point incremental step to move from one scanline to the next, as our first rasterizer did, this rasterizer will use an error-term DDA (much like the Bresenham line-drawing algorithm, covered in most graphics books). However, unlike most DDA rasterizers you've probably seen, our DDA parameters will be initialized with fixed-point numbers instead of integers.

We'll start by defining exactly what we mean by fractional endpoints. From here on out, x and y are real numbers, not integers, and their values are m/F and n/F, respectively. The numbers m and n are integers, and F is the scaling

Figure 2. Equation 4

$$x_{\text{int}} = \left\lfloor \frac{F \Delta m y - \Delta m n_0 + \Delta n m_0 - 1 + F \Delta n}{F \Delta n} \right\rfloor \qquad (4)$$

factor for whatever fixed-point format you're using. For 28.4 fixed-point, F = 16, for 16.16, F would be 65,536, and so on. To convert from the fixed-point values to real numbers we divide by the scaling factor. You commonly see the opposite of this when you multiply a floating point number by the scaling factor to get its fixedpoint value. Here are some useful equations:

$$\Delta x = x_1 - x_0 = \frac{\Delta m}{F} = \frac{m_1 - m_0}{F}$$

$$\Delta y = y_1 - y_0 = \frac{\Delta n}{F} = \frac{n_1 - n_0}{F}$$

We'll be reusing some of the formulas we derived in the first two columns. Refer to Figure 1 for Equations 1 through 3. The variables a and b are integers in these equations—remember that the mathematically defined `mod` operator (used in Equation 3) probably behaves slightly differently than the modulus operator in your chosen programming language. See the `FloorDivMod` function in Listing 1 for the correct implementation and "Perspective Texture Mapping, Part II: Rasterization" for an in-depth discussion. Equation 1 shows the real formula for a left edge under our fill convention (right edges are the same equation minus one). Let's rewrite Equation 1 to use fixed point:

$$x_{\text{int}} = \left\lceil \left(\frac{\frac{\Delta m}{F}}{\frac{\Delta n}{F}}\right)\left(y - \frac{n_0}{F}\right) + \frac{m_0}{F} \right\rceil$$

If we do some basic algebra and use the ceiling-to-floor conversion in Equation 2 (you can move integers into and out of a ceiling or floor) on this we get the equation pictured in Figure 2.

Next we'll introduce the symbol R (Why R? I don't know, mostly because I'm running out of letters in the alphabet!), set it equal to the numerator so things look pretty, and finally use Equation 3 (the relationship between a rational number and its floor and `mod`) on the R/FΔn term inside the floor to give us our initial condition:

Listing 1. Changes for Fractional Endpoints (Continued on p. 21)

```
typedef long fixed28_4;
inline fixed28_4 FloatToFixed28_4( float Value ) {
    return Value * 16;
}
inline float Fixed28_4ToFloat( fixed28_4 Value ) {
    return Value / 16.0;
}
inline fixed28_4 Fixed28_4Mul( fixed28_4 A, fixed28_4 B ) {
                            // could make this asm to prevent overflow
    return (A * B) / 16;    // 28.4 * 28.4 = 24.8 / 16 = 28.4
}
inline long Ceil28_4( fixed28_4 Value ) {
    long ReturnValue;
    long Numerator = Value - 1 + 16;
    if(Numerator >= 0) {
        ReturnValue = Numerator/16;
    } else {
        // deal with negative numerators correctly
        ReturnValue = -((-Numerator)/16);
        ReturnValue -= ((-Numerator) % 16) ? 1 : 0;
    }
    return ReturnValue;
}
struct POINT3D {
    fixed28_4 X, Y;
    float Z;
    float U, V;
};
inline void FloorDivMod( long Numerator, long Denominator, long &Floor,
                long &Mod )
{
    assert(Denominator > 0);                        // we assume it´s positive
    if(Numerator >= 0) {
        // positive case, C is okay
        Floor = Numerator / Denominator;
        Mod = Numerator % Denominator;
    } else {
        // Numerator is negative, do the right thing
        Floor = -((-Numerator) / Denominator);
        Mod = (-Numerator) % Denominator;
        if(Mod) {
            // there is a remainder
            Floor--; Mod = Denominator - Mod;
        }
    }
}
gradients::gradients( POINT3D const *pVertices )
{
    int Counter;
    fixed28_4 X1Y0 = Fixed28_4Mul(pVertices[1].X - pVertices[2].X,
                                pVertices[0].Y - pVertices[2].Y);
    fixed28_4 X0Y1 = Fixed28_4Mul(pVertices[0].X - pVertices[2].X,
                                pVertices[1].Y - pVertices[2].Y);
    float OneOverdX = 1.0 / Fixed28_4ToFloat(X1Y0 - X0Y1);
    float OneOverdY = -OneOverdX;
    for(Counter = 0;Counter < 3;Counter++)
    {
        float const OneOverZ = 1/pVertices[Counter].Z;
        aOneOverZ[Counter] = OneOverZ;
        aUOverZ[Counter] = pVertices[Counter].U * OneOverZ;
        aVOverZ[Counter] = pVertices[Counter].V * OneOverZ;
    }
    dOneOverZdX = OneOverdX * (((aOneOverZ[1] - aOneOverZ[2]) *
            Fixed28_4ToFloat(pVertices[0].Y - pVertices[2].Y)) -
            ((aOneOverZ[0] - aOneOverZ[2]) *
            Fixed28_4ToFloat(pVertices[1].Y - pVertices[2].Y)));
    dOneOverZdY = OneOverdY * (((aOneOverZ[1] - aOneOverZ[2]) *
            Fixed28_4ToFloat(pVertices[0].X - pVertices[2].X)) -
            ((aOneOverZ[0] - aOneOverZ[2]) *
```

Listing 1. Fractional Endpoints (Continued from p. 20)

```
            Fixed28_4ToFloat(pVertices[1].X - pVertices[2].X)));
    dUOverZdX = OneOverdX * (((aUOverZ[1] - aUOverZ[2]) *
            Fixed28_4ToFloat(pVertices[0].Y - pVertices[2].Y)) -
            ((aUOverZ[0] - aUOverZ[2]) *
            Fixed28_4ToFloat(pVertices[1].Y - pVertices[2].Y)));
    dUOverZdY = OneOverdY * (((aUOverZ[1] - aUOverZ[2]) *
            Fixed28_4ToFloat(pVertices[0].X - pVertices[2].X)) -
            ((aUOverZ[0] - aUOverZ[2]) *
            Fixed28_4ToFloat(pVertices[1].X - pVertices[2].X)));
    dVOverZdX = OneOverdX * (((aVOverZ[1] - aVOverZ[2]) *
            Fixed28_4ToFloat(pVertices[0].Y - pVertices[2].Y)) -
            ((aVOverZ[0] - aVOverZ[2]) *
            Fixed28_4ToFloat(pVertices[1].Y - pVertices[2].Y)));
    dVOverZdY = OneOverdY * (((aVOverZ[1] - aVOverZ[2]) *
            Fixed28_4ToFloat(pVertices[0].X - pVertices[2].X)) -
            ((aVOverZ[0] - aVOverZ[2]) *
            Fixed28_4ToFloat(pVertices[1].X - pVertices[2].X)));
}
edge::edge( gradients const &Gradients, POINT3D const *pVertices,
            int Top, int Bottom )
{
    Y = Ceil28_4(pVertices[Top].Y);
    int YEnd = Ceil28_4(pVertices[Bottom].Y);
    Height = YEnd - Y;
    if(Height)
    {
        long dN = pVertices[Bottom].Y - pVertices[Top].Y;
        long dM = pVertices[Bottom].X - pVertices[Top].X;
        long InitialNumerator = dM*16*Y - dM*pVertices[Top].Y +
                    dN*pVertices[Top].X - 1 + dN*16;
        FloorDivMod(InitialNumerator,dN*16,X,ErrorTerm);
        FloorDivMod(dM*16,dN*16,XStep,Numerator);
        Denominator = dN*16;
        float YPrestep = Fixed28_4ToFloat(Y*16 - pVertices[Top].Y);
        float XPrestep = Fixed28_4ToFloat(X*16 - pVertices[Top].X);
        OneOverZ = Gradients.aOneOverZ[Top]
                        + YPrestep * Gradients.dOneOverZdY
                        + XPrestep * Gradients.dOneOverZdX;
        OneOverZStep = XStep * Gradients.dOneOverZdX
                        + Gradients.dOneOverZdY;
        OneOverZStepExtra = Gradients.dOneOverZdX;
        UOverZ = Gradients.aUOverZ[Top]
                        + YPrestep * Gradients.dUOverZdY
                        + XPrestep * Gradients.dUOverZdX;
        UOverZStep = XStep * Gradients.dUOverZdX
                        + Gradients.dUOverZdY;
        UOverZStepExtra = Gradients.dUOverZdX;
        VOverZ = Gradients.aVOverZ[Top]
                        + YPrestep * Gradients.dVOverZdY
                        + XPrestep * Gradients.dVOverZdX;
        VOverZStep = XStep * Gradients.dVOverZdX
                        + Gradients.dVOverZdY;
        VOverZStepExtra = Gradients.dVOverZdX;
    }
}
```

$$R = F\Delta my - \Delta mn_0 + \Delta nm_0 - 1 + F\Delta n$$

$$x_{\text{int}} = \left\lfloor \left\lfloor \frac{R}{F\Delta n} \right\rfloor + \frac{R \bmod F\Delta n}{F\Delta n} \right\rfloor = \left\lfloor \frac{R}{F\Delta n} \right\rfloor + \left\lfloor \frac{R \bmod F\Delta n}{F\Delta n} \right\rfloor \qquad (5)$$

Notice that we moved the floored R/FΔn term outside the main floor; we can do this because a floored term is an integer by the definition of the floor function, and you can always move an integer into and out of a floor.

Equation 5 is our initial DDA condition. If we plug in integer y values and do the math correctly, the floored R/FΔn term will be our initial integer x starting location, and the numerator of the `mod` term will be our initial DDA error term. Before we plug it into this equation, our y should be prestepped to the first scanline according to our fill convention, which defines the starting integer y as the ceiling of the fractional y. Alternatively, if you're two-dimensionally clipping the polygon at rasterization time, you'd make y be the first scanline you want to draw after clipping.

To calculate our DDA step variables for x to step to x', we plug y = y + 1 into Equation 4 and see that our equation changes by FΔm/FΔn. We use Equation 3 to convert this ratio into an integer and a fractional part:

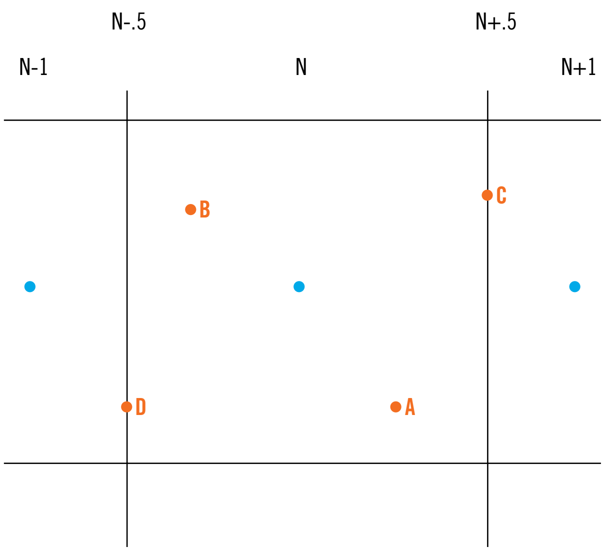
$$x'_{\text{int}} = x_{\text{int}} + \left\lfloor \frac{F\Delta m}{F\Delta n} \right\rfloor + \left\lfloor ErrorTerm + \frac{F\Delta m \bmod F\Delta n}{F\Delta n} \right\rfloor \qquad (6)$$

Equations 5 and 6 will step us on the integer raster grid, but we will step according to the fractional edge, so we'll get the extra precision. As I've mentioned before, it's important to notice that the `mod` terms are always positive, so when our error term rolls over our DDA will always step by 1 (in contrast with some other DDAs you've probably seen where you step by 1 for right-going edges and by -1 for left-going edges). This slightly odd behavior drops out of the math when you do the flooring divide and `mod` correctly, as we discussed previously.

We're still doing the same DDA step as in my last column (the step code is identical), but the various DDA values are determined by the real fractional endpoints, not by the truncated integers. More importantly, the gradients are calculated with the fractional endpoints, which avoids the jitter problems that brought up this fractional mess in the first place.

The results are surprising. Visually, you can't tell the difference between our original floating-point rasterizer and the new fractional endpoint rasterizer—they both are completely solid and jitter-free—

Figure 3. A Pixel



describing the very problem!

The bug is in the only part of the code where I didn't rigidly define the math before I started out: the real-to-integer source-texture coordinate mapping.

As we've seen, we have rock-solid mathematical descriptions of the rasterization, the subpixel stepping, the perspective projection, and the gradient calculations. But when it comes time in the code to take our real texture coordinates for the current pixel center and map them into integer-source-texture coordinates, we simply truncate with no explanation of whether this is the correct thing to do or not. It's not, and we're going to figure out why. Here's the suspect code from our original `DrawScanline` function (the variables on the right are floating point numbers):

```
int U = UOverZ * Z;
int V = VOverZ * Z;
```

To understand why this code is wrong, we need to understand how the mapping from the source to the destination (or vice versa) works, and to understand this, we need to understand the lowliest element in the graphics pipeline, the pixel. As we hinted in previous issues, a pixel isn't a single point as we're used to thinking, it's really a box; a small box, but a box nonetheless. Like every other box (with sides of nonzero length), this one covers an area, and we need to take that area into account when we texture map our polygon.

Figure 3 shows one complete pixel and portions of a couple of its neighbors. We'll call N our integer pixel coordinate, and you can see the edges, or walls, of the middle pixel are each a half-pixel away from the center. This geometry gives our pixel a total area of one, as you'd expect. The other pixel centers are exactly one unit away on either side.

As we rasterize our polygon on the destination grid, we're very careful to only light destination pixels when they're "in," according to our precisely defined fill convention, and we're also very careful to only generate source texture coordinates (using the truncation code for the time being) when we're exactly on a destination pixel center. We're basically projecting the destination pixel center back into the source to figure out the source pixel color with which to light the destination pixel. The `UOverZ * Z` expression generates this real source texture coordinate, and our "mapping rule," such as it is, converts this real number into an integer source coordinate we can use to look up the texture pixel (sometimes called a texel) value.

Figure 4, which shows the source texture and its position on the destination, gives us a way of visualizing the problem. You can also see each of the source texel boundries drawn in the destination for the purposes of this illustration. If we were to rasterize this destination polygon to texture map our source, we'd generate source coordinates for each of the destination pixel centers. As you can see (with the help of the arrow showing one of the destination pixel centers mapping back into the source) those pixel centers rarely, if ever, map to the source pixel centers.

Let's look at how our current truncation mapping rule affects various source coordinates by viewing the pixels in Figure 3 as the source texels. If our perspective projection for a given mapping takes a destination pixel center and maps it to the real point denoted by the A in Figure 3, our truncation rule maps this to pixel N (A's value is greater than N, but less than N + 1, so truncation maps it to N), which looks about right. In general, we'd like our mapping rule to take the real source coordinates and map them to the nearest integer pixel center, which means any points that fall within the box for a given source pixel get mapped to that pixel's integer coordinate.

Next let's say our projection takes our destination pixel center to B in Figure 3.

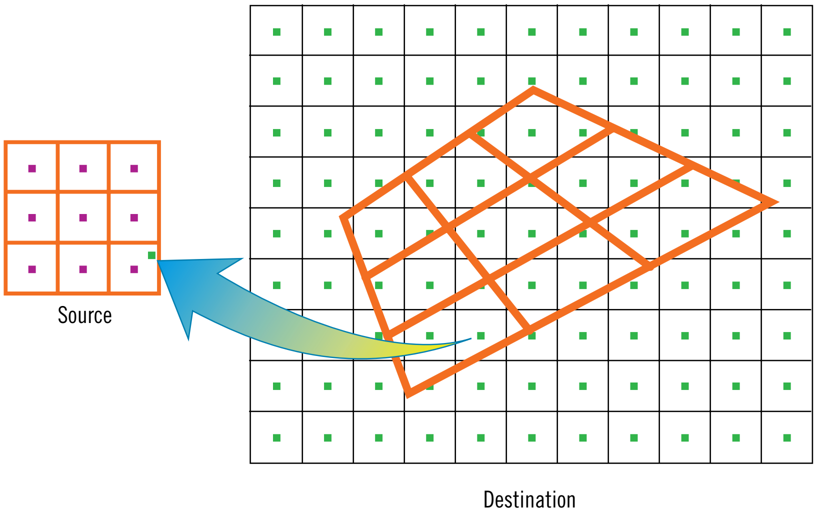
and we get all the benefits of doing error-term integer DDAs.

Listing 1 shows the changes to our texture mapper to use fractional endpoints. One thing to watch for is overflow in these equations, particularly in the numerator of Equation 4. If your polygons get really big, and your scaling factor is large, you can overflow beyond 32 bits. Most architectures make it possible to keep a 64-bit numerator around for the divide, so you can usually handle this if the need arises.

Off The Map

Let me just come out and say it: there's a bug in the code from my first column on perspective texture mapping. No, it's not a bug in any of the rasterization math or implementation we've been poring over for the last two issues, and it's subtle enough that you'd have to know what you were looking for and look pretty hard to find it. In fact, Michael Abrash and I were talking about a related issue when we realized there actually was a bug in the code and math. We even tossed around the idea of having a contest to spot the bug, but I decided against it because I assumed it was so subtle nobody would figure it out. Of course, within the next day or so Walt Donovan (walt@rendition.com) from Rendition Inc. sent me e-mail

Figure 4. Mapping



Our truncation generates N - 1 for our source coordinate, when N is clearly the right answer! Oops.

Now that you see the bug, let me tell you how we figured out it was there in the first place. We were talking about various texture mapping one day when Michael asked how I would implement a `blt` (a block transfer, or pixel copy) with my texture mapper. In other words, how would I allocate the source-texture coordinates and the destination-screen coordinates so that the source-to-destination mapping was one to one? It seems logical to be able to do this, not so much because you'll use the texture code when you simply want to `blt`, but because if the math is completely right you should be able to get an exact 1:1 mapping just like all the other arbitrary mappings you can get with perspective projections.

I thought about this problem for a second, and answered that I'd allocate the corner texture coordinates and the destination coordinates at the exact same coordinates on the screen, (-0.5,-0.5) and (TextureWidth-0.5,TextureHeight-0.5). (Our code only handles triangles so I'd obviously need to call it twice to `blt` the whole source, but the top-left and bottom-right corners are all I needed to describe the destination rectangle.) As soon as I said this I realized I hadn't bothered to define the mapping rule from real source coordinates to integer source coordinates.

Figure 5 helps illustrate why I chose the coordinates I did. I can't stress enough in this discussion that to get the correct mappings we need to view pixels as areas, not just as points. With this in mind, the coordinates I gave Michael are the coordinates of the inifinitely thin edge that completely surrounds our source texture bitmap, as you can see in Figure 5. The point (-0.5,-0.5) is the upper left-hand corner of the texture (the upper left-hand pixel center is at [0,0]), and (9.5,9.5) is the lower right-hand corner—the edges totally enclose the texture pixels. If I had chosen integer source coordinates for the corners we'd be cutting the edge pixel areas in half, which you can see if you take Figure 5 and draw an imaginary edge through the pixel center.

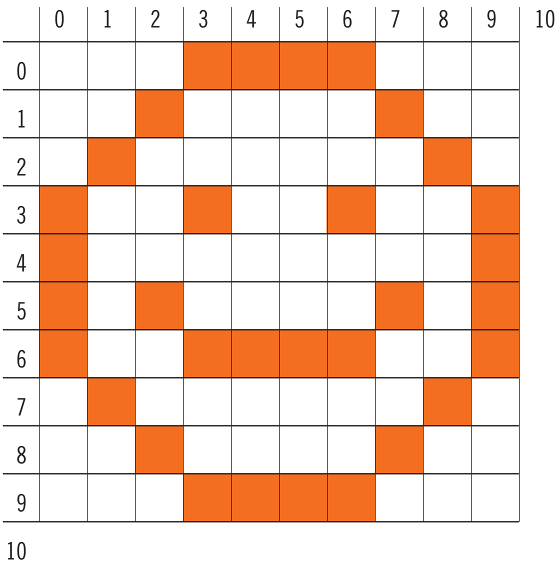
Similarly, I chose the corresponding screen coordinates for the destination. I wanted the destination pixel centers to map exactly to the source pixel centers, so it was necessary to completely enclose the destination pixels in the same manner as the source pixels.

I realized I needed to define a mapping that took a real texture coordinate and mapped it to the closest integer pixel center. This is basically a rounding operation, and the function for rounding is:

$$C_{\text{int}} = \left\lfloor C + \frac{1}{2} \right\rfloor \qquad (7)$$

Equation 7 is the familiar rounding rule where you add a half and floor the result. I looked at what effect this rule would have on the texture mapper and it fixes the problem with B in Figure 3, that's for sure. However, implicit in any rounding rule is a tie-breaking rule that kicks in when the value to be rounded is exactly halfway between two integers, like C and D in Figure 3. Equation 7 will map C to N + 1 and D to N, so it's pretty clear that this is a

Figure 5. Source Texture



top-left rounding rule, meaning if the pixel center falls on the top or left edge of the pixel it is considered "in," and if it falls on the bottom or right edge it is considered part of the neighboring pixel. This is obviously very similar to our fill convention.

At first glance, a top-left rounding rule seems to work well with our top-left fill convention. It looks like the only way for our mapping rule to generate a pixel that's out of bounds is for the right or bottom edge of the texture to fall directly on a pixel center in the destination (think about shifting our `blt` destination coordinates down and to the right by a half pixel so they're on integers in the destination), but if the right or bottom edge of the texture corresponds to a right or bottom edge of the destination polygon we don't draw those pixel centers anyway because of our fill convention. This beautiful harmony is broken when you visualize rotating the destination polygon by 180 degrees so that its edges still correspond with integer destination pixel centers, but the left edge of the polygon corresponds to the right edge of the texture. Now, if we apply our rounding mapping rule we'll start with the right edge of the texture (TextureWidth - 0.5, or 9.5 in Figure 3), add a half, and floor, resulting in a texture coordinate off the edge of our texture! Are we back where we started?

Same Time, Same Channel

The answer to that question is no, but to find out how we're going to solve the problem you'll have to tune in next time because I'm out of space. I will give you a hint, however. Equation 7 is just one rounding rule. Here's another:

$$C_{\text{int}} = \left\lceil C - \frac{1}{2} \right\rceil \qquad (8)$$

Equation 8 works out to be a bottom-right rounding rule, which would have avoided the problem mentioned at the end of the last paragraph, but would result in a similar problem with our original orientation! Think about what criteria we have for choosing between the two rounding rules and join me next issue.

By the way, our texture mapper is doing a form of resampling called *point sampling*. We map the destination pixel centers into the source and just take whatever texel in which we land. There are other forms of resampling where you take the corners of the destination pixel and map them back into the source to form a quadrilateral, and then filter the resulting area into a single pixel color. *Digital Image Warping* (IEEE Computer Society, 1990) by George Wolberg covers a bunch of these resampling techniques.

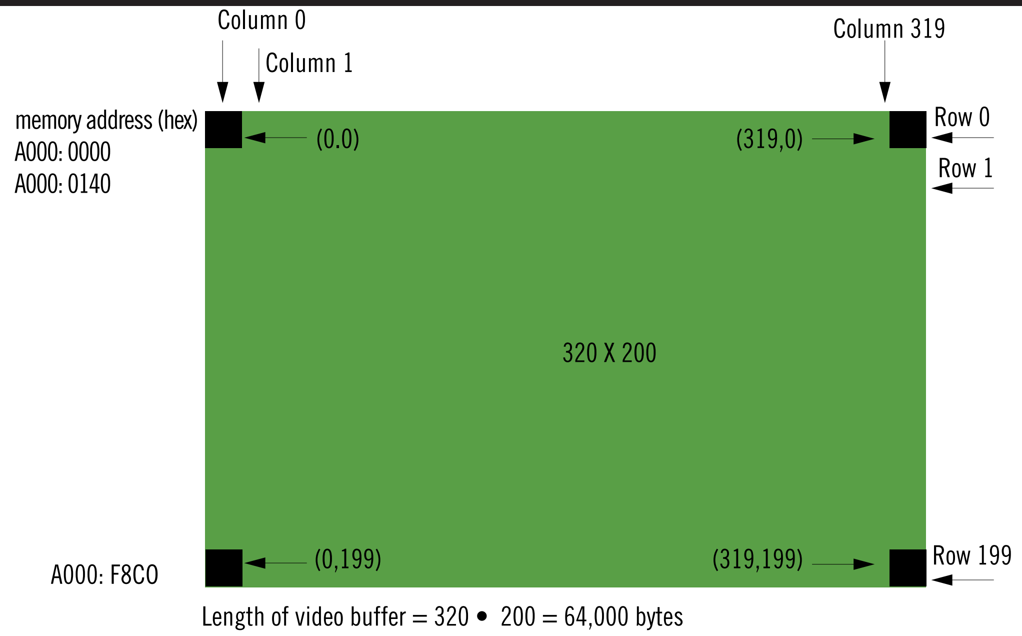
Once again I'd like to thank Kirk Olynyk. He's the reason I know the Windows NT 28.4 rasterizer is correct—he did the original math. ■

Chris Hecker thinks that regardless of the outcome of the video game console wars, it's unlikely anyone will beat Sega's "sphincter" advertisement for pure comedy value. Discussion of various body parts and their relationship to video games is available at checker@bix.com.

Back

Building Brains into Your Games

Figure 1. The Layout of Mode 13h



Game developers have always pushed the limits of the hardware when it comes to graphics and sound, but I think we all agree that when it's time to implement artificial intelligence for a game, AI always gets the short end of the stick! In this article, we are going to study a potpourri of AI topics ranging from the simple to the complex.

Along the way, we are going to try out a few demos that use a very rudimentary graphics interface to illustrate some of the simpler concepts. However, most of our discussion will be quasi-theoretical and abstract. This is because AI is not as simple as an algorithm, a data structure, or similar things. Artificial intelligence is a fluid concept that must be shaped by the game it is to be used on. Granted, you may use the same fundamental techniques on myriad games, but the form and implementation may be radically different.

Let's begin our discussion with some simple statements that define what AI is in the context of games. Artificial intelligence in the arena of computer games implies that the computer-controlled opponents and game objects seem to show some kind of cognitive process when taking actions or reacting to the player's actions. These actions may be implemented in a million different ways, but the bottom line, from an observers point of view, is that they seem to show intelligence.

This brings us to the fundamental definition of intelligence. For our pur-

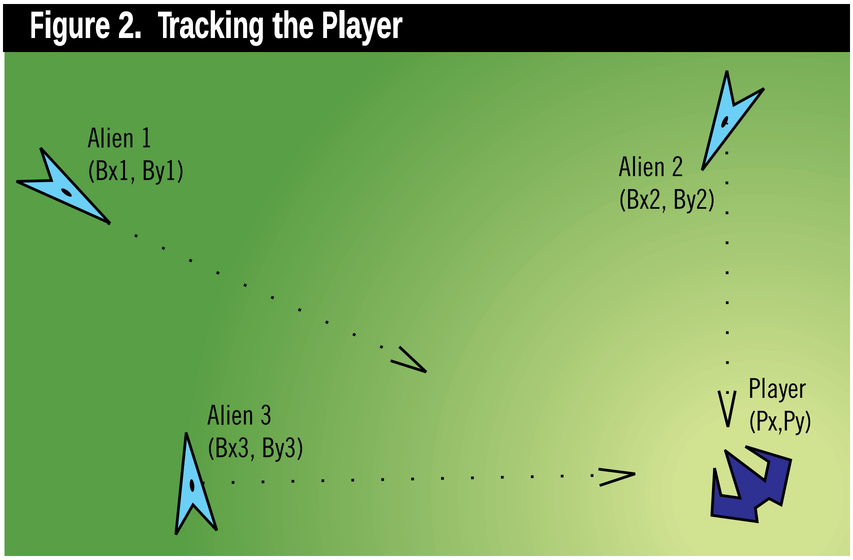
Listing 1. The Graphics Module GMOD.H

```
// GMOD.H graphics module for demos

unsigned char far *video_buffer = (unsigned char far *)0xA0000000L;

void Plot_Pixel(int x,int y,int color)
{
// plots the pixel in the desired color a little quicker using binary shifting
// to accomplish the multiplications

video_buffer[((y<<8) + (y<<6)) + x] = (unsigned char )color;

} // end Plot_Pixel

void Set_Graphics_Mode(int mode)
{
// use the video interrupt 10h and the C interrupt function to set
// the video mode

union REGS inregs,outregs;

inregs.h.ah = 0;                   // set video mode sub-function
inregs.h.al = (unsigned char)mode; // video mode to change to
int86(0x10, &inregs, &outregs);

} // end Set_Graphics_Mode

void Time_Delay(int clicks)
{
// this function uses the internal timer to wait a specified number of "clicks"
// the actual amount of real time is the number of clicks * (time per click)
// usually the time per click is set to 1/18th of a second or 55ms

long far *clock = (long far *)0x0000046CL, // address of timer
    start_time; // starting time

// get current time
start_time = *clock;

// when the current time minus the starting time >= the requested delay then
// the function can exit
while(labs(*clock - start_time) < (long)clicks){}

} // end Time_Delay
```

André LaMothe

Any element you add to a game to make it more realistic can only enhance it. Artificial intelligence is the next wave to make your game think and respond like a living, breathing, (shooting) opponent.

Figure 2. Tracking the Player



poses, intelligence is simply the ability to survive and perform tasks in an environment. The tasks may be to hunt down and destroy the player, find food, navigate an asteroid field, or whatever. Nevertheless, this will be our loose definition of intelligence.

Now that we have an idea of what we are trying to accomplish, where on earth should we begin? We will begin by using humans as our models of intelligence because they seem to be reasonably intelligent for carbon units. If we observe a human in an environment, we can extrapolate a few key behaviors of intelligence that we can model using fairly simple computer algorithms and techniques.

These behaviors are blind reflexes, random selection, use of known patterns, environmental analysis, memory-based selections and sequential behaviors that may encompass some or all of the other behaviors. We'll take a look at all of these behaviors and explore how we might implement them in a computer game, but first let's talk about the graphics module we are going to use for some of the demos.

The Graphics Module

Half the world uses Microsoft C and C++ compilers and the other half uses Borland C and C++ compilers—so it's always a problem publishing demos that depend on the use of either. Hence, we are going to write C code that is totally compiler independent, based on a graphics interface that we are going to write ourselves, and that will work on both compilers. The graphics interface will be based on graphics mode 13h, which is 320 by 200 pixels with 256 colors as shown in Figure 1. For the simple demos we are going to write, all we want to do is place the VGA/SVGA card in mode 13h and plot single pixels on the screen. Thus we need two functions:

```
Set_Video_Mode(int mode);
```

and

```
Plot_Pixel(int x, int y,unsigned
  char color);
```

We will use the video BIOS function 10h to set the video mode, but how can we plot pixels? Plotting pixels in mode 13h is very simple because the graphics are fully memory mapped. Basically, mode 13h is a totally linear array of memory that represents each pixel with a single byte. Further, this video memory starts at location A000:0000. Figure 1 shows that there are 200 rows and 320 columns. Therefore, to compute the address of any pixel at (x,y) we simply multiply the Y component by 320 and add the X. Or in other words:

```
memory offset = y*320+x;
```

Adding this memory offset to A000:0000 gives us the final memory location to access the desired screen pixel. Hence, if we alias a FAR pointer to the video memory like this:

```
unsigned char far* video_buffer =
  (unsigned char far*)A0000000L;
```

Then we can access the video memory using a syntax like:

```
video_buffer[y*320+x] = color;
```

And that's it. So, using that information, we can then write a simple pixel-plotting function and graphics-mode function. These two functions should be added to each demo so that the demos can perform the graphics-related functions without help from the compiler-dependent graphics library. We're also going to add a little time-delay function based on the PC's internal timer. The function is called Time_Delay() and takes a single parameter, which is the number of clicks to wait for. Listing 1 shows the complete graphics interface named GMOD.H for the demos contained within this article. Simply include the code of the graphics module with each demo and everything should work fine. Now that we have the software we need to do graphics, let's begin our discussion of AI.

Deterministic Algorithms

Deterministic algorithms are the simplest of the AI techniques used in games. These algorithms use a set of variables as the input and then use some simple rules to drive the computer-controlled enemies or game objects based on these inputs. We can think of deterministic algorithms as reflexes or very low-level instincts. Activated by some set of conditions in the environment, the algorithms then perform the desired behavior relentlessly without concern for the outcome, the past, or future events.

The chase algorithm is a classic example of a deterministic algorithm. The chase algorithm is basically a method of intelligence used to hunt down the player or some other object of interest in a game by applying the spatial coordinates of the computer-controlled object and the object to be tracked. Figure 2 illustrates a good example of this. It depicts a top view of a typical battleground, on which three computer-controlled bad guys and one player are fighting. The question is, how can we make the computer-controlled bad guys track and move toward the player? One way is to use the coordinates of the bad guys and the coordinates of the player as inputs into a deterministic algorithm that outputs direction changes or direction vectors for the bad guys in real time.

Let's use bad guy one as the example. We see that he is located at coordinates (bx1,by1) and the player is located at coordinates (px,py). Therefore, a simple algorithm to make the bad guy move toward the player would be:

```
// process x-coords
if (px>bx1)
   bx1++;
else
if (px<bx1)
   bx1--;
// process y-coords
if (py>by1)
   by1++;
else
if (py<by1)
   by1--;
```

That's all there is to it. If we wanted to reverse the logic and make the bad guy run then the conditional logic could be inverted or the outcome increment operators could be inverted. As an example of deterministic logic, Listing 2 is a complete program that will make a little computer-controlled dot chase a player-controlled dot. Use the numeric keypad to control your player and press ESC to exit the program.

Now let's move on to another typical behavior, which we can categorize as random logic.

Listing 2. A Demo of Deterministic Logic

```
// Deterministic chasing algorithm demo
// use numeric keypad to move player

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <math.h>
#include <string.h>

#include "gmod.h" // include our graphics
module

int main(void)
{

int px=160,    // starting position of
player
    py=100,
    bx=0,      // starting position of
bad guy
    by=0,
    done=0;    // exit flag

// set the video mode to 13h
Set_Graphics_Mode(0x13);

// main event loop
while(!done)
        {
            // perform player logic

            // get input from keyboard
    if (kbhit())

            // which way is player moving?
        switch(getch())
                {
                case ´8´: // up
                    {

                    if ((py-=2)<0)
                    py+=200;

                    } break;

                case ´2´: // down
                    {

                    if ((py+=2)>=200)
                    py-=200;

                    } break;

            case ´6´: // right
                {

                if ((px+=2)>=320)
                    px-=320;

                } break;

            case ´4´: // left
                {

                if ((px-=2)<0)
                    px+=320;

                    } break;

                case 27: // exit
                        {
                        done=1;
                        } break;

                } // end switch

            } // end if

        // perform bad guy logic
        if (px>bx)
            bx++;
        else
        if (px<bx)
            bx--;

        if (py>by)
            by++;
        else
        if (py<by)
            by--;

        // draw player and bad guy
        Plot_Pixel(bx,by,12);
        Plot_Pixel(px,py,9);

        // wait a bit
        Time_Delay(1);

        } // end main while

    // reset graphics back to text
    Set_Graphics_Mode(0x03);

    // return success to DOS
    return(0);

    } // end main
```

Random Logic

Sometimes an intelligent creature exhibits almost random behaviors. These random behaviors may be the result of any one of a number of internal processes, but there are two main ones that we should touch upon—lack of information and desired randomness.

The first premise is an obvious one. Many times an intelligent creature does not have enough information to make a decision or may not have any information at all. The creature then simply does the best it can, which is to select a random behavior in hopes that it might be the correct one for the situation. For example, let's say you were dropped into a dungeon and presented with four identical doors. Knowing that all but one meant certain death, you would simply have to randomly select one!

The second premise that brings on a random selection is intentional. For example, say you are a spy trying to make a getaway after acquiring some secret documents (this happens to me all the time). Now, imagine you have been seen, and

Listing 3. A Bunch of Dumb Flies

```
// Random logic demo
// moves a flock of flies around
// hit any key to exit


#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <math.h>
#include <string.h>

#include "gmod.h" // include our graphics
module

#define NUM_FLIES 64 // start off with 64
flies

typedef struct fly_typ
        {
        int x,y; // position of fly
        } fly;

int main(void)
{

fly flys[NUM_FLIES]; // the array of flies

int index; // looping variable

// set the video mode to 13h
Set_Graphics_Mode(0x13);

// initialize all flies to random position
for (index=0; index<NUM_FLIES; index++)
    {
    flys[index].x = rand()%320;
    flys[index].y = rand()%200;
    } // end for index

// main event loop
while(!kbhit())
    {
        // erase flies

        for (index=0; index<NUM_FLIES;
  index++)

Plot_Pixel(flys[index].x,flys[index].y,0);

        // perform fly logic, translate
  each fly +-2 pixels
        for (index=0; index<NUM_FLIES;
  index++)
            {
            flys[index].x+=(-2+rand()%5);
            flys[index].y+=(-2+rand()%5);
        } // end for index

        // draw flies
        for (index=0; index<NUM_FLIES;
  index++)

Plot_Pixel(flys[index].x,flys[index].y,10)
;

        // wait a bit

            Time_Delay(2);

            } // end main while

    // reset graphics back to text
    Set_Graphics_Mode(0x03);

    // return success to DOS
    return(0);

    } // end main
```

the bad guys start shooting at you! If you run in a straight line, chances are you are going to get shot. However, if during your escape you make many random direction changes and zigzag a bit, you will get away every time!

What we learn from that example is that many times random logic and selections are good because it makes it harder for the player to determine what the bad guys are going to do next, and it's a good way to help the bad guys make a selection when there isn't enough information to use a deterministic algorithm. Motion control is a typical place to apply random logic in bad-guy AI. You can use a random number or probability to select a new direction for the bad guy as a function of time. Let's envision a multiplayer game with a single, computer-controlled bad guy surrounded by four human players. This is a great place to apply random motion, using the following logic:

```
// select a random translation
   for X axis
bx1 = bx1 + rand()%11 - 5;
// select a random translation
   for Y axis
by1 = by1 + rand()%11 - 5;
```

The position of the bad guy is translated by a random amount in both X and Y, which in this case is +-5 pixels or units.

Of course, we can use random logic for a lot of other things besides direction changes. Starting positions, power levels, and probability of firing weapons are all good places to apply random logic. It's definitely a good technique that adds a bit of unpredictability to game AI. Listing 3 is a demo of random logic used to control motion. The demo creates an array of flies and uses random logic to move them around. Press ESC to exit the demo.

Now let's talk about patterns.

Encoded List Processing

Many intelligent creatures have prerecorded patterns or lists of behaviors that they have either learned from experience or are instinctive. We can think of a pattern as a sequence of steps we perform to accomplish a task. Granted, this sequence may be interrupted if something happens during the sequence that needs attention. But in general, if we forget about interruptions then we can think of patterns as a list of encoded instructions that an intelligent creature consumes to accomplish some task.

For example, when you drive to work, school, or your girlfriend's or boyfriend's house, you are following a pattern. You get into your car, start it, drive to the destination, stop the car, turn it off, get out, and finally do whatever it is you're going to do. This is a pattern of behavior. Although during the entire experience a billion things may have gone through your head, the observed behavior was actually very simple. Hence, patterns are a good way to implement seemingly complex thought processes in game AI. In fact, many games today still use patterns for much of the game logic.

So how can we implement patterns for game AI? Simply by using an input array to a list processor. The output of the processor is the control of a game object or bad guy. In this case, the encoded list has the following set of valid instructions:

- Turn right
- Turn left
- Move forward
- Move backward
- Sit still
- Fire weapon.

Even though we only have six selections, we can construct quite a few patterns with a short input list of 16 elements as in the example. In fact there are 6^{16} different possible patterns or roughly 2.8 trillion different behaviors. I think that's enough to make something look intelligent! So how can we use encoded lists and patterns in a game for the AI? One solid way is to use them to control the motion of a bad guy or game object. For example, a deterministic algorithm might decide it's time to make a bad guy perform some complex motion that would be difficult if we used standard conditional logic. Thus, we could use that pattern, which simply reads an encoded list directing the bad guy to make some tricky moves. For example, we might have a simple algorithm like this:

```
int move_x[16] = {-2,0,0,0,3,3,2,1,0,
  -2,-2,-,0,1,2,3,4};
int move_y[16] = {0,0,0,1,1,1,0,0,-1,-1,
   2,3,4,0,0.-1};
// encoded pattern logic for a
   16 element list
for (index=0; index<16; index++)
     {
     bx1+=move_x[index];
     by1+=move_y[index];
     } // end for index
```

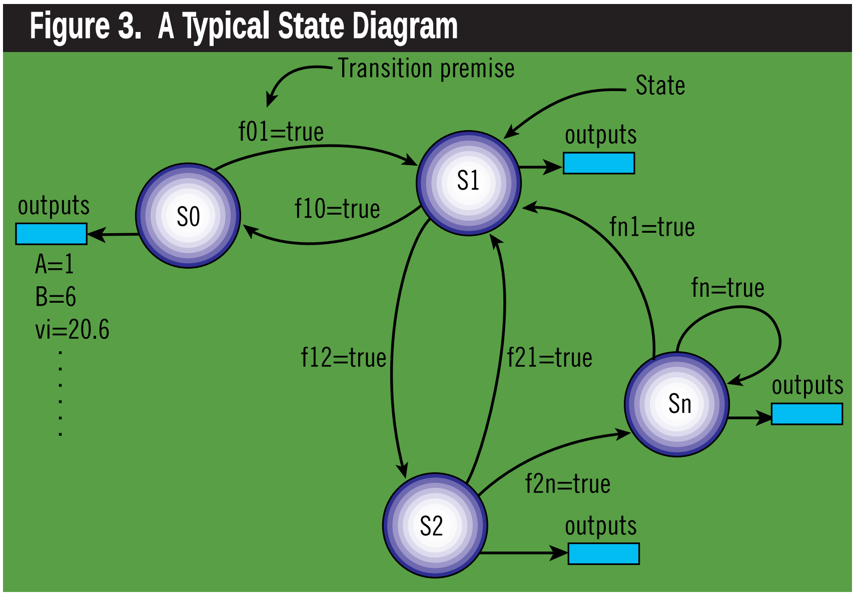
You'll notice that the encoded pattern is made up simply of X and Y translations. The pattern could just as well have contained complex records with a multitude of data fields. I've written detailed code that will create an example of patterns and list processing, a demo of an ant that can process one of four patterns selected by the keys 1-4. Unfortunately, it's too long to print here. Go to the *Game Developer* ftp site, though (ftp://ftp.mfi.com/gdmag/src), and you can download it there.

Now we're starting to get somewhere, but we need an overall control unit with some form of memory, and we must select the appropriate types of behaviors.

Finite State Machines

Finite state machines, or FSMs, are abstract models that can be implemented either in hardware or software. FSMs are not really machines in the mechanical sense of the word, but rather abstract models with a set of "states" that can be traversed. Within these states, the FSM not only has a special set of outputs but remembers the state and can transition to another state, if and only if a set of inputs or premises are met. Figure 3 is a typical

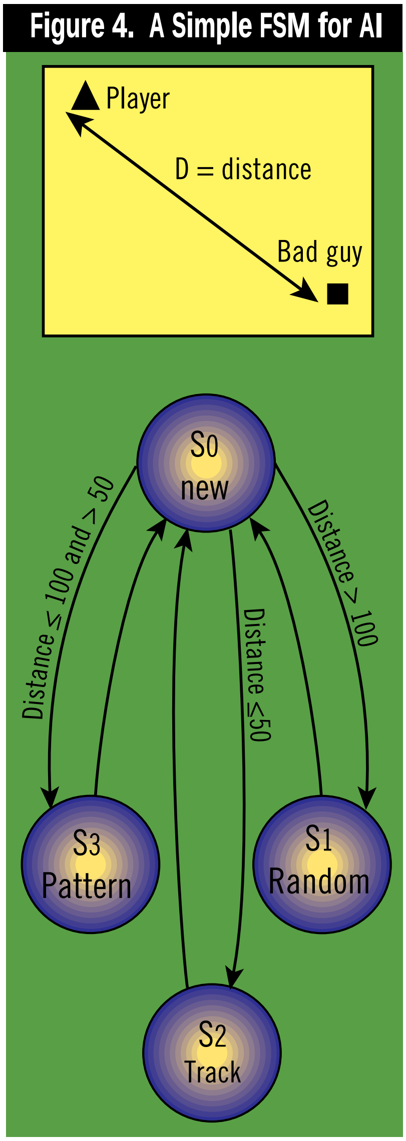
Figure 3. A Typical State Diagram



Listing 4. The Core FSM Logic for Figure 7

```
typedef unsigned short DISTANCE;
const DISTANCE Tracking_Threshold = 50;
const DISTANCE Random_Threshold = 100;
DISTANCE theDistance;
//Define states and initialize
enum states{new, random, track, pattern};
states currentState = new;
//FSM loop
for(;;){
    switch (currentState){
                case new:
                    //Note: Switchbox only, causes no behavior
                    theDistance = CalcDistanceToPlayer();
                    if (theDistance > Random_Threshold){
                            currentState = random;
                    }else{
                            if (theDistance > Tracking_Threshold){
                                    currentState = pattern;
                            }else{
                                    currentState = track;
                            }
                    }
                    break;
                case track:
                    DoTrackBehavior();
                    currentState = new;
                    break;
                case pattern:
                    DoPatternBehavior();
                    currentState = new;
                    break;
                case random:
                    DoRandomBehavior();
                    currentState = new;
                    break;
                case default:
                    cerr<<"state machine has entered an unknown
state\n";
                    assert(FAIL);
    }
}
```

Figure 4. A Simple FSM for AI



depiction of a finite state machine. We see that there is a set of states labeled, S_0, S_1, S_2, and S_n. We also see that there is a set of connecting edges or arcs. These are called transition arcs and are the premises that must be met for the FSM to move from state to state. Finally, within each state is a set of outputs. These outputs can be anything we wish—from motion controls for a game's bad guys to hard disk commands.

So how do we model an FSM in software and use it to control the game AI? Let's begin with the first question.

We can model an FSM with a single variable and a set of logical conditions used to make state transitions along with the output for each state. For example, let's actually build a simple software state machine that controls a computer bad guy differently based on the bad guy's distance to the player. The state machine will have the following four states:

- *State 0*: Select new state = STATE_NEW
- *State 1*: Move randomly = STATE_RANDOM
- *State 2*: Track player = STATE_TRACK
- *State 3*: Use a pattern = STATE_PATTERN

The FSM's transition diagram is shown in Figure 4. We can see that if the bad guy is within 50 units of the player, then the bad guy moves into State 2 and simply attacks. If the bad guy is in the range of 51 to 100 units from the player, then the bad guy goes into State 3 and moves in a pattern. Finally, if the bad guy is farther than 100 units from the player then chances are the bad guy can't even see the player (in the imaginary computer universe). In that case, the bad guy moves into State 1, which is random motion.

So how can we implement this simple FSM machine? All we need is a variable to record the current state and some conditional logic to perform the state transitions and outputs. Listing 4 shows a rough algorithm that will do all this.

Note that S_0 (the new state) does not trigger any behavior on the part of the opponent. Rather, it acts as a state "switchbox," to which all states (except itself) transition. This allows you to localize in a single control block all the decision making about transitions

Although this requires two cycles through the FSM loop to create one behavior, it's well worth it. In the case of a small FSM, the entire loop can stay in the cache, and in the case of a large FSM loop, the localization of the transition logic will more than pay for the performance penalty. If you absolutely refuse to double-loop, you can handcraft the transitions between states. A finite-state machine diagram will vividly illustrate, in the form of spaghetti transitions, when your transition logic is out of control.

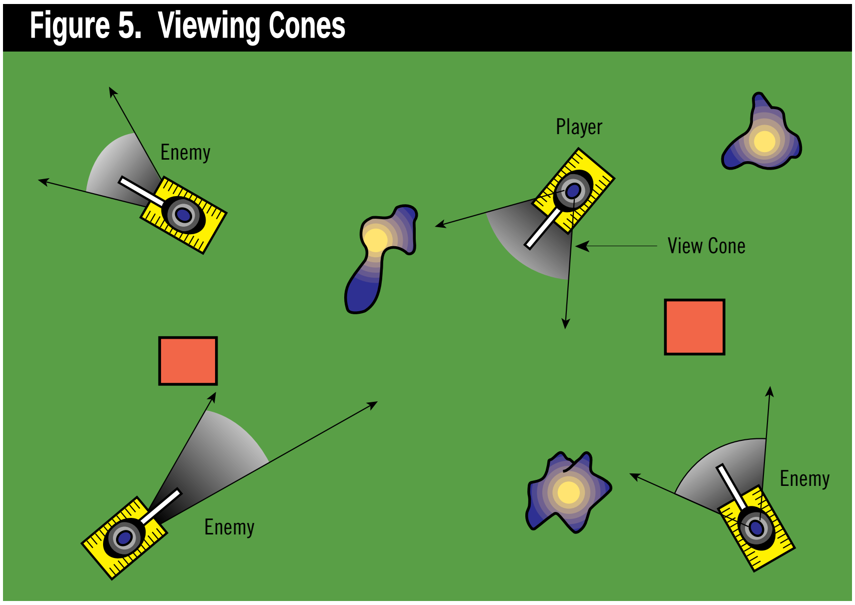
Now that we have an overall thought controller, that is, an FSM, we should discuss simulating sensory excitation in a virtual world.

Environmental Sensing

One problem that plagues AI game programming is that it can be very unfair—at least to the player. The reason for this is that the player can only "see" what's on the computer screen, whereas the computer AI system has access to all variables and data that the player can't access.

This brings us to the concept of simulated sensory organs for the bad guys and game objects. For example, in a three-dimensional tank game that takes place on a flat plain, the player can only see so far based on his or her field of view. Further, the player can't see through rocks, buildings, and obstacles. However, because the game logic has

Figure 5. Viewing Cones



access to all the system variables and data structures, it is tempting for it to use this extra data to help with the AI for the bad guys.

The question is, is this fair to the player? Well, of course not. So how can we make sure we supply the AI engine of the bad guys and game objects with the same information the player has? We must use simulated sensory inputs such as vision, hearing, vibration, and the like. Figure 5 is an example of one such imaginary tank game. Notice that each opponent and the player has a cone of vision associated with it. Both the bad guys and the player can only see objects within this cone. The player can only see within this cone as a function of the 3D graphics engine, but the bad guys can only see within this cone as a function of their AI program. Let's be a little more specific about this.

Since we know that we must be fair to the player, what we can do is write a simple algorithm that scans the area in front of each bad guy and determines if the player is within view. This scanning is similar to the player viewing the viewport or looking out the virtual window. Of course, we don't need to perform a full three-dimensional scan with ray tracing or the like—we can simply make sure the player is within the view angle of the bad guy in question by using trigonometry of any technique we wish.

Based on the information obtained from each bad guy scan, the proper AI decision can be made in a more uniform and fair manner. Of course, we may want to give the computer-controlled AI system more advantage than the human player to make up for the AI system itself being rather primitive when compared to the 100 billion-cell neural network it is competing against, but you get the idea.

Finally, we might ask, "Can we perform other kinds of sensing?" Yes. We can create simulated light detectors, sound detectors, and so forth. I have been experimenting with an underwater game engine, and in total darkness the only way the enemy creatures can "see" you is to listen to your propulsion units. Based on the power level of the player's engines the game AI determines the sound level that the bad guys hear and moves them toward the sound source or sources.

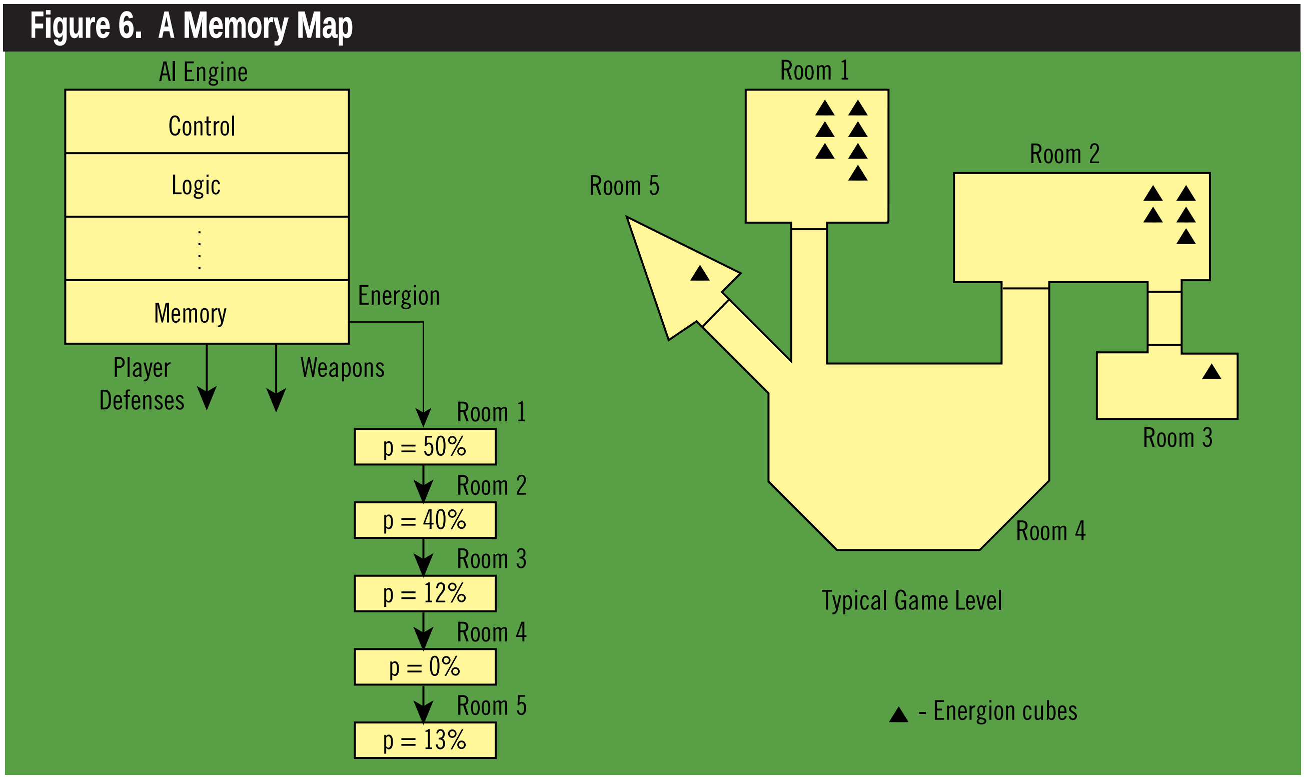
Memory and Learning

The final topic we're going to touch upon is memory and learning. Memory is easy enough to understand, but learning is a bit more nebulous. Learning as far as we are concerned is the ability to interact in an environment in such a way that behaviors that seem to work better than others under certain conditions are "memorized" and used more often. In essence, learning is based on memory of past actions being good or bad or whatever. Imagine that we have written a fairly complex game composed of computer-controlled aliens. These aliens use an FSM-based AI engine and environmental sensing. The problem is that one of the resources in the game is energion cubes and the player and aliens must compete for these cubes.

As the player is moving around in the environment, he or she can create a mental map of where energion cubes seem to be plentiful, but the alien creatures have no such ability; they can only stand and are at a disadvantage. Can we give them a memory and teach them where these energion cubes are? Of course we can, we are cybergods!

One such implementation would

Figure 6. A Memory Map



work as follows: We could use a simple data structure that would track the number of times an alien found energion in each geographical region of the game.(Figure 6 illustrates one such memory map.) Then, when an alien was power hungry, instead of randomly bouncing around, the alien would refer to this memory data structure and select the geographical region with the highest probability of finding energion and set its trajectory for this region.

The previous example is a simple one, but as we can see, memory and learning are actually very easy to implement. Moreover, we can make the computer AI learn much more than where energion is. It could learn the most common defensive moves of the player and use this information against the player.

Well that's enough for basic AI techniques. Let's take a quick look at how we can put it all together.

Building Monsters from the Id

We have quite a repertoire of computer AI tricks at our fingertips, so how should we use it all? Basically, when you write a game and are implementing the AI, you should list the types of behaviors that each game object or bad guy needs to exhibit. Simple creatures should use deterministic logic, randomness, and patterns. Complex creatures that will interact with the player should use an FSM-based AI engine. And the main game objects that harass and test the player should use an FSM and sensory inputs and memory. Figure 7 illustrates a final model of the most advanced AI engine we can construct with what we have to work with.

The Future

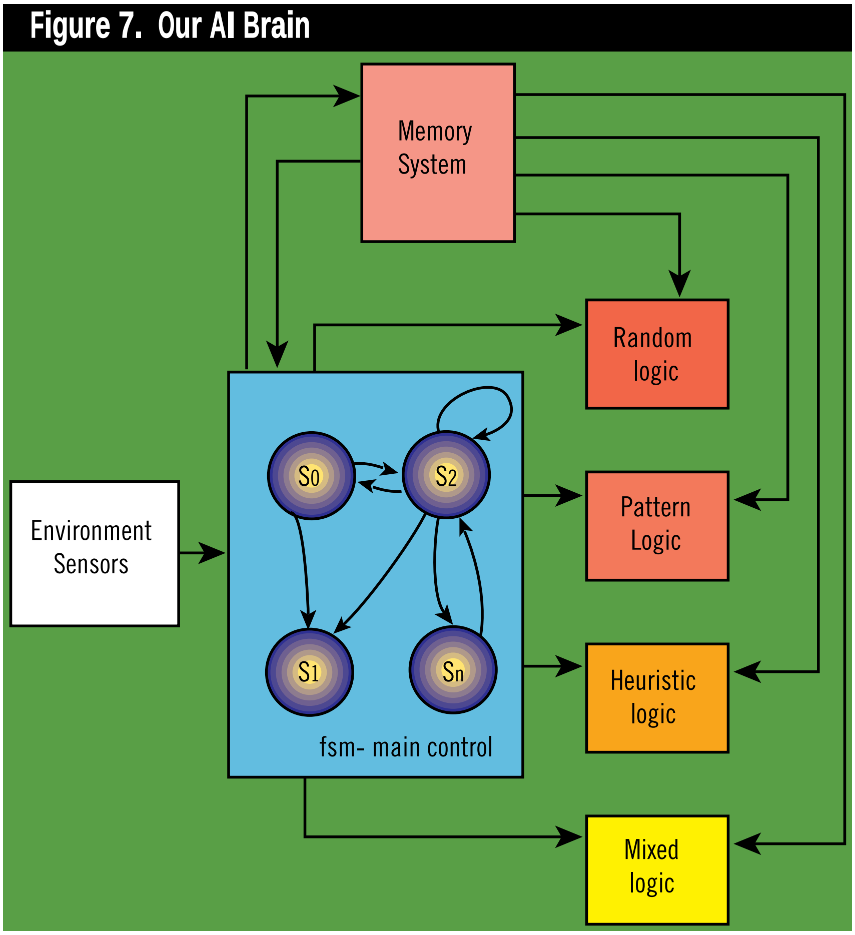
I see AI as the next frontier to explore. Without a doubt, most game programmers have focused so much on graphics that AI hasn't been researched much. The irony is that researchers have been making leaps and bounds in AI research and Artificial Life or A-Life.

I'm sure you've heard the common terms "genetic algorithms" and "neural networks." Genetic algorithms are simply a method of representing some aspect of a computer-based AI model with a set of "genes," which can represent whatever we wish—aggressiveness, maximum speed, maximum vision distance, and so on. Then, a population of creatures is generated using an algorithm that adds a little randomness in each of the output creatures' genes.

Our game world is then populated with these gene-based creatures. As the creatures interact in the environment, they are killed, survive, and are reborn. The biological analog comes into play during the rebirthing phase. Either manually or by some other means, the computer AI engine "mates" various pairs of creatures and mixes their genes. The resulting offspring then survive another generation and the process continues. This causes the creatures to evolve so that they are most adapted for the given environment.

Neural networks, on the other hand, are computer abstractions of a collection of brain cells that have firing thresholds. You can enhance or diminish these thresholds and the connections between cells. By "teaching" a neural network or strengthening and weakening these connections, the neural net can learn something. So we can use these nets to help make decisions and even come up with new methods.

Figure 7. Our AI Brain



André LaMothe is the author of the best-selling Tricks of the Game Programming Gurus *(SAMS Publishing, 1994) and* Teach Yourself Game Programming in 21 Days *(SAMS Publishing, 1994). His latest creation is the* Black Art of 3D Game Programming *(Waite Group Press, 1995).*

Back

TinyMUSH: Implementing A MUD

Though the roots of multiuser dungeons (MUDs) go back to games like Zork and Adventure, Roy Trubshaw and Richard Bartle created the first true MUD in 1979. As the Internet grew, becoming more accessible, MUDs made themselves at home on the Internet, evolving into a diverse family of multiuser games.

The most common type of MUD is the combat MUD, where players visit unusual and mythic lands alone or in groups, searching for creatures to slay and quests to solve. In social MUDs, people hang out, chatting in the virtual pub of their choice.

Some MUD users, called mudders, enjoy recreating their favorite fantasy settings, using such authors as J.R.R. Tolkein, Anne McCaffery, and C.S. Lewis as inspiration. For the mudders who crave storytelling in their favorite fictional setting, there is the dramatic roleplay MUD, in which people create compelling drama in a kind of interactive theater where stories can take months or years to tell.

MUDs and Multimedia

With CD ROM-induced multimedia, today's games typically emphasize slick, three-dimensional, ray-traced graphics; high-quality voice recording; and catchy MIDI scores. This media dependence is so common, we often forget that the most enduring games had little or no wow potential—it was gameplay that made the classics great.

Some people may think the popularity of MUDs is simply due to their being multiuser networked applications, but that is only part of it—albeit, an important part. For students who have Internet access and are perpetually short on cash, MUDs are dirt cheap entertainment. But fiscal hardship alone doesn't explain why so many people pass up conventional games to romp about some virtual world.

This article covers the implementation of TinyMUSH, a popular type of MUD. Due to time and space constraints, we will cover functionality issues such as the network interface, database, and in-game programming.

The MUD Basics

Regardless of type, all MUDs have a common concept of location (virtual, of course) that affects how players communicate. This strong concept of location affects communications on a MUD. For example, to speak aloud to a group using the `say` command, players must be in the same room as the group.

TinyMUSH is representative of MUD servers leaning toward social rather than combat interactions. TinyMUSH is an open-ended application that has no resolution, unlike a combat MUD. Though TinyMUSH is typically used for social hangouts and interactive theaters, MUSHs provide a suitable communications solution for small groups of people who wish to communicate and collaborate online.

Usually UNIX

MUDs do not use fancy graphics or audio scores, but they do allow multiple players to interact. MUDs need some form of interprocess communication so players on different machines at separate physical locations are able to play together. This requirement has made networking a core technology in MUD development.

The need for reliable networking is one reason MUDs have gravitated toward UNIX—TCP/IP, a popular networking protocol, came bundled with UNIX. Another reason is many of the TinyMUSH developers are, or were, undergraduates in universities where UNIX is widely available. TCP is not the only networking protocol, but TinyMUSH's heritage links it to UNIX. On UNIX, the reliable, connection-oriented communication protocol of choice is TCP.

The developers of TinyMUSH wrote the application in their spare time as a form of recreation and did not design it for portability to every computer architecture and operating system. Making MUSH run on several flavors of UNIX and VMS was more than enough to occupy the developers.

The Network Interface

TCP/IP is really a group of protocols, spanning the network and transport layers of the Open Systems Interconnection model (OSI). Two TCP/IP application program interfaces (APIs) exist on UNIX systems—the Berkeley Socket interface, the choice for most programmers, and the less popular Transport Layer Interface (TLI) developed at Bell Labs. Though TinyMUSH supports both APIs, due to space requirements, this article covers only the Berkeley Socket interface.

From a player's perspective, a TinyMUSH server resides on a specific machine at a specific port. This port address is a 16-bit integer destination port number, which is part of the TCP/IP header packet, along with a source port number and a source, and a 32-bit destination address. These addresses provide communication endpoints for TCP.

For historical reasons, TinyMUSH comes with a default port setting of 4,201 but uses a value greater than 5,000. This minimizes the chance that a conflict can occur between an operating system port assigned to a miscellaneous service and the TinyMUSH port number. UNIX reserves ports 0 to 1023 for privileged operating system processes, so selection of a port greater than 1023 is mandatory.

The function `make_socket()` creates a TCP/IP socket on the port for external processes to connect to the server. If this call fails, the routine exits. This action is critical; MUSH exits on failure of `bind()`. A server that cannot bind a socket cannot allow incoming connections.

TinyMUSH uses a simple method to handle communications between the TinyMUSH server and the players. A user connects to a MUD using a client like Telnet. Using Telnet, players send commands consisting of character strings to TinyMUSH. The MUD, often called the server, receives and acts upon the text. The MUD processes and parses the player's actions, sending the results back to the player.

When a player's client program connects to a TinyMUSH server, the

Ed Meinfelder and
Jeffrey Vance

Multiuser gamers: Don't flounder in the MUD. This article explains the workings of TinyMush, a complete system you can download for free!

operating system marks the file descriptor corresponding to the socket as having input pending. During the main loop of the server MUSH calls `select()` to see if any file descriptors are ready for input or output. After `select()` returns, the MUD services player connections with pending I/O and handles new connections.

TinyMUSH may disallow or restrict connections from specific computers or entire networks. After the `accept()` call returns, the server makes the connection to the player's client. Once connected, the server knows the source address of the client. TinyMUSH checks the source address against a list of disallowed addresses and mask pairs. For example, TinyMUSH rejects the connection from all clients at the host `128.92.15.15` if the address pair `128.92.0.0 255.255.0.0` is in the reject list.

After restriction checks for the connecting player are made, a `DESC` structure for the player is filled in. The `DESC` includes the file descriptor, host information, a player `dbref`, and `doing` message, among other information.

The server sets the file descriptor to nonblocking, representing the player's socket. A `write()` to a nonblocking file descriptor won't result in the server waiting, pending completion of the input or output operation. TinyMUSH does not know the state of the player's client, so it is possible that the operating system's output buffer could otherwise fill, blocking the server and hanging the game.

Each `DESC` has input and output queue pointers, allowing fast processing of the user's data. The `DESC` stores the size of the input and output queues and the amount of input and output. Because sockets to player's clients may block at any time for arbitrary lengths of time, the output queues are finite. After the player's output queues exceed a limit, the server discards further player output.

After the player successfully connects, TinyMUSH completes and returns the `DESC`. Next, the server displays a standard welcome message to the user with a brief description of the commands available. The commands available at this point are only those provided by the networking code.

TinyMUSH has a programmable interface and it allows objects to run many of the same commands as the players. But some commands, like `LOGOUT` and `QUIT`, only make sense for connected players. Thus, the networking code implements and handles these commands, bypassing the rest of the game. When the player connects to the game, using either the command `connect` or `create`, all the commands are available.

The complete list of network-level commands are:

- create
- connect
- QUIT
- LOGOUT
- WHO
- DOING
- SESSION
- OUTPUTPREFIX
- OUTPUTSUFFIX.

The commands listed in uppercase letters are case sensitive; `create` and `connect` are not. The capitalized commands indicate that commands are executable by players over a network connection only.

The commands `create` and `connect` are only available to players not yet associated with a player object. Once the connection is associated with a player object, a flag in the `DESC` structure is set, indicating that socket is connected. Once the player connects, the `create` and `connect` commands are no longer available. The uppercase network commands, available through the game, exist in a small hash table.

The `QUIT` command disconnects the user from the server. `LOGOUT` is more complex; it disassociates a `DESC` structure from a player object and places the user back into a logged-out state, as if he or she had just connected to the server. `WHO`, `DOING`, and `SESSION` are all variants of one command, showing information on connected players.

`OUTPUTPREFIX` and `OUTPUTSUFFIX` are among the less-used commands in TinyMUSH. As the output from commands such as `WHO` spans many lines, it is difficult to determine where the output of a single command begins and ends. The `OUTPUTPREFIX` command takes a string as an argument, like:

```
OUTPUTPREFIX ThisIsTheStart
```

and prints `ThisIsTheStart` on a separate line before the output of every command. Programs, called robots, connect to MUDs and require an easy way to parse the results of their actions. `OUTPUTPREFIX` and `OUTPUTSUFFIX` are simple ways for robots to find the start and end of their command output. Robots function as mailers, mappers, off-MUD database interfaces, and amusing AI simulacrums.

After the networking code parses the user text into a command buffer, the server searches for the text in a hash table with the networking commands. Should the hash routines not find the command, TinyMUSH checks the `DESC` structure to see if it is associated with a player object. If the `DESC` indicates a connected player, the function `do_command` passes the command along to the function `process_command`, as it may be a game-related command, like `Say Hello`.

If the `DESC` indicates no commands associated with a player object, the server assumes it must use either the `create` or `connect` command. The function `check_connect()` parses the input into three parameters. `check_connect()` then checks the first two characters of the first parameter of the command for the strings `co` and `cr` to see if the player entered either a `connect` or `create` command. If not, the server flags the input as invalid and shows the user the standard welcome message again.

When the server recognizes a `connect` command, MUSH searches for and authenticates the player name password. If the player name or password is invalid, the server responds with a fail message to the user. After three failed attempts, the server closes and shuts down the user's connection. If the player's name and password are correct, the server checks to see if players are allowed to log in and that the number of players connected is less than the maximum. If both conditions are true or the player is a wizard (a term for a TinyMUSH administrator), then the MUSH allows the user to log in.

The user is associated with a player object by storing the player object's dbref in the DESC structure along with the connect time. The player is shown the welcome message for new players or the message of the day for existing players. The game administrator receives an additional message often set by another administrator as a note. Should the server be refusing logins or have more than the maximum number of players connected, the player receives a reject message.

Once the player has been connected, the server initializes various attributes on the player, such as the player's last site and last connect time. When the player connects, all objects at the player's location receive the message:

```
<Player> has connected.
```

where <Player> is the player's name.

The final act in connecting a player is the queued execution of a set of commands when the player logs in to the game. These commands are stored in an attribute called aconnect on the player, objects in the master room, and the master room itself.

The Command Interface

Learning TinyMUSH is not easy—there are more than 200 commands and 150 functions—but the basics of MUSH are not hard. TinyMUSH provides users with an interpreted, shell-like language. The internal language to TinyMUSH, often called MUSH-code, is both a bane and a boon—programming simple things is easy, enabling new users to start right away; but for complex tasks, MUSH-code is tedious. Most users never really learn to program MUSH—instead, these players, like unsophisticated DOS users, stick to a minimal set of commands they use often.

The TinyMUSH command interface is user extensible. Users call player-created commands "$ commands" because the definition is preceded by a $. $ commands appear no different than the built-in commands to the user. Attributes store the $ commands on objects.

TinyMUSHes have the capability to provide global $ commands in a master room. When a player-typed command matches nothing, the server checks the master room and its contents for $ commands. The Tiny MUSH master room enables user-defined global commands in the master room to be available everywhere on the MUD.

TinyMUSH has exits providing a path from one object to another. Players use exits like commands, traversing exits by typing the name of the exit only. Users can lock exits against some or all players. When anyone tries to traverse a locked exit, the server triggers the fail action attributes for the exit. An action is a set of three attributes: <name>, o<name>, and a<name>. For the failure to traverse an exit, the action attributes are: fail, ofail, and afail. And, in the case of players and things, objects trigger the fail action when failing to pick up an object that is locked.

A player opens an exit, linking and locking it to location #0. Linking an exit gives it a destination, and locking is a way of restricting who can and cannot use the exit. Locking the exit to #0 means that any object traversing the cave exit must either be #0 or carrying #0. Because #0 is a room on nearly every MUSH, neither case is likely. When Alaric attempts to traverse the exit, he triggers the fail action. All players other than Alaric will see the ofail text following Alaric's name, and only Alaric will see the fail message. The server executes any commands in the exit's afail attribute, so any number of actions can happen.

Executing the TinyMUSH Commands

When a player types in the command say Hello everyone!, the networking code buffers the text in the player's DESC. After the network layer parses the command into a distinct command buffer, the main loop eventually arrives at the code that handles interactive commands. The server calls the function do_command() with the player's DESC and the command typed.

Each player's DESC contains a buffer for interactively typed commands, but there are noninteractive commands, too. These commands are triggered by $ commands or other nondirect methods, like an object programmed to bark like a dog whenever anyone enters the room. Noninteractive commands take a back seat to commands from the sockets, so the turnaround time for the players is short. All objects take a backseat to players, as the server adds commands typed by objects to the end of a low-priority queue.

In the old days of TinyMUSH, the command parser was one huge switch() statement. As the command set grew, the organization of the cases became unwieldy and inefficient. The developers took the next logical next step—a hash table was put in place for version 2.0.

In TinyMUSH, there are two command hash tables. The first hash table contains commands reserved only for connected players. The second table contains the commands available to both players and MUSH objects alike. Listing 1 shows the in-game command hash table.

When the function do_command() receives a player's command, the code checks the command to see if it exists in the command table for connected players. If the command is not in the hash table with connected player commands, the function do_command() calls process_command(). The handling of player commands has to be fast. The addition of the hash table to TinyMUSH was a plus, but the developers took another shortcut—the recognition of single character lead-ins.

Single character leads are a group of characters in a 256-character array, where each character is the unique first character of a commonly typed command or abbreviation of a command. A command say has an abbreviation of ", used like so:

Listing 1. The In-Game Hash Table

```
typedef struct cmdentry CMDENT;
    struct cmdentry {
            char *cmdname;
            NAMETAB *switches;
            int perms;
            int extra;
            int callseq;
            void (*handler) ();
    };

NAMETAB, defined in htab.h, is a list of switches that modifies the command behavior. Rather
than sending text to an object, a message could be sent to an object´s contents with the use of
/contents on the @pemit command.

typedef struct name_table NAMETAB;
    struct name_table {
            char *name;
            int minlen;
            int perm;
            int flag;
    };

Example CMDENT entries from command.c:

For the command, @pemit, which sends text to an object, in the form:
@pemit <object>=<text>.  The command:

            @pemit The Statue=The pigeons are coming!

Would send the text, "The pigeons are coming!" to the object named "The Statue."

{
                (char *) "@pemit",
                pemit_sw,
                CA_NO_GUEST | CA_NO_SLAVE,
                PEMIT_PEMIT,
                CS_TWO_ARG | CS_INTERP,
                do_pemit
},

The switches are:

NAMETAB pemit_sw[] =
{
        {(char *) "contents", 1, CA_PUBLIC, PEMIT_CONTENTS},
        {(char *) "list", 1, CA_PUBLIC, PEMIT_LIST | SW_MULTIPLE},
        {(char *) "object", 1, CA_PUBLIC, 0},
        {NULL, 0, 0, 0}
};
```

```
"Hello everyone
You say "Hello everyone!"
```

Both `say` and " exist in the command hash table, but only " is a single-character lead—`s` would make a lousy single character lead, blocking any user-defined commands or other built-in commands starting with “s”. When `process_command()` receives the command buffer, the function uses the first character of that buffer as an index into the array of single-character lead-ins, called `prefix_cmds[]`. This indexing will quickly acquire the pointer to the entry of the command in the hash table, that is, if the entry in `prefix_cmds[]` isn't null. If the `prefix_cmds[]` entry is null, the server looks up the command in the hash table.

If the command typed did not have a character lead-in, the server has more work to do. It must match the player's text against all possibilities. Is the player typing in a built-in command, like traversing an exit, or triggering a $ command? And, if the command is an exit or a $ command, where are they—on the player, in the room, or in the master room?

If the matching of a lead-in produces a `NULL`, TinyMUSH starts with the `home` command, which deserves some special treatment because it instantly transports all players home. The game checks the `home` command right after the lead-ins because exits, matched before commands, could redefine all the movement commands, allowing the user to be trapped.

To execute the player's command, the server must know the type of command. Is `north` an exit, or is it a local command on a nearby vehicle to make it go north? As exits and $ commands can have any name, the server matches commands to all the various possibilities. The order of command matching is:

- Reserved commands for connected players
- Common commands with unique single-character lead-ins
- The home command
- Exits in the same room as the player
- Global exits in the master room
- Built-in commands (like `say`)
- Leave aliases for objects
- Enter aliases for objects
- $ commands on the player
- $ commands on nearby objects and the location
- $ commands on objects in player's inventory
- $ commands on the master room and any objects in the master room.

Because the game matches $ commands last, those commands cannot redefine anything, but players can use exits to redefine built-in commands.

The function `process_command()` handles the matching order of the commands after the connected player-only commands. Once the processing function is called, the server will handle each command depending on what sort of command is entered. The server handles all built-in commands with `process_cmdent()` and exits by `move_exit()`. The $ commands are first matched in `atr_match1()` and placed on the low-priority queue for execution after the interactive commands.

Because the MUSH built-ins are the commands used most often, a look into how `process_cmdent()` handles commands is worthwhile, especially since the built-ins compose the core functionality of MUSH.

The function process_cmdent() provides the sanity checking before calling the command handler pointed to in the CMDENT type. When the hash location routine returns a pointer to the built-in command table, the entry is of the type CMDENT. CMDENT contains a detailed description of a command, including possible command switches that alter the command behavior, permissions for who may invoke the command, special flags for the command handling function, flags describing the arguments, and a pointer to the command handling function. A hash table of type CMDENT describes all the MUSH built-in commands.

Before the server calls the command handler function, one last check remains: should the command's arguments be evaluated? Evaluation will expand the substitution characters' functional expressions in a recursive-descent fashion. After the server decides whether or not to evaluate the arguments, TinyMUSH calls the command handler via the function pointer in CMDENT, handling the player's request.

The Database Layer

Hackers treat the database layer of TinyMUSH with more respect—even awe—than any other part of TinyMUSH. New, aspiring TinyMUSH hackers, called mushhacks, avoid modifying this section with good reason—mess the database up and the best you can hope for is that large sections of your virtual world will become corrupted. At worst, one wrong move can result in the premature end of the MUD, mangling the database into a worthless pattern of bits. It's not just the programmer's lost work, either—it is everyone's. Database corruption causes many human years of player efforts to be lost.

Dbm makes the TinyMUSH database possible. UNIX usually comes with the dbm library, and if dbm is not present, it is replaceable with gdbm, a publicly available version of dbm by the Free Software Foundation. Dbm is able to maintain huge numbers of key and

Listing 2. The MUSH Main Loop Function (Continued on p. 42)

```
void
shovechars(port)
        int port;
{
        [...]

        while (mudstate.shutdown_flag == 0) {
                get_tod(&current_time);
                last_slice = update_quotas(last_slice, current_time);

                process_commands();
                if (mudstate.shutdown_flag)
                        break;

                /* test for events */

                dispatch();

                /* any queued robot commands waiting? */

                timeout.tv_sec = que_next();
                timeout.tv_usec = 0;
                next_slice = msec_add(last_slice, mudconf.timeslice);
                slice_timeout = timeval_sub(next_slice, current_time);

                FD_ZERO(&input_set);
                FD_ZERO(&output_set);

                /* Listen for new connections if there are free descriptors */

                if (ndescriptors < avail_descriptors) {
                        FD_SET(sock, &input_set);
                }
                /* Mark sockets that we want to test for change in status */

                DESC_ITER_ALL(d) {
                        if (!d->input_head)
                                FD_SET(d->descriptor, &input_set);
                        if (d->output_head)
                                FD_SET(d->descriptor, &output_set);
                }

                /* Wait for something to happen */
                found = select(maxd, &input_set, &output_set, (fd_set *) NULL, &timeout);
                if (found < 0) {
                        if (errno != EINTR) {
                                log_perror("NET", "FAIL", "checking for activity", "select");
                        }
                        continue;
                }
                /* if !found then time for robot commands */

                if (!found) {
                        do_top(mudconf.queue_chunk);
                        continue;
                } else {
                        do_top(mudconf.active_q_chunk);
                }

                /* Check for new connection requests */

                check = CheckInput(sock);
                if (check) {
                        newd = new_connection(sock);
                        if (!newd) {
                                check = (errno && (errno != EINTR) &&
                                                  (errno != EMFILE) &&
                                                  (errno != ENFILE));
                                if (check) {
```

Listing 2. The MUSH Main Loop Function (Continued from p. 41)

```
                    log_perror("NET", "FAIL", NULL, "new_connection");
                }
            } else {
                if (newd->descriptor >= maxd)
                    maxd = newd->descriptor + 1;
            }
        }
    /* Check for activity on user sockets
    /* DESC_SAFEITER_ALL() is a define which steps through the player
    /* descriptors, which contain the sockets.
    */
        DESC_SAFEITER_ALL(d, dnext) {

            /* Process input from sockets with pending input */

            check = CheckInput(d->descriptor);
            if (check) {

                [...]

                /* Process received data */

                if (!process_input(d)) {
                    shutdownsock(d, R_SOCKDIED);
                    continue;
                }
            }
            /* Process output for sockets with pending output */

            check = CheckOutput(d->descriptor);
            if (check) {
                if (!process_output(d)) {
                    shutdownsock(d, R_SOCKDIED);
                }
            }
        }
    }
}
```

content pairs in a database, accessing a keyed item quickly using a hash routine.

The database consists of a few files, but the main file has the .db suffix. This file contains the size, version, the list of user-named attributes, and some information for each object. The DB file stores the following object data:

- name
- location
- exits
- contents
- attributes contained
- parent
- lock
- owner
- monetary worth
- status flags.

Sound familiar? It should—this information is stored in the in-memory object array of type `OBJ`.

The database is simply formatted because it's read in sequentially and checkpointed at intervals during run time. The part of the database containing the attribute data, sometimes called the attribute database, is more complex. Attribute text may exist anywhere in the attribute file, so all reads are random-access. The attribute database uses dbm. The dbm database uses the attribute and object numbers combined for a key. The data in the dbm database associated with the key is an offset and length into the flat file that stores the attribute text.

When an attribute is being referenced, the server calls `atr_get_raw()`. MUSH creates a unique key for the attribute and calls `FETCH()`. `FETCH()` is a define attribute, referencing the function `cache_get()`. The cache is a transparent layer over the database. When an object is referenced and not present in the cache, the object is read in from disk.

During run time, the game employs a bitmap of the file storing the attribute text, with each bit corresponding to a block within the file. The server uses the bitmap to allocate and deallocate space from the file. If an attribute being written to the file is larger than any free area in the file, TinyMUSH calls the function `grow_bit()`. `grow_bit()` extends the database bitmap. Then, the game writes the attribute to the end of the file, the position in the attribute file, and the length of the key into the dbm database.

Putting It All Together

TinyMUSH, like all C programs, has a `main()` function where execution begins. In `main()`, the server performs startup initializations; it checks to see if a new database is to be created, initializes memory allocation structures, hash tables, reads in the file of configuration parameters, loads the database, and calls `shovechars()`. The loop in `shovechars()` is where the game remains until shutdown, when the database is written out, all file descriptors written to, and the server exits.

Listing 2 shows the main loop of TinyMUSH; the function `shovechars()` continues until an administrator types `@shutdown`, setting the shutdown flag that terminates the game. In this loop, the current time is important—players may enter only so many commands per time slice, and `select()` waits until the end of the next time slice or when any network activity occurs, whichever comes first.

Because TinyMUSH bases command quotas on time, the server acquires the current time in the main loop. The server uses command quotas to limit the number of interactive commands one player may execute in a period of time, called a time slice. With the quotas updated to reflect the current time, the MUSH server executes as many commands as users can afford with their quotas.

During its execution, TinyMUSH must perform regular maintenance tasks periodically. The database must be checkpointed, disconnected, and floating rooms identified, idle players booted, garbage objects collected and so on. So the main loop of shovechars() calls dispatch(). The function dispatch() checks the time to see if any periodic events need to be performed.

Once the game queues its commands and miscellaneous events, it waits until either network activity occurs (on the player sockets), or the current time slice expires with the select() function. If there is network activity, MUSH typically processes no commands from the queue of noninteractive commands and processes three commands from the queue if no network activity exists. As I mentioned, the bias to interactive commands exists to give the players first crack at the CPU.

After select() in shovechars() returns, two fd_sets return, informing the server which file descriptors have input ready and are ready to receive output. These fd_sets enable the networking code to handle the player connections by exception rather than polling.

At the end of the main loop in shovechars(), the server checks the existence of pending input and output on the player's sockets. TinyMUSH checks all the sockets, reading those with input and placing commands in a linked list. The server now writes to sockets ready to receive pending output until they block or until MUSH has written all pending output to the socket. Then, the loop begins again and repeats until a wizard types @shutdown, ending the game.

MUDS: Nonstop Playgrounds

MUDs are international activities, running for days or weeks at a time. The MUSH may stop to make a stable backup or go down with a system crash. But other than that, TinyMUSH, like most MUDs, just keeps looping about, processing commands, allowing virtual playgrounds on the Internet to continue.

By necessity, we couldn't discuss many significant parts of the server, including the operation of the in-memory database cache, the operation of the command queues, the method of argument evaluation, and in-game game help systems. We have also included only a small sampling of appropriate code listings. To access the remainder of my listings, go to the *Game Developer* ftp site at ftp://ftp.mfi.com/gdmag/src. The code to TinyMUSH is freely available via anonymous ftp at caisr2.caisr.cwru.edu in the directory /pub/mush. On the Web, try ftp://caisr2.caisr.cwru.edu/pub/mush/. ■

Edmond L. Meinfelder works at the Naval Research Laboratory in the Center for Computing Science's Research Networks Group. He can be reached via e-mail at edmond@cmf.nrl.navy.mil, on the Web at http://www.nrl.navy.mil/CCS/people/edmond/, or through Game Developer *magazine.*

Jeffrey A. Vance works for TRW. You can contact him via e-mail at shadow@eng.umd.edu or through Game Developer *magazine.*

Back

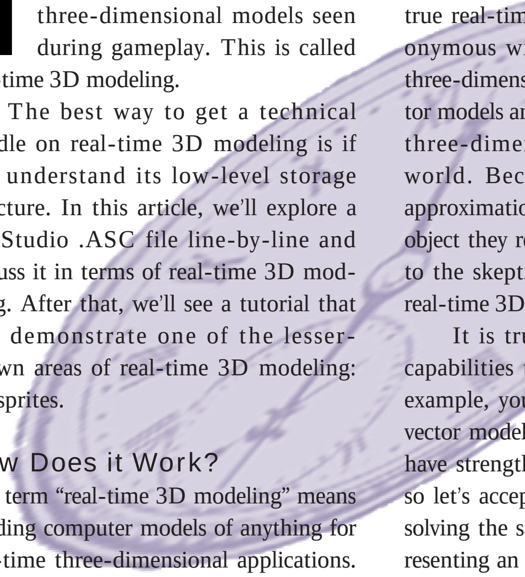
Real-Time 3D Modeling

Listing 1. Square.ASC

```
Ambient light color: Red=0.3 Green=0.3 Blue=0.3

Named object: "Flat Square"
Tri-mesh, Vertices: 8   Faces: 12
Vertex list:
Vertex 0:   X:1.000000  Y:1.000000  Z:1.000000
Vertex 1:   X:-1.000000 Y:1.000000  Z:1.000000
Vertex 2:   X:-1.000000 Y:-1.000000 Z:1.000000
Vertex 3:   X:1.000000  Y:-1.000000 Z:1.000000
Face list:
Face 0: A:2 B:1 C:0 AB:1 BC:1 CA:1
Material:"Default"
Face 1: A:3 B:2 C:0 AB:1 BC:1 CA:1
Material:"Default"
```

The hottest new games are immersive three-dimensional environments controlled by the players—games known as real-time 3D games. In making this type of game, an artist must create the three-dimensional models seen during gameplay. This is called real-time 3D modeling.

The best way to get a technical handle on real-time 3D modeling is if you understand its low-level storage structure. In this article, we'll explore a 3D Studio .ASC file line-by-line and discuss it in terms of real-time 3D modeling. After that, we'll see a tutorial that will demonstrate one of the lesser-known areas of real-time 3D modeling: 3D sprites.



How Does it Work?

The term "real-time 3D modeling" means building computer models of anything for real-time three-dimensional applications. In our case, we specifically mean games. The term is really quite specific. Real-time means the code computes, on the fly, the scene players see while they are playing. Descent, Flight Unlimited, Indycar Racing, and most virtual reality games are examples of real-time 3D games. These games contrast with prerendered three-dimensional games like Myst or 7th Guest.

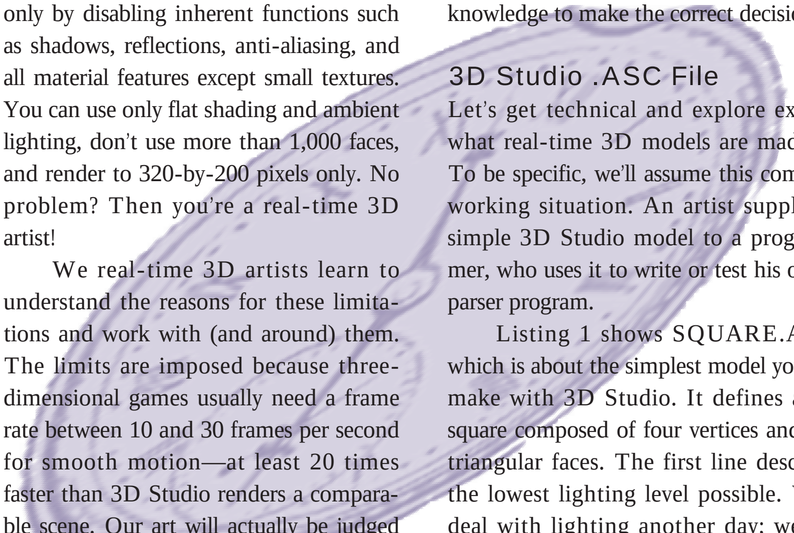
Of course, games do exist that span this division. For example, Alone in the Dark features a real-time three-dimensional human model walking around in prerendered rooms. If you've seen this game, the difference between the two types of art is immediately obvious.

Doom is also an example of a real-time 3D game—sort of. The characters are what I call *3D sprites*—a series of small bitmaps, which are pictures of a person viewed from several angles. Some people would not call the graphics used in Doom true real-time 3D, which is usually synonymous with vector-based polygonal three-dimensional models. However, vector models and 3D sprites both represent a three-dimensional object in a virtual world. Because they are both gross approximations of the three-dimensional object they represent, I submit them both to the skeptical reader as valid forms of real-time 3D models.

It is true that vector models have capabilities that 3D sprites do not. For example, you can change the shape of a vector model in real time. 3D sprites also have strengths that polygon models lack, so let's accept each as a different way of solving the same problem: accurately representing an object in a real-time simulated environment. We'll explore 3D sprites in detail later on. For now, let's talk about vector models.

Josh White

The newest games increasingly rely on impressively rendered three-dimensional graphics. Here are some tips for creating efficient three-dimensional models in real time.



Limitations of Real-Time Modeling

Game artists beware—real-time modeling is very limiting compared to normal three-dimensional modeling. If you're working in 3D Studio, you can experience the limitations that real-time modeling imposes, only by disabling inherent functions such as shadows, reflections, anti-aliasing, and all material features except small textures. You can use only flat shading and ambient lighting, don't use more than 1,000 faces, and render to 320-by-200 pixels only. No problem? Then you're a real-time 3D artist!

We real-time 3D artists learn to understand the reasons for these limitations and work with (and around) them. The limits are imposed because three-dimensional games usually need a frame rate between 10 and 30 frames per second for smooth motion—at least 20 times faster than 3D Studio renders a comparable scene. Our art will actually be judged in another dimension—smoothness. As our model gets simpler the frame rate improves, and the gameplay gets smoother and better.

If we understand the basics of how our models are rendered at run time, we can make sure trade-off decisions benefit the art in the game. For example, many game architects often avoid smooth shading in favor of textured, flat shading; however, this deserves some thought.

Untextured smooth shading does have strengths. If a model is Gouraud shaded, you can very simply model curved surfaces without affecting the appearance of the model, which frees up faces to be used elsewhere. Smooth shading also yields excellent results on curved objects of a single color, such as a pool cue or car body.

So, there is a tradeoff between the detailed surfaces provided by textured, flat-shaded faces and the speed and smoothness of Gouraud shading. We need both artistic judgment and technical knowledge to make the correct decision.

3D Studio .ASC File

Let's get technical and explore exactly what real-time 3D models are made of. To be specific, we'll assume this common working situation. An artist supplies a simple 3D Studio model to a programmer, who uses it to write or test his or her parser program.

Listing 1 shows SQUARE.ASC, which is about the simplest model you can make with 3D Studio. It defines a flat square composed of four vertices and two triangular faces. The first line describes the lowest lighting level possible. We'll deal with lighting another day; we can safely ignore it for now.

Line 3 starts the definition of an object. 3D Studio, in the simplest case, stores its data by organizing objects that contain faces and vertices.

We cannot have a vertex that does not belong to any object. If we want a single vertex in space, we must define a one-vertex object. For object-oriented programmers, this structure should make sense, and similar structures are used in many three-dimensional engines. (The terms "three-dimensional engine" or "graphics engine" refer to the basic code that takes a three-dimensional model and renders it in real-time. It is separated from the rest of the game code, for example physics simulation, scoring, and so on.)

3D Studio objects may optionally contain other data, such as hierarchy relationships between objects (parent/child) and animation path information, but our example does not.

Line 4 summarizes the object that follows, describing what kind of object it is (trimesh) and its basic size.

Lines 5 through 9 show a group of X, Y, and Z coordinates defining the three-dimensional points (vertices) in space. These vertex definitions can include optional information such as UV coordinates, but our example does not.

Lines 10 through 14 define a finite planar triangular surface known as a face. It is defined by referencing the vertex numbers (`A:2 B:1 C:0`). Several faces can reference the same vertex, which is called "vertex sharing." This is common in 3D Studio files and real-time three-dimensional files alike. The faces have other important data such as textures or smoothing information, but all we care about for now is the vertex references and the material name.

Each face is followed by a text name that describes a 3D Studio "material." This is a reference to a set of colors, texture maps, and other attributes that 3D Studio uses to render a surface. Most of the fancier settings (for example, bump mapping, specular highlight mapping, and so on) are not possible in real-time games, so most developers only parse the texture map name and settings, the basic color values, and perhaps the opacity mapping (similar to alpha channel support).

That's it! If this information is supported by the programmers' code, the simplest real-time three-dimensional models can be read in.

So what?

In the .ASC file, the key is the basic structure of a vertex list followed by a face list that refers back to the vertices. Unlike other three-dimensional structures such as the crude `3DFACE` entity found in .DXF files, a 3D Studio face cannot exist without vertices to connect to. This is a good thing for real-time games because the simple, small data structure allows niceties like easy real-time manipulation of models (that is, three-dimensional morphing). You simply move the vertices, and the faces follow automatically.

The downside is that 3D Studio does have some annoying limits in its data structures, mainly that it supports only triangular faces. Many game developers find that four-sided faces are a good solution to many problems, including convenient surfaces to texture and even more efficient storage of models.

Now, let's explore another kind of three-dimensional modeling—3D Sprites.

3D Sprites Tutorial

A 3D sprite is made of a series of pictures of the same object from different viewing angles. 3D sprites are a powerful technique for representing objects that are difficult to build with polygons, such as human figures, clouds, or fire. Using detailed source art (even small photographs) lets you portray convincing detail without slowing the game down with lots of polygons.

The bad news has to do with memory: The many bitmaps that compose the 3D sprite eat up RAM very quickly. Thus the number of bitmaps are usually under very strict budgets, limiting the smoothness of motion. Also, animating a 3D sprite takes up even more memory. Memory use is our main limitation, so this tutorial will concentrate on ways to reduce memory use in 3D sprites. We can make a 3D sprite by using the following five steps:

- Get a source
- Figure the number of frames considering range and symmetry

Table 1. 3D Memory Use Table

We assume a spheric camera range with 16 points around the equator and eight levels from pole to pole, yielding 128 frames maximum:

Description	Frames	RAM
Original sprite (complete sphere)	128	524K
Limit camera range (half sphere)	64	262K
One plane of symmetry (quarter sphere)	32	131K

WHERE ARE THE QUADS?

One common question that developers ask is, "Why doesn't 3D Studio support four-sided faces?" There are some problems with four-sided faces (known as "quads") compared to three-sided faces (triangles). The most common problem, warping, occurs when the four corner vertices do not lie on the same plane. It is very hard for an artist to avoid making warped quads when creating a complex, organic shape. Programmers know warping results in inaccurate normals, which cause many problems. For example, the warped quad will disappear when viewed edge-on.

Another problem unique to quads is an invalid shape such as a bow tie. These degenerate shapes often occur when a model is deformed in real time, though they can be created by accident (especially by ex-3D Studio modelers who are not used to quads).

USING QUADS WITH 3D STUDIO

If quads are supported, a simple, effective solution is to incorporate a "face combiner" feature in the three-dimensional file parser. This code searches for coplanar triangular faces that share two vertices and combines pairs of them into quads.

If more than two faces could be combined, the parser must make a decision. Some parsers decide randomly; others use the "edge visibility" settings stored in 3D Studio files to determine which pairs of faces to combine. This allows the artist to set the divisions inside 3D Studio using Modify/Edge/Visible in the 3D Editor.

- Set the camera angles and make the camera position list
- Automate the rendering and render the frames
- Hand off the sprite.

Get A Source

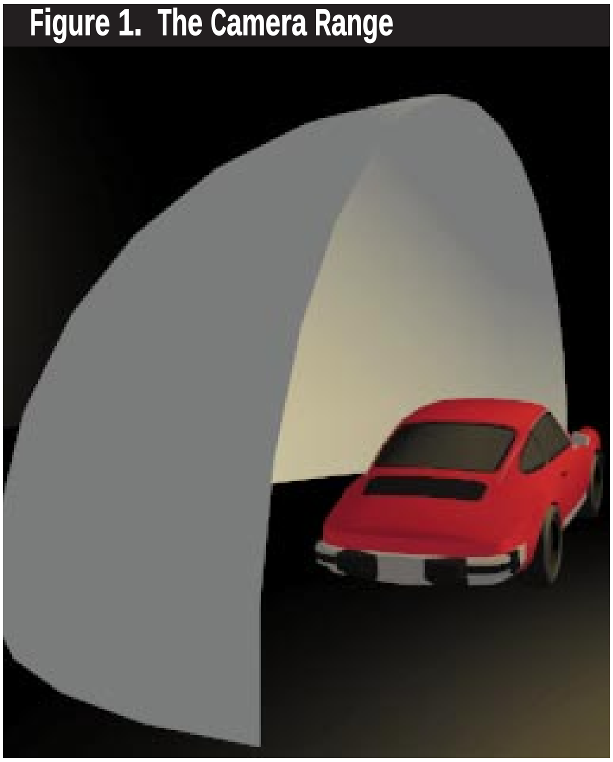
To make a 3D sprite, we start with a three-dimensional source. In theory, this can be a real-world object, like a sports car or a person. However, the source is usually a normal three-dimensional computer model intended for standard rendering. This model will never be imported into the game directly, so its polygon count and other limitations don't apply. For this tutorial, we'll use a three-dimensional model of a Porsche 911.

Decide the Camera Range

Next, we must decide from which possible positions the sprite could be seen during game play. To accomplish this, imagine that our Porsche is enclosed in a sphere, the surface of which is called the camera range. Each point on the camera range represents a possible position of the player viewing the car, and each view is called a frame. Figure 1 shows the camera range on our source model, the Porsche.

Can our Porsche be seen from any direction (that is, any point on the camera range)? In the worst case, such as an enemy plane in a flight simulator, it will, but in most cases, some areas can be eliminated. This is good because the less we can see, the more frames we can use to show the rest of the model.

Figure 1. The Camera Range



Listing 2. Camera Position List: CAMPPOS.ASC

```
Ambient light color: Red=0.0 Green=0.0 Blue=0.0

Named object: "CamPPositon"
Tri-mesh, Vertices: 30 Faces: 0
Vertex list:
Vertex 0:   X: 0.001081    Y: 0.002534    Z: 0.994797
Vertex 1:   X: -0.155636   Y: -0.268908   Z: 0.944466
Vertex 2:   X: -0.296596   Y: -0.513057   Z: 0.798534
Vertex 3:   X: -0.407624   Y: -0.705364   Z: 0.571674
Vertex 4:   X: -0.477557   Y: -0.826492   Z: 0.286697
Vertex 5:   X: -0.499364   Y: -0.864262   Z: -0.007422
Vertex 6:   X:  0.0013     Y:  0.002534   Z:  0.944466
Vertex 7:   X: -0.155636   Y:  0.273977   Z:  0.944466
Vertex 8:   X: -0.296596   Y:  0.518126   Z:  0.798533
Vertex 9:   X: -0.407624   Y:  0.710432   Z:  0.571674
Vertex 10:  X: -0.477557   Y:  0.83156    Z:  0.286696
Vertex 11:  X: -0.499364   Y:  0.86933    Z: -0.007422
Vertex 12:  X:  0.001286   Y: -0.298214   Z:  0.932635
Vertex 13:  X:  0.001274   Y: -0.610524   Z:  0.783235
Vertex 14:  X:  0.001266   Y: -0.885267   Z:  0.455405
Vertex 15:  X:  0.001258   Y: -0.992485   Z: -0.007423
Vertex 16:  X:  0.001286   Y:  0.303283   Z:  0.932634
Vertex 17:  X:  0.001274   Y:  0.615592   Z:  0.783235
Vertex 18:  X:  0.001266   Y:  0.890335   Z:  0.455405
Vertex 19:  X:  0.001258   Y:  0.997552   Z: -0.007424
Vertex 20:  X: -0.346193   Y:  0.002534   Z:  0.932634
Vertex 21:  X: -0.557221   Y:  0.261643   Z:  0.783235
Vertex 22:  X: -0.557221   Y: -0.256575   Z:  0.783235
Vertex 23:  X: -0.753137   Y:  0.47179    Z:  0.455405
Vertex 24:  X: -0.852744   Y:  0.002534   Z:  0.516384
Vertex 25:  X: -0.753137   Y: -0.466722   Z:  0.455405
Vertex 26:  X: -0.80824    Y:  0.590785   Z: -0.007424
Vertex 27:  X: -0.977282   Y:  0.210601   Z: -0.007424
Vertex 28:  X: -0.977282   Y: -0.205533   Z: -0.007424
Vertex 29:  X: -0.80824    Y: -0.585717   Z: -0.007423
```

For example, if the car is placed in a open (but inaccessible) garage, the garage blocks the viewer from ever seeing the back of the car. We don't have to store any frames that show the back, because it can't be seen during the game. This leaves us with just half (or less) of the camera range we had.

Use Planes of Symmetry

Our car has a plane of symmetry, that is, one half is a mirror image of the other half. By writing game code that mirrors the frames at run time, the game can simulate both sides of a symmetrical object.

If our object is radially symmetrical, like a glass bottle, we can reduce our frame count even further. We only need one viewing angle in the horizontal direction, decreasing our camera range drastically. Of course, this won't work for our car. With its plane of symmetry, our camera range is now a quarter-sphere.

Figure the Number of Frames

Next, we must decide how many frames of our sprite will be available within the game. This depends on the available RAM, the size of the bitmap, the color

Figure 2a. Top View on a Sphere

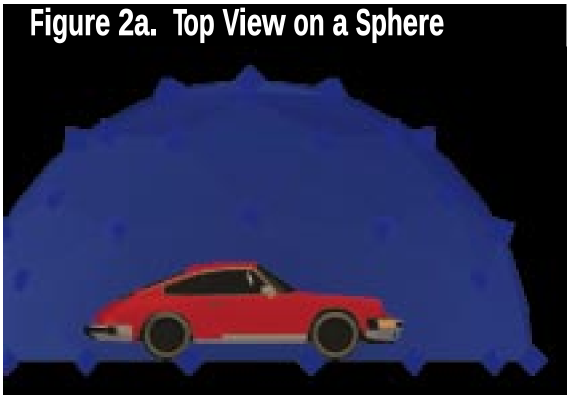
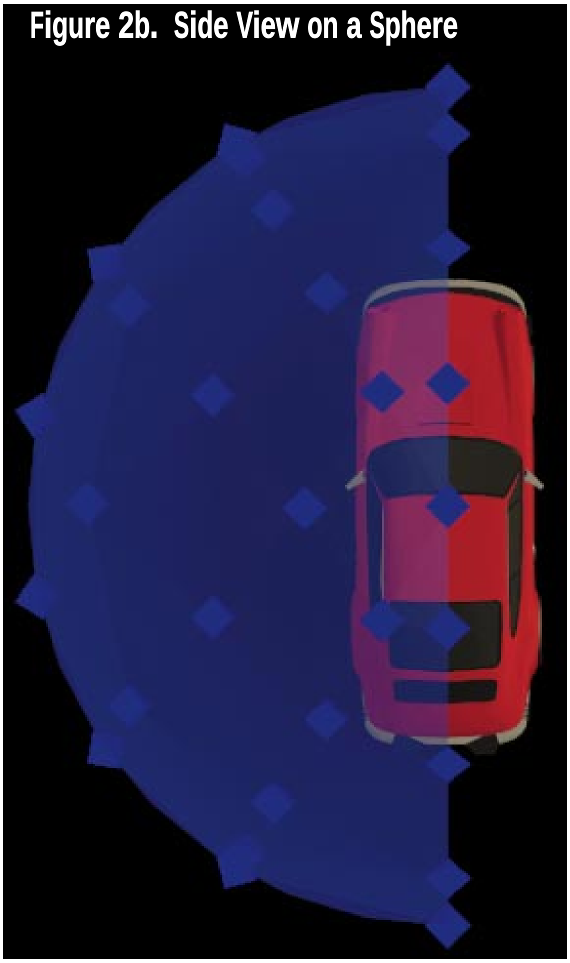


Figure 2b. Side View on a Sphere



depth, and the compression (if any) of the bitmap in memory.

For our example, let's say we will have 128K of memory available for this sprite, bitmaps will not be compressed, and our game will use 8-bit color. Table 1 shows a 3D sprite memory use table. This gives us 30 bitmaps of 64-by-64 pixels. If the programmers don't know how much memory will be available for this object, don't render the sprites until the coding is farther along and the memory resources are more defined.

Set the Camera Angles

Once we know how many frames we can use, we must decide which angle they will show; that is, which 30 viewing directions in the camera range are most important? Figures 2a and 2b show the camera positions on a sphere (top and side views). The obvious starting point is a simple latitude and longitude grid over the quarter sphere, but there are other factors to consider:

The latitude and longitude distribution is not ideal because the points are closer to the poles. A better starting point would be a geodesic sphere. This soccer-ball-type arrangement distributes the points evenly over the surface of the sphere.

The most likely viewing angles should have more frames because we prefer the model to move smoothly when it is seen most often.

A complete coverage is also important. If we completely leave out large areas of the camera's range, the model will appear to jump when viewed from those areas.

The shape and appearance of the model must be considered. If our object was a lollipop, we would use fewer frames in which the stick wasn't visible because jerky motion would be less noticeable with the stick absent.

Here's how to create the camera locations using 3D Studio. (Before actually doing this, set the coordinate labels correctly). When 3D Studio creates .DXF files—and .ASC files, too—the Y and Z coordinates are swapped by default. You can fix this by setting the coordinate labels in the 3DS.SET file as follows. (Remember to save the old settings!):

```
H-LABEL = Z
W-LABEL = X
D-LABEL = Y
```

After you've reset the settings, follow these steps:

1. Use Create/GSphere/Faceted to create a geodesic sphere around your object. The "Values" setting should be set to 196 faces, yielding 30 points on a quarter-sphere. The exact number of faces you want will vary (when in doubt, choose more). The sphere should have a radius of 1 and be centered at 0,0,0. Choose an appropriate name like `CAM-POSITION`.
2. Erase the faces of the sphere that are totally outside your camera range; that is, three-quarters of the sphere. Erase the isolated vertices.
3. Choose Modify/Object/Attributes to check how many vertices remain. If the number of vertices is less than your target amount, you must use a `GSphere` with more vertices in Step 1. Otherwise, use Modify/Vertex/Delete to erase the extras that are farthest from your camera range.
4. Move the vertices into the camera locations you've chosen. Use Modify/Axis/Place to set the axis in the center of the sphere and use Modify/Vertex/Rotate to move the vertices. Using a rotation around the center of the sphere will keep the vertices on the sphere's surface. If you use other editing commands and accidentally move the vertices off the surface, create a second sphere (repeat Step 1), freeze it, and compare your distorted vertices to its correct surface.
5. Select the object and save it to a .3DS file (in the File menu, choose Save Selected). Load that 3DS file.
6. Delete all the faces in the object, keeping the isolated vertices.

This is your model of the camera positions.

Make the Camera Position List

Depending on what we plan to do with the camera position list, we will save it as a .3DS , .ASC, or .DXF file. We'll have to know more about how the rendering engine incorporates sprites before we decide, but somehow we have to provide this list. The rendering engine will use the list to determine which frame to show when the player is able to see the sprite.

For the simplest format, save your information as an .ASC file, then open the file in a normal text editor. The list of vertices that follow the object name is your camera position list. They should look something like the CAMPOS.ASC file listing, shown in Listing 2.

Automate the Rendering

The next step is to render each frame from each camera position. We could simply create a camera at each location and render it, but there's a tricky, handy way to automate this. The idea is to create a single camera that jumps from point to point in the Keyframer. Follow these steps:

1. Save your original file as a .DXF file with one layer.
2. Fix the .DXF file. Listing 3 shows the CAMPOSE.DXF file.

   When you save a .DXF file from the 3D Editor, 3D Studio cannot import it as a path because it assumes you're saving a surface, not a line. Here's how to get around this problem:

   1. Open the .DXF file in a text editor. Search for `ENTITIES`. Look on the screen for `POLYLINE`, a few lines down.
   2. Move down to the line that has 70 all by itself (about five lines down from `POLYLINE`).
   3. The number on the next line after 70 should be 8, not 64. Change this number, save, and exit.
3. Create a camera, go to the Keyframer, and set the number of animation frames (animation length) to one more than the number of camera positions you have.
4. Choose Paths/Get/Disk. In the file dialog box, choose the .DXF file you just saved. In the next dialog box, make the camera move to the path start, and do not allow the keys to be adjusted for constant speed.
5. Choose Paths/Show-Hide and click on the camera to see the path. The camera will now have a path with a keyframe for each position in the list.
6. Check to see that the last two animation frames are the same camera position; if so, reduce the animation length by one. If they are not, there may be more camera positions that are not shown. Increase the animation length and reload the path.

Listing 3. CAMPOS.DXF

```
[ Lines omitted: The first 147 lines are
the "header" section and are not used by
3DS; in fact, they can be deleted. ]

0
SECTION
2
ENTITIES
0
POLYLINE
8
CAMPOSITON
66
1
70
8
71
30
72
0
0
VERTEX
8
CAMPOSITON
10
0.001081
20
0.002534
30
0.994763
70
192
0
VERTEX
8
CAMPOSITON
10
-0.155636
20
-0.268908
30
0.944432
70
192

[ Lines omitted: The "0 | VERTEX ..."
block repeats for each of the 30 vertices.
]

0
VERTEX
8
CAMPOSITON
10
-0.296596
20
-0.513057
30
0.798500
70
192
0
VERTEX
8
CAMPOSITON
10
-0.808240
20
-0.585717
30
-0.007457
70
192
0
SEQEND
8
CAMPOSITON
0
ENDSEC
0
EOF
```

Render the Frames

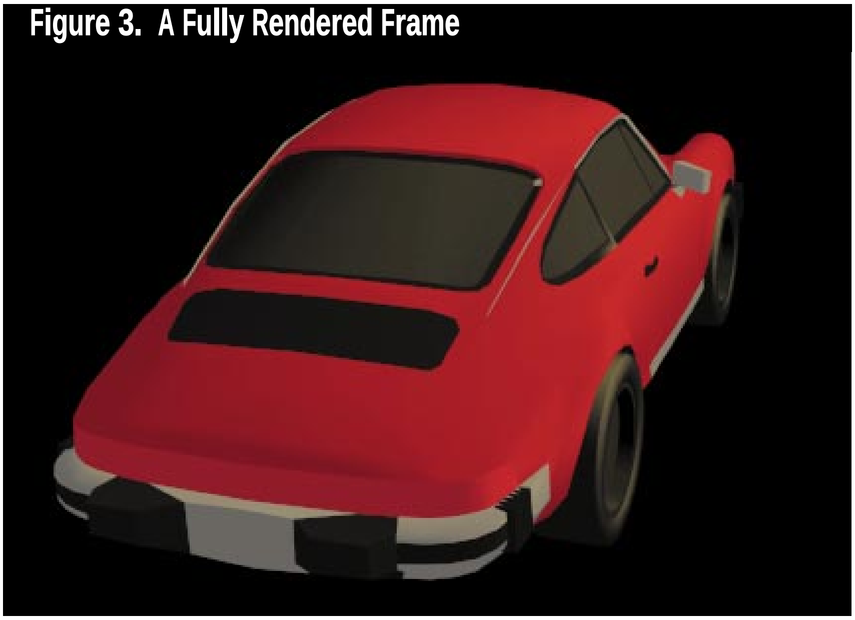
Once the camera is set up, we render the images, saving each one to a separate file. When rendering, we will have to deal with the following problems:

- *Backgrounds*. The object must be isolated from its background during rendering. Using alpha channel is ideal but is not often supported in real-time games. If alpha channel is unavailable, set the background color to an RGB value that will not appear in the rendered object. This is the easiest way for graphics engines to separate the sprite from its background.
- *Perspective*. We have to choose a compromised perspective angle ("fish-eye vs. zoom" camera setting) when we render because the perspective doesn't change during gameplay. Base the amount of perspective on how the object will be most commonly viewed. Generally, err on the side of less perspective.
- *Rendering Size*. The goal is to make full use of the final 64-by-64-pixel frame.

Minimize empty space by spending time with the field-of-view camera setting, carefully getting as many frames as possible to completely fill the rendered bitmap. Usually, it is necessary to use the same camera settings for all frames; otherwise, the sprite will appear to grow or shrink during gameplay. Because many frames will inevitably have large areas of background, it will pay to store the frames using some simple compression algorithm like RLE memory.

- *Lighting and Effects*. There aren't any real limits here, but lighting will appear to be symmetrical if we are taking advantage of planes of symmetry. For example, if we put a blue spotlight on the left side of the model and we use a plane of symmetry, blue lights will appear on both sides of the model when it's in the game. This also applies to shadows and other effects.
- *Redundant Frames*. If we can't see a difference between any two frames, one of the frames is redundant and unnecessary, even if the frames were rendered from very different angles. We can have the two different viewing angles reference the same bitmap, which saves memory.

Figure 3. A Fully Rendered Frame



Hand Off the 3D Sprite

Once we've rendered, all that remains is for us to check all the files for errors and give them, with the camera position list file, to the programmers. This is all the material they need to integrate the 3D sprite into the game. Figure 3 shows our first rendered frame.

Real-Time Art—It's Great!

Let's end with something for the artiste in all of us. As creative developers, especially artists, the limits of real-time three-dimensional modeling are a stiff price to pay, but what we get is the freedom of a whole new medium. Our art is not locked onto a camera path or frozen into a single frame.

The players are free to explore our art like kids on a playground—crawl around under it, fly over it, or bump along with their virtual noses pressed against it. It's truly the next step in computer art, and well worth learning. ■

WHY 3D STUDIO?

In this article, I refer specifically to AutoDesk's modeling software 3D Studio because it is commonly used to build real-time models. Many other significant modelers—for example, Caligari TrueSpace—also support 3D Studio's file formats. I mention this because it is only fair to point out that there are other modelers you can use.

WHY .ASC?

3D Studio allows many types of three-dimensional file formats to be saved, such as .PPRJ, .DXF, .ASC, and .3DS. This article explores the .ASC format, simply because it is stored as a text file and is the most self-explanatory. Many programmers support the binary .3DS format instead, but it's beyond the scope of this article.

WRITING 3D STUDIO FILE PARSERS

If you want to write code that reads or writes the .3DS files that 3D Studio generates, the .3DS file format is documented in AutoDesk's 3*D Studio R3 File Toolkit,* and the author, Grant Blaha, usually answers questions on the free 3D Studio e-mail list at majordomo@autodesk.com.

FREE INFORMATION

Documentation on file formats, including 3D Studio, is also available via ftp on avalon.chinalake.navy.mil and on the ftp site for game programmers, x2ftp.oulu.fi.

To learn more about 3D Studio from other users:

- Send e-mail to majordomo@autodesk.com with the subject "subscribe 3dstudio." To unsubscribe, e-mail majordomo@autodesk.com with the subject line "unsubscribe 3dstudio." Don't write the list alias itself; this will annoy the natives.
- Check out the USENET newsgroup alt.3d.studio if you can. This is better than subscribing via majordomo@autodesk.com because you don't get bombarded by e-mails whether you want them or not.

Josh White is a partner in Vector Graphics, building real-time three-dimensional models for game and virtual reality developers. He can be reached via e-mail at vectorg@crl.com or through Game Developer *magazine. Josh is a frequent poster on rec.games.programmers and comp.graphics.packages.3dstudio, so look out for him there.*

Rise of the Triad

Wayne Sikes

The world within Rise of the Triad is largely possible thanks to innovative ray-casting techniques. Wayne Sikes looks at Apogee's use of masked walls, floor sound tiles, and columnar storage of image data.

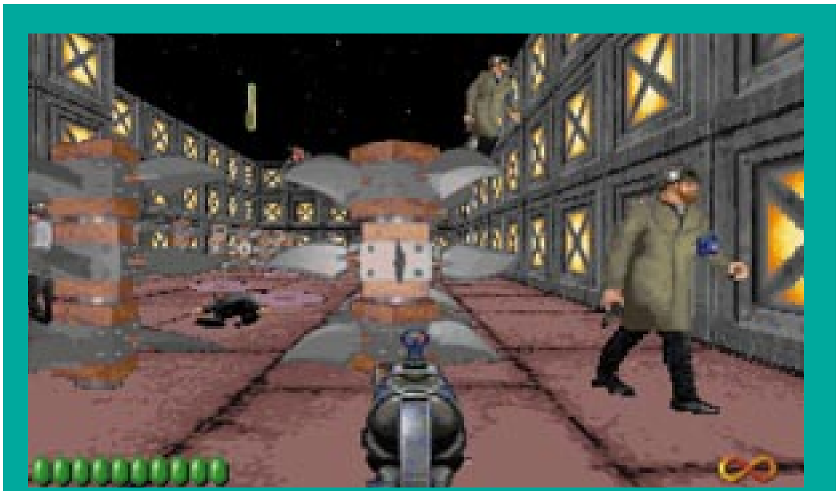
On the Chopping Block this month is Rise of the Triad by Apogee Software. Rise of the Triad is a scrolling, first-person perspective three-dimensional action game. The fast action constantly challenges your reflexes. Triad has a high entertainment factor as well. You'll come up against clever devices and hazards such as fire chutes and jets that can burn you to a crisp, poison gas jets that can be triggered by opening doors or throwing a wall switch, rotating blades that appear out of the floor or ceiling, and movable walls that can crush you against another wall.

The use of features such as jump pads that propel your character into the air to grab suspended objects and Gravitational Anomaly Disks that move up and down or travel over the ground in predefined pathways adds an extra level of creativity and originality. I found some elements of the game such as the "Dog Mode" power-up (as opposed to "God Mode") very amusing. And we mustn't overlook the Violence Level adjustment for tailoring the blood-and-guts graphics.

The primary Triad executable, ROTT.EXE, is a little over 1MB. Triad was written with the Watcom C/C++ Run-Time system. The game runs in protected mode using the DOS4GW DPMI system by Rational Systems. The engine contains a large number of system diagnostics and error detection routines, which are especially useful when you're designing new game levels.

How Does it Look and Sound?

The graphics techniques used in creating the Triad environment are very



Apogee has upped the ante on Rise of the Triad's ray-casting engine, which was originally intended for Wolfenstein 2.

interesting. The Triad world is rendered with a ray-casting algorithm that uses a 90° field of view. Do not assume that this is just another run-of-the-mill ray-casting engine. Apogee has upped the ante on ray casters with its Triad engine. Originally, Apogee projected this engine for use in Wolfenstein 2. Later, the company refined it into the Triad game engine. In fact, many of the graphic images in Rise of the Triad were originally intended for Wolfenstein 2.

Further rendering on Triad is achieved using 320-by-200-pixel graphics in Mode X. (Many game engines that originally rendered their graphics using 320-by-200-pixel Mode 13h are now switching to Mode X.) Triad's developers optimized the ray caster for speed and used well-developed methods for achieving high frame rates while keeping the graphics clean.

For example, consider the case of masked walls—walls you can see through. These walls may be actual walls containing glass windows or gratings you can look through or they may take the form of physical barriers such as picket fences. Older ray casters rendered these walls using two separate castings. The first casting would locate and render the wall farthest away from the viewpoint, and the next casting would locate and render the masked wall on top of the wall rendered with the first cast. You could then look through a transparent area in the closest wall and see the background wall.

This method demands a large chunk of system time and achieves marginal frame rates. The Triad ray caster renders masked walls as "patches" placed on top of background walls or other objects. Essentially, only one ray cast is done, the background is rendered, and finally the masked wall is scaled, rotated, and drawn to fit on top of everything else. The patch looks great and can be done quickly.

Another feature of the ray caster Triad uses is its ability to draw walls of varying heights. Older ray casters generated environments that had a very blocky appearance. The variable-height environment of the Triad world makes it appear much more realistic. Each Triad game map contains height information that instructs the engine to render the world using one of 16 possible heights.

Listing 1. WAD File Structures

```
// Header structure.
typedef struct
{
    char    identification[4];
    long    numlumps;
    long    infotableofs;
} WADHEADER, *PWADHEADER;

// Directory structure.
typedef struct
{
        long    filepos;
        long    size;
        char    name[8];
} LUMPS, *PLUMPS;

// Masked object (actors and sprites) structure.
typedef struct
{
        short origsize;         // the orig size of "grabbed" gfx
        short width;                // bounding box size
        short height;
        short leftoffset;           // pixels to the left of origin
        short topoffset;            // pixels above the origin
        short collumnofs[320];  // only [width] used, the [0] is
                                    // &collumnofs[width]
} PATCH, *PPATCH;

// Transparent object structure.
typedef struct
{
        short origsize;         // the orig size of "grabbed" gfx
        short width;                // bounding box size
        short height;
        short leftoffset;           // pixels to the left of origin
        short topoffset;            // pixels above the origin
        short translevel;
        short collumnofs[320];  // only [width] used, the [0] is
                                    // &collumnofs[width]
} TRANSPATCH, *PTRANSPATCH;

// Structure used for floor and ceiling data.
typedef struct
{
        short   Width,Height;
        short   Orgx,Orgy;
} FLOORCEILING, *PFLOORCEILING;

Note: These structure examples are given in the Official ROTT Specifications file, ROTSP1, by
Apogee Software.
```

The sky background is another area where the engine really shines. Rather than simply drawing background sky and ground or floor textures followed by the ray caster overlaying its data on top, the Triad engine draws the sky as a series of image tiles. Areas of visible sky are tiled into the Triad world. Using the player's current viewing angle and altitude, the engine renders the appropriate piece of the sky area. The end result is a realistic sky background that required only a small amount of system time to generate.

Because the sky background represents the boundary of the Triad world, some interesting effects can be designed into the game's maps. For example, because the Triad world ends at the edge of the floor or ground map tiles, you can fall off the edge of the world when you intentionally (or accidentally) step off a floor tile.

The tiles used for mapping player floor area are one of the more interesting aspects of Triad levels. How can floors be interesting you ask? Some grid-based games delegate floor space by assuming that all area enclosed by walls is floor space. In Triad, the floor space is specifically marked with floor tiles. These tiles are not images or floor textures; instead, they are referred to as “floor sound tiles.” The engine uses the values of the floor tiles to determine how to distribute the various game sounds. For example, if you're standing on a floor tile that has the same value as the tile under a hazard such as a fire chute, you will hear the sound made by the fire chute. Alternatively, if you're on a floor tile that is different from that under the hazard, you will not hear the hazard. You can creatively exploit this feature when designing maps.

Primary Data Storage

Most of the graphic and sound data is stored in DARKWAR.WAD. This file is about 14.6MB and has the same format as the Id Software WAD file used in Doom. Listing 1 shows example code for several WAD file structures. (The structures in Listing 1 are contained in Apogee's Official ROTT Specs file, ROTSP1. You can find this file on CompuServe in the 3-D Action Games library of the Action Games forum (GO ACTION). I included the patch, transparent patch, and floor and ceiling structures just in case you want to look at these objects while examining the WAD file. The data in DARKWAR.WAD is grouped according to type. For example, most of the wall data is stored between the WALLSTRT and WALLSTOP directory entries.

Most of the graphic data in Triad is stored using a particularly interesting method. The Triad bitmap data is stored in columnar format. Most drawing programs usually store their image files using formats such as PCX. The bitmap data is stored in row x column format. This is not the best storage format for rendering images in ray-cast environments, because ray-cast data is rendered as a sequence of vertical lines. When storing data in its final WAD format, Apogee converts the row x column bitmap data into column x row format, which speeds the rendering process.

Map Fundamentals

Triad map files have either an RTL or RTC suffix. RTL files are used mainly for single-player games, and RTC files are designed for ROTT Comm-bat (multiplayer) games. The primary difference between these two files is that the RTC files contain no computer-generated enemies or exits.

Both map file types have the same internal format: an 8-byte version structure, a block of 100 map header structures that contain (among other things) the size and location of the map data, and the map data stored in compressed Run Length Encoded (RLE) format. Listing 2 gives version and header structures for Triad map files. (The structures given in Listing 2 are contained in Apogee's Official ROTT Specs file, ROTSP1.)

The `VERSION` structure contains the file's RTL or RTC signature plus the file format version specification. Each map file contains 100 `RTLMAP` structures with each structure corresponding to a stored map. The `Used` variable indicates whether or not a map is stored in this map slot. `RLEWtag` is the encoding tag used for compressing and decompressing the map data. The `Planestart` and `Planelength` arrays contain the offset positions and lengths of the map data.

When uncompressed, each Triad map consists of three planes of data. Each plane is an array of 128-by-128, 16-bit words. At first glance, the map data format is somewhat confusing. It's easiest to view the map as a grid of 128-by-128 map cells. (I use the reference to Triad map “cells” because most of us are familiar with spreadsheet programs that have their data in cells.) A map cell is composed of three values—one value from each of the three map planes. Each plane contains unique data, and we can label the planes as the Wall Plane, the Sprite Plane, and the Info Plane to help simplify this concept even more.

The Wall Plane contains (as you probably already deduced) the data for the vertical walls. Triad walls also consist of objects such as wall switches, doors,

Listing 1. Map File Structures

```
#define NUMPLANES                   3
#define ALLOCATEDLEVELNAMELENGTH    24
#define WALL_PLANE                  0
#define SPRITE_PLANE                1
#define INFO_PLANE                  2

// Version structure.
typedef struct
    {
    char             Signature[4];
    unsigned long   Version;
    } VERSION, *PVERSION;

// Header structure. There are 100 of these in each file.
typedef struct
    {
    unsigned long   Used;
    unsigned long   CRC;
    unsigned long   RLEWtag;
    unsigned long   MapSpecials;
    unsigned long   Planestart[  NUMPLANES ];
    unsigned long   Planelength[ NUMPLANES ];
    char             Name[ ALLOCATEDLEVELNAMELENGTH ];
    } RTLMAP, *PRTLMAP;


Note: These structure examples are given in the Official ROTT Specifications file, ROTSP1, by
Apogee Software.
```

transparent windows (you can shoot the glass out), gratings, fences, and archways. For mapping purposes, the Wall Plane also contains the floor sound tiles.

The Sprite Plane holds most of the visible objects in the game such as enemy players, light fixtures, power-up objects that give you health, hazards such as rotating blades and knives that periodically cycle in and out of the floor or ceiling, jump pads, boxes, and trees. Path information is also stored in this plane and is used for controlling the movement of game-generated enemies and walls that can be moved or pushed.

The Info Plane provides a very versatile method for storing height, song, X and Y coordinates, exit, and minute and second time data. For example, some game objects can have height above the player floor level. To specify an object's height, the object is loaded into the Sprite Plane, and its height is stored in the Info Plane. In another example, you can control many game objects using wall switches. To program a door that is controlled by a switch, the door object is placed in the Wall Plane, and the X and Y coordinates of the controlling wall switch are placed in the door's Info Plane. Special data such as sky tiling are also stored in the Info Plane.

Want To Design Your Own Maps?

Triad map data can be very tedious to edit by hand, plus you can easily make many mistakes (as I found out). I found the Triad maps interesting enough (and fun enough) to warrant writing a map utility that edits existing maps and also lets you create new ones. The editor, ROTTED, is a full-featured, easy-to-use Windows system that edits Triad maps using a simple image tiling system. (The attractive image tiles were provided by Apogee Software.) You can find ROTTED.ZIP on CompuServe in the 3-D Action Games library of the Action Games forum (GO ACTION).

Not Another Doom Wannabe

Many Doom clones have appeared in recent months, but Triad isn't one of them. It has enough outstanding features to be in a category by itself. The ray-casting engine is unique, the game world contains several hundred creative objects and actors, and the mapping system is one of the most flexible that I have seen. The Apogee "Developers Of Incredible Power" are definitely living up to their name! ■

Wayne Sikes has been a computer hardware and software engineer for the last 12 years. He has an extensive background in C, C++, and assembly language programming. He also has several years experience as a computer systems intelligence analyst, a field in which he specialized in deciphering and disassembling computer code while working on classified government projects. He has written numerous computer gaming help utilities. You can reach him via e-mail at 70733.1562@compuserve.com or through Game Developer *magazine.*

Back

Make Your Mark

David Sieks

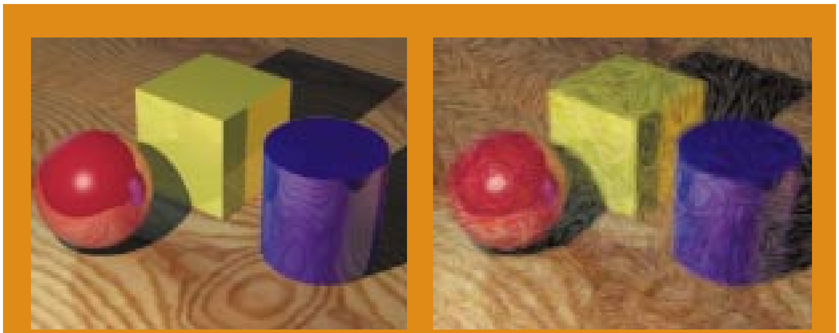
When you first studied art, were you using a computer? Probably not. Here are some tools that let three-dimensional artists get back to their roots and apply many traditional techniques electronically.

There is something intrinsically convincing about a three-dimensional rendering. Even when an extremely photorealistic effect has not been attempted, the visual cues inherent to three-dimensional art still manage to satisfy the subconscious in some basic way. For the artist, a good three-dimensional design package can facilitate the realization of creative vision and remove barriers to expression, imbuing even fantastic images or effects with uncanny authenticity.

I enjoy working in three dimensions because it allows me to flex different creative muscles than "working flat." When I'm working in the three-dimensional environment, my role is more akin to that of sculptor, lighting and set designer, costumer, makeup effects artist, and puppeteer all rolled into one. However, beyond the challenges of these various roles and the challenge of mastering the modeling and rendering software itself, three-dimensional design presents the artist with another more insidious challenge: maintaining an identity.

Convincingly solid though the images may be, and despite the possibilities of numerous special effects, many three-dimensional images can tend to display a certain homogeneity. I do not imply that creativity and talent are in less demand when you use this medium, nor do I suggest that these qualities are less evident in a three-dimensional rendering.

Too often what is lacking, however, is what we might call evidence of the artist's hand. In a two-dimensional work, the personality of the artist resides closer to the surface; even when the computer is used to create the image, a less digital gimmickry exists between concept and product. When



Handcrafted character is a mouse click away. It took less than a minute to transform a ray-traced truSpace image into an impressionistic painting using the Auto Van Gogh feature in Fractal Design's Painter 3. An essentially infinite array of tools and user-defined settings let you make subtle or sweeping changes automatically or one brush stroke at a time, to imitate all sorts of natural media materials and surfaces.

depth is created not by a sophisticated algorithm but by the mind, eye, and hand of the artist, that is a small sort of magic. In many cases, an artist's sketchbook will contain more of the unique flair and flavor of that artist than the final texture-mapped, light-sourced, fully rendered image or animation.

The blurring of individual style that is, not infrequently, a by-product of three-dimensional art can even be viewed as advantageous. When several artists contribute to a game's graphics, the unintentional homogenizing effect of three-dimensional rendering can work to lend a uniformity of appearance to the final whole. Yet while it may be desirable for the different graphic elements of your game to have constancy, it is less than desirable for that game to look like everyone else's game. Distinctive graphics can be one of the first hooks to attract a game buyer to your title.

Three-dimensional modeling and rendering tools should not be viewed as a shortcut or a high-tech trick, but as one more medium that we as artists must work to make the most of. The novelty has been worn thin enough that we—and the game-buying public—can begin to see beyond the superficial gratification provided by shadowmaps and perfect specular highlights and now want more. Merely using three-dimensional tools to render title screens, cut scenes, backgrounds, sprites, or any other game element is not in itself going to score anyone a lot of points.

The coming generation of fast-twitch, real-time-rendered-on-the-fly, three-dimensional games will still be buoyed by the "Gee whiz" factor. But prerendered three-dimensional material is no longer news and viewers will look at it with an increasingly jaded eye as the inevitable "Ho-hum" factor sets in. As artists, we must continually strive to keep the visual elements of our games—from title sequence to gameplay and everything in between—as distinctive and compelling as we know how.

Using What We Know

Before coming to the computer, most of us had our artistic roots in drawing and painting on paper or canvas. For me, those experiences—and the years spent developing my own approach to that work—remain one of the cornerstones of my artistic sensibility and continue to influence my work in digital media today. If you're interested in translating the hands-on quality and distinctive personal style of natural media to the digital medium, then you'll want to take a look at Painter. In its third release, Painter, from Fractal Design, brings to the computer screen a staggering array of effects amazingly similar to those created with traditional art materials like pen and ink, airbrush, watercolor, or even glossy, gloppy impasto oils.

Primarily an illustration package, Painter—with your help—can create stunning images that are all the more amazing because they look for all the world like they were created not on the computer but at an easel. The range of possibilities is essentially limitless, with so many variables provided to affect the look of an image that you will while away countless happy hours experimenting with different techniques and still discover new surprises—surprises you can then pass on to the players of your game.

In Painter, you work with "brushes" of user-defined shape and size, even allowing control over the action of the virtual bristles—with settings for thickness, clumpiness, and hair scale. You are also able to control the action of the brush stroke, which can be predefined to be made up of smaller multiple strokes of different colors for a soft, impressionistic effect. The marks you make mimic the appearance of pencil, crayon, marker, chalk, charcoal, oil pastel, pen and ink, scratchboard, watercolors, oils, and airbrush—each with its own menu of variations.

You can further manipulate the image with the simulated effect of an eraser, droplets of water, bleach, the darkroom techniques of dodging and burning, and masking that allows hard or feathered edges or even conforms to the texture of the surface you work on. Yes, I know you're working on paper or canvas, but your audience won't: Painter lets you select a virtual surface on which to work. This can be as smooth and featureless as the screen of your typical painting program, or can look like a wide variety of interesting textures that can be customized to suit your needs. The art materials you use interact with the selected surface much like their natural media counterparts.

There's almost too much in Painter 3, and the system of "drawers" and dropdown menus and pop-up dialogs with multiple slider controls can be a bit bewildering to navigate at times. Documentation is a bit thin but is generally sufficient to point you in the right direction. More help can be found in *Inside Fractal Design Painter 3.0* (Macmillan) and *Artistry*, a Painter newsletter published 10 times a year, both by Karen Sperling. Happily, your customized brushes, textures, color sets, and "sessions" of applied effects can be named and saved, so once you figure something out you can keep it for later use.

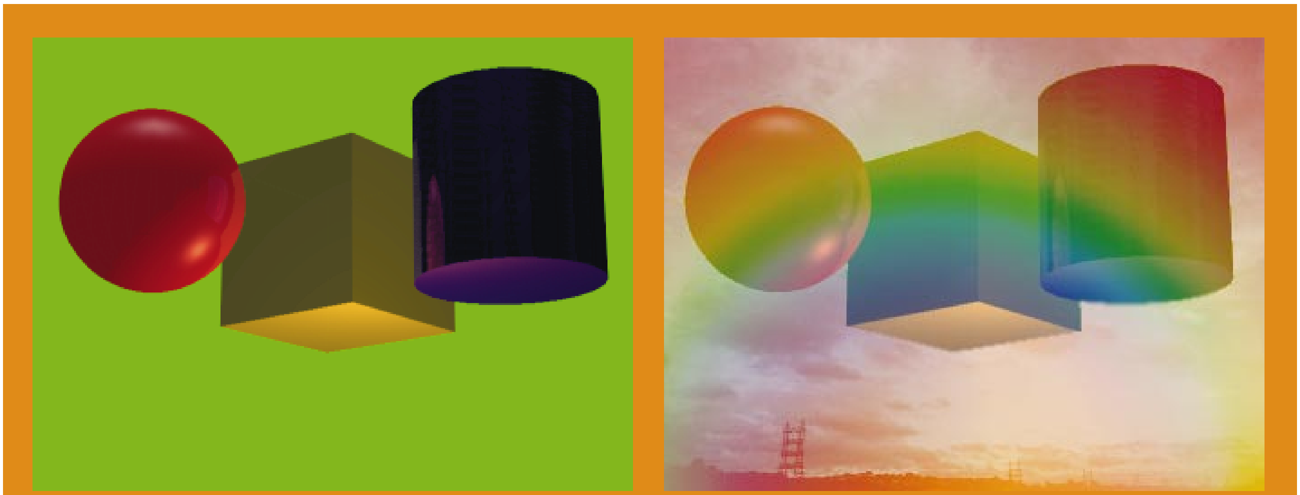
The sheer range of visual styles possible with Painter and the unexpectedness of such natural media techniques on the computer screen should start some ideas percolating in your head already. But more than just a way to paint without making a mess, this program throws in the abilities of a powerful image editor and a capable animation tool.

Existing images or animations can be opened in Painter and treated to subtle retouching using any of the painting tools or plug-ins including most Photoshop filters. Or they can be cloned and completely reworked. By recording a session, it's possible to apply an effect or multiple effects to an entire animation, introducing a surface texture or causing the whole to appear rendered by the staccato dabblings of impressionist master George Seurat, for example. I've been experimenting with various painterly effects to rework three-dimensional stills and even whole animations. The results nicely combine the authenticity of the three-dimensional rendering with the freshness and vitality of an actual painting. You wouldn't use this sort of effect for everything, but it can be a great way to tone down the slickness of a three-dimensional image for a historical period setting or whenever you might want a more painterly, personal look.

Another neat crossover between Painter and your work in three-dimensional graphics is the ability to create your own seamless tiles with Painter's Capture Pattern utility. A wraparound color technique causes marks that stray off the edge of your pattern to continue on the opposite side. So you can easily use Painter's various media looks and surface textures to create your own materials to map to three-dimensional objects.

Creating an animation in Painter is very much akin to the traditional cel techniques of film animation. That is, it lacks a lot of the improvements that more dedicated animation software can provide, like tweening between keyframes and morphing one shape into another.

If you're experienced in cel animation, moving the sketching, inking, and painting of frames off the peg bar and onto the computer will seem a real boon. (Most animation houses have abandoned cels and computerized these processes as well.) If you're used to three-dimensional animation or more sophisticated animation effects software, however, Painter may seem a bit underfeatured in this department. Except when applying a prerecorded session to an entire animation (called a Frame Stack in Painter), you're pretty much drawing frame by frame. Onionskinning lets a series of subsequent frames show through, one on top of the other (as many as five at a time), which aids in progressing a movement through its various stages. A tracing paper function also allows rotoscoping, so that digitized video can be reinterpreted by hand, which can be a lifesaver when it comes to capturing the timing and nuances of human or animal movement in a hand-drawn scene.



Your three-dimensional renderer doesn't have to do all the work. These geometric forms were created in Caligari's truSpace and the lurid background added at rendering time to act as a "green screen" for later compositing. The sophisticated selection tools in Autodesk's Animator Studio easily separate the forms from the background in every frame, without any hint of a halo. Composited with a background animation, the whole is treated with Kai's Power Tool filters for the final effect.

Editing from Autodesk

Another way to get the most out of your digital animations is to polish them in post with a suite of sophisticated editing tools, and your old friends at Autodesk want to help. "Sound, paint, and motion...Complete 2D animation for Windows" is how the company bills its brand new Animator Studio. This is more than just an update of Autodesk's popular Animator Pro—indeed, it is all you need to make walking, talking animations. Users of that earlier title will be comfortable with most of the methods and conventions utilized here, though the switch from DOS to Windows has simplified and improved some processes. To get Animator Pro users quickly up to speed, the changes have been well documented and the tutorials take nothing for granted.

Moving beyond the capabilities of Animator Pro, the animation functions now exist as the main of four supporting modules: Animator, Soundlab, Scriptor, and Player. Soundlab is a very usable editor that lets you create and filter audio files and synch them with your animations. Player runs AVI, FLI, FLC, and Quicktime files and can be freely distributed. Scriptor is intended to string together animation and audio files for presentations or self-playing shows, but in its first release I found it still quite buggy. Hopefully, Autodesk will have released patches to remedy these problems before you even have a chance to read about them here. While these three modules round out the package for general use, they are of less concern to us than Animator, which is packed with useful animating and editing features that can help you introduce interesting effects to existing animation files or create new ones from the ground up.

The first thing that struck me about this new product was the manual. That is, the—as in one, singular—manual, rather than the collection of weighty tomes traditionally received with an Autodesk title. We're not talking about a thick book here, either; as software reference goes, this verges on slender. Online help duplicates information from the manual and adds to it, and as a nice extra allows you to add your own notes and bookmarks to return you to sections you use most. As this package is not short on features, it is a testimony to the usable nature of the interface that one slim volume proves sufficient.

That interface remains admirably uncluttered despite the wealth of features it manages to support. Movable selection boxes for tools, filters, and colors default to a neat, out-of-the way row to screen left. When an animation is opened, it appears as a filmstrip across the top of the screen, while the active frame is shown at full size in a roomy central work area. Pop-up dialogs provide fine-tuning control for almost every aspect of this program. On a 1,024-by-780 screen, everything fit with room to spare. After working your way through a couple of tutorials you'll feel right at home with Animator's tools and workspace.

To start with, you can paint into a frame to create a new image or alter an existing one. To do so, you select one of several available brushes. Brushes can also be customized, though not to the degree possible in Painter. Pressure sensitivity is supported for stylus users, and there is also an airbrush tool with variable controls, and the usual straight-line, curved-line, and rectangle tools. Compared to Painter, these provisions seem somewhat skimpy, but they fulfill the basics. Using a stylus and the onion-skin feature you can create nice hand-drawn animations, and a multiple undo lets you easily fix slip-ups. The real strengths of the program, however, are its labor-and-time-saving animation shortcuts and a useful array of sophisticated editing capabilities.

A range of about thirty "inks" are really filters for applying special effects. These include such nifty items as Clone ink, which copies brushstroke by brushstroke the area you paint into with it; Colorize ink, which adds only hue, leaving lightness values unaffected (sort of like coloring over a black-and-white photo with a translucent marker); Soften ink, which gently blurs an area; and Jumble ink, which adds a more chaotic blurring effect. These and the rest of the ink choices allow you to really tailor the appearance of an animation to meet your needs.

Another useful ink selection provided by Animator is Alpha ink. In addition to color-keyed alpha channel support, this allows you to affect the opacity of a specific area by painting into it. In addition to increased compositing possibilities, this lets you remove unwanted elements from an animation by rendering them transparent. So you can, for instance, mark up the frame with guidelines to indicate the paths of animated objects, then use clone ink to make the guidelines invisible when you're done with them. Used in conjunction with onionskinning, this can really facilitate hand-drawn animations. Anime, anyone?

Perhaps best of all, the selection tools in Animator are flexible and sophisticated. By keying on the background color with the Magic Wand tool, for example, you can select out the entire background, leaving the foreground object for you to cut and composite or modify with some other effect. You can instantly apply this sort of selection over time to affect a segment so that you don't need to work frame by frame. It is also possible to add to a selection before acting on it, so several areas within a frame can be indicated and edited as one.

A selection—even one spanning multiple frames—can be saved as a sprite and then painted into a different animation. When you add an animated sprite to a movie, its movement is automatically extrapolated over the segment. You can then modify the sprite's motion path by clicking and dragging handles along its course, which can really help fine-tune a movement. Animator also provides predefined Action Envelopes that can be applied to a sprite to automatically cause its movement to gradually accelerate or decelerate as it progresses.

Defining a segment creates two keyframes as the start and finish of that segment and you can easily apply desired changes over time between these keyframes. These changes can include color interpolation (a gradual shift from one color to another), morphing one shape into another (as long as both comprise the same number of vertices), and movement of an object from one area of the picture frame to another. All that is required is to set the first and last keyframes. The in-between transition is then automated.

The selecting and editing features of Animator provide some powerful and flexible tools for altering and compositing digitized video or animations you might already have made in a three-dimensional program or elsewhere. To expand on that flexibility, Animator is compatible with Photoshop plug-ins and, for its initial release, is being bundled with a special Kai's Power Tools package that includes Gradient Designer. This adds such an abundance of functionality to Animator right out of the box that you'll not soon run out of new looks to bring to your animations.

When there are no surprises left for the artist, few are to be expected for the audience. It should be our task to contribute what we can to keep game players in a continual state of surprise, wonderment, and delight, and that means never getting too comfortable with the way we do things so that players never get bored with what they're seeing. These are just two possible tools that might help you bring your graphics to the next level, and bring the players with you. Which is, after all, where you want to keep them. ■

David Sieks is a contributing editor to Game Developer. *Contact him via e-mail at dsieks@arnarb.harvard.edu or through* Game Developer *magazine.*